3

Cryptographic Techniques

3.1 BLOCK AND STREAM CIPHERS

Let M be a plaintext message. A **block cipher** breaks M into successive blocks M_1, $M_2, \ldots$, and enciphers each M_1 with the same key K; that is,

$$E_K(M) = E_K(M_1)E_K(M_2) \ldots .$$

Each block is typically several characters long. Examples of block ciphers are shown in Table 3.1. Simple substitution and homophonic substitution ciphers are block ciphers, even though the unit of encipherment is a single character. This is because the same key is used for each character. We shall return to block ciphers in Section 3.4.

A **stream cipher** breaks the message M into successive characters or bits m_1, $m_2, \ldots$, and enciphers each m_i with the ith element k_i of a **key stream** $K = k_1k_2 \ldots$; that is,

TABLE 3.1 Block ciphers.

Cipher	Block size
Transposition with period d	d characters
Simple substitution	1 character
Homophonic substitution	1 character
Playfair	2 characters
Hill with $d \times d$ matrix	d characters
DES	64 bits
Exponentiation mod n	$\log_2 n$ bits (664 bits recommended)
Knapsacks of length n	n bits (200 bits recommended)

$$E_K(M) = E_{k_1}(m_1)E_{k_2}(m_2) \ldots .$$

A stream cipher is **periodic** if the key stream repeats after d characters, for some fixed d; otherwise, it is nonperiodic. Ciphers generated by Rotor and Hagelin machines are periodic stream ciphers. The Vernam cipher (one-time pad) and running-key ciphers are nonperiodic stream ciphers.

A periodic substitution cipher with a short period (e.g., Vigenère cipher) is normally regarded as a stream cipher because plaintext characters are enciphered one by one, and adjacent characters are enciphered with a different part of the key. But it has characteristics in common with both types of ciphers. Let $K = k_1k_2 \ldots k_d$, where d is the period of the cipher. The cipher can be regarded as a block cipher, where each M_i is a block of d letters:

$$E_K(M) = E_K(M_1)E_K(M_2) \ldots ,$$

or as a stream cipher, where each m_i is one letter, and K is repeated in the key stream; that is, the key stream is:

$$\overbrace{k_1 \ldots k_d}^{K} \quad \overbrace{k_1 \ldots k_d}^{K} \quad \overbrace{k_1 \ldots k_d}^{K} \ldots$$

For short periods, the cipher is more like a block cipher than a stream cipher, but it is a weak block cipher because the characters are not diffused over the entire block. As the length of the period increases, the cipher becomes more like a stream cipher.

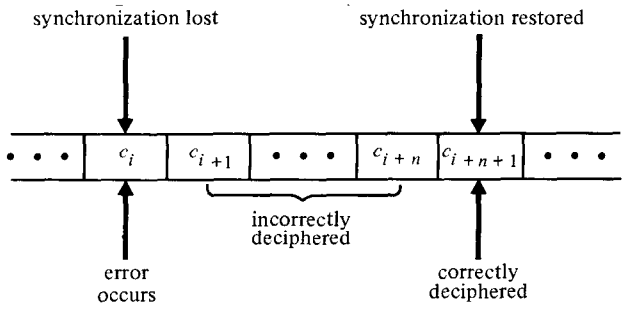
There are two different approaches to stream encryption: synchronous methods and self-synchronous methods (see Table 3.2). In a **synchronous stream cipher**, the key stream is generated independently of the message stream. This means that if a ciphertext character is lost during transmission, the sender and receiver must

TABLE 3.2 Stream ciphers.

Synchronous stream ciphers	Period
Vigenère with period d	d
Rotor machines with t rotors	26^t
Hagelin machines with t wheels, each having p_i pins	$p_1p_2 \cdots p_t$
Running-key	—
Vernam	—
Linear Feedback Shift Registers with n-bit register	2^n
Output-block feedback mode with DES†	2^{64}
Counter method with DES	2^{64}
Self-synchronous methods	
Autokey cipher	
Cipher feedback mode	

† It can be less; see [Hell80].

FIGURE 3.1 Propagation of error with self-synchronous stream cipher.



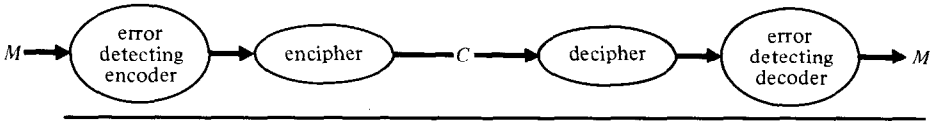
resynchronize their key generators before they can proceed further. Furthermore, this must be done in a way that ensures no part of the key stream is repeated (thus the key generator should not be reset to an earlier state). All the stream ciphers we have discussed so far are synchronous. Section 3.2 describes three methods better suited for digital computers and communications: Linear Feedback Shift Registers, output-block feedback mode, and the counter method. These ciphers are also periodic.

In a **self-synchronous stream cipher**, each key character is derived from a fixed number n of preceding ciphertext characters. Thus, if a ciphertext character is lost or altered during transmission, the error propagates forward for n characters, but the cipher resynchronizes by itself after n correct ciphertext characters have been received (see Figure 3.1). Section 3.3 describes two self-synchronous stream ciphers: autokey ciphers and cipher feedback mode. Self-synchronous stream ciphers are nonperiodic because each key character is functionally dependent on the entire preceding message stream.

Even though self-synchronous stream ciphers do not require resynchronization when errors occur, transmission errors cannot be ignored. The errors could be a sign of active wiretapping on the channel. Even if the errors are not caused by wiretapping, retransmission is necessary if the application requires recovery of lost or damaged characters.

Although protocols for recovering from transmission errors are beyond the scope of this book, we shall briefly describe the role of error detecting and correcting codes in cryptographic systems. Diffie and Hellman [Diff79] observe that if errors are propagated by the decryption algorithm, applying error detecting codes before encryption (and after decryption—see Figure 3.2) provides a mechanism for authenticity, because modifications to the ciphertext will be detected by the

FIGURE 3.2 Encryption used with error detecting codes.



error decoder. Block ciphers and self-synchronous stream ciphers propagate errors, so this strategy is applicable for both of these modes of operation. Synchronous stream ciphers, on the other hand, do not propagate errors because each ciphertext character is independently enciphered and deciphered. If a fixed linear error detecting code is used, then an opponent could modify the ciphertext character and adjust the parity bits to match the corresponding changes in the message bits. To protect against this, a keyed or nonlinear error detecting code can be used. Error correcting codes must be applied after encryption (because of the error propagation by the decryption algorithm), but can be used with error detecting codes (applied before encryption) for authentication.

Communication protocols for initiating and terminating connections and synchronizing key streams are also beyond the scope of this book. It is worth noting, however, that such protocols must require message acknowledgement to detect deletion of messages. (For a more detailed description of the cryptography-communications interface, see [Bran75,Bran78,Feis75,Kent76,Pope79].)

All the stream ciphers discussed in this chapter use a simple exclusive-or operation for enciphering and deciphering (as in the Vernam cipher). Thus, the enciphering algorithm is given by:

$$c_i = E_{k_i}(m_i) = m_i \oplus k_i \ ,$$

where each k_i, m_i, and c_i is one bit or character. The deciphering algorithm is the same:

$$\begin{aligned} D_{k_i}(c_i) &= c_i \oplus k_i \\ &= (m_i \oplus k_i) \oplus k_i \\ &= m_i \ . \end{aligned}$$

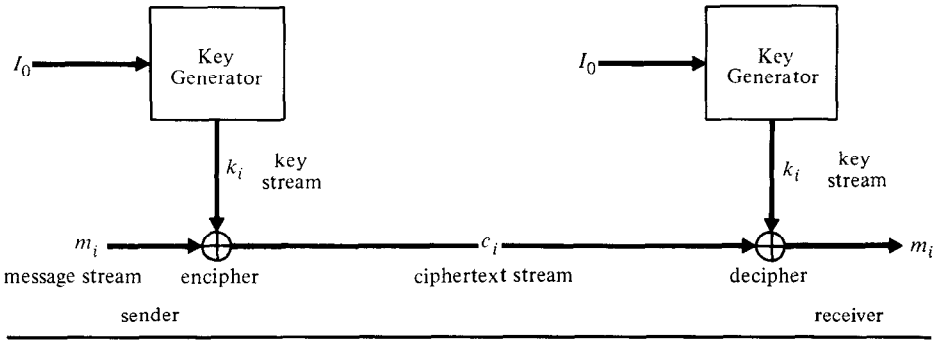
3.2 SYNCHRONOUS STREAM CIPHERS

A synchronous stream cipher is one in which the key stream $K = k_1 k_2 \ldots$ is generated independently of the message stream. The algorithm that generates the stream must be deterministic so the stream can be reproduced for decipherment. (This is unnecessary if K is stored, but storing long key streams may be impractical.) Thus, any algorithm that derives the stream from some random property of the computer system is ruled out. The starting stage of the key generator is initialized by a "seed" I_0. Figure 3.3 illustrates.

We saw in Chapter 2 that stream ciphers are often breakable if the key stream repeats or has redundancy; to be unbreakable, it must be a random sequence as long as the plaintext. Intuitively, this means each element in the key alphabet should be uniformly distributed over the key stream, and there should be no long repeated subsequences or other patterns (e.g., see [Knut69,Brig76] for a discussion of criteria for judging randomness).

No finite algorithm can generate truly random sequences [Chai74]. Although this does not rule out generating acceptable keys from pseudo-random number generators, the usual congruence type generators are unacceptable.

FIGURE 3.3 Synchronous stream cipher.



Even a good pseudo-random number generator is not always suitable for key generation. Linear Feedback Shift Registers are an example; given a relatively small amount of plaintext-ciphertext pairs, a cryptanalyst can easily derive the entire key stream. Because this technique illustrates the possible pitfalls of key generators, we shall describe it before turning to methods that appear to be much stronger.

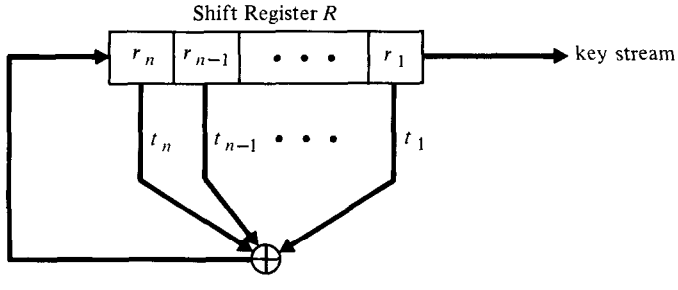
3.2.1 Linear Feedback Shift Registers

An n-stage **Linear Feedback Shift Register** (LFSR) consists of a shift register $R = (r_n, r_{n-1}, \ldots, r_1)$ and a "tap" sequence $T = (t_n, t_{n-1}, \ldots, t_1)$, where each r_i and t_i is one binary digit. At each step, bit r_1 is appended to the key stream, bits $r_n, \ldots, r_2$ are shifted right, and a new bit derived from T and R is inserted into the left end of the register (see Figure 3.4). Letting $R' = (r'_n, r'_{n-1}, \ldots, r'_1)$ denote the next state of R, we see that the computation of R' is thus:

$$r'_i = r_{i+1} \quad i = 1, \ldots, n-1$$

$$r'_n = TR = \sum_{i=1}^{n} t_i r_i \bmod 2 = t_1 r_1 \oplus t_2 r_2 \oplus \ldots \oplus t_n r_n \ .$$

FIGURE 3.4 Linear Feedback Shift Register (LFSR).



Thus,

$$R' = HR \bmod 2 \ , \tag{3.1}$$

where H is the $n \times n$ matrix:

$$H = \begin{pmatrix} t_n & t_{n-1} & t_{n-2} & \cdots & t_3 & t_2 & t_1 \\ 1 & 0 & 0 & & 0 & 0 & 0 \\ 0 & 1 & 0 & & 0 & 0 & 0 \\ 0 & 0 & 1 & & 0 & 0 & 0 \\ & . & & . & & . & \\ & . & & . & & . & \\ & . & & . & & . & \\ 0 & 0 & 0 & & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{pmatrix} .$$

An n-stage LFSR can generate pseudo-random bit strings with a period of $2^n - 1$ (e.g., see [Golu67]). To achieve this, the tap sequence T must cause R to cycle through all $2^n - 1$ nonzero bit sequences before repeating.This will happen if the polynomial

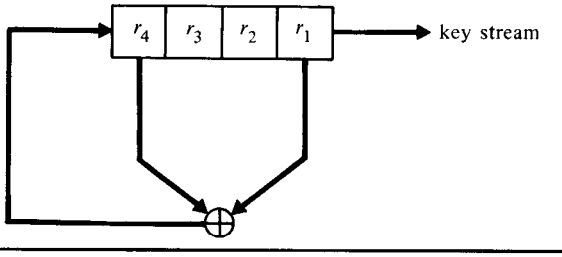
$$T(x) = t_n x^n + t_{n-1} x^{n-1} + \cdots + t_1 x + 1 \ ,$$

formed from the elements in the tap sequence plus the constant 1, is primitive. A **primitive polynomial** of degree n is an irreducible polynomial that divides $x^{2^n-1} + 1$, but not $x^d + 1$ for any d that divides $2^n - 1$. Primitive trinomials of the form $T(x) = x^n + x^a + 1$ are particularly appealing, because only two stages of the feedback register need be tapped. (See [Golu67,Pete72,Zier68,Zier69] for tables of primitive polynomials.)

Example:
Figure 3.5 illustrates a 4-stage LFSR with tap sequence $T = (1, 0, 0, 1)$; thus there are "taps" on bits r_1 and r_4. The matrix H is given by

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} .$$

FIGURE 3.5 Four-stage LFSR.



The polynomial $T(x) = x^4 + x + 1$ is primitive, so the register will cycle through all 15 nonzero bit combinations in $\mathbf{GF}(2^4)$ before repeating. Starting R in the initial state 0001, we have

0	0	0	1
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
0	1	1	0
0	0	1	1
1	0	0	1
0	1	0	0
0	0	1	0

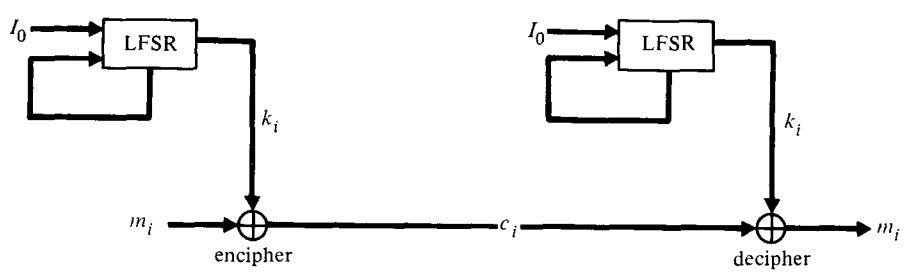
The rightmost column gives the key stream $K = 100011110101100$. ■

A binary message stream $M = m_1 m_2 \ldots$ is enciphered by computing $c_i = m_i \oplus k_i$ as the bits of the key stream are generated (see Figure 3.6). Deciphering is done in exactly the same way; that is, by regenerating the key stream and computing $c_i \oplus k_i = m_i$. The seed I_0 is used to initialize R for both encipherment and decipherment.

The feedback loop attempts to simulate a one-time pad by transforming a short key (I_0) into a long pseudo-random sequence K. Unfortunately, the result is a poor approximation of the one-time pad.

The tap sequence T is easily determined in a known-plaintext attack [Meye 73,Meye72]. Following the description in [Diff79], we shall show how this is done using just $2n$ bits of plaintext-ciphertext pairs. Let $M = m_1 \ldots m_{2n}$ be the plaintext corresponding to ciphertext $C = c_1 \ldots c_{2n}$. We can determine the key sequence $K = k_1 \ldots k_{2n}$ by computing

FIGURE 3.6 Encryption with LFSR.



$$m_i \oplus c_i = m_i \oplus (m_i \oplus k_i) = k_i \ ,$$

for $i = 1, \ldots, 2n$.

Let R_i be a column vector representing the contents of register R during the ith step of the computation. Then

$$\begin{aligned} R_1 &= (k_n, k_{n-1} \ \ldots, k_1) \\ R_2 &= (k_{n+1}, k_n \ \ldots, k_2) \\ &\vdots \\ R_{n+1} &= (k_{2n}, k_{2n-1} \ \ldots, k_{n+1}) \ . \end{aligned}$$

Let X and Y be the following matrices:

$$\begin{aligned} X &= (R_1, R_2, \ldots, R_n) \\ Y &= (R_2, R_3, \ldots, R_{n+1}) \ . \end{aligned}$$

Using Eq. (3.1), we find that X and Y are related by

$$Y = HX \bmod 2 \ .$$

Because X is always nonsingular, H can be computed from

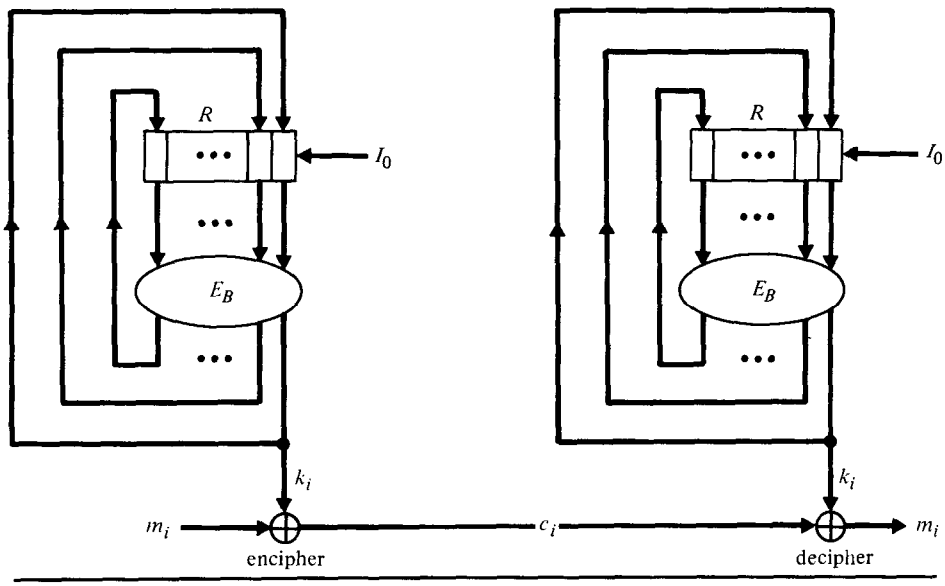
$$H = YX^{-1} \bmod 2 \qquad \textbf{(3.2)}$$

and T can be obtained from the first row of H. The number of operations required to compute the inverse matrix X^{-1} is on the order of n^3, whence the cipher can be broken in less than 1 day on a machine with a 1 μsec instruction time for values of n as large as 1000.

3.2.2 Output-Block Feedback Mode

The weakness of LFSRs is caused by the linearity of Eq. (3.1). A better approach is to use a nonlinear transformation. Nonlinear block ciphers such as the DES seem to be good candidates for this. Figure 3.7 illustrates an approach called **output-block feedback mode** (OFM). The feedback register R is used as input to a block encryption algorithm E_B with key B. During the ith iteration, $E_B(R)$ is computed, the low-order (rightmost) character of the output block becomes the ith key character k_i, and the entire block is fed back through R to be used as input during the next iteration. Note that each k_i is one character rather than just a single bit; this is to reduce the number of encipherments with E_B, which will be considerably more time-consuming than one iteration of a LFSR. A message stream is broken into characters and enciphered in parallel with key generation as described earlier. The technique has also been called **internal feedback** [Camp78] because the feedback is internal to the process generating the key stream; by contrast, the self-synchronous method described in Section 3.3.2 uses a feedback loop derived from the ciphertext stream. (See [Gait77] for a discussion of using DES in OFM.)

FIGURE 3.7 Synchronous stream cipher in output-block feedback mode (OFM).

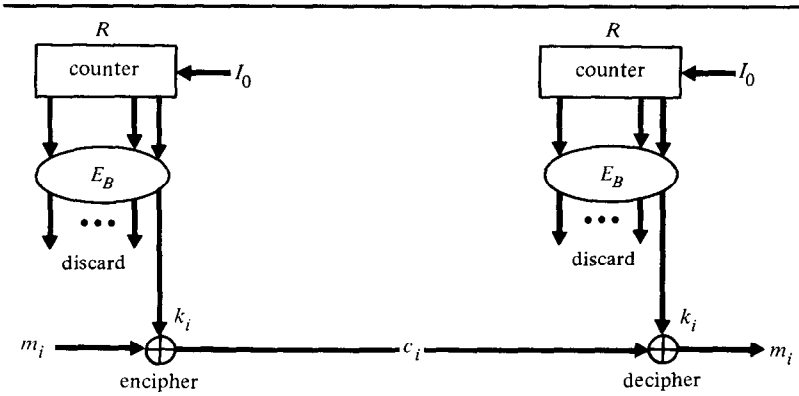


3.2.3 Counter Method

Diffie and Hellman [Diff79,Hell80] have suggested a different approach called the **counter method**. Rather than recycling the output of E_B back through E_B, successive input blocks are generated by a simple counter (see Figure 3.8).

With the counter method, it is possible to generate the ith key character k_i without generating the first $i - 1$ key characters by setting the counter to $I_0 + i - 1$. This capability is especially useful for accessing the ith character in a direct access file. With OFM, it is necessary to first compute $i - 1$ key characters.

FIGURE 3.8 Synchronous stream cipher in counter mode.



In Section 2.4.4, we saw that stream ciphers could be broken if the key stream repeated. For this reason, synchronous stream ciphers have limited applicability to file and database encryption; if an element is inserted into the middle of a file, the key stream cannot be reused to reencipher the remaining portion of the file. To see why, suppose an element m' is inserted into the file after the ith element. We have:

original plaintext:	$\ldots$	m_i	m_{i+1}	m_{i+2} $\ldots$
key stream:	$\ldots$	k_i	k_{i+1}	k_{i+2} $\ldots$
original ciphertext:	$\ldots$	c_i	c_{i+1}	c_{i+2} $\ldots$
updated plaintext:	$\ldots$	m_i	m'	m_{i+1} $\ldots$
key stream:	$\ldots$	k_i	k_{i+1}	k_{i+2} $\ldots$
updated ciphertext:	$\ldots$	c_i	c'_{i+1}	c'_{i+2} $\ldots$

Bayer and Metzger [Baye76] show that if m' is known, then all key elements k_j and plaintext elements m_j $(j > i)$ can be determined from the original and updated ciphertext:

$$k_{i+1} = c'_{i+1} \oplus m' \qquad m_{i+1} = c_{i+1} \oplus k_{i+1}$$
$$k_{i+2} = c'_{i+2} \oplus m_{i+1}, \qquad m_{i+2} = c_{i+2} \oplus k_{i+2}$$
$$\vdots$$

Note that a cryptanalyst does not need to know the position in the file where the insertion is made; this can be determined by comparing the original and updated versions of the file.

Synchronous stream ciphers protect against ciphertext searching, because identical blocks of characters in the message stream are enciphered under a different part of the key stream. They also protect against injection of false ciphertext, replay, and ciphertext deletion, because insertions or deletions in the ciphertext stream cause loss of synchronization.

Synchronous stream ciphers have the advantage of not propagating errors; a transmission error affecting one character will not affect subsequent characters. But this is also a disadvantage in that it is easier for an opponent to modify (without detection) a single ciphertext character than a block of characters. As noted earlier, a keyed or nonlinear error detecting code helps protect against this.

3.3 SELF-SYNCHRONOUS STREAM CIPHERS

A self-synchronous stream cipher derives each key character from a fixed number n of preceding ciphertext characters [Sava67]. The genesis of this idea goes back to the second of two autokey ciphers invented by Vigenère in the 16th Century [Kahn67]. We shall first describe Vigenère's schemes, and then describe a method suited for modern cryptographic systems.

3.3.1 Autokey Ciphers

An **autokey cipher** is one in which the key is derived from the message it enciphers. In Vigenère's first cipher, the key is formed by appending the plaintext $M = m_1 m_2 \ldots$ to a "priming key" character k_1; the ith key character ($i > 1$) is thus given by $k_i = m_{i-1}$.

Example:
Given the priming key D, the plaintext RENAISSANCE is enciphered as follows, using a shifted substitution cipher (Vigenère actually used other substitutions):

$$\begin{aligned} M &= \text{RENAISSANCE} \\ K &= \text{DRENAISSANC} \\ E_K(M) &= \text{UVRNIAKSNPG}. \end{aligned}$$ ■

In Vigenère's second cipher, the key is formed by appending each character of the ciphertext to the priming key k_1; that is, $k_i = c_{i-1}$ $(i > 1)$.

Example:
Here the plaintext RENAISSANCE is enciphered with the priming key D as follows:

$$\begin{aligned} M &= \text{RENAISSANCE} \\ K &= \text{DUYLLTLDDQS} \\ E_K(M) &= \text{UYLLTLDDQSW}. \end{aligned}$$ ■

Of course, neither of these ciphers is strong by today's standards. But Vigenère's discovery that nonrepeating key streams could be generated from the messages they encipher was a significant contribution to cryptography.

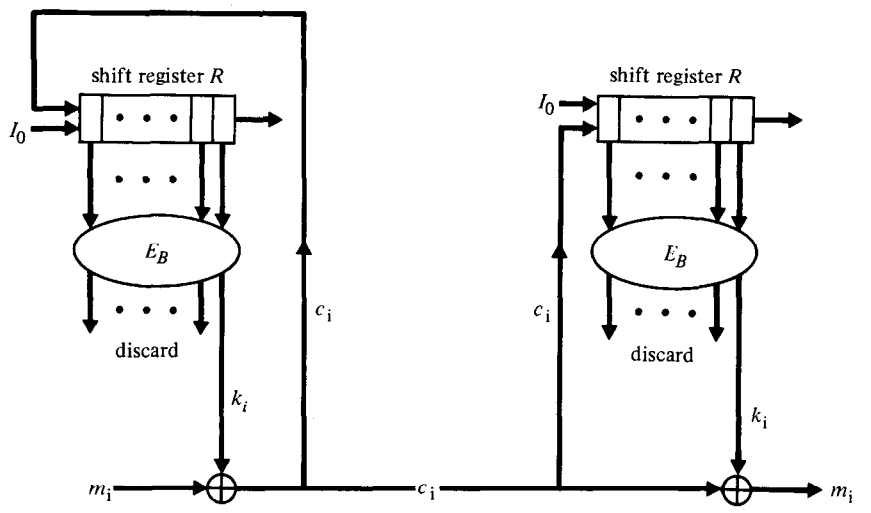
Vigenère's second autokey cipher is a self-synchronous system in the sense that each key character is computed from the preceding ciphertext character (here, the computation is a simple identity operation).

Even though each key character can be computed from its preceding ciphertext character, it is functionally dependent on all preceding characters in the message plus the priming key. Thus, each ciphertext character is functionally dependent on the entire preceding message. This phenomenon, sometimes called "garble extension", makes cryptanalysis more difficult, because the statistical properties of the plaintext are diffused across the ciphertext.

3.3.2 Cipher Feedback

Vigenère's system is weak because it exposes the key in the ciphertext stream. This problem is easily remedied by passing the ciphertext characters through a nonlinear block cipher to derive the key characters. The technique is called **cipher feed-**

FIGURE 3.9 Self-synchronous stream cipher in cipher feedback mode (CFB).



back mode (CFB) because the ciphertext characters participate in the feedback loop. It is sometimes called "chaining", because each ciphertext character is functionally dependent on (chained to) preceding ciphertext characters. This mode has been approved by the National Bureau of Standards for use with DES [GSA77].

Figure 3.9 illustrates. The feedback register R is a shift register, where each ciphertext character c_i is shifted into one end of R immediately after being generated (the character at the other end is simply discarded). As before, R is initialized to the seed I_0. During each iteration, the value of R is used as input to a block encryption algorithm E_B, and the low-order character of the output block becomes the next key character.

With CFB, transmission errors affect the feedback loop. If a ciphertext character is altered (or lost) during transmission, the receiver's shift register will differ from the transmitter's, and subsequent ciphertext will not be correctly deciphered until the erroneous character has shifted out of the register. Because the registers are synchronized after n cycles (where n is the number of characters per block), an error affects at most n characters; after that, the ciphertext is correct (see Figure 3.1).

CFB is comparable to the counter method in its ability to access data in random access files. To decipher the ith ciphertext character c_i, it suffices to load the feedback register with the n preceding ciphertext characters $c_{i-n}, \ldots, c_{i-1}$, and execute one cycle of the feedback loop to get k_i.

With CFB, it is possible to make an insertion or deletion in a file without reenciphering the entire file. It is, however, necessary to reencipher all characters after the place of insertion or deletion, or the following block of characters will not be decipherable. Reencipherment can be confined to a single record in a file by reinitializing the feedback loop for each record. Although this exposes identical

records, identical records can be concealed by prefixing each record with a random block of characters, which is discarded when the record is deciphered for processing. Note that cipher feedback is not vulnerable to the insertion/deletion attack described for synchronous ciphers. This is because the key stream is automatically changed by any change in the message stream.

Self-synchronous ciphers protect against ciphertext searching because different parts of the message stream are enciphered under different parts of the key stream. They also protect against all types of authenticity threats because any change to the ciphertext affects the key stream. Indeed, the last block of ciphertext is functionally dependent on the entire message, serving as a checksum for the entire message.

A **checksum** refers to any fixed length block functionally dependent on every bit of the message, so that different messages have different checksums with high probability. Checksums are frequently appended to the end of messages for authentication. The method of computing the checksum should ensure that two messages differing by one or more bits produce the same checksum with probability only 2^{-n}, where n is the length of the checksum. CFB can be used to compute checksums for plaintext data when authenticity is required in the absence of secrecy. Although a checksum does not usually provide the same level of security as encrypting the entire message, it is adequate for many applications.

3.4 BLOCK CIPHERS

We have seen how a block encryption algorithm E can be used to generate a key stream in either synchronous or self-synchronous mode. This raises an obvious question: Is it better to use a block encryption algorithm for block encryption, or to use it for stream encryption? Although the answer to this question depends on the requirements of the particular application, we can make some general observations about the efficiency and security of the different approaches.

Using block encryption directly is somewhat faster than either stream mode, because there will be only one execution of the encryption algorithm per n characters rather than n executions. This may not be an important factor, however, when the algorithm is implemented in special-purpose hardware capable of encrypting several million bits per second (as for some DES implementations). Such data rates are well beyond the capabilities of slow-speed telecommunications lines. Furthermore, block encryption is not inherently faster than stream encryption; stream encryption can be speeded up by using a faster generator, and, for applications requiring high speed, a key stream can be generated in advance with synchronous stream encryption (see [Brig80]).

With block encryption, transmission errors in one ciphertext block have no affect on other blocks. This is comparable to cipher feedback mode, where a transmission error in one ciphertext character affects only the next n characters, which is equivalent to one block.

Block encryption may be more susceptible to cryptanalysis than either stream mode. Because identical blocks of plaintext yield identical blocks of cipher-

text, blocks of blanks or keywords, for example, may be identifiable for use in a known-plaintext attack. This is not a problem with stream encryption because repetitions in the plaintext are enciphered under different parts of the key stream. With block encryption, short blocks at the end of a message must also be padded with blanks or zeros, which may make them vulnerable to cryptanalysis.

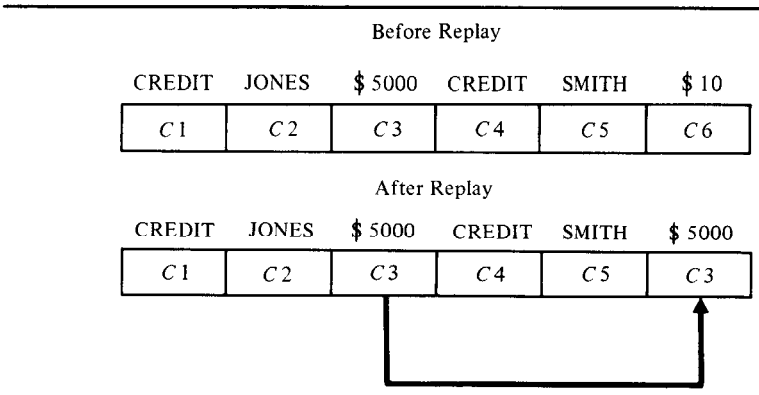
In database systems, enciphering each field of a record as a separate block is not usually satisfactory. If the fields are short, padding increases the storage requirements of the database, and may leave the data vulnerable to cryptanalysis. Enciphering the fields with an algorithm that operates on short blocks does not solve the problem, because the algorithm will be weaker.

Even if the fields are full size and cryptanalysis impossible, information can be vulnerable to ciphertext searching. Consider a database containing personnel records. Suppose the *Salary* field of each record is enciphered as a single block, and that all salaries are enciphered under the same key. The *Name* fields are in the clear, so that a particular individual's record can be identified. A user can determine which employees earn salaries equal to Smith's by searching for those records with ciphertext *Salary* fields identical to Smith's. This problem demonstrates the need to protect nonconfidential information as well as confidential information, and the possible pitfalls of enciphering database records by fields, especially when multiple records are enciphered under one key.

Block encryption is more susceptible to replay than stream encryption. If each block is independently enciphered with the same key, one block can be replayed for another. Figure 3.10 shows how a transaction "CREDIT SMITH $10" can be changed to "CREDIT SMITH $5000" by replaying a block containing the ciphertext for $5000 (see [Camp78]). This type of replay is not possible with a stream cipher (assuming the key stream is not repeated). One simple solution is to append checksums to the end of messages.

Replay can also be a problem in databases. Consider again the database of employee records, where each record contains a *Salary* field enciphered as a separate block, and all salaries are enciphered under one key. Suppose a user can

FIGURE 3.10 Replay of ciphertext block.



identify the records in the database belonging to Smith and to Jones, and that Jones's salary is known to be higher than Smith's. By copying Jones's enciphered *Salary* field into Smith's record, Smith's salary is effectively increased. The change will not be detected. Adding a checksum to each record can thwart this type of attack, but cannot protect against attacks based on ciphertext searching as described earlier.

Block encryption is also vulnerable to insertion and deletion of blocks, because these changes to the message stream do not affect surrounding blocks. Although it may be difficult to create false ciphertext for textual data, it may not be for numeric data. If, for example, the objective is simply to make the balance in an account nonzero, any positive integer will do. As noted earlier, applying error detecting codes before encryption protects against this threat. Checksums can also be added.

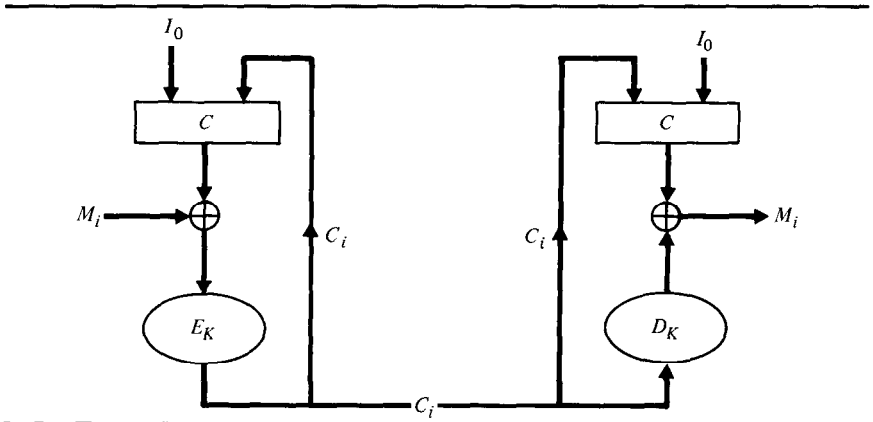
A database system with secure access controls can prevent unauthorized users from searching or modifying ciphertext (or plaintext) as described in these examples. Access controls are not always foolproof, however, and cannot generally prevent users from browsing through data on removable storage devices such as tapes and disks, or from tapping communications channels.

The following subsections describe two strategies for making block encryption more resistant to attack.

3.4.1 Block Chaining and Cipher Block Chaining

Feistel [Feis73] showed that block ciphers could be made more resistant to cryptanalysis and ciphertext substitution (including replay) using a technique called **block chaining**. Before enciphering each plaintext block M_i, some of the bits of the previous ciphertext block C_i are inserted into unused bit positions of M_i, thereby chaining the blocks together. Kent [Kent76] proposed a similar approach using sequence numbers. Both strategies protect against insertions, deletions, and modi-

FIGURE 3.11 Cipher block chaining (CBC).



fications in the message stream in much the same way as a stream cipher in cipher feedback mode. But, unlike cipher feedback mode, they reduce the number of available message bits per block.

This is remedied by an approach called **cipher block chaining** (CBC), which has been suggested for use with the DES [GSA77]. CBC is similar to cipher feedback (CFB), except that an entire block of ciphertext is fed back through a register to be exclusive-ored with the next plaintext block. The result is then passed through a block cipher E_K with key K (see Figure 3.11). The ith plaintext block M_i is thus enciphered as

$$C_i = E_K(M_i \oplus C_{i-1}),$$

where $C_0 = I_0$. Deciphering is done by computing

$$\begin{aligned} D_K(C_i) \oplus C_{i-1} &= D_K\big(E_K(M_i \oplus C_{i-1})\big) \oplus C_{i-1} \\ &= (M_i \oplus C_{i-1}) \oplus C_{i-1} \\ &= M_i . \end{aligned}$$

Because each ciphertext block C_i is computed from M_i and the preceding ciphertext block C_{i-1}, one transmission error affects at most two blocks. At the same time, each C_i is functionally dependent on all preceding ciphertext blocks, so the statistical properties of the plaintext are diffused across the entire ciphertext, making cryptanalysis more difficult. Like cipher feedback mode, the last block serves as a checksum, so the method can also be used to compute checksums for messages encrypted under another scheme or stored as plaintext.

CBC is similar to CFB in its resistance to all forms of attack (including ciphertext searching, replay, insertion, and deletion). It is more efficient than CFB in that it uses a single execution of the block encryption algorithm for each message block. It also protects a block cipher against the time-memory tradeoff attack described in Section 2.6.3. To see why, let C_i be the ciphertext corresponding to the chosen plaintext M_0. Since $D_K(C_i) = M_0 \oplus C_{i-1}$, to determine K a cryptanalyst would have to generate the tables of starting and ending points using $M_0 \oplus C_{i-1}$ rather than M_0. But this would rule out the possibility of precomputing the tables or of using the same tables to break more than one cipher.

CBC is used in the Information Protection System (IPS), a set of cryptographic application programs designed by IBM [Konh80]. The algorithms E and D are implemented with DES. The IPS facilities allow users to encrypt and decrypt entire files, and to call the encryption functions from their programs. Chaining may be applied either to a single record (called block chaining in IPS) or to a collection of records (called record chaining). Under block chaining, the feedback register is reset to I_0 at the beginning of each record, while under record chaining, it retains its value across record boundaries. Record chaining has the advantage of concealing identical lines in a file. Block chaining may be preferable for direct access files and databases, however, because records can be accessed directly without deciphering all preceding records, and records can be inserted and deleted without reenciphering the remaining records. Identical records can be concealed by prefixing each record with a random block of characters, as described earlier for cipher feedback.

The chaining procedure is slightly modified to deal with short blocks. To

encipher a trailing short block M_t of j bits, the preceding ciphertext block C_{t-1} is reenciphered, and the first j bits exclusive-ored with M_t; thus $C_t = M_t \oplus E_K(C_{t-1})$. Because C_{t-1} depends on all preceding blocks of the record, the trailing short ciphertext block is as secure as the preceding full ones.

With block chaining, there is a problem securing short records, because there is no feedback from the previous record. To encipher a single short record M_1 of j bits, it is exclusive-ored with the first j bits of I_0; thus, $C_1 = M_1 \oplus I_0$. Although this is not strong, it does superficially conceal the plaintext.

Cipher block chaining is somewhat less efficient for databases than direct block encryption because changing a field requires reenciphering all succeeding fields. Nevertheless, the added protection may offset the performance penalties for many applications. Moreover, if an entire record is fetched during retrieval anyway, encryption and decryption need not degrade performance if implemented in hardware.

3.4.2 Block Ciphers with Subkeys

Davida, Wells, and Kam [Davi81] introduced a new type of block cipher suitable for databases. A database is modeled as a set of records, each with t fields. Each record is enciphered as a unit, so that all fields are diffused over the ciphertext. The individual fields can be separately deciphered, though doing so requires access to an entire ciphertext record. Like cipher block chaining, the scheme protects against all types of attack.

Access to a particular field requires a special **subkey** for the field. There are separate **read subkeys** $d_1, \ldots, d_t$ for deciphering each field, and separate **write subkeys** $e_1, \ldots, e_t$ for enciphering each field. The subkeys are global to the database; that is, all records are enciphered with the same subkeys. Each user is given subkeys only for the fields that user is allowed to read or write.

We shall first describe a simplified, but insecure, version of the scheme, and then discuss the modifications needed for security. The scheme is based on the Chinese Remainder Theorem (see Theorem 1.8 in Section 1.6.2). To construct subkeys, each read subkey d_j is chosen to be a random prime number larger than the maximum possible value for field j. Letting $n = d_1 d_2 \cdots d_t$, the write subkeys are as follows:

$$e_j = \left(\frac{n}{d_j}\right) y_j \ , \tag{3.3}$$

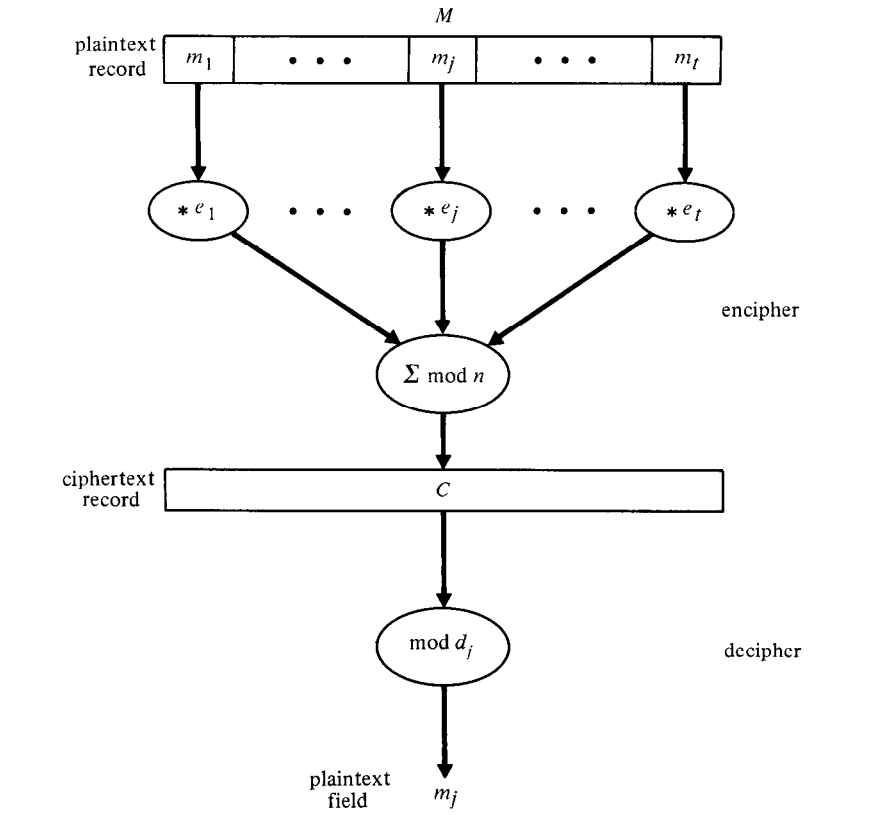
where y_j is the inverse of $(n/d_j) \bmod d_j$:

$$y_j = inv(n/d_j, d_j) \ .$$

Let M be a plaintext record with fields $m_1, \ldots, m_t$. The entire record is enciphered as

$$C = \sum_{j=1}^{t} e_j m_j \bmod n \ . \tag{3.4}$$

FIGURE 3.12 Enciphering and deciphering with subkeys.



Because C is the solution to the equations

$$C \bmod d_j = m_j \ , \tag{3.5}$$

for $j = 1, \ldots, t$, the jth field is easily deciphered using only the read subkey d_j. Figure 3.12 illustrates the enciphering and deciphering of a record.

The jth field can be updated using only the read and write subkeys d_j and e_j. Letting m_j' denote the new value, the updated ciphertext C' is given by

$$C' = [C - e_j (C \bmod d_j) + e_j m_j'] \bmod n \ . \tag{3.6}$$

Example:
Let $t = 3$, $d_1 = 7$, $d_2 = 11$, and $d_3 = 5$. Then $n = 7 * 11 * 5 = 385$ and

$$\begin{aligned} y_1 &= inv(385/7, 7) &&= inv(55, 7) &&= 6 \\ y_2 &= inv(385/11, 11) &&= inv(35, 11) &&= 6 \\ y_3 &= inv(385/5, 5) &&= inv(77, 5) &&= 3 \ . \end{aligned}$$

The write subkeys are thus:

$$e_1 = 55 * 6 = 330$$
$$e_2 = 35 * 6 = 210$$
$$e_3 = 77 * 3 = 231 \ .$$

Let M be the plaintext record $M = (4, 10, 2)$. Using Eq. (3.4), M is enciphered with the write subkeys as

$$\begin{aligned} C &= (e_1m_1 + e_2m_2 + e_3m_3) \bmod n \\ &= (330 * 4 + 210 * 10 + 231 * 2) \bmod 385 \\ &= (1320 + 2100 + 462) \bmod 385 = 3882 \bmod 385 \\ &= 32 \ . \end{aligned}$$

Using Eq. (3.5), the fields of M are deciphered with the read subkeys as follows:

$$\begin{aligned} m_1 &= C \bmod d_1 = 32 \bmod 7 \ = \ 4 \\ m_2 &= C \bmod d_2 = 32 \bmod 11 = 10 \\ m_3 &= C \bmod d_3 = 32 \bmod 5 \ = \ 2 \ . \end{aligned}$$

Using Eq. (3.6), the contents of the second field can be changed from 10 to 8 as follows:

$$\begin{aligned} C' &= [C - e_2(C \bmod d_2) + e_2 * 8] \bmod n \\ &= [32 - 210 * 10 + 210 * 8] \bmod 385 \\ &= -388 \bmod 385 = -3 \bmod 385 \\ &= 382 \ . \end{aligned}$$

The reader should verify that all three fields are still retrievable. ■

There are two weaknesses with the scheme as described thus far. The first lies in the method of enciphering as defined by Eq. (3.4). Let m_{ij} denote the value of the jth field in record i. A user knowing the plaintext values m_{1j} and m_{2j} for two records M_1 and M_2 can determine the read key d_j from the ciphertext C_1 and C_2. Observe that Eq. (3.5) implies

$$C_1 - m_{1j} = u_1d_j$$
$$C_2 - m_{2j} = u_2d_j$$

for some u_1 and u_2. Thus, d_j can be determined with high probability by computing the greatest common divisor of $(C_1 - m_{1j})$ and $(C_2 - m_{2j})$. The solution is to append a random 32-bit (or longer) value x_j to each m_j before enciphering with Eq. (3.4).

The second weakness is that the method of updating fields individually, as defined by Eq. (3.6), can expose the read keys (see exercises at end of chapter). The solution here is to reencipher the entire record, using new random values x_j for all fields.

Because both the read and write subkeys are required to write a field, user A automatically has read access to any field that A can write. This is consistent with most access control policies. But to write a single field, A must also have access to n; this gives A the ability to compute the write subkey for any field for which A has

the read subkey. This is not consistent with most access control policies. To prevent this, n must be hidden in the programs that access the database. To read field j, A would invoke a read procedure, passing as parameter the read subkey d_j; to write the field, A would invoke a write procedure, passing as parameters d_j, e_j, and the new value m_j'.

The write subkey e_j for field j is computed from n and the read subkey d_j. Therefore, any group of users with access to all the read subkeys can compute n and determine all the write subkeys. To prevent collusion, dummy fields are added to each record. The read and write subkeys for these fields are not given to any user.

3.5 ENDPOINTS OF ENCRYPTION

Data protected with encryption may be transmitted over several links before it arrives at its final destination. For example, data may be input from a user's terminal to the user's program, where it is processed and then transmitted to a disk file for storage. Later, the user may retrieve the data and have it displayed on the terminal. In computer networks, data may be transmitted from one location on the network to another for processing or for storage. In the discussion that follows, we use the terminology **node** for any location (computer, terminal, front-end, or program) where data may be input, stored, encrypted, processed, routed (switched), or output; and **link** for any communication line or data bus between two nodes.

3.5.1 End-to-End versus Link Encryption

There are many possible choices of endpoints for the encryption. At one extreme, **link encryption** enciphers and deciphers a message M at each node between the source node 0 and the destination node n [Bara64]. The message is processed as plaintext at the ith node, and transmitted as ciphertext $E_i(M)$ over the ith link (see Figure 3.13). Each link i has its own pair of transformations E_i and D_i, and different encryption algorithms and message formats may be used on the different links. This strategy is generally used with physical (hard-wired) connections. At the other extreme, **end-to-end encryption** enciphers and deciphers a message at the source and destination only (see Figure 3.14).

There are advantages and disadvantages to both extremes. With link encryp-

FIGURE 3.13 Link encryption.

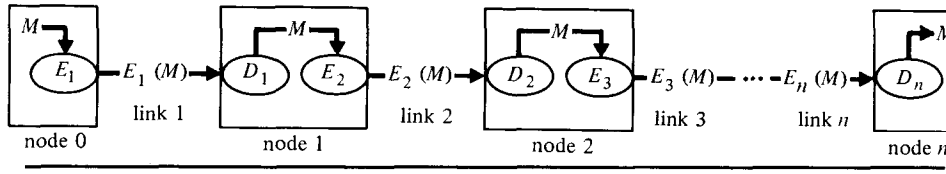
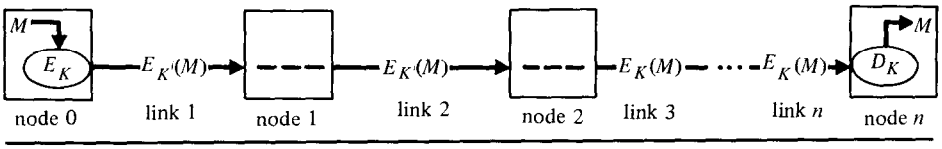


FIGURE 3.14 End-to-end encryption.



tion users need only one key for communicating with their local systems. With end-to-end encryption, users need separate keys for communicating with each correspondent. In addition, protocols are needed whereby users (or nodes) can exchange keys to establish a secure connection (see Section 3.7).

End-to-end encryption provides a higher level of data security because the data is not deciphered until it reaches its final destination. With link encryption, the data may be exposed to secrecy and authenticity threats when it is in plaintext at the intermediate nodes. Thus, end-to-end encryption is preferable for electronic mail and applications such as electronic-funds transfer requiring a high level of security.

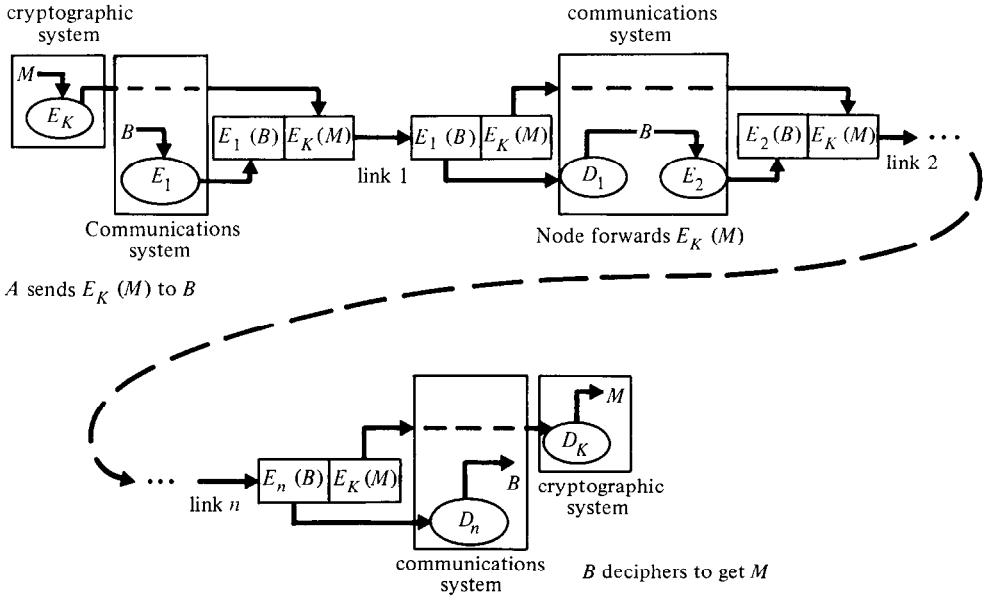
End-to-end encryption, however, is more susceptible to attacks of traffic flow analysis. With link encryption, the final destination addresses of messages can be transmitted as ciphertext along each link. This is not possible with end-to-end encryption, because the intermediate nodes need the addresses for routing (unless there is a single physical connection). The addresses could be used, for example, to learn whether an important business transaction was taking place.

For applications requiring high security, the two approaches can be combined. Figure 3.15 shows how an end-to-end cryptographic system can be interfaced with a communications system using link encryption to encipher addresses. The cryptographic system is on the outside, enciphering messages before they are processed by the communications system, and deciphering messages after they have been processed by the communications system. Each "packet" sent over a link has a header field and a data field. The header field contains the final destination of the message, enciphered using the key for the link, plus other control information used by the communications system. If the channel is used for more than one connection (as in a ring network), the header must also contain the immediate source and destination addresses in plaintext (the source address is needed so the receiver will know which key to use to decipher the final destination address).

Because a packet arriving at a device contains both plaintext and ciphertext, the device must be capable of operating in two modes: normal (plaintext) mode in which arriving control characters are interpreted by the device, and a "transparent" (ciphertext) mode in which arriving characters are not interpreted [Feis75]. With this feature, messages can also be exchanged in either plaintext or ciphertext.

Enciphering the final destinations of messages along the way does not obscure activity on a particular channel. The only solution here is to pad the channel with dummy traffic to make it appear constantly busy.

FIGURE 3.15 End-to-end data encryption with link adress-encryption.



Chaum [Chau81] uses a combination of endpoints to design an electronic mail system based on public-key encryption. His system allows messages to be sent anonymously through a node S (called a "mix"), which collects and shuffles messages to obscure their flow through the network.

Suppose user A (at location A) wishes to send an anonymous message M to user B (at location B). First A enciphers M using B's public transformation E_B (for end-to-end encryption). A then enciphers the destination B plus the enciphered message using S's public transformation E_S, and transmits the result to S:

$$C = E_S(B, E_B(M)) .$$

S deciphers C using its private transformation D_S, and forwards the enciphered message $E_B(M)$ to B, which B deciphers using B's private transformation D_B. The sender A is not revealed to B, and the path from A to B is concealed through encryption of B's address on the path from A to S and shuffling at S.

If A wants to receive a reply from B, A sends along an **untraceable return address** together with a key K that B uses to encipher the reply; the message transmitted through S is thus:

$$C = E_S(B, E_B(M, U, K)) ,$$

where

$$U = E_S(A)$$

is A's untraceable return address. Because A's address is enciphered with S's public key, B cannot decipher it. B can, however, send a reply M' to A through S:

$$C' = E_S(U, E_K(M'))\ .$$

S deciphers U and forwards the reply $E_K(M')$ to A.

There could be a problem with the scheme as we have described it if there is not enough uncertainty about the message M or the return address (see Section 1.4.1). Because A cannot sign M (doing so would divulge A's identity), someone might be able to determine M by guessing an X and checking whether $E_B(X) = E_B(M)$ using B's public key [this comparison cannot be made if A signs the message, transmitting $D_A(E_B(M))$] . Similarly, B might be able to guess the sender A from the untraceable return address, where the number of candidate addresses may be relatively small. Chaum solves the problem by appending random bit strings to all messages before enciphering.

If the output of one mix is used as the input for a second mix, then both mixes would have to conspire or be compromised for any message to be traced. With a series of mixes, any single mix can ensure the security of the messages. Thus, in the limiting case, each sender or receiver is a mix and need only trust itself.

3.5.2 Privacy Homomorphisms

To process data at a node in the system, it is usually necessary to decipher the data first, and then reencipher it after it has been processed. Consequently, it may be exposed to secrecy or authenticity threats while it is being processed. There are two possible safeguards. The first is to encapsulate the computation in a physically secure area. The second is to process the data in its encrypted state.

Rivest, Adleman, and Dertouzos [Rive78a] describe how the latter might be accomplished with a **privacy homomorphism**. The basic idea is that encrypted data, after processing, should be the same as if the data were first deciphered, processed in plaintext, and finally reenciphered. Suppose the plaintext data is drawn from an algebraic system consisting of

1. Data elements, denoted by a, b, etc.
2. Operations, denoted by f.
3. Predicates, denoted by p.
4. Distinguished constants, denoted by s.

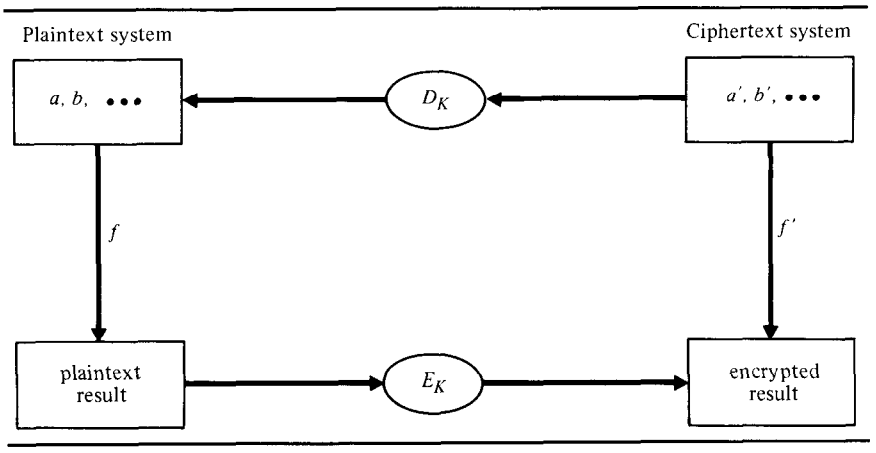
Similarly, suppose the ciphertext is drawn from an algebraic system with corresponding data elements a', b', operations f', predicates p', and distinguished constants s'. Let E_K be the enciphering transformation, and let D_K be the corresponding deciphering transformation. Then D_K is a **homomorphism** from the ciphertext system to the plaintext system if and only if for all cipher elements a', b' the following hold:

1. For all f and corresponding f':

$$f'(a', b', \ldots) = E_K(f(D_K(a'), D_K(b'), \ldots))\ .$$

2. For all p and corresponding p':

FIGURE 3.16 Privacy homomorphism.



$p'(a', b', \ldots)$ if and only if $p(D_K(a'), D_K(b'), \ldots)$.

3. For all s and corresponding s':

$$D_K(s') = s .$$

Figure 3.16 illustrates the first requirement.

Example:
Consider an exponentiation cipher (see Section 2.7) with enciphering transformation

$$E_K(a) = a^e \bmod n ,$$

and corresponding deciphering transformation

$$D_K(a') = (a')^d \bmod n ,$$

where $ed \bmod \phi(n) = 1$. Then D_K is a homomorphism from a ciphertext system consisting of the integers modulo n, with multiplication and test for equality, to an identical plaintext system. Given elements

$a' = a^e \bmod n$, and
$b' = b^e \bmod n$,

we have

$$\begin{aligned} a' * b' &= (a^e \bmod n) * (b^e \bmod n) \bmod n \\ &= (a * b)^e \bmod n \end{aligned}$$

$a' = b'$ if and only if $a = b$. ■

Privacy homomorphisms have inherent limitations. The most significant of these is described in Theorem 3.1.

Theorem 3.1:
It is not possible to have a secure enciphering function for an algebraic system that includes the ordering predicate "$\leq$" when the encrypted version of the distinguished constants can be determined.

Proof:
Consider the plaintext system over the natural numbers with $+$, $\leq$, and the constants 0 and 1. Let $+'$, $\leq'$, $0'$, and $1'$ denote the corresponding operators and elements in the ciphertext system. Given ciphertext element i', it is possible to determine the corresponding plaintext element i without computing $D_K(i')$ using a simple binary search strategy. First, determine $1'$ (by assumption this is possible). Next, compute

$$2' = 1' +' 1'$$
$$4' = 2' +' 2'$$
$$8' = 4' +' 4'$$
$$\vdots$$
$$(2^j)' = (2^{j-1})' +' (2^{j-1})'$$

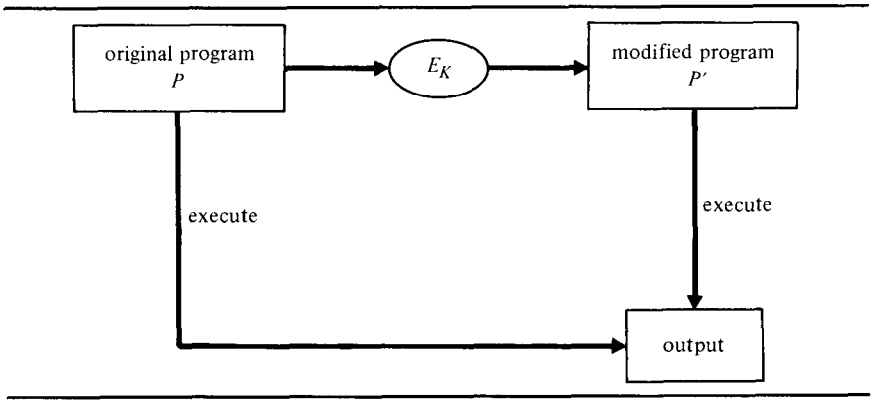
until $i' \leq' (2^j)'$ for some j. At this point, it is known that i falls in the interval $[2^{j-1} + 1, 2^j]$. To determine i, apply a similar binary search strategy to search the interval between 2^{j-1} and 2^j (the details of this are left as an exercise for the reader). ■

Privacy homomorphisms are an intuitively attractive method for protecting data. But they are ruled out for many applications because comparisons cannot in general be permitted on the ciphertext. In addition, it is not known whether it is possible to have a secure privacy homomorphism with a large number of operations.

Approximations of the basic principle, however, are used to protect confidential data in statistical databases. A one-way (irreversible) **privacy transformation** $E_K(M)$ transforms ("enciphers") a confidential data value M into a value that can be disclosed without violating the privacy of the individual associated with M [Turn73]. Examples of privacy transformations are "data perturbation", which distorts the value of M (e.g., by rounding), "aggregation", which replaces M with a group average, and "data swapping", which swaps values in one record with those in another (see Chapter 6). A privacy transformation is an approximation of a privacy homomorphism in that statistics computed from the transformed data are estimates of those computed from the original ("deciphered") data. But because it is impossible to restore the data to its original state and the transformation introduces errors, it is not a true privacy homomorphism. (Reed [Reed73] shows how the amount of distortion introduced by a privacy transformation can be measured using information theory, in particular, rate distortion theory.)

The basic principle is also used to protect proprietary software. Many software vendors distribute the source code for their programs so their customers can

FIGURE 3.17 Proprietary software protection.



tailor the code to their needs and make corrections. The problem is that a customer may illegally sell or give away copies of the program. If the copies have been modified (e.g., by changing the names of variables and rearranging code), it can be difficult to prove an agreement was breached. Indeed, the vendor might not even be aware of the illegal distribution.

One solution is to transform ("encipher") a source program in such a way that the transformed program executes the same as the original but is more difficult to copy [DeMi78]. Let P' be the transformed version of a program P. The transformation should satisfy the following properties:

1. P' should have the same output as P for all valid inputs.
2. P' should have approximately the same performance characteristics as P.
3. P' should have distinguishing features that are difficult to conceal in copies.

Property (1) is similar to the first property of a privacy homomorphism, where the elements a and b are programs and the function f corresponds to program execution (see Figure 3.17); instead of processing data in an encrypted state, we now execute programs in an encrypted state.

One method is to pad a program with code that does not affect the output (at least on valid inputs). Map makers employ a similar technique, introducing minor errors that do not seriously impair navigation, but make copies easily discernible. A second method is to transform the code or constants of the program to obscure the algorithm. A third method is to transform the program so that it will not run on other systems. This could be done by hiding in the code information about the customer; this information would be checked when the program runs. None of these transformations need be cryptographically strong; the objective is only to make it more costly to change a program than to develop it from scratch or obtain it from the vendor.

It is easier to protect proprietary software when the source is not distributed. In a sense, program compilation is a form of encipherment, because the algorithm is obscured in the object code. Object code can be decompiled, however, so the

encipherment is cryptographically weak. Still, it is easier to hide customer dependent information in object code than in the source, and many customers will lack the skills needed to decompile and modify the code.

Kent [Kent80] proposes special hardware and cryptographic techniques to protect proprietary software in small computer systems. Externally supplied software would run in tamper-resistant modules that prevent disclosure or modification of the information contained therein. Outside the module, information would be stored and transmitted in encrypted form.

3.6 ONE-WAY CIPHERS

A **one-way cipher** is an irreversible function f from plaintext to ciphertext. It is computationally infeasible to systematically determine a plaintext message M from the ciphertext $C = f(M)$.

One-way ciphers are used in applications that do not require deciphering the data. One such class of applications involves determining whether there is a correspondence between a given message M and a ciphertext C stored in the system. This correspondence is determined by computing $f(M)$, and comparing the result with C. For this to be effective, f should be one-to-one, or at least not too degenerate. Otherwise, a false message M' may pass the test $f(M') = C$.

A one-way cipher can be implemented using a computationally secure block encryption algorithm E by letting

$$f(M) = E_M(M_0)\,,$$

where M_0 is any given, fixed message. The message M serves as the key to E. As long as E is secure, it is computationally infeasible to determine the enciphering key M with a known plaintext attack by examining pairs $(M_0, E_M(M_0))$.

Purdy [Purd74] suggests implementing a one-way cipher using a sparse polynomial of the form:

$$f(x) = (x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0) \bmod p\,,$$

where p is a large prime and n is also large. Because a polynomial of degree n has at most n roots, there can be at most n messages enciphering to the same ciphertext. The time to invert f (i.e., find its roots) is $O(n^2(\log p)^2)$; for $n \simeq 2^{24}$ and $p \simeq 2^{64}$, this will exceed 10^{16} operations. (See [Evan74] for other methods of implementing one-way ciphers.)

Note that a one-way cipher cannot be implemented using a stream cipher with keystream M and plaintext stream M_0. We would have $C = M_0 \oplus M$, whence M is easily computed by $M = C \oplus M_0$.

3.6.1 Passwords and User Authentication

Password files are protected with one-way ciphers using a scheme developed by Needham [Wilk75]. Rather than storing users' passwords in the clear, they are

transformed by a one-way cipher f, and stored as ciphertext in a file which cannot be deciphered even by the systems staff. Each entry in the password file is a pair $(ID, f(P))$, where ID is a user identifier and P is the user's password. To log into the system, a user must supply ID and P. The system computes the enciphered password $f(P)$, and checks this against the password file; the login is permitted only if there is a match.

Because the stored passwords cannot be deciphered, they are completely safe even if the entire password file is (accidentally or maliciously) disclosed. This also implies that a forgotten password P cannot be recovered. A new password P' must be created, and $f(P')$ entered into the password file.

A strong one-way cipher can protect passwords only if users select passwords at random. In practice, users select short, easily remembered passwords. Such passwords are often simple to find by exhaustive search. Sequences of letters are systematically generated, enciphered, and then looked up in the table. In a study of password security on Bell Labs' UNIX, Morris and Thompson [Morr79] discovered that about 86% of all passwords were relatively easy to compromise. The system-supplied passwords were no better; because they were generated by a pseudo-random number generator with only 2^{15} possible outputs, all possible passwords could be tried in about 1 minute on a DEC PDP-11/70.

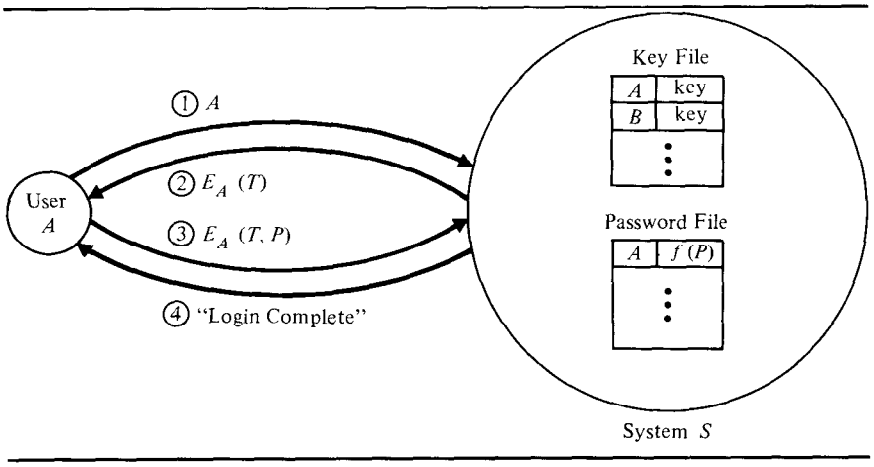
Two improvements were made to the UNIX password security. First, the password entry program was modified to encourage users to use longer passwords. Second, each password is concatenated with a 12-bit random number (called the **salt**) before encryption, effectively lengthening the password by 12 bits. When a password P is created, a salt X is generated and concatenated to P. Letting PX denote the concatenation, both X and $f(PX)$ are stored in the password file along with the user's ID. When a user logs in, the system gets X from the file, forms the concatenation PX using the password P supplied by the user, and checks $f(PX)$ against the password file.

This does not increase the work factor for finding a particular user's password, because the salt is not protected. But it substantially increases the work factor for generating random passwords and comparing them with the entire password file, since each possible password could be enciphered with each possible salt. If passwords are n bits long, there are 2^{n+12} possibilities for the entries $f(PX)$ in the password table; thus, the effort required to find the password associated with one of these entries is 2^{12} times greater than for a file containing only enciphered passwords.

Suppose a user logs into the system and supplies password P as described earlier. If P is transmitted from the user's terminal to the system in the clear, it could be compromised on the way (e.g., by wiretapping). It may not even make it to the system: a program masquerading as the login procedure might trick the user into typing ID and P.

Feistel, Notz, and Smith [Feis75] describe a login procedure that does not expose the user's password, and allows the user and system to mutually authenticate each other. They assume each user A has a private key on some digital storage medium (e.g., a magnetic-stripe card) which can be inserted into A's terminal, and that a copy of the key is stored on file at the system. To log into the

FIGURE 3.18 Login protocol with passwords.



system, A transmits ID in the clear (for simplicity, we assume $ID = A$). The system responds with a "challenge-reply" test that allows A to determine whether the communication is "live" (and not a replay of an earlier login), and allows the system to establish A's authenticity. Letting S denote the system, the protocol is as follows (see also Figure 3.18):

Login protocol using passwords

1. A transmits $ID = A$ to S.
2. S sends to A:

   $$X = E_A(T),$$

   where T is the current date and time, and E_A is the enciphering transformation derived from A's private key.
3. A deciphers X to get T, and checks that T is current. If it is, A replies to S by sending

   $$Y = E_A(T, P)$$

   where P is A's password.
4. S deciphers Y to get T and P. It checks T against the time transmitted to A in Step 2, and checks $f(P)$ against the password file. If both check, the login completes successfully.

The protocol is easily modified for a public-key system. In Step 2, the system S uses its private transformation D_S (for sender authenticity) to create $X = D_S(T)$, which A can validate using S's public transformation E_S. In Step 3, A uses the system's public enciphering transformation (for secrecy) to create $Y = E_S(T, P)$. Only S can decipher Y to obtain A's password and complete the login. Note that A's private transformation (i.e., digital signature) can be used for authentication instead of A's password; in this case, the protocol becomes:

Login protocol using digital signatures

1. A transmits $ID = A$ to S.
2. S sends to A:

$$X = D_S(T),$$

where T is the current date and time, and D_S is S's private transformation.

3. A computes $E_S(X) = T$ using S's public transformation, and checks that T is current. If it is, A replies to S by sending

$$Y = D_A(T),$$

where D_A is A's private transformation.

4. The system validates Y using A's public transformation E_A. If it is valid, the login completes successfully.

One possible weakness with the digital signature protocol is that users can be impersonated if their private keys are stolen. The password protocol has the advantage that memorized passwords are less susceptible to theft than physical keys—provided users do not write them down. The digital signature protocol can be enhanced by combining it with passwords or with a mechanism that uses personal characteristics (e.g., a handprint) for identification. If passwords are used, then A sends to S

$$D_A(T), E_S(T, P)$$

in Step 3 of the protocol.

3.7 KEY MANAGEMENT

A troublesome aspect of designing secure cryptosystems is key management. Unless the keys are given the same level of protection as the data itself, they will be the weak link. Even if the encryption algorithm is computationally infeasible to break, the entire system can be vulnerable if the keys are not adequately protected. In this section we consider various techniques for safeguarding and distributing keys.

3.7.1 Secret Keys

We first consider the management of keys that are used by a single user (or process) to protect data stored in files. The simplest approach is to avoid storing cryptographic keys in the system. In IPS [Konh80], for example, keys do not reside permanently in the system. Users are responsible for managing their own keys and entering them at the time of encipherment or decipherment.

IPS offers users two formats for entering the 56-bit keys needed for DES. The first format is the direct entry of an 8-byte key (56 key bits plus 8 parity bits).

This format should be used only if the 56 key bits are randomly selected; a key formed from English letters only (or even letters and digits) is too easy to find by exhaustive search. Because it is easier for users to remember meaningful strings, IPS provides a second format whereby a key can be entered as a long character string. The string is reduced to a 56-bit key by enciphering it with DES using cipher block chaining, and keeping the rightmost 56 bits (i.e., the checksum). The process is called "key crunching".

Keys could also be recorded in Read Only Memory (ROM) or on magnetic stripe cards [Feis75,Flyn78,Denn79]. The hardware-implemented key could then be entered simply by inserting it into a special reader attached to the user's terminal.

In conventional systems, users may register private keys with the system to establish a secure channel between their terminals and the central computer (master terminal keys are used in some systems for this purpose). These keys must be protected, and the simplest strategy is to store them in a file enciphered under a system master key. Unlike passwords, encryption keys cannot be protected with one-way functions, because it would then be impossible to recover them.

In public-key systems, a user A need not register a private transformation D_A with the system to establish a secure channel. This does not mean D_A requires no security. If it is used for signatures, it must be protected from disclosure to prevent forgery. Indeed, it must be protected from deliberate disclosure by A. If A can give away D_A—or even just claim to have lost D_A—then A has a case for disavowing any message (see [Salt78,Lipt78]). To prevent this, Merkle [Merk80] suggests that A's signature key should not be known to anyone, including A. A single copy of D_A would be kept in a dedicated microcomputer or sealed in a ROM.

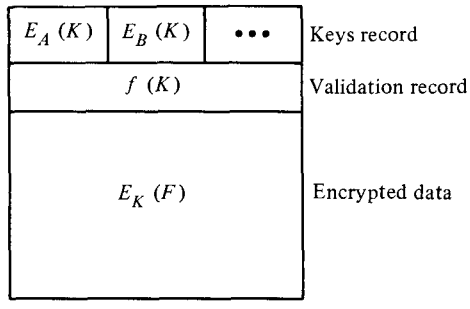
Even if D_A is given a high level of protection, some mechanism is needed to handle the case where D_A is compromised. This case can be handled as for lost or stolen credit cards: liability would be limited once the loss or theft is reported to the system. A signature manager S would keep a record of A's past and present public transformations E_A, and the times during which these transformations were valid. To determine whether a message was signed before the loss, messages could be timestamped. A cannot affix the timestamp, however, because A could knowingly affix an incorrect time, and if D_A is compromised, someone else could forge a message and affix a time when D_A was valid. The solution is for S to affix the current time T to a message signed by A, and then add its own signature D_S, thereby playing the role of a notary public [Pope79,Merk80]. A message M would thus be doubly signed as follows:

$$C = D_S(D_A(M), T) \ .$$

Another user receiving C can check (with S) that A's corresponding public transformation E_A was valid at time T before accepting C.

A similar strategy can be used to protect a signature transformation D_A in a conventional system. But because the corresponding enciphering transformations E_A and E_S are secret, the receiver of a signed message $C = D_S(D_A(M), T)$ cannot validate the signature (see Section 1.3.3).

FIGURE 3.19 File encryption with keys record.



The keys that unlock a database require special protection. If the database is shared by many users, it is usually better to store the keys in the system under the protection of a key manager than to distribute the keys directly to the users, where they would be vulnerable to loss or compromise. More importantly, it relieves the users of the burden of key management, providing "cryptographic transparency". Database keys could be enciphered under a database master key and stored either in the database or in a separate file.

Gudes [Gude80] has proposed a scheme for protecting file encrypting keys that can be integrated with the access control policies of a system. Let K be the encryption key for a file F. For every user A allowed access to F, the system computes

$$X = E_A(K) \ ,$$

using A's private transformation E_A, and stores X in a **keys record** at the beginning of the encrypted file (see Figure 3.19). When A requests access to F, the system finds A's entry X in the keys record, and computes $D_A(X) = D_A(E_A(K)) = K$; the file is then deciphered using the recovered key. An additional level of protection against unauthorized updates is provided by storing $f(K)$ in a **validation record** of the file, where f is a one-way function. A user is not allowed to update the file unless $f(D_A(X))$ matches the value in the authentication field. If the file is enciphered using a two-key system, with separate read and write keys as described in Section 1.2 (see Figure 1.8), users can be given read access without write access to the file. One drawback with the scheme is that if a user's access rights to a file are revoked, the file must be reenciphered under a new key, and the keys record and validation record recomputed.

Ehrsam, Matyas, Meyer, and Tuchman [Ehrs78,Maty78] describe a complete key management scheme for communication and file security. They assume that each host system has a **master key** $KM0$ with two variants, $KM1$ and $KM2$. The variants can be some simple function of $KM0$. The master keys are used to encipher other encryption keys and to generate new encryption keys. Each terminal also has a **master terminal key** KMT, which provides a secure channel between the terminal and the host system for key exchange. The system stores its copies of

these keys in a file enciphered under $KM1$. Other key encrypting keys, such as file master keys (called **secondary file keys**), are stored in files enciphered under $KM2$.

The master key $KM0$ is stored in the nonvolatile storage of a special cryptographic facility or security module. ($KM1$ and $KM2$ are computed from $KM0$ as needed.) The facility is secured so that users cannot access the master key or its derivatives. Data encrypting keys are stored and passed to the cryptographic facility as ciphertext to protect them from exposure.

Special operations are provided by the cryptographic facilities in the host systems and terminals. The following operations are used by the key management scheme in a host:

1. *Set master key* (*smk*). A master key $KM0$ is installed with the operation

$$smk(KM0) \ . \tag{3.7}$$

Clearly, this operation requires special protection.

2. *Encipher under master key* (*emk*). A key K is protected by encrypting it under $KM0$ with the operation

$$emk(K) = E_{KM0}(K) \ . \tag{3.8}$$

3. *Encipher* (*ecph*). To encipher a message M using key K, K is passed to the cryptographic facility enciphered under $KM0$. Letting $X = E_{KM0}(K)$ be the enciphered key, M is enciphered with the operation

$$ecph(X, M) = E_K(M) \ , \tag{3.9}$$

where K = $D_{KM0}(X)$ (see Figure 3.20). This instruction allows keys to be stored and passed to the cryptographic facility as ciphertext.

4. *Decipher* (*dcph*). Similarly, a ciphertext message C is deciphered using a key K with the operation

$$dcph(X, C) = D_K(C) \ , \tag{3.10}$$

where $K = D_{KM0}(X)$ (see Figure 3.21).

5. *Reencipher from master key* (*rfmk*). A terminal master key KMT is stored under $KM1$ encipherment as $W = E_{KM1}(KMT)$. The reencipher from master key operation allows the key manager to take a key K enciphered under $KM0$ as $X = E_{KM0}(K)$ and put it under KMT encipherment:

FIGURE 3.20 Encipher (*ecph*).

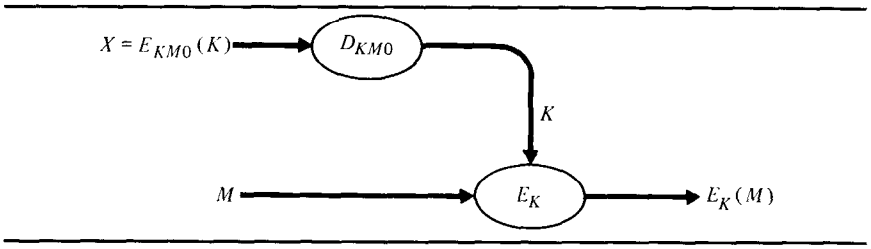
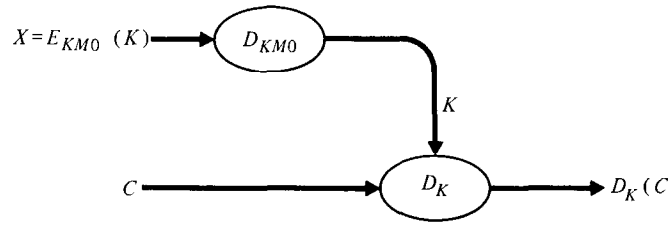


FIGURE 3.21 Decipher (*dcph*).



$$rfmk(W, X) = E_{KMT}(K) \ , \tag{3.11}$$

where $KMT = D_{KM1}(W)$, and $K = D_{KM0}(X)$ (see Figure 3.22). This operation is used by the key manager to transmit keys to terminals and other host systems for end-to-end encryption (see Section 3.7.4). As with the preceding operations, none of the keys entering or leaving the cryptographic facility is ever in the clear.

KMT is stored under $KM1$ encipherment to protect K from exposure. If KMT were stored under $KM0$ encipherment as $W' = E_{KM0}(KMT)$, then K could be obtained from $Y = E_{KMT}(K)$ using the *dcph* operation:

$$dcph(W', Y) = D_{KMT}(Y) = K \ .$$

Thus, an eavesdropper obtaining Y, W', and access to the cryptographic facility could decipher messages enciphered under K.

6. *Reencipher to master key* (*rtmk*). A file encrypting key K is likewise not stored under $KM0$ encipherment; if it were, then any user who obtained access to the encrypted key and the cryptographic facility could decipher the file using the *dcph* operation. Rather, K is stored under the encipherment of a secondary file key KNF as $X = E_{KNF}(K)$; KNF, in turn, is stored under $KM2$ encipherment as $W = E_{KM2}(KNF)$. To use K, it must first be placed under $KM0$ encipherment. This is done with the reencipher to master key operation:

$$rtmk(W, X) = E_{KM0}(K) \tag{3.12}$$

FIGURE 3.22 Reencipher from master key (*rfmk*).

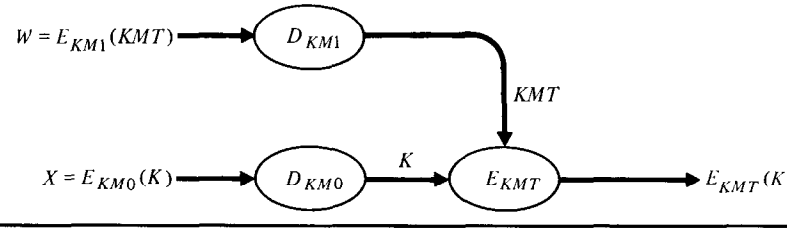
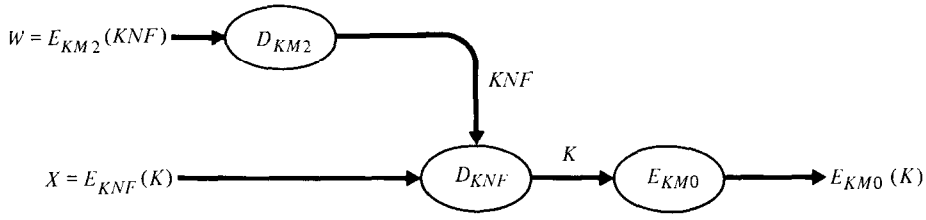


FIGURE 3.23 Reeincipher to master key (*rtmk*).



where $K = D_{KNF}(X)$ and $KNF = D_{KM2}(W)$ (see Figure 3.23).

Note that KNF is stored under $KM2$ encipherment rather than under $KM1$ encipherment. This is done to separate the communications and file systems. If the *rtmk* operation used $KM1$ instead of $KM2$, it would be the inverse of *rfmk*. This would allow the key management operations for the file system to be used to obtain keys used by the communications system. In fact, neither *rfmk* nor *rtmk* have inverse operations.

The master key KMT for a terminal is stored in the cryptographic facility of the terminal. The following operations are provided by the facility:

1. *Decipher from master key* (*dmk*). This operation takes a key K transmitted to the terminal under KMT encipherment as $E_{KMT}(K)$, deciphers it, and stores it in a special working register of the facility. K remains in the working register until it is changed or the terminal is turned off.
2. *Encipher* (*ecph*). This operation enciphers data using the key stored in the working register. The encrypted data is transmitted to its destination, providing end-to-end encryption.
3. *Decipher* (*dcph*). This operation deciphers data using the key stored in the working register.

3.7.2 Public Keys

Public keys can be distributed either outside the system or from a public-key directory within the system. In the former case, they could be recorded on some digital medium such as a magnetic stripe card, which can be inserted into a special reader attached to a user's terminal. Users would exchange public keys by giving out copies of their cards.

In the latter case, they could be stored in a file managed by a system directory manager S. It is unnecessary to safeguard the keys from exposure, because their secrecy is not required for secure communication or digital signatures. But it is essential to maintain their integrity so they can be used to transmit messages in secrecy and validate digital signatures. If S (or an imposter) supplies a valid but incorrect public key, a user could unknowingly encipher confidential data that

would be decipherable by foe rather than friend, or be tricked into accepting a message with the wrong signature.

Imposters can be thwarted by requiring a signature from S [Rive78b, Need78]. This does not, however, protect against a faulty or untrustworthy system. To deal with both problems, Kohnfelder [Konf78] proposes **certificates**. Upon registering a public transformation E_A with the system, a user A receives a signed certificate from S containing E_A. Using public-key encryption to implement the signature, A's certificate is thus:

$$C_A = D_S(A, E_A, T) \tag{3.13}$$

(strictly speaking, C_A would contain the key to E_A), T is a timestamp giving the current time, and D_S is S's private signature transformation. A can verify that the certificate came from S and contains the correct public key by computing

$$E_S(C_A) = (A, E_A, T) ,$$

using the public transformation E_S of S. Certificates can be distributed either through S or by their owners. The receiver of a certificate C_A can verify its authenticity the same way as the owner. In addition, the receiver can check the timestamp to determine if C_A is current.

There is a problem if the secret signature key used by the system directory manager is compromised. Merkle's [Merk80] **tree authentication** scheme solves this problem by eliminating the secret signature key. All entries in the public file are signed as a unit using a one-way hashing function. Users can authenticate their own keys in a file of n keys knowing $O(\log_2 n)$ intermediate values of the hashing function. These intermediate values form an **authentication path** of a tree, and serve as a form of certificate.

Let $K_1, \ldots, K_n$ denote the file of public enciphering keys. Let $f(x, y)$ be a function that returns a value in the domain of f. The hashing function H is defined recursively by:

$$H(i,j) = \begin{cases} f(H(i, m), H(m + 1, j)) & \text{if } i < j, \text{ where } m = \lfloor (i + j)/2 \rfloor \\ f(K_i, K_i) & \text{if } i = j \end{cases}$$

where $H(1, n)$ is the hash value of the entire public file.

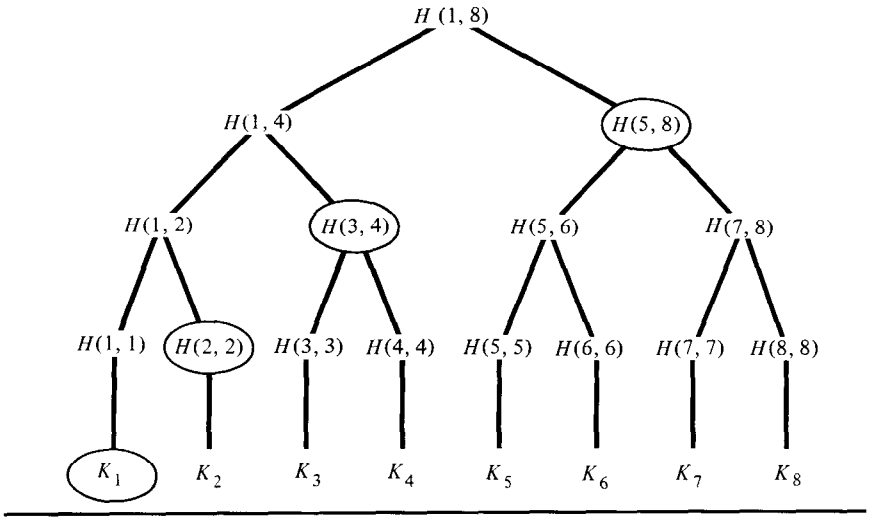
Figure 3.24 illustrates the computation for $n = 8$. Users can compute $H(1, 8)$ by following a path from their key to the root, provided they know the intermediate values of H needed to follow the path. For example, the user with key K_1 can compute $H(1, 8)$ from $H(2, 2)$, $H(3, 4)$, and $H(5, 8)$ by computing the following sequence:

$$\begin{aligned} H(1, 1) &= f(K_1, K_1) \\ H(1, 2) &= f(H(1, 1), H(2, 2)) \\ H(1, 4) &= f(H(1, 2), H(3, 4)) \\ H(1, 8) &= f(H(1, 4), H(5, 8)) . \end{aligned}$$

The values K_1, $H(2, 2)$, $H(3, 4)$, and $H(5, 8)$ form this user's authentication path.

Because the entire file is hashed as a unit, any modification of the file invalidates every certificate. Thus, it is easy for a user to determine whether someone

FIGURE 3.24 Tree authentication.



else is attempting to masquerade as the user. There is a drawback, though; the entire file must be rehashed each time an entry is added or changed.

3.7.3 Generating Block Encryption Keys

In Section 3.2 we discussed methods for generating pseudo-random key streams for stream encryption. The streams were generated from a seed I_0 using a block encryption algorithm. We shall now discuss methods for generating single keys. These values should satisfy the same properties of randomness as key streams, and should not be repeated across power failures.

Matyas and Meyer [Maty78] suggest methods for generating keys from the master keys $KM0$, $KM1$, and $KM2$ described in Section 3.7.1. They recommend the master key $KM0$ be generated outside the system by some random process such as tossing coins or throwing dice. The key could be recorded on some digital medium, entered from the medium, and installed with the set master key operation *smk*.

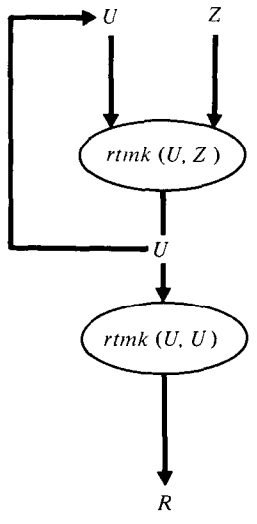
A set of key-encrypting keys K_i ($i = 1, 2, \ldots$) is derived from a random number R, which is generated outside the system in the same manner as the master key. Each K_i is generated from the operation *rfmk* [Eq.(3.11)]:

$$K_i = rfmk(R, rfmk(R, T + i)),$$

where T is the current time (the parity bits of K_i are adjusted as needed). Because *rfmk* is a function of $KM0$, K_i is a function of a random number R, the time, and the master key.

Data encrypting keys are generated inside the system using the operation

FIGURE 3.25 Key generation.



rtmk [Eq. (3.12)]. Consecutive random numbers R_i ($i = 1, 2, \ldots$) are generated from two independent values U_{i-1} and Z_i by

$$R_i = rtmk(U_i, U_i)\,, \tag{3.14}$$

where

$$U_i = rtmk(U_{i-1}, Z_i) \quad \text{for } i > 0\,,$$

and each Z_i is derived from two or more independent clock readings (see Figure 3.25). The initial value U_0 is derived from a combination of user and process dependent information. Each U_i is a function of $KM0$ and all preceding values of U and Z, and each R_i is a function of $KM0$ and U_i; thus, it should be computationally infeasible to determine one R_i from another R_j, and computationally infeasible to determine R_i from U_i or vice versa.

Each random number R_i is defined as the encipherment of a data encrypting key K_i under some other key X; that is,

$$R_i = E_X(K_i)\,.$$

Thus, K_i can be generated without ever being exposed in plaintext. For example, if K is to be used to encipher data stored in a system file, R_i would be defined as

$$R_i = E_{KNF}(K_i)\,,$$

where KNF is a secondary file key.

Keys that are used more than once (e.g., database keys or private keys) can be stored in files, enciphered under some other key. If an application uses a large number of such keys, it is more efficient (in terms of space) to regenerate them

when they are needed. This cannot be done using the preceding methods, because the keys are a function of the current state of the system. Bayer and Metzger [Baye76] describe an encipherment scheme for paged file and database structures (including indexed structures) where each page is enciphered under a different page key. The key K for a page P is derived from a file key KF by $K = E_{KF}(P)$. Keys for other objects could be generated by similar means. In capability-based systems (see Chapter 4), each object is identified by a unique name; this name could be used to derive the encryption key for the object. By encrypting each object under a separate key, exposure of a key endangers only one object.

Each record of a database could be enciphered under a different key using a scheme suggested by Flynn and Campasano [Flyn78]. A record key would be a function of a database key and selected plaintext data stored in the record (or supplied by the user), and would be generated at the time the record is accessed. Enciphering each record under a separate key would protect against ciphertext searching and replay as described in Section 3.4. A user authorized to access the encrypted fields of the records either would use a special terminal equipped with the database key, or would be given a copy of the key sealed in a ROM, which could be inserted into any terminal. The scheme could also be used to encipher each field of a record under a separate key, though encrypting the fields of a record separately can introduce security problems as described earlier.

Keys shared by groups of users in a computer network can be generated from information about the group members. Let G be a group of users in a network of N users. Members of G can share a secret **group key** K_G, which allows them to broadcast and receive messages from other members of G, and to access and update files private to G. Users not in G are not allowed access to K_G. There can be at most $2^N - N - 1$ groups of two or more users in the system. Denning and Schneider [Denn81a] and Denning, Meijer, and Schneider [Denn81b] describe schemes for deriving all possible group keys from a list of N user values; thus, the schemes generate an exponential number of keys from a linear number of values. One simple method computes the key K_G for a group G of m users from

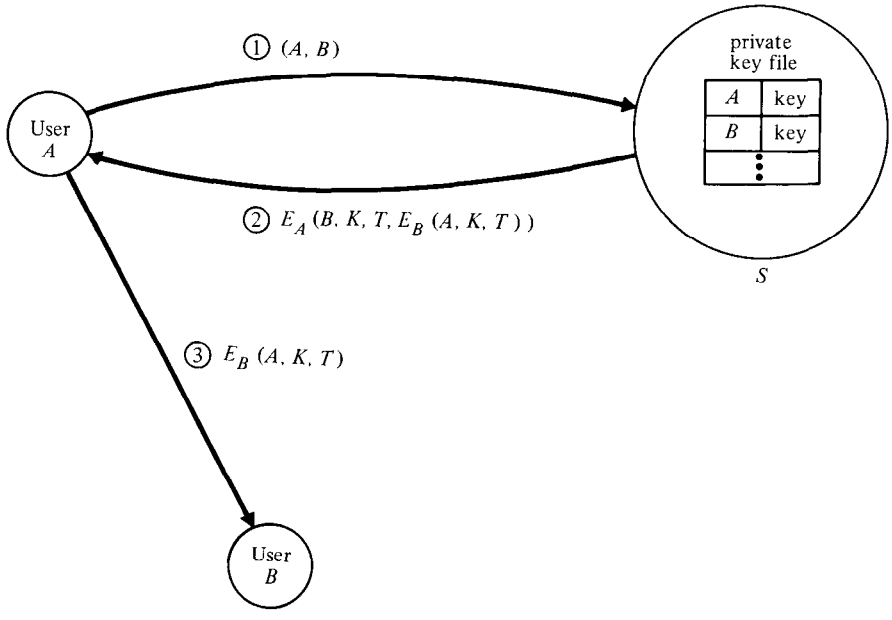
$$K_G = 2^{K_1 K_2 \cdots K_m} \bmod p ,$$

where $K_1, \ldots, K_m$ are the user's private keys. A user not in G cannot determine K_G or any of the K_i of the members in G without computing a discrete logarithm. In Section 2.7.1, we saw this was infeasible when p was 200 decimal digits long. If a key shorter than 200 digits is needed, K_G can be compressed by enciphering it with a stream cipher in cipher feedback mode or with cipher block chaining, and keeping the rightmost bits.

3.7.4 Distribution of Session Keys

If two users wish to communicate in a network using conventional encryption, they must share a secret **session key** (also called "communication key"). Either a system key manager must supply this key (in which case security of the key distribu-

FIGURE 3.26 Centralized key distribution protocol.



tion facility is required), or else the users must find an independent but secure method for exchanging the key by themselves. We shall show how each of these approaches may be implemented.

We shall first describe a protocol for obtaining session keys from a centralized key distribution facility, sometimes called an "authentication server", S. The protocol is based on one introduced by Needham and Schroeder [Need78] and modified by Denning and Sacco [Denn81c]. We assume each user has a private key, registered with S. If two users wish to communicate, one of them obtains a session key K from S and distributes it to the other. A new key is obtained for each session, so that neither the users nor S need keep lists of session keys. Although S could keep a list of session keys for all possible pairs of users, the storage requirements would be enormous [for n users there are $\binom{n}{2}$ possible pairs], and the keys might be vulnerable to attack. The key distribution facility could be distributed over the network.

User A acquires a key K from S to share with another user B by initiating the following steps (also refer to Figure 3.26):

Centralized key distribution protocol

1. A sends to S the plaintext message

   (A, B)

   stating both A's and B's identity.

2. S sends to A the ciphertext message

$$E_A(B, K, T, C) ,$$

where E_A is the enciphering transformation derived from A's private key, K is the session key, T is the current date and time, and

$$C = E_B(A, K, T) ,$$

where E_B is the enciphering transformation derived from B's private key.

3. A sends to B the message C from Step 2.

In Step 1, A could conceal B's identity by sending $(A, E_A(B))$ to S. A's identifier cannot be encrypted under E_A, however, because then S would not know with whom to correspond.

In Step 2, S returns the session key K enciphered under B's private key as well as A's so that A can securely send the key to B without knowing B's private key. The timestamp T guards against replays of previous keys.

For added protection, the protocol can be extended to include a "handshake" between B and A [Need78]:

4. B picks a random identifier I and sends to A

$$X = E_K(I) .$$

5. A deciphers X to obtain I, modifies it in some predetermined way to get I' (e.g., $I' = I + 1$), and returns to B

$$E_K(I') .$$

This confirms receipt of I and the use of K.

With the handshake, A can begin transmitting data to B at Step 5; without the handshake, A can begin at Step 3.

To protect K from exposure between the time it is generated and the time it is enciphered under A's and B's private keys, K should be generated as a random number R in ciphertext as described in the preceding section [see Eq. (3.14)]. Ehrsam, Matyas, Meyer, and Tuchman [Ehrs78] suggest that R be defined as the encipherment of K under the master key $KM0$; that is,

$$R = E_{KM0}(K) .$$

If users' private keys are enciphered under a variant master key $KM1$, as described in Section 3.7.1 for terminal master keys, K can be deciphered and reenciphered under A's and B's private keys using the reencipher from master key operation *rfmk* [Eq. (3.11)].

It is desirable to distribute the server S over the network so that all keys do not have to be registered at a single site, and so that no single file contains all private keys. This general approach is used in the protocols given by Ehrsam, Matyas, Meyer, and Tuchman [Ehrs78] for use in the system described in Section 3.7.1. Here the host systems for the users exchange an enciphered session key R.

Each host then uses the *rfmk* instruction to transmit R to its respective user's terminal, enciphered under the terminal master key KMT (users' private keys are used for data encryption only, not for key exchange). The hosts exchange R using the *rfmk* and *rtmk* operations as follows. Let H_A be the host for A and H_B the host for B, and let $KM0_A$ and $KM0_B$ be their respective master keys. H_A and H_B share a **secondary communication key** KNC. KNC is stored as $W_A = E_{KM1_A}(KNC)$ at H_A (for key forwarding) and as $W_B = E_{KM2_B}(KNC)$ at H_B (for key receiving). The encrypted key K is given by $R = E_{KM0_A}(K)$. H_A uses the *rfmk* instruction to forward R to H_B, and H_B uses the *rtmk* instruction to receive R. The complete protocol is as follows:

Key distribution by host systems

1. H_A generates $R = E_{KM0_A}(K)$, and sends to H_B:

$$rfmk(W_A, R) = E_{KNC}(R) \ .$$

2. H_B obtains R by computing

$$rtmk(W_B, E_{KNC}(R)) = R \ .$$

3. H_A sends to A:

$$Z_A = rfmk(Y_A, R) = E_{KMT_A}(K) \ ,$$

where KMT_A is A's terminal master key, and $Y_A = E_{KM1_A}(KMT_A)$. A obtains K by deciphering Z_A.

4. Similarly, H_B sends to B:

$$Z_B = rfmk(Y_B, R) = E_{KMT_B}(K) \ ,$$

where KMT_B is B's terminal master key, and $Y_B = E_{KM1_B}(KMT_B)$. B obtains K by deciphering Z_B.

The scheme is also used to distribute session keys to nodes or processes rather than terminals.

Diffie and Hellman [Diff76] and Merkle [Merk78] have proposed an approach that allows users to exchange session keys directly. Diffie's and Hellman's scheme is based on the computational difficulty of computing discrete logarithms (see Section 2.7.1). The first user, A, picks a random value x_A in the interval $[0, p - 1]$, where p is prime (p may be publicly available to all users). The second user, B, also picks a random value x_B in the interval $[0, p - 1]$. Then A sends to B the value

$$y_A = a^{x_A} \bmod p,$$

and B sends to A the value

$$y_B = a^{x_B} \bmod p,$$

for some constant a. For sufficiently large values of x_A and x_B (e.g., 664 bits), the fastest known algorithms for computing the discrete logarithm function are intractable, whence x_A and x_B cannot be practically computed from y_A and y_B (see Section 2.7.1). After the y's are exchanged, A computes the session key

$$K = (y_B)^{x_A} \bmod p$$
$$= (a^{x_B} \bmod p)^{x_A} \bmod p$$
$$= a^{x_A x_B} \bmod p \ ,$$

and similarly for B.

Although an eavesdropper cannot compute K from either y_A or y_B, there is no way A and B can be sure that they are communicating with each other with this mechanism alone. This problem is remedied in the public-key distribution schemes described next.

MITRE is developing a demonstration system that uses public-key distribution for key exchange and the DES for message encryption [Scha79,Scha80]. Each user A obtains a "private key" x_A and registers the "public key" $y_A = a^{x_A} \bmod p$ with a public-key distribution center (y_A and x_A are not encryption keys in the usual sense since they are not used to encipher and decipher in a public-key cryptosystem). The MITRE system provides three modes of operating the DES for message encryption: standard block mode, cipher feedback mode, and cipher block chaining. For the latter two modes, two users A and B must exchange initialization bit vectors (seeds) I_{AB} and I_{BA} in addition to a session key K; I_{AB} is used for transmission from A to B and I_{BA} for transmissions from B to A. A secure channel is established between A and B as follows:

Exchange of keys using public-key distribution

1. A obtains y_B from the public directory and computes a key $K_{AB} = (y_B)^{x_A} \bmod p$ as described earlier. A generates random values R and R_{AB} and computes the session key K and initialization vector I_{AB}:

   $$K = D_{AB}(R),\ I_{AB} = D_K(R_{AB}) \ ,$$

   where D_{AB} is the DES deciphering transformation with key K_{AB}. A sends to B the plaintext

   $$(A,\ R,\ R_{AB}) \ .$$

2. B obtains y_A and computes $K_{AB} = (y_B)^{x_A} \bmod p$. B computes

   $$K = D_{AB}(R),\ I_{AB} = D_K(R_{AB}) \ .$$

   B modifies I_{AB} in a predetermined way to get I'_{AB} and computes

   $$X = E_K(I'_{AB}) \ .$$

   B generates a random value R_{BA}, computes

   $$I_{BA} = D_K(R_{BA}) \ ,$$

   and sends $(X,\ R_{BA})$ to A.

3. A deciphers X to get $D_K(X) = I'_{AB}$. This confirms receipt of I_{AB} by B. A computes

   $$I_{BA} = D_K(R_{BA}) \ .$$

   A modifies I_{BA} in a predetermined way to get I'_{BA}, computes

   $$Y = E_K(I'_{BA}) \ ,$$

and sends Y to B.

4. B deciphers Y to get $D_K(Y) = I'_{BA}$, confirming receipt of I_{BA} by A.

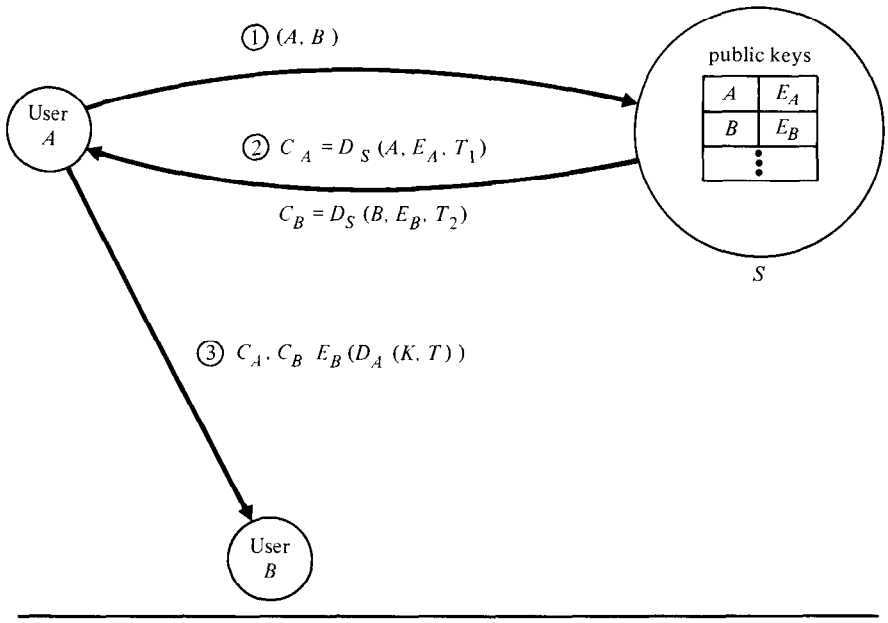
Note that all ciphertext transmitted between A and B is enciphered under the session key K, which is likely to have a much shorter lifetime than the shared key K_{AB}.

Computation of the public keys y_A and y_B and the private key K_{AB} is implemented in $\mathbf{GF}(2^n)$ using $n = 127$ and irreducible polynomial $p(x) = x^{127} + x + 1$ (see Section 1.6.3). Recall that implementation in $\mathbf{GF}(2^n)$ was also suggested for the Pohlig-Hellman exponentiation cipher, where computation of $y_A = a^{x_A} \bmod p$ corresponds to the encryption of plaintext a using encryption key x_A (see Section 2.7.1).

The MITRE demonstration system provides both central and local public-key distribution modes. With central key distribution, a user acquires public keys directly from the key distribution center each time the user wants to establish a secure connection. With local key distribution, the public key directory is downloaded to the user's system and stored in local memory. This has the advantage of relieving the central facility of participating in every connection. The demonstration system also provides a non-directory mode corresponding to the original Diffie-Hellman proposal, where the public values are exchanged directly by the users. This mode is suitable when active wiretapping does not pose a serious threat.

A public-key cryptosystem can also be used to exchange session keys [Diff76,Merk80]. Following the timestamp protocol in [Denn81b], let C_A and C_B

FIGURE 3.27 Public-key distribution protocol.



be A's and B's certificates [Eq. (3.13)], signed by S. A and B can exchange a key K with the following protocol (also refer to Figure 3.27):

Public-key distribution protocol

1. A sends the plaintext message (A, B) to S, requesting certificates for A and B.
2. S sends to A the certificates

$$C_A = D_S(A, E_A, T_1)$$
$$C_B = D_S(B, E_B, T_1) \ .$$

3. A checks C_A and gets B's public transformation E_B from C_B. A then generates a random key K, gets the time T, and sends the following to B:

$$(C_A, C_B, X) \ ,$$

where $X = E_B(D_A(K, T))$, and D_A is A's private deciphering transformation.

4. B checks C_B, gets A's public transformation E_A from C_A, and computes

$$E_A(D_B(X)) = (K, T) \ ,$$

where D_B is B's private transformation.

For added protection, the protocol can be extended to include a handshake between B and A, following the approach described for the centralized key distribution protocol. Of course, in a public-key system users can send messages directly using each other's public keys (obtained from certificates). The protocol would be used only if conventional encryption of messages is needed for performance reasons.

If the public keys are distributed among multiple hosts in a network, then the protocol must be extended slightly to allow exchange of certificates among the hosts. In Step 2, if A's host S does not have B's public key, it could dispatch a message to B's host requesting a certificate for B.

For additional reading on protocols for secure communication and key exchange, see Davies and Price [Davs79,Pric81], Kent [Kent78,Kent76], and Popek and Kline [Pope79]. Protocols for secure broadcast scenario are examined in Kent [Kent81].

3.8 THRESHOLD SCHEMES

Ultimately, the safety of all keys stored in the system—and therefore the entire system—may depend on a single master key. This has two serious drawbacks. First, if the master key is accidently or maliciously exposed, the entire system is vulnerable. Second, if the master key is lost or destroyed, all information in the system becomes inaccessible. The latter problem can be solved by giving copies of the key to "trustworthy" users. But in so doing, the system becomes vulnerable to betrayal.

The solution is to break a key K into w **shadows** (pieces) $K_1, \ldots, K_w$ in such a way that:

1. With knowledge of any t of the K_i, computing K is easy; and
2. With knowledge of any $t-1$ or fewer of the K_i, determining K is impossible because of lack of information.

The w shadows are given to w users. Because t shadows are required to reconstruct the key, exposure of a shadow (or up to $t-1$ shadows) does not endanger the key, and no group of less than t of the users can conspire to get the key. At the same time, if a shadow is lost or destroyed, key recovery is still possible (as long as there are at least t valid shadows). Such schemes are called (t, w) **threshold schemes** [Sham79], and can be used to protect any type of data. Blakley [Blak79] published the first threshold scheme, which was based on projective geometry. The following subsections describe two other approaches.

3.8.1 Lagrange Interpolating Polynomial Scheme

Shamir [Sham79] has proposed a scheme based on Lagrange interpolating polynomials (e.g., see [Ardi70,Cont72]). The shadows are derived from a random polynomial of degree $t-1$:

$$h(x) = (a_{t-1}x^{t-1} + \cdots + a_1 x + a_0) \bmod p \tag{3.15}$$

with constant term $a_0 = K$. All arithmetic is done in the Galois field $\mathbf{GF}(p)$, where p is a prime number larger than both K and w (long keys can be broken into smaller blocks to avoid using a large modulus p). Given $h(x)$, the key K is easily computed by

$$K = h(0) \ .$$

The w shadows are computed by evaluating $h(x)$ at w distinct values $x_1, \ldots, x_w$:

$$K_i = h(x_i) \quad i = 1, \ldots, w \ . \tag{3.16}$$

Each pair (x_i, K_i) is thus a point on the "curve" $h(x)$. The values $x_1, \ldots, x_w$ need not be secret, and could be user identifiers or simply the numbers 1 through w. Because t points uniquely determine a polynomial of degree $t-1$, $h(x)$ and, therefore, K can be reconstructed from t shadows. There is not enough information, however, to determine $h(x)$ or K from fewer than t shadows.

Given t shadows $K_{i_1}, K_{i_2}, \ldots, K_{i_t}$, $h(x)$ is reconstructed from the Lagrange polynomial:

$$h(x) = \sum_{s=1}^{t} K_{i_s} \prod_{\substack{j=1 \\ j \neq s}}^{t} \frac{(x - x_{i_j})}{(x_{i_s} - x_{i_j})} \bmod p \ . \tag{3.17}$$

Because arithmetic is in **GF**(p), the divisions in Eq. (3.17) are performed by computing inverses mod p and multiplying.

Example:
Let $t = 3$, $w = 5$, $p = 17$, $K = 13$, and

$$h(x) = (2x^2 + 10x + 13) \bmod 17$$

with random coefficients 2 and 10. Evaluating $h(x)$ at $x = 1, \ldots, 5$, we get five shadows:

$$\begin{aligned}
K_1 &= h(1) = (2 + 10 + 13) \bmod 17 = 25 \bmod 17 = 8 \\
K_2 &= h(2) = (8 + 20 + 13) \bmod 17 = 41 \bmod 17 = 7 \\
K_3 &= h(3) = (18 + 30 + 13) \bmod 17 = 61 \bmod 17 = 10 \\
K_4 &= h(4) = (32 + 40 + 13) \bmod 17 = 85 \bmod 17 = 0 \\
K_5 &= h(5) = (50 + 50 + 13) \bmod 17 = 113 \bmod 17 = 11 \ .
\end{aligned}$$

We can reconstruct $h(x)$ from any three of the shadows. Using K_1, K_3, and K_5, we have:

$$\begin{aligned}
h(x) &= \left[8\frac{(x-3)(x-5)}{(1-3)(1-5)} + 10\frac{(x-1)(x-5)}{(3-1)(3-5)} + 11\frac{(x-1)(x-3)}{(5-1)(5-3)}\right] \bmod 17 \\
&= \left[8\frac{(x-3)(x-5)}{(-2)(-4)} + 10\frac{(x-1)(x-5)}{(2)(-2)} + 11\frac{(x-1)(x-3)}{(4)(2)}\right] \bmod 17 \\
&= [8 * inv(8, 17) * (x-3)(x-5) \\
&\qquad + 10 * inv(-4, 17) * (x-1)(x-5) \\
&\qquad + 11 * inv(8, 17) * (x-1)(x-3)] \bmod 17 \\
&= [8 * 15 * (x-3)(x-5) + 10 * 4 * (x-1)(x-5) \\
&\qquad + 11 * 15 * (x-1)(x-3)] \bmod 17 \\
&= [(x-3)(x-5) + 6(x-1)(x-5) + 12(x-1)(x-3)] \bmod 17 \\
&= [19x^2 - 92x + 81] \bmod 17 \\
&= 2x^2 + 10x + 13 \ . \ \blacksquare
\end{aligned}$$

Blakley [Blak80] shows the scheme can be more efficiently implemented in **GF**(2^n) with modulus $p(x) = x^n + x + 1$ if $p(x)$ is irreducible (which it usually is not—see Section 1.6.3). Here, the coefficients a_i of $h(x)$ are elements of **GF**(2^n), and $h(x)$ is reduced mod $p(x)$. Similarly, the coordinates x_i and K_i are elements of **GF**(2^n), and Eq. (3.17) is evaluated in **GF**(2^n). Note that whereas the "x" in $h(x)$ is a variable or unknown, each "x" in $p(x)$ represents a placeholder for the binary representation of p.

Example:
Consider the field **GF**(2^3) with irreducible polynomial

$$p(x) = x^3 + x + 1 = 1011 \text{ (in binary)}$$

The elements of **GF**(2^3) are binary strings of length 3. Let $t = 2$, $w = 3$, $K = 011$, and

$$h(x) = (101x + 011) \bmod 1011$$

with random coefficient 101. Evaluating $h(x)$ at $x = 001$, 010, and 011, we get:

$$\begin{aligned} K_1 = h(001) &= (101 * 001 + 011) \bmod 1011 \\ &= 110 \\ K_2 = h(010) &= (101 * 010 + 011) \bmod 1011 \\ &= 001 + 011 = 010 \\ K_3 = h(011) &= (101 * 011 + 011) \bmod 1011 \\ &= 100 + 011 = 111 \ . \end{aligned}$$

We can reconstruct $h(x)$ from any two of the shadows. Using K_1 and K_2, we have

$$\begin{aligned} h(x) &= \left[110\frac{(x - 010)}{(001 - 010)} + 010\frac{(x - 001)}{(010 - 001)}\right] \bmod 1011 \\ &= \left[110\frac{(x - 010)}{011} + 010\frac{(x - 001)}{011}\right] \bmod 1011 \ . \end{aligned}$$

Because the inverse of 011 is 110, and subtraction is equivalent to addition, this reduces to:

$$\begin{aligned} h(x) &= [110 * 110 * (x + 010) + 010 * 110 * (x + 001)] \bmod 1011 \\ &= [010 * (x + 010) + 111 * (x + 001)] \bmod 1011 \\ &= 010x + 100 + 111x + 111 \\ &= 101x + 011 \ . \ \blacksquare \end{aligned}$$

Shamir observes that the general approach [implemented in either **GF**(p) or **GF**(2^n)] has several useful properties, namely:

1. The size of each shadow K_i does not substantially exceed the size of the key K [there may be some key expansion in **GF**(p), because p must be larger than K].
2. For fixed K, additional shadows can be created without changing existing ones just by evaluating h(x) at more values of x. A shadow can also be destroyed without affecting other shadows.
3. A shadow can be voided without changing K by using a different polynomial $h(x)$ with the same constant term. Voided shadows cannot be used unless there are at least t of them derived from the same polynomial.
4. A hierarchical scheme is possible, where the number of shadows given to each user is proportional to the user's importance. For example, a company president can be given three shadows, each vice-president two, and so forth.
5. Key recovery is efficient, requiring only $O(t^2)$ operations to evaluate Eq. (3.17) using standard algorithms, or $O(t \log^2 t)$ operations using methods discussed in [Knut69,Aho74].

3.8.2 Congruence Class Scheme

Asmuth and Bloom [Asmu80] have proposed a threshold scheme based on the Chinese Remainder Theorem (see Theorem 1.8 in Section 1.5.2). In their scheme the shadows are congruence classes of a number associated with K. Let

$$\{p, d_1, d_2, \ldots, d_w\}$$

be a set of integers such that

1. $p > K$
2. $d_1 < d_2 < \cdots < d_w$
3. $gcd(p, d_i) = 1$ for all i
4. $gcd(d_i, d_j) = 1$ for $i \neq j$
5. $d_1 d_2 \cdots d_t > p d_{w-t+2} d_{w-t+3} \cdots d_w$.

Requirements (3) and (4) imply the set of integers is pairwise relatively prime. Requirement (5) implies the product of the t smallest d_i is larger than the product of p and the $t-1$ largest d_i. Let $n = d_1 d_2 \cdots d_t$ be the product of the t smallest d_i. Thus n/p is larger than the product of any $t-1$ of the d_i. Let r be a random integer in the range $[0, (n/p) - 1]$. To decompose K into w shadows, $K' = K + rp$ is computed; this puts K' in the range $[0, n-1]$. The shadows are then as follows:

$$K_i = K' \bmod d_i \quad i = 1, \ldots, w \ . \tag{3.18}$$

To recover K, it suffices to find K'. If t shadows $K_{i_1}, \ldots, K_{i_t}$ are known, then by the Chinese Remainder Theorem K' is known modulo

$$n_1 = d_{i_1} d_{i_2} \cdots d_{i_t} \ .$$

Because $n_1 \geq n$, this uniquely determines K', which can be computed using algorithm *crt* in Figure 1.24:

$$K' = crt(n_1, d_{i_1}, \ldots, d_{i_t}, K_{i_1}, \ldots, K_{i_t}) \bmod n \ . \tag{3.19}$$

Finally, K is computed from K', r, and p:

$$K = K' - rp \ . \tag{3.20}$$

If only $t-1$ shadows $K_{i_1}, \ldots, K_{i_{t-1}}$ are known, K' can only be known modulo

$$n_2 = d_{i_1} d_{i_2} \ldots d_{i_{t-1}} \ .$$

Because $n/n_2 > p$ and $gcd(n_2, p) = 1$, the numbers x such that $x \leq n$ and $x \equiv_{n_2} K'$ are evenly distributed over all the congruence classes modulo p; thus, there is not enough information to determine K'.

Example:

Let $K = 3$, $t = 2$, $w = 3$, $p = 5$, $d_1 = 7$, $d_2 = 9$, and $d_3 = 11$. Then

$$n = d_1 d_2 = 7 * 9 = 63 > 5 * 11 = p d_3$$

as required. We need a random number r in the range $[0, (63/5) - 1] = [0, 11]$. Picking $r = 9$, we get

$$K' = K + rp = 3 + 9 * 5 = 48 .$$

The shadows are thus:

$$K_1 = 48 \bmod 7 = 6$$
$$K_2 = 48 \bmod 9 = 3$$
$$K_3 = 48 \bmod 11 = 4 .$$

Given any two of the shadows, we can compute K. Using K_1 and K_3, we have

$$n_1 = d_1 d_3 = 7 * 11 = 77 .$$

Applying algorithm *crt*, we first compute

$$y_1 = inv(n_1/d_1, d_1) = inv(11, 7) = 2$$
$$y_3 = inv(n_1/d_3, d_3) = inv(7, 11) = 8 .$$

Thus

$$\begin{aligned} K' &= \left[\left(\frac{n_1}{d_1}\right) y_1 K_1 + \left(\frac{n_1}{d_3}\right) y_3 K_3\right] \bmod n_1 \\ &= [11 * 2 * 6 + 7 * 8 * 4] \bmod 77 \\ &= 356 \bmod 77 \\ &= 48 . \end{aligned}$$

Thus,

$$K = K' - rp = 48 - 9 * 5 = 3 . \quad \blacksquare$$

Asmuth and Bloom describe an efficient algorithm for reconstructing K that requires only $O(t)$ time and $O(w)$ space. Thus their scheme is asymptotically more efficient than Shamir's polynomial scheme, which requires $O(t \log^2 t)$ time. For small t, this may not be an important consideration.

Asmuth and Bloom also describe a modification to their scheme for detecting defective shadows before their use. The basic idea is to remove the requirement that the moduli d_i be pairwise relatively prime [requirement (4)]. Then two shadows K_i and K_j will be congruent modulo $gcd(d_i, d_j)$ if both shadows are correct. Because an error in one shadow K_i will change its congruence class modulo $gcd(d_i, d_j)$ for most (if not all) $j \neq i$, defective shadows are easily detected. With this approach, requirement (5) must be changed to require that the least common multiple (*lcm*) of any t of the d_i is larger than the product of p and the *lcm* of any $t - 1$ of the d_i. (The general approach can also be incorporated in the polynomial interpolation scheme.)

Other threshold schemes have been proposed. Davida, DeMillo, and Lipton [Davi80] have proposed a scheme based on error correcting codes. Bloom [Bloo81] has outlined a class of schemes approaching optimal speed when the key length is large compared with the threshold value t; his schemes are based on linear maps

over finite fields. (See [Blak81] for a study of security proofs for threshold schemes.)

Whereas cryptosystems achieve computational security, threshold schemes achieve unconditional security by not putting enough information into any $t - 1$ shadows to reconstruct the key. Blakley [Blak80] has shown the one-time pad can be characterized as a $t = 2$ threshold scheme protecting a message M. Here, the sender and receiver each has a copy of the pad M_1, which serves as one shadow. The second shadow, constructed by the sender and transmitted to the receiver, is given by the ciphertext $M_2 = M \oplus M_1$. Both M_1 and M_2 are needed to reconstruct M. The interesting point is that classifying the one-time pad as a key threshold scheme explains how it can achieve perfect security when no other cryptosystem can.

EXERCISES

3.1 Suppose the ith ciphertext c_i is deleted from a direct access file enciphered with a synchronous stream cipher, and that the remainder of the file is deciphered and reenciphered using the original key stream, where c_{i+1} is now enciphered under k_i, c_{i+2} under k_{i+1}, and so forth. Show that all key elements k_j and plaintext elements m_j ($j \geq i$) can be determined from the original and updated ciphertext if m_i is known.

3.2 Let $M = 100011$ and $C = 101101$ be corresponding bit streams in a known plaintext attack, where the key stream was generated with a 3-stage linear feedback register. Using Eq. (3.2) solve for H to get the tap sequence T.

3.3 Describe methods for breaking Vigenère's autokey ciphers.

3.4 Let $d_1 = 3$, $d_2 = 5$, and $d_3 = 7$ be read subkeys in the scheme described in Section 3.4.2. Using Eq. (3.3), find write subkeys e_1, e_2, and e_3. Using Eq. (3.4), encipher a record having fields $m_1 = 2$, $m_2 = 4$, and $m_3 = 3$ (do not bother adding random strings to each field). Show that all three fields can be deciphered using only their read subkeys in Eq. (3.5). Using e_2 and d_2 in Eq. (3.6), modify the ciphertext so that the second field has the value 3 instead of 4. Show that all three fields can be extracted from the modified ciphertext using only their read subkeys.

3.5 Consider the subkey scheme described in Section 3.4.2. Let C_1 and C_2 be two ciphertext records of t fields each, and suppose the first $t - 1$ fields in each record are updated as defined by Eq. (3.6). Letting C_1' and C_2' denote the updated ciphertext records, show how the read key d_t can be determined for the tth field from the original and updated ciphertext records.

3.6 Complete the proof of Theorem 3.1 in Section 3.5.2 by showing how the exact value of i can be determined.

3.7 Consider a privacy homomorphism for the system of natural numbers with addition, multiplication, and test for equality. Given a ciphertext element i', show that it is possible to determine whether the corresponding plaintext element i is equal to an arbitrary constant n in $O(n)$ time without using any constants.

3.8 Give the authentication path for the user with key K_6 in Figure 3.24. Show how these values can be used to compute $H(1, 8)$.

3.9 Consider Shamir's key threshold scheme based on Lagrange interpolating polynomials in **GF**(p). Let $t = 4$, $p = 11$, $K = 7$, and

$$h(x) = (x^3 + 10x^2 + 3x + 7) \bmod 11 \ .$$

Using Eq. (3.16), compute shadows for $x = 1, 2, 3$, and 4. Using Eq. (3.17), reconstruct $h(x)$ from the shadows.

3.10 Consider Shamir's key threshold scheme based on Lagrange interpolating polynomials in **GF**(2^3) with irreducible polynomial $p(x) = x^3 + x + 1 = 1011$. Let $t = 3$, $K = 010$, and

$$h(x) = (001x^2 + 011x + 010) \bmod 1011 \ .$$

Compute shadows for $x = 001$, 011, and 100. Reconstruct $h(x)$ from the shadows.

3.11 Consider Asmuth's and Bloom's key threshold scheme based on the Chinese Remainder Theorem. Let $t = 2$, $w = 4$, $p = 5$, $d_1 = 8$, $d_2 = 9$, $d_3 = 11$, and $d_4 = 13$. Then $n = 8 * 9 = 72$. Let $K = 3$ and $r = 10$, whence $K' = 53$. Using Eq. (3.18), decompose K' into four shadows, K_1, K_2, K_3, and K_4. Using Eq. (3.19) and Eq. (3.20), reconstruct K from K_1 and K_2, and from K_1 and K_4.

3.12 *Class Computer Project:* Implement one of the key threshold schemes.

3.13 Consider a synchronous stream cipher where the ith key element k_i of a stream K is a block given by

$$k_i = (i + 1)^d \bmod n \ ,$$

where d is the private key to an RSA cipher and n is public. Thus,

$$\begin{aligned} K &= k_1, k_2, k_3, k_4, k_5, \ldots \\ &= 2^d, 3^d, 4^d, 5^d, 6^d, \ldots \pmod n \ . \end{aligned}$$

The ith block m_i of a message stream M is enciphered as $c_i = m_i \oplus k_i$. As shown by Shamir [Sham81], this stream is vulnerable to a known-plaintext attack. Show how a cryptanalyst knowing the plaintext-ciphertext pairs (m_1, c_1) and (m_2, c_2) can determine k_3 and k_5. Given many plaintext-ciphertext pairs, can the cryptanalyst determine d and thereby derive the entire key stream?

3.14 Shamir [Sham81] proposes the following key stream as an alternative to the one given in the preceding exercise:

$$\begin{aligned} K &= k_1, k_2, k_3, \ldots \\ &= S^{1/d_1}, S^{1/d_2}, S^{1/d_3}, \ldots \pmod n, \end{aligned}$$

where $n = pq$ for large primes p and q, the d_i are pairwise relatively prime and relatively prime to $\phi(n)$, S is secret, and $S^{1/d_i} \bmod n$ is the d_ith root of S mod n. An example of a stream is

$$K = S^{1/3}, S^{1/5}, S^{1/7}, \ldots \pmod n.$$

Shamir shows this stream is cryptographically strong; in particular, the diffi-

culty of determining unknown key elements is equivalent to breaking the RSA cipher. Show how the d_ith root of S can be computed to give k_i. [*Hint:* Find the inverse of d_i mod $\phi(n)$.] Show why this technique cannot be used to compute the square root of S.

3.15 Suppose users A and B exchange message M in a public-key system using the following protocol:

1. A encrypts M using B's public transformation, and sends the ciphertext message to B along with plaintext stating both A's and B's identity:

   $(A, B, E_B(M))$.

2. B deciphers the ciphertext, and replies to A with

   $(B, A, E_A(M))$.

Show how an active wiretapper could break the scheme to determine M. (See Dolev [Dole81] for a study on the security of this and other public key protocols.)

REFERENCES

Aho74. Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass. (1974).

Ardi70. Ardin, B. W. and Astill, K. N., *Numerical Algorithms,* Addison-Wesley, Reading, Mass. (1970).

Asmu80. Asmuth, C. and Bloom, J., "A Modular Approach to Key Safeguarding," Math. Dept., Texas A&M Univ., College Station, Tex. (1980).

Bara64. Baran, P., "On Distributed Communications: IX. Security, Secrecy, and Tamper-Free Considerations," RM-3765-PR, The Rand Corp., Santa Monica, Calif. (1964).

Baye76. Bayer, R. and Metzger, J. K., "On the Encipherment of Search Trees and Random Access Files," *ACM Trans. on Database Syst.* Vol. 1(1) pp. 37–52 (Mar. 1976).

Blak79. Blakley, G. R., "Safeguarding Cryptographic Keys," *Proc. NCC,* Vol. 48, AFIPS Press, Montvale, N.J., pp. 313–317 (1979).

Blak80. Blakley, G. R., "One-Time Pads are Key Safeguarding Schemes, Not Cryptosystems," *Proc. 1980 Symp. on Security and Privacy,* IEEE Computer Society, pp. 108–113 (Apr. 1980).

Blak81. Blakley, G. R. and Swanson, L., "Security Proofs for Information Protection Systems," in *Proc. 1981 Symp. on Security and Privacy,* IEEE Computer Society pp. 75–88 (Apr. 1981).

Bloo81. Bloom, J. R., "A Note on Superfast Threshold Schemes," Math. Dept., Texas A&M Univ., College Station, Tex. (1981).

Bran75. Branstad, D. K., "Encryption Protection in Computer Communication Systems," *Proc. 4th Data Communications Symp.,* pp. (8–1)–(8–7) (1975).

Bran78. Branstad, D. K., "Security of Computer Communication," *IEEE Comm. Soc. Mag.* Vol. 16(6) pp. 33–40 (Nov. 1978).

Brig76. Bright, H. S. and Enison, R. L., "Cryptography Using Modular Software Elements," *Proc. NCC,* Vol. 45, AFIPS Press, Montvale, N.J., pp. 113–123 (1976).

Brig80. Bright, H. S., "High-Speed Indirect Cryption," *Cryptologia* Vol. 4(3) pp. 133–139 (July 1980).

Camp78. Campbell, C. M., "Design and Specification of Cryptographic Capabilities," *IEEE Comm. Soc. Mag.* Vol. 16(6) pp. 15–19 (Nov. 1978).

Chai74. Chaitin, G. J., "Information-Theoretic Limitations of Formal Systems," *J. ACM* Vol. 21(3) pp. 403–424 (July 1974).

Chau81. Chaum, D. L., "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Comm. ACM* Vol. 24(2) pp. 84–88 (Feb. 1981).

Cont72. Conte, S. D. and deBoor, C., *Elementary Numerical Analysis,* McGraw-Hill, New York (1972).

Davi80. Davida, G. I., DeMillo, R. A., and Lipton, R. J., "Protecting Shared Cryptographic Keys," *Proc. 1980 Symp. on Security and Privacy,* IEEE Computer Society, pp. 100–102 (Apr. 1980).

Davi81. Davida, G. I., Wells, D. L., and Kam, J. B., "A Database Encryption System with Subkeys," *ACM Trans. on Database Syst.,* Vol. 6(2) pp. 312–328 (June 1981).

Davs79. Davies, D. W. and Price, W. L., "A Protocol for Secure Communication," NPL Report NACS 21/79, National Physical Lab., Teddington, Middlesex, England (Nov. 1979).

DeMi78. DeMillo, R., Lipton, R., and McNeil, L., "Proprietary Software Protection," pp. 115–131 in *Foundations of Secure Computation,* Academic Press, New York (1978).

Denn79. Denning, D. E., "Secure Personal Computing in an Insecure Network," *Comm. ACM* Vol. 22(8) pp. 476–482 (Aug. 1979).

Denn81a. Denning, D. E. and Schneider, F. B., "Master Keys for Group Sharing," *Info. Proc. Let.* Vol. 12(1) pp. 23–25 (Feb. 13, 1981).

Denn81b. Denning, D. E., Meijer, H., and Schneider, F. B., "More on Master Keys for Group Sharing," *Info. Proc. Let.* (to appear).

Denn81c. Denning, D. E. and Sacco, G. M., "Timestamps in Key Distribution Protocols," *Comm. ACM* Vol. 24(8) pp. 533–536 (Aug. 1981).

Diff76. Diffie, W. and Hellman, M., "New Directions in Cryptography," *IEEE Trans. on Info. Theory* Vol. IT-22(6) pp. 644–654 (Nov. 1976).

Diff79. Diffie, W. and Hellman, M., "Privacy and Authentication: An Introduction to Cryptography," *Proc. IEEE* Vol. 67(3) pp. 397–427 (Mar. 1979).

Dole81. Dolev, D. and Yao, A. C., "On the Security of Public Key Protocols," *Proc. 22nd Annual Symp. on the Foundations of Computer Science,* (1981).

Ehrs78. Ehrsam, W. F., Matyas, S. M., Meyer, C. H., and Tuchman, W. L., "A Cryptographic Key Management Scheme for Implementing the Data Encryption Standard," *IBM Syst. J.* Vol. 17(2) pp. 106–125 (1978).

Evan74. Evans, A. Jr., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM* Vol. 17(8) pp. 437–442 (Aug. 1974).

Feis73. Feistel, H., "Cryptography and Computer Privacy," *Sci. Am.* Vol. 228(5) pp. 15–23 (May 1973).

Feis75. Feistel, H., Notz, W. A., and Smith, J. L., "Some Cryptographic Techniques for Machine to Machine Data Communications," *Proc. IEEE* Vol. 63(11) pp. 1545–1554 (Nov. 1975).

Flyn78. Flynn, R. and Campasano, A. S., "Data Dependent Keys for a Selective Encryption Terminal," pp. 1127–1129 in *Proc. NCC,* Vol. 47, AFIPS Press, Montvale, N.J. (1978).

GSA77. "Telecommunications: Compatibility Requirements for Use of the Data Encryp-

tion Standard," Proposed Federal Standard 1026, General Services Administration Washington, D.C. (Oct. 1977).

Gait77. Gait, J., "A New Nonlinear Pseudorandom Number Generator," *IEEE Trans. on Software Eng.* Vol. SE-3(5) pp. 359–363 (Sept. 1977).

Golu67. Golumb, S. W., *Shift Register Sequences,* Holden-Day, San Francisco, Calif. (1967).

Gude80. Gudes, E., "The Design of a Cryptography Based Secure File System," *IEEE Trans. on Software Eng.* Vol. SE-6(5) pp. 411–420 (Sept. 1980).

Hell80. Hellman, M. E., "On DES-Based, Synchronous Encryption," Dept. of Electrical Eng., Stanford Univ., Stanford, Calif. (1980).

Kahn67. Kahn, D., *The Codebreakers,* Macmillan Co., New York (1967).

Kent76. Kent, S. T., "Encryption-Based Protection Protocols for Interactive User-Computer Communication," MIT/LCS/TR-162, MIT Lab. for Computer Science, Cambridge, Mass. (May 1976).

Kent78. Kent, S. T., "Protocol Design Considerations for Network Security," *Proc. of the NATO Advanced Studies Inst. on the Interlinking of Computer Networks,* D. Reidel, pp. 239–259 (1978).

Kent80. Kent, S. T., "Protecting Externally Supplied Software in Small Computers," Ph.D. Thesis, Dept. of Electrical Eng. and Computer Science, MIT, Cambridge, Mass. (Sept. 1980).

Kent81. Kent, S. T., "Security Requirements and Protocols for a Broadcast Scenario," *IEEE Trans. on Communications* Vol. COM-29(6) pp. 778–786 (June 1981).

Knut69. Knuth, D., *The Art of Computer Programming;* Vol. 2, *Seminumerical Algorithms,* Addison-Wesley, Reading, Mass. (1969).

Konf78. Kohnfelder, L. M., "A Method for Certification," MIT Lab. for Computer Science, Cambridge, Mass. (May 1978).

Konh80. Konheim, A. G., Mack, M. H., McNeill, R. K., Tuckerman, B., and Waldbaum, G., "The IPS Cryptographic Programs," *IBM Syst. J.* Vol. 19(2) pp. 253–283 (1980).

Lipt78. Lipton, S. M. and Matyas, S. M., "Making the Digital Signature Legal—and Safeguarded," *Data Communications,* pp. 41–52 (Feb. 1978).

Maty78. Matyas, S. M. and Meyer, C. H., "Generation, Distribution, and Installation of Cryptographic Keys," *IBM Syst. J.* Vol. 17(2) pp. 126–137 (1978).

Merk78. Merkle, R. C., "Secure Communication Over an Insecure Channel," *Comm. ACM* Vol. 21(4) pp. 294–299 (Apr. 1978).

Merk80. Merkle, R. C., "Protocols for Public Key Cryptosystems," *Proc. 1980 Symp. on Security and Privacy,* IEEE Computer Society, pp. 122–133 (Apr. 1980).

Meye72. Meyer, C. H. and Tuchman, W. L., "Pseudo-Random Codes Can Be Cracked," *Electronic Design* Vol. 23 (Nov. 1972).

Meye73. Meyer, C. H., "Design Considerations for Cryptography," *Proc. NCC,* Vol. 42 AFIPS Press, Montvale, N.J. pp. 603–606 (1973).

Morr79. Morris, R. and Thompson, K., "Password Security: A Case History," *Comm. ACM* Vol. 22(11) pp. 594–597 (Nov. 1979).

Need78. Needham, R. M. and Schroeder, M., "Using Encryption for Authentication in Large Networks of Computers," *Comm. ACM* Vol. 21(12) pp. 993–999 (Dec. 1978).

Pete72. Peterson, W. W. and Weldon, E. J., *Error Correcting Codes* MIT Press, Cambridge, Mass. (1972).

Pope79. Popek, G. J. and Kline, C. S., "Encryption and Secure Computer Networks," *Computing Surveys* Vol. 11(4) pp. 331–356 (Dec. 1979).

Pric81. Price, W. L. and Davies, D. W., "Issues in the Design of a Key Distribution

Centre," NPL Report DNACS 43/81, National Physical Lab., Teddington, Middlesex, England (Apr. 1981).

Purd74. Purdy, G. P., "A High Security Log-in Procedure," *Comm. ACM* Vol. 17(8) pp. 442–445 (Aug. 1974).

Reed73. Reed, I. S., "Information Theory and Privacy in Data Banks," *Proc. NCC* Vol. 42, AFIPS Press, Montvale, N.J., pp. 581–587 (1973).

Rive78a. Rivest, R. L., Adleman, L., and Dertouzos, M. L., "On Data Banks and Privacy Homomorphisms," pp. 169–179 in *Foundations of Secure Computation,* ed. R. A. DeMillo et al., Academic Press, New York (1978).

Rive78b. Rivest, R. L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM* Vol. 21(2) pp. 120–126 (Feb. 1978).

Salt78. Saltzer, J., "On Digital Signatures," *Oper. Syst. Rev.* Vol. 12(2) pp. 12–14 (Apr. 1978).

Sava67. Savage, J. E., "Some Simple Self-Synchronizing Digital Data Scramblers," *Bell System Tech. J.,* pp. 448–487 (Feb. 1967).

Scha79. Schanning, B. P., "Data Encryption with Public Key Distribution," *EASCON '79 Conf. Record,* pp. 653–660 (Oct. 1979).

Scha80. Schanning, B. P., Powers, S. A., and Kowalchuk, J., "Memo: Privacy and Authentication for the Automated Office," *Proc. 5th Conf. on Local Computer Networks,* pp. 21–30 (Oct. 1980).

Sham79. Shamir, A., "How to Share a Secret," *Comm. ACM* Vol. 22(11) pp. 612–613 (Nov. 1979).

Sham81. Shamir, A., "On the Generation of Cryptographically Strong Pseudo-Random Sequences," Dept. of Applied Math., The Weizmann Institute of Science, Rehovot, Israel (1981).

Turn73. Turn, R., "Privacy Transformations for Databank Systems," *Proc. NCC,* Vol. 42, AFIPS Press, Montvale, N.J., pp. 589–600 (1973).

Wilk75. Wilkes, M. V., *Time-Sharing Computing Systems,* Elsevier/MacDonald, New York (1968; 3rd ed., 1975).

Zier68. Zierler, N. and Brillhart, J., "On Primitive Trinomials (Mod 2)," *Info. and Control* Vol. 13 pp. 541–554 (1968).

Zier69. Zierler, N. and Brillhart, J., "On Primitive Trinomials (Mod 2)," *Info. and Control* Vol. 14 pp. 566–569 (1969).