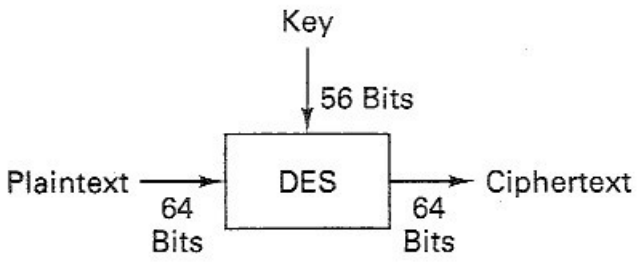

Figure 14.10 Data encryption standard (DES) viewed as a block encryption system.

the standard building block (SBB) shown in Figure 14.12. The standard building block uses 48 bits of key to transform the 64 input data bits into 64 output data bits, designated as 32 left-half bits and 32 right-half bits. The output of each building block becomes the input to the next building block. The input right-half 32 bits (R_{i-1}) are copied unchanged to become the output left-half 32 bits (L_i). The R_{i-1} bits are also *extended* and transformed into 48 bits with the E-table (Table 14.2), and then modulo-2 summed with the 48 bits of the key. As in the case of the IP-table, the E-table is read from left to right and from top to bottom. The table expands bits

$$R_{i-1} = x_1, x_2, \ldots, x_{32}$$

into

$$(R_{i-1})_E = x_{32}, x_1, x_2, \ldots, x_{32}, x_1 \tag{14.22}$$

Notice that the bits listed in the first and last columns of the E-table are those bit positions that are used twice to provide the 32 bit-to-48 bit expansion.

Next, $(R_{i-1})_E$ is modulo-2 summed with the ith key selection, explained later, and the result is segmented into eight 6-bit blocks

$$B_1, B_2, \ldots, B_8$$

That is,

$$(R_{i-1})_E \oplus K_i = B_1, B_2, \cdots, B_8 \tag{14.23}$$

Each of the eight 6-bit blocks, B_j, is then used as an input to an S-box function which returns a 4-bit block, $S_j(B_j)$. Thus the input 48 bits are transformed by the S-box to 32 bits. The S-box mapping function, S_j, is defined in Table 14.3. The transformation of $B_j = b_1, b_2, b_3, b_4, b_5, b_6$ is accomplished as follows. The integer corresponding to bits, b_1, b_6 selects a row in the table, and the integer corresponding to bits $b_2\, b_3\, b_4\, b_5$ selects a column in the table. For example, if $b_1 = 110001$, then S_1 returns the value in row 3, column 8, which is the integer 5 and is represented by the bit sequence 0101. The resulting 32-bit block out of the S-box is then permuted using the P-table (Table 14.4). As in the case of the other tables, the P-table is read from left to right and from top to bottom, so that bits $x_1, x_2, \ldots, x_{32}$ are permuted to $x_{16}, x_7, \ldots, x_{25}$. The 32-bit output of the P-table is modulo-2 summed with the input left-half 32 bits (L_{i-1}), forming the output right-half 32 bits (R_i).

The algorithm of the standard building block can be represented by

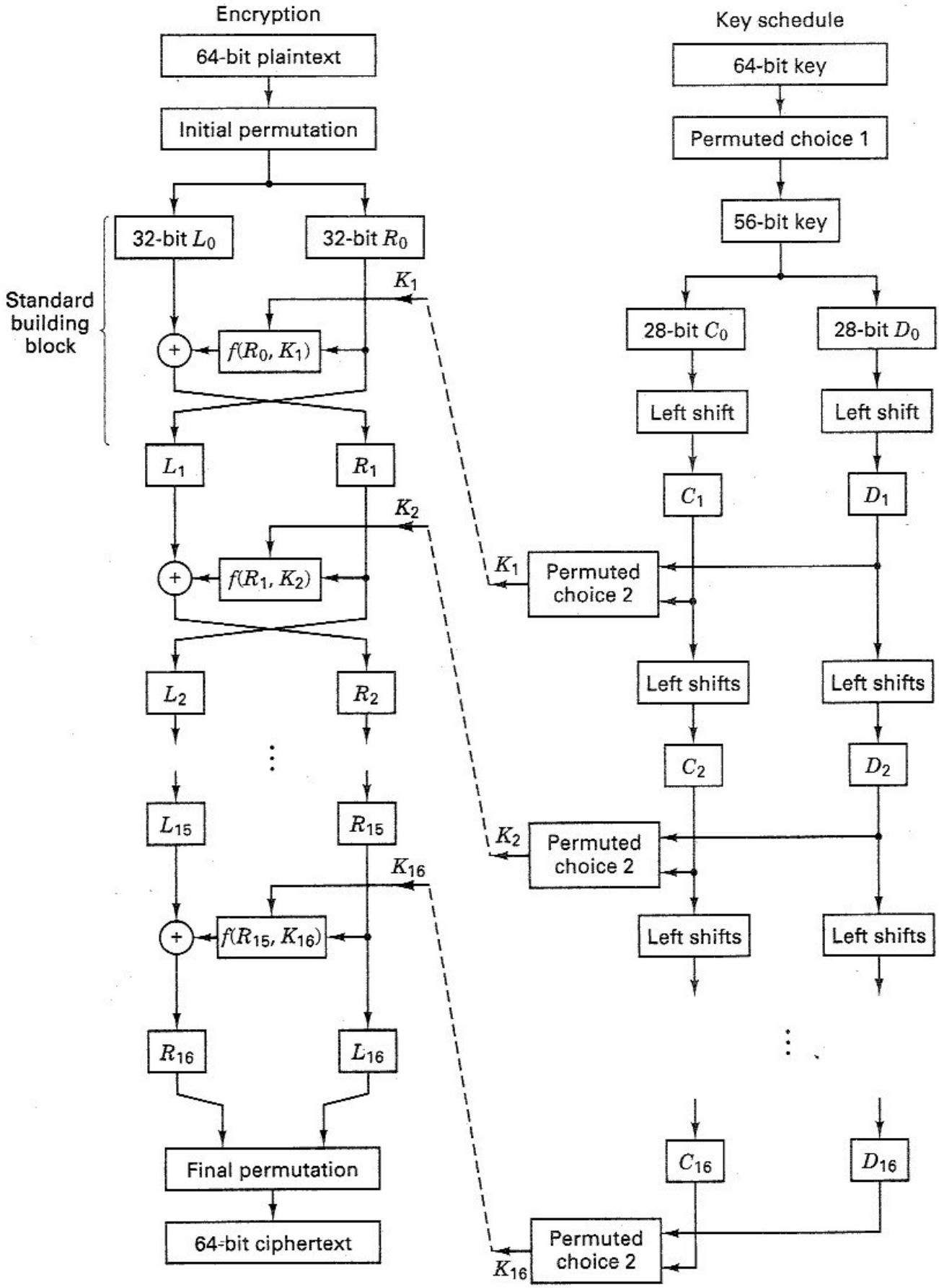

Figure 14.11 Data encryption standard.

TABLE 14.1 Initial Permutation (IP)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

$$L_i = R_{i-1} \tag{14.24}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \tag{14.25}$$

where $f(R_{i-1}, K_i)$ denotes the functional relationship comprising the E-table, S-box, and P-table we have described. After 16 iterations of the SBB, the data are transposed according to the final inverse permutation (IP^{-1}) described in the IP^{-1}-table (Table 14.5), where the output bits are read from left to right and from top to bottom, as before.

To decrypt, the same algorithm is used but the key sequence that is used in the standard building block is taken in the reverse order. Note that the value of $f(R_{i-1}, K_i)$ which can also be expressed in terms of the output of the ith block as $f(L_i, K_i)$, makes the decryption process possible.

14.3.5.1 Key Selection

Key selection also proceeds in 16 iterations, as seen in the key schedule portion of Figure 14.11. The input key consists of a 64-bit block with 8 parity bits in positions 8, 16, . . . , 64. The permuted choice 1 (PC-1) discards the parity bits and permutes the remaining 56 bits as shown in Table 14.6. The output of PC-1 is split into two halves, C and D, of 28 bits each. Key selection proceeds in 16 iterations in

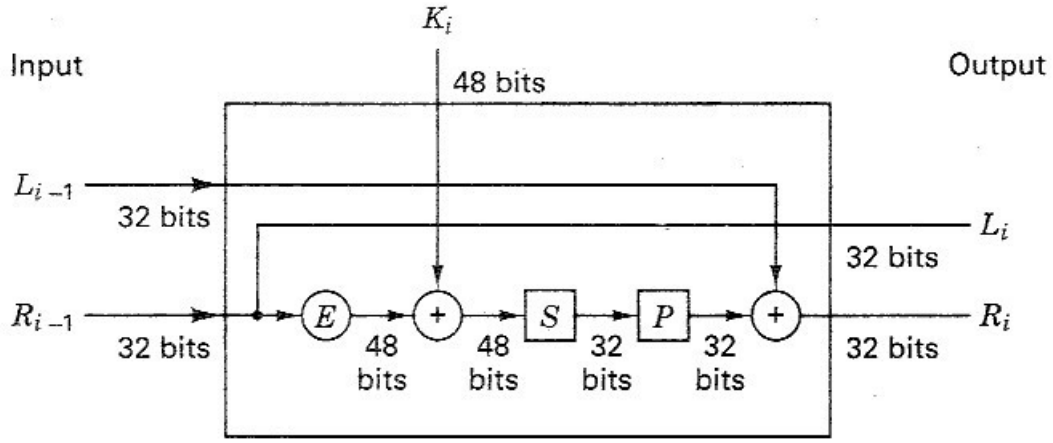

Figure 14.12 Standard building block (SBB).

TABLE 14.2 *E*-Table Bit Selection

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

TABLE 14.3 *S*-Box Selection Functions

	Column															
Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	S_1
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0	
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13	
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10	
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	S_2
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15	
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9	
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8	
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	S_3
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7	
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12	
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15	
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	S_4
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4	
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14	
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9	
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	S_5
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14	
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3	
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11	
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	S_6
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6	
3	4	3	2	12	9	5	15	0	11	14	1	7	6	0	8	13	
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1	
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	S_7
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2	
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12	
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7	
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	S_8
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8	
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11	

TABLE 14.4 *P*-Table Permutation

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

order to provide a different set of 48 key bits to each SBB encryption iteration. The C and D blocks are successively shifted according to

$$C_i = \mathrm{LS}_i(C_{i-1}) \quad \text{and} \quad D_i = \mathrm{LS}_i(D_{i-1}) \tag{14.26}$$

where LS_i is a left circular shift by the number of positions shown in Table 14.7. The sequence C_i, D_i is then transposed according to the permuted choice 2 (PC-2) shown in Table 14.8. The result is the key sequence K_i, which is used in the ith iteration of the encryption algorithm.

The DES can be implemented as a block encryption system (see Figure 14.11), which is sometimes referred to as a *codebook* method. A major disadvantage of this method is that a given block of input plaintext will always result in the same output ciphertext (under the same key). Another encryption mode, called the *cipher feedback* mode, encrypts single bits rather than characters, resulting in a stream encryption system [3]. With the cipher feedback scheme (described later), the encryption of a segment of plaintext not only depends on the key and the current data, but also on some of the earlier data.

Since the late 1970s, two points of contention have been widely publicized about the DES [10]. The first concerns the key variable length. Some researchers felt that 56 bits are not adequate to preclude an exhaustive search. The second concerns the details of the internal structure of the *S*-boxes, which were never released by IBM. The National Security Agency (NSA), which had been involved in the testing of the DES algorithm, had requested that the information not be publicly discussed, because it was sensitive. The critics feared that NSA had been involved in design selections that would allow NSA to "tap into" any DES-encrypted messages [10]. DES is no longer a viable choice for strong encryption. The 56-bit key

TABLE 14.5 Final Permutation (IP^{-1})

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

TABLE 14.6 Key Permutation PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

can be found in a matter of days with relatively inexpensive computer tools [11]. (Some alternative algorithms are discussed in Section 14.6.)

14.4 STREAM ENCRYPTION

Earlier, we defined a *one-time pad* as an encryption system with a random key, used one time only, that exhibits unconditional security. One can conceptualize a stream encryption implementation of a one-time pad using a truly random key stream (the key sequence never repeats). Thus, perfect secrecy can be achieved for an infinite number of messages, since each message would be encrypted with a different portion of the random key stream. The development of stream encryption schemes represents an attempt to emulate the one-time pad. Great emphasis was placed on generating key streams that appeared to be random, yet could easily be implemented for decryption, because they could be generated by algorithms. Such stream encryption techniques use pseudorandom (PN) sequences, which derive their name from the fact that they appear random to the casual observer; binary

TABLE 14.7 Key Schedule of Left Shifts

Iteration, i	Number of left shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

TABLE 14.8 Key Permutation PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

pseudorandom sequences have statistical properties similar to the random flipping of a fair coin. However, the sequences, of course, are deterministic (see Section 12.2). These techniques are popular because the encryption and decryption algorithms are readily implemented with feedback shift registers. At first glance it may appear that a PN key stream can provide the same security as the one-time pad, since the period of the sequence generated by a maximum-length linear shift register is $2^n - 1$ bits, where n is the number of stages in the register. If the PN sequence were implemented with a 50-stage register and a 1-MHz clock rate, the sequence would repeat every $2^{50} - 1$ microseconds, or every 35 years. In this era of large-scale integrated (LSI) circuits, it is just as easy to provide an implementation with 100 stages, in which case the sequence would repeat every 4×10^{16} years. Therefore, one might suppose that since the PN sequence does not repeat itself for such a long time, it would appear truly random and yield perfect secrecy. There is one important difference between the PN sequence and a truly random sequence used by a one-time pad. The PN sequence is generated by an algorithm; thus, knowing the algorithm, one knows the entire sequence. In Section 14.4.2 we will see that an encryption scheme that uses a linear feedback shift register in this way is very vulnerable to a *known plaintext attack.*

14.4.1 Example of Key Generation Using a Linear Feedback Shift Register

Stream encryption techniques generally employ shift registers for generating their PN key sequence. A shift register can be converted into a pseudorandom sequence generator by including a feedback loop that computes a new term for the first stage based on the previous n terms. The register is said to be linear if the numerical operation in the feedback path is linear. The PN generator example from Section 12.2 is repeated in Figure 14.13. For this example, it is convenient to number the stages as shown in Figure 14.13, where $n = 4$ and the outputs from stages 1 and 2 are modulo-2 added (linear operation) and fed back to stage 4. If the initial state of stages (x_4, x_3, x_2, x_1) is 1 0 0 0, the succession of states triggered by clock pulses would be 1 0 0 0, 0 1 0 0, 0 0 1 0, 1 0 0 1, 1 1 0 0, and so on. The output sequence is made up of the bits shifted out from the rightmost stage of the register, that is, 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0, where the rightmost bit in this sequence is the earliest output and the leftmost bit is the most recent output. Given any linear feedback shift register of degree n, the output sequence is ultimately periodic.

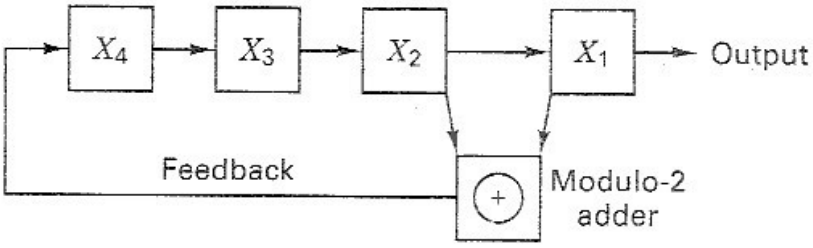

Figure 14.13 Linear feedback shift register example.

14.4.2 Vulnerabilities of Linear Feedback Shift Registers

An encryption scheme that uses a linear feedback shift register (LFSR) to generate the key stream is very vulnerable to attack. A cryptanalyst needs only $2n$ bits of plaintext and its corresponding ciphertext to determine the feedback taps, the initial state of the register, and the entire sequence of the code. In general, $2n$ is very small compared with the period $2^n - 1$. Let us illustrate this vulnerability with the LFSR example illustrated in Figure 14.13. Imagine that a cryptanalyst who knows nothing about the internal connections of the LFSR manages to obtain $2n = 8$ bits of ciphertext and its plaintext equivalent:

Plaintext: 0 1 0 1 0 1 0 1
Ciphertext: 0 0 0 0 1 1 0 0

Where, the rightmost bit is the earliest received and the leftmost bit is the most recent that was received.

The cryptanalyst adds the two sequences together, modulo-2, to obtain the segment of the key stream, 0 1 0 1 1 0 0 1, illustrated in Figure 14.14. The key stream sequence shows the contents of the LFSR stages at various times. The rightmost border surrounding four of the key bits shows the contents of the shift register at time t_1. As we successively slide the "moving" border one digit to the left, we see the shift register contents at times $t_2, t_3, t_4, \ldots$. From the linear structure of the four-stage shift register, we can write

$$g_4x_4 + g_3x_3 + g_2x_2 + g_1x_1 = x_5 \tag{14.27}$$

where x_5 is the digit fed back to the input and g_i (= 1 or 0) defines the ith feedback connection. For this example, we can thus write the following four equations with four unknowns, by examining the contents of the shift register at the four times shown in Figure 14.14:

$$\begin{aligned} g_4(1) + g_3(0) + g_2(0) + g_1(1) &= 1 \\ g_4(1) + g_3(1) + g_2(0) + g_1(0) &= 0 \\ g_4(0) + g_3(1) + g_2(1) + g_1(0) &= 1 \\ g_4(1) + g_3(0) + g_2(1) + g_1(1) &= 0 \end{aligned} \tag{14.28}$$

The solution of Equations (14.28) is $g_1 = 1, g_2 = 1, g_3 = 0, g_4 = 0$, corresponding to the LFSR shown in Figure 14.13. The cryptanalyst has thus learned the connections of

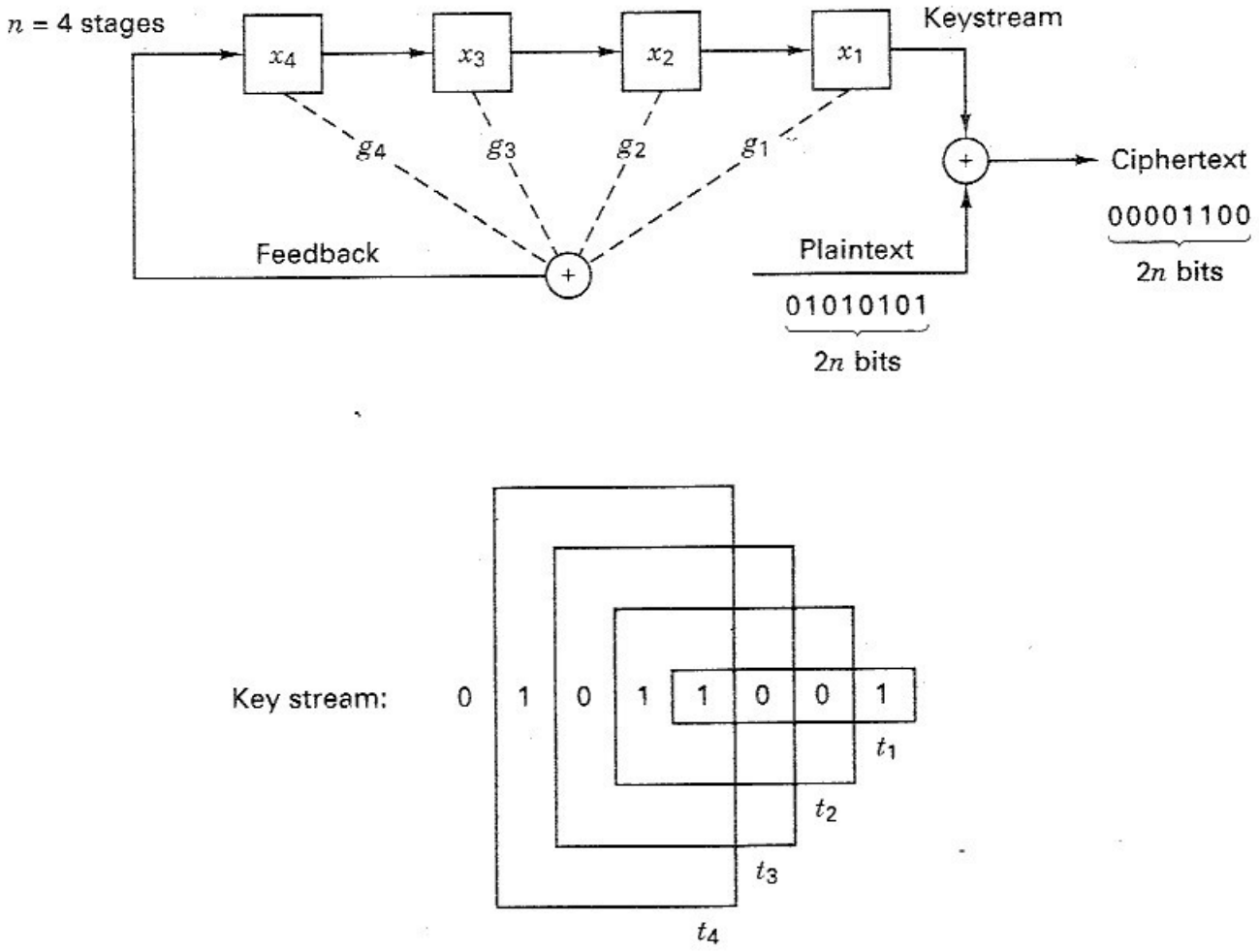

Figure 14.14 Example of vulnerability of a linear feedback shift register.

the LFSR, together with the starting state of the register at time t_1. He can therefore know the sequence for all time [3]. To generalize this example for any n-stage LFSR, we rewrite Equation (14.27) as follows:

$$x_{n+1} = \sum_{i=1}^{n} g_i x_i \tag{14.29}$$

We can write Equation (14.29) as the matrix equation

$$\mathbf{x} = \mathbf{X}\mathbf{g} \tag{14.30}$$

where

$$\mathbf{x} = \begin{bmatrix} x_{n+1} \\ x_{n+2} \\ \vdots \\ x_{2n} \end{bmatrix} \quad \mathbf{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$$

and

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & & \vdots \\ x_n & x_{n+1} & \cdots & x_{2n-1} \end{bmatrix}$$

It can be shown [3] that the columns of $\mathbf{X}$ are linearly independent; thus $\mathbf{X}$ is nonsingular (its determinant is nonzero) and has an inverse. Hence,

$$\mathbf{g} = \mathbf{X}^{-1}\,\mathbf{x} \tag{14.31}$$

The matrix inversion requires at most on the order of n^3 operations and is thus easily accomplished by computer for any reasonable value of n. For example, if $n = 100$, $n^3 = 10^6$, and a computer with a 1-μs operation cycle would require 1 s for the inversion. The weakness of a LFSR is caused by the linearity of Equation (14.31). The use of *nonlinear feedback* in the shift register makes the cryptanalyst's task much more difficult, if not computationally intractable.

14.4.3 Synchronous and Self-Synchronous Stream Encryption Systems

We can categorize stream encryption systems as either *synchronous* of *self-synchronous*. In the former, the key stream is generated independently of the message, so that a lost character during transmission necessitates a resynchronization of the transmission and receiver key generators. A synchronous stream cipher is shown in Figure 14.15. The starting state of the key generator is initialized with a known input, I_0. The ciphertext is obtained by the modulo addition of the ith key character, k_i, with the ith message character, m_i. Such synchronous ciphers are generally designed to utilize *confusion* (see Section 14.3.1) but not *diffusion*. That is, the encryption of a character is not diffused over some block length of message. For this reason, synchronous stream ciphers do not exhibit *error propagation*.

In a *self-synchronous* stream cipher, each key character is derived from a fixed number, n, of the preceding ciphertext characters, giving rise to the name *cipher feedback*. In such a system, if a ciphertext character is lost during

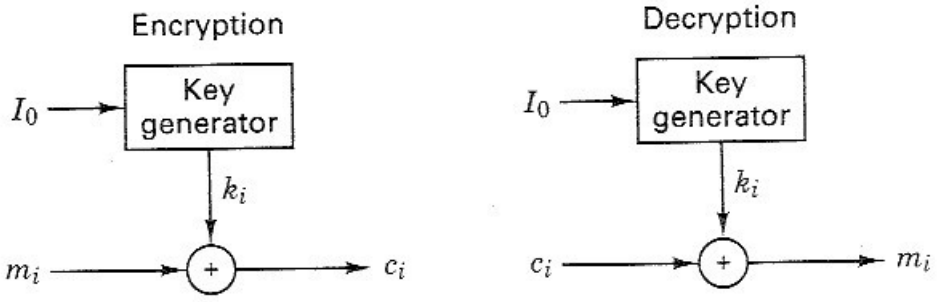

Figure 14.15 Synchronous stream cipher.

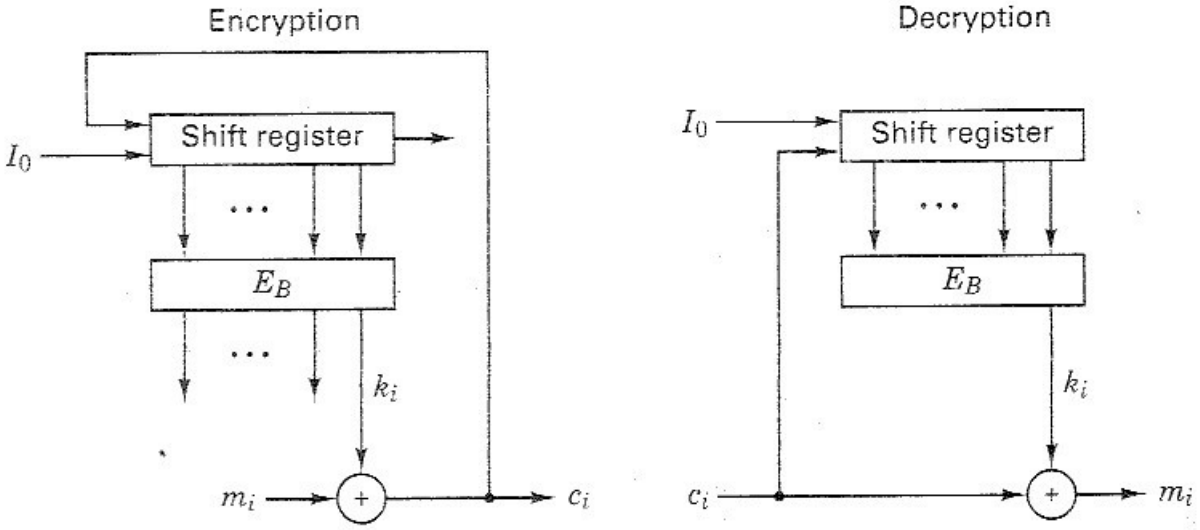

Figure 14.16 Cipher feedback mode.

transmission, the error propagates forward for n characters, but the system resynchronizes itself after n correct ciphertext characters are received.

In Section 14.1.4 we looked at an example of cipher feedback in the Vigenere auto key cipher. We saw that the advantages of such a system are that (1) a nonrepeating key is generated, and (2) the statistics of the plaintext message are diffused throughout the ciphertext. However, the fact that they key was exposed in the ciphertext was a basic weakness. This problem can be eliminated by passing the ciphertext characters through a nonlinear block cipher to obtain the key characters. Figure 14.16 illustrates a shift register key generator operating in the cipher feedback mode. Each output ciphertext character, c_i (formed by the modulo addition of the message character, m_i, and the key character, k_i), is fed back to the input of the shift register. As before, initialization is provided by a known input, I_0. At each iteration, the output of the shift register is used as input to a (nonlinear) block encryption algorithm E_B. The low-order output character from E_B becomes the next key character, k_{i+1}, to be used with the next message character, m_{i+1}. Since, after the first few iterations, the input to the algorithm depends only on the ciphertext, the system is self-synchronizing.

14.5 PUBLIC KEY CRYPTOSYSTEMS

The concept of public key cryptosystems was introduced in 1976 by Diffie and Hellman [12]. In conventional cryptosystems the encryption algorithm can be revealed since the security of the system depends on a safeguarded key. The same key is used for both encryption and decryption. Public key cryptosystems utilize *two different* keys, one for encryption and the other for decryption. In public key cryptosystems, not only the encryption algorithm but also the encryption key can be publicly revealed without compromising the security of the system. In fact, a public directory, much like a telephone directory, is envisioned, which contains the

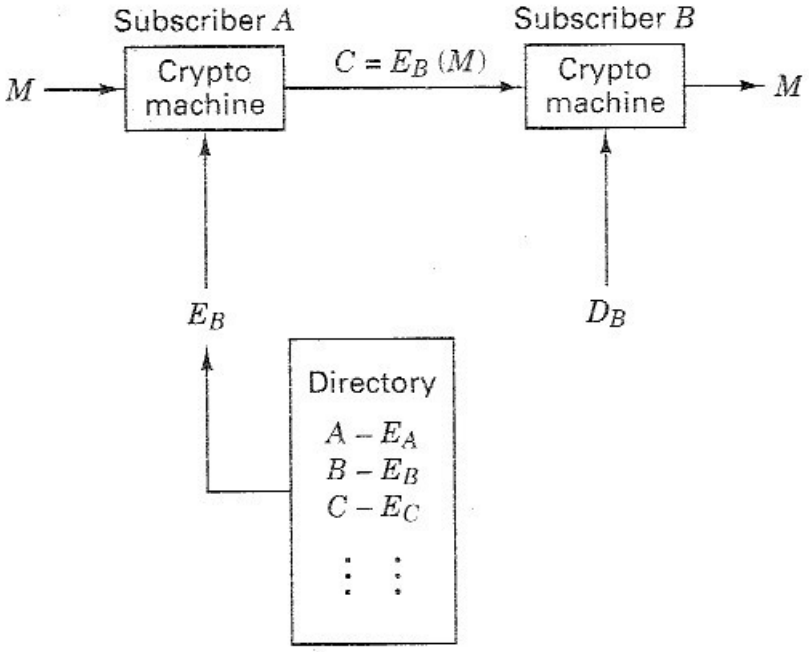

Figure 14.17 Public key cryptosystem.

encryption keys of all the subscribers. Only the decryption keys are kept secret. Figure 14.17 illustrates such a system. The important features of a public key cryptosystem are as follows:

1. The encryption algorithm E_K and the decryption algorithm D_K are invertible transformations on the plaintext M, or the ciphertext C, defined by the key K. That is, for each K and M, if $C = E_K(M)$, then $M = D_K(C) = D_K[E_K(M)]$.
2. For each K, E_K and D_K are easy to compute.
3. For each K, the computation of D_K from E_K is computationally intractable.

Such a system would enable secure communication between subscribers who have never met or communicated before. For example, as seen in Figure 14.17, subscriber A can send a message, M, to subscriber B by looking up B's encryption key in the directory and applying the encryption algorithm, E_B, to obtain the ciphertext $C = E_B(M)$, which he transmits on the public channel. Subscriber B is the only party who can decrypt C by applying his decryption algorithm, D_B, to obtain $M = D_B(C)$.

14.5.1 Signature Authentication Using a Public Key Cryptosystem

Figure 14.18 illustrates the use of a public key cryptosystem for signature authentication. Subscriber A "signs" his message by first applying his decryption algorithm, D_A, to the message, yielding $S = D_A(M) = E^{-1}_A(M)$. Next, he uses the encryption algorithm, E_B, of subscriber B to encrypt S, yielding $C = E_B(S) = E_B[E^{-1}_A(M)]$, which he transmits on a public channel. When subscriber B receives C, he first decrypts it using his private decryption algorithm, D_B, yielding $D_B(C) = E^{-1}_A(M)$. Then he applies the encryption algorithm of subscriber A to produce $E_A[E^{-1}_A(M)] = M$.

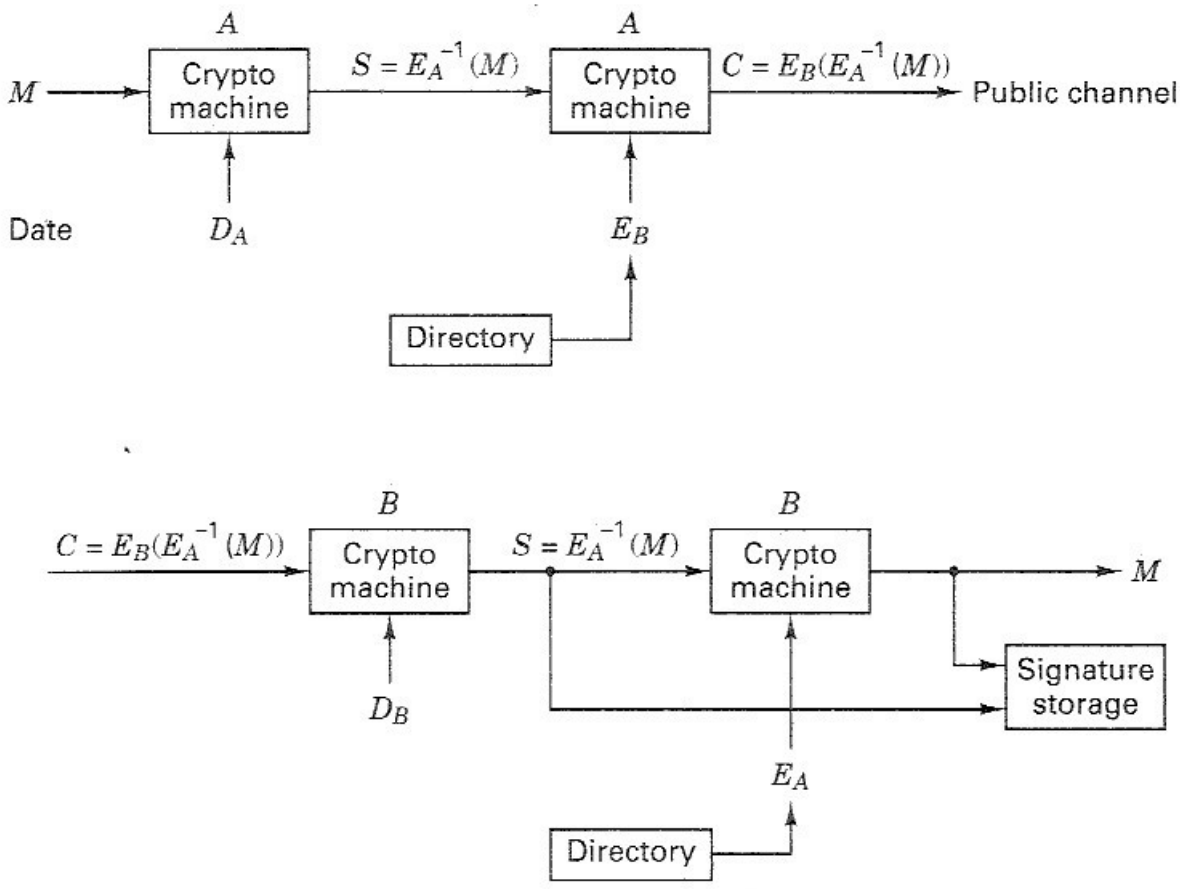

Figure 14.18 Signature authentication using a public key cryptosystem.

If the result is an intelligible message, it must have been initiated by subscriber A, since no one else could have known A's secret decryption key to form $S = D_A(M)$. Notice that S is both message dependent and signer dependent, which means that while B can be sure that the received message indeed came from A, at the same time A can be sure that no one can attribute any false messages to him.

14.5.2 A Trapdoor One-Way Function

Public key cryptosystems are based on the concept of trapdoor one-way functions. Let us first define, a *one-way function* as an easily computed function whose inverse is computationally infeasible to find. For example, consider the function $y = x^5 + 12x^3 + 107x + 123$. It should be apparent that given x, y is easy to compute, but given y, x is relatively difficult to compute. A *trapdoor one-way function* is a one-way function whose inverse is easily computed if certain features, used to design the function, are known. Like a trapdoor, such functions are easy to go through in one direction. Without special information the reverse process takes an impossibly long time. We will apply the concept of a trapdoor in Section 14.5.5, when we discuss the Merkle–Hellman scheme.

14.5.3 The Rivest–Shamir–Adelman Scheme

In the Rivest–Shamir–Adelman (RSA) scheme messages are first represented as integers in the range $(0, n-1)$. Each user chooses his own value of n and another pair of positive integers e and d, in a manner to be described below. The user places his encryption key, the number pair (n, e), in the public directory. The decryption key consists of the number pair (n, d), of which d is kept secret. Encryption of a message M and decryption of a ciphertext C are defined as follows:

$$\begin{aligned} &\text{Encryption:} \quad C = E(M) = (M)^e \text{ modulo-}n \\ &\text{Decryption:} \quad M = D(C) = (C)^d \text{ modulo-}n \end{aligned} \tag{14.32}$$

They are each easy to compute and the results of each operation are integers in the range $(0, n-1)$. In the RSA scheme, n is obtained by selecting *two large prime numbers* p and q and multiplying them together:

$$n = pq \tag{14.33}$$

Although n is made public, p and q are kept hidden, due to the great difficulty in factoring n. Then

$$\phi(n) = (p-1)(q-1) \tag{14.34}$$

called *Euler's totient function*, is formed. The parameter $\phi(n)$ has the interesting property [12] that for any integer X in the range $(0, n-1)$ and any integer k,

$$X = X^{k\phi(n)+1} \text{ modulo-}n \tag{14.35}$$

Therefore, while all other arithmetic is done modulo-n, arithmetic in the exponent is done modulo-$\phi(n)$. A large integer, d, is randomly chosen so that it is relatively prime to $\phi(n)$, which means that $\phi(n)$ and d must have no common divisors other than 1, expressed as

$$\gcd[\phi(n), d] = 1 \tag{14.36}$$

where gcd means "greatest common divisor." Any prime number greater than the larger of (p, q) will suffice. Then the integer e, where $0 < e < \phi(n)$, is found from the relationship

$$ed \text{ modulo-}\phi(n) = 1 \tag{14.37}$$

which, from Equation (14.35), is tantamount to choosing e and d to satisfy

$$X = X^{ed} \text{ modulo-}n \tag{14.38}$$

Therefore,

$$E[D(X)] = D[E(X)] = X \tag{14.39}$$

and decryption works correctly. Given an encryption key (n, e), one way that a cryptanalyst might attempt to break the cipher is to factor n into p and q, compute $\phi(n) = (p-1)(q-1)$, and compute d from Equation (14.37). This is all straightforward except for the factoring of n.

The RSA scheme is based on the fact that it is easy to generate two large prime numbers, p and q, and multiply them together, but it is very much more difficult to factor the result. The product can therefore be made public as part of the encryption key, without compromising the factors that would reveal the decryption key corresponding to the encryption key. By making each of the factors roughly 100 digits long, the multiplication can be done in a fraction of a second, but the exhaustive factoring of the result should take billions of years [2].

14.5.3.1 Use of the RSA Scheme

Using the example in Reference [13], let $p = 47$, $q = 59$. Therefore, $n = pq = 2773$ and $\phi(n) = (p - 1)(q - 1) = 2668$. The parameter d is chosen to be relatively prime to $\phi(n)$. For example, choose $d = 157$. Next, the value of e is computed as follows (the details are shown in the next section):

$$ed \text{ modulo } \phi(n) = 1$$

$$157e \text{ modulo } 2688 = 1$$

Therefore, $e = 17$. Consider the plaintext example

ITS ALL GREEK TO ME

By replacing each letter with a two-digit number in the range (01, 26) corresponding to its position in the alphabet, and encoding a blank as 00, the plaintext message can be written as

0920 1900 0112 1200 0718 0505 1100 2015 0013 0500

Each message needs to be expressed as an integer in the range $(0, n - 1)$; therefore, for this example, encryption can be performed on blocks of four digits at a time since this is the maximum number of digits that will always yield a number less than $n - 1 = 2772$. The first four digits (0920) of the plaintext are encrypted as follows:

$$C = (M)^e \text{ modulo-}n = (920)^{17} \text{ modulo-}2773 = 948$$

Continuing this process for the remaining plaintext digits, we get

$$C = 0948\ 2342\ 1084\ 1444\ 2663\ 2390\ 0778\ 0774\ 0219\ 1655$$

The plaintext is returned by applying the decryption key, as follows:

$$M = (C)^{157} \text{ modulo-}2773$$

14.5.3.2 How to Compute e

A variation of Euclid's algorithm [14] for computing the gcd of $\phi(n)$ and d is used to compute e. First, compute a series $x_0, x_1, x_2, \ldots$, where $x_0 = \phi(n)$, $x_1 = d$, and $x_{i+1} = x_{i-1}$ modulo-x_i, until an $x_k = 0$ is found. Then the gcd $(x_0, x_1) = x_{k-1}$. For each x_i compute numbers a_i and b_i such that $x_i = a_i x_0 + b_i x_1$. If $x_{k-1} = 1$, then b_{k-1} is the multiplicative inverse of x_1 modulo-x_0. If b_{k-1} is a negative number, the solution is $b_{k-1} + \phi(n)$.

Example 14.5 Computation of e from d and $\phi(n)$

For the previous example, with $p = 47$, $q = 59$, $n = 2773$, $\phi(n) = 2688$, and d chosen to be 157, use the Euclid algorithm to verify that $e = 17$.

Solution

i	x_i	a_i	b_i	y_i
0	2668	1	0	
1	157	0	1	16
2	156	1	−16	1
3	1	−1	17	

where

$$y_i = \left\lfloor \frac{x_{i-1}}{x_i} \right\rfloor$$

$$x_{i+1} = x_{i-1} - y_i x_i$$

$$a_{i+1} = a_{i-1} - y_i a_i$$

$$b_{i+1} = b_{i-1} - y_i b_i$$

Hence

$$e = b_3 = 17$$

14.5.4 The Knapsack Problem

The classic knapsack problem is illustrated in Figure 14.19. The knapsack is filled with a subset of the items shown with weights indicated in grams. Given the weight of the filled knapsack (the scale is calibrated to deduct the weight of the empty knapsack), determine which items are contained in the knapsack. For this simple example, the solution can easily be found by trial and error. However, if there are 100 possible items in the set instead of 10, the problem may become computationally infeasible.

Let us express the knapsack problem in terms of a knapsack vector and a data vector. The knapsack vector is an n-tuple of distinct integers (analogous to the set of possible knapsack items)

$$\mathbf{a} = a_1, a_2, \ldots, a_n$$

The data vector is an n-tuple of binary symbols

$$\mathbf{x} = x_1, x_2, \ldots, x_n$$

The knapsack, S, is the sum of a subset of the components of the knapsack vector:

$$S = \sum_{i=1}^{n} a_i x_i \quad \text{where } x_i = 0, 1 \tag{14.40}$$

$$= \mathbf{ax}$$

The knapsack problem can be stated as follows: Given S and knowing $\mathbf{a}$, determine $\mathbf{x}$.

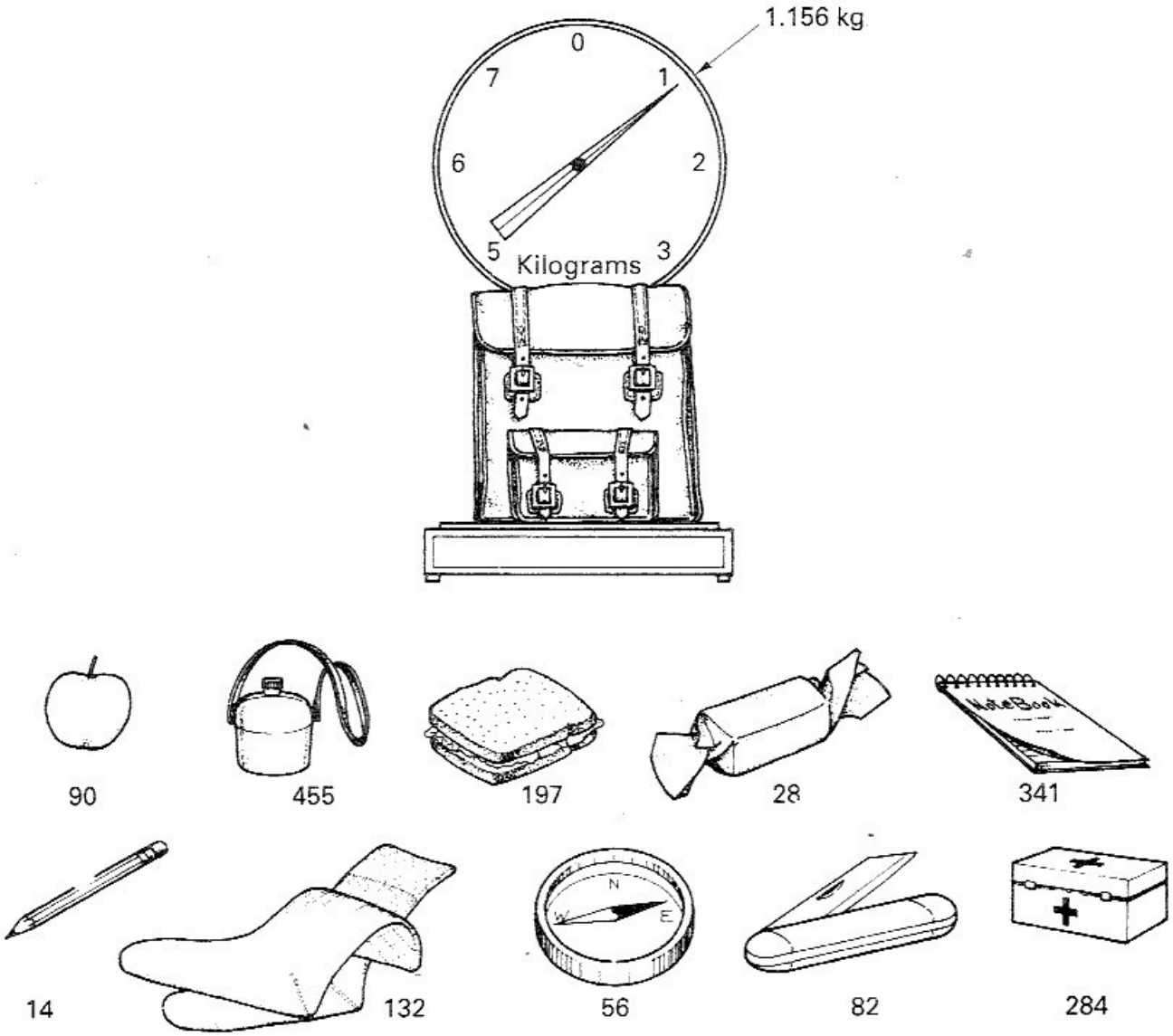

Figure 14.19 Knapsack problem.

Example 14.6 Knapsack Example

Given $\mathbf{a} = 1, 2, 4, 8, 16, 32$ and $S = \mathbf{ax} = 26$, find $\mathbf{x}$.

Solution

In this example $\mathbf{x}$ is seen to be the *binary* representation of S. The decimal-to-binary conversion should appear more familiar with $\mathbf{a}$ expressed as $2^0, 2^1, 2^2, 2^3, 2^4, 2^5$. The data vector $\mathbf{x}$ is easily found since $\mathbf{a}$ in this example is *super-increasing*, which means that each component of the n-tuple $\mathbf{a}$ is larger than the sum of the preceding components. That is,

$$a_i > \sum_{j=1}^{i-1} a_j \qquad i = 2, 3, \ldots, n \tag{14.41}$$

When $\mathbf{a}$ is super-increasing, the solution of $\mathbf{x}$ is found by starting with $x_n = 1$ if $S \geq a_n$ (otherwise $x_n = 0$) and continuing according to the relationship

$$x_i = \begin{cases} 1 & \text{if } S - \sum_{j=i+1}^{n} x_j a_j \geq a_i \\ 0 & \text{otherwise} \end{cases} \tag{14.42}$$

where $i = n - 1, n - 2, \ldots, 1$. From Equation (14.42) it is easy to compute $\mathbf{x} = 0\,1\,0\,1\,1\,0$.

Example 14.7 Knapsack Example

Given $\mathbf{a} = 171, 197, 459, 1191, 2410, 4517$ and $\mathbf{S} = \mathbf{ax} = 3798$, find $\mathbf{x}$.

Solution

As in Example 14.6, $\mathbf{a}$ is super-increasing; therefore, we can compute $\mathbf{x}$ using Equation (14.42), which again yields

$$\mathbf{x} = 0\,1\,0\,1\,1\,0$$

14.5.5 A Public Key Cryptosystem Based on a Trapdoor Knapsack

This scheme, also known as the Merkle–Hellman scheme [15], is based on the formation of a knapsack vector that is not super-increasing and is therefore not easy to solve. However, an essential part of this knapsack is a *trapdoor* that enables the authorized user to solve it.

First, we form a super-increasing n-tuple $\mathbf{a}'$. Then we select a prime number M such that

$$M > \sum_{i=1}^{n} a_i' \tag{14.43}$$

We also select a random number W, where $1 < W < M$, and we form W^{-1} to satisfy the following relationship:

$$WW^{-1} \text{ modulo-}M = 1 \tag{14.44}$$

the vector $\mathbf{a}'$ and the numbers M, W, and W^{-1} are all kept hidden. Next, we form $\mathbf{a}$ with the elements from $\mathbf{a}'$, as follows:

$$a_i = Wa_j' \text{ modulo-}M \tag{14.45}$$

The formation of $\mathbf{a}$ using Equation (14.45) constitutes forming a knapsack vector with a *trapdoor*. When a data vector $\mathbf{x}$ is to be transmitted, we multiply $\mathbf{x}$ by $\mathbf{a}$, yielding the number S, which is sent on the public channel. Using Equation (14.45), S can be written as follows:

$$S = \mathbf{ax} = \sum_{i=1}^{n} a_i x_i = \sum_{i=1}^{n} (Wa_i' \text{ modulo-}M)x_i \tag{14.46}$$

The authorized user receives S and, using Equation (14.44), converts it to S':

$$S' = W^{-1}S \text{ modulo-}M = W^{-1} \sum_{i=1}^{n} (Wa_i' \text{ modulo-}M)x_i \text{ modulo-}M$$

$$= \sum_{i=1}^{n} (W^{-1} W a'_i \text{ modulo-}M) x_i \text{ modulo-}M \tag{14.47}$$

$$= \sum_{i=1}^{n} a'_i x_i \text{ modulo-}M$$

$$= \sum_{i=1}^{n} a'_i x_i$$

Since the authorized user knows the secretly held super-increasing vector $\mathbf{a}'$, he or she can use S' to find $\mathbf{x}$.

14.5.5.1 Use of the Merkle–Hellman Scheme

Suppose that user A wants to construct public and private encryption functions. He first considers the super-increasing vector $\mathbf{a}' = (171, 197, 459, 1191, 2410, 4517)$

$$\sum_{i=1}^{6} a'_i = 8945$$

He then chooses a prime number M larger than 8945, a random number W, where $1 \le W < M$, and calculates W^{-1} to satisfy $WW^{-1} = 1$ modulo-M.

$$\left.\begin{array}{l} \text{Choose } M = 9109 \\ \text{choose } W = 2251 \\ \text{then } W^{-1} = 1388 \end{array}\right\} \text{kept hidden}$$

He then forms the trapdoor knapsack vector as follows:

$$a_i = a'_i 2251 \text{ modulo-}9109$$

$$\mathbf{a} = 2343, 6215, 3892, 2895, 5055, 2123$$

User A makes public the vector $\mathbf{a}$, which is clearly not super-increasing. Suppose that user B wants to send a message to user A.

If $\mathbf{x} = 0\,1\,0\,1\,1\,0$ is the message to be transmitted, user B forms

$$S = \mathbf{ax} = 14{,}165 \text{ and transmits it to user } A$$

User A, who receives S, converts it to S':

$$S' = \mathbf{a}'\mathbf{x} = W^{-1}S \text{ modulo-}M$$

$$= 1388 \cdot 14{,}165 \text{ modulo-}9109$$

$$= 3798$$

Using $S' = 3798$ and the super-increasing vector $\mathbf{a}'$, user A easily solves for $\mathbf{x}$.

The Merkle–Hellman scheme is now considered broken [16], leaving the RSA scheme (as well as others discussed later) as the algorithms that are useful for implementing public key cryptosystems.

14.6 PRETTY GOOD PRIVACY

Pretty Good Privacy (PGP) is a security program that was created by Phil Zimmerman [17] and published in 1991 as free-of-charge shareware. It has since become the "de facto" standard for electronic mail (e-mail) and file encryption. PGP, widely used as version 2.6, remained essentially unchanged until PGP version 5.0 (which is compatible with version 2.6) became available. Table 14.9 illustrates the algorithms used in versions 2.6, 5.0, and later.

As listed in Table 14.9, PGP uses a variety of encryption algorithms, including both private-key- and public-key-based systems. A private-key algorithm (with a new session key generated at each session) is used for encryption of the message. The private-key algorithms offered by PGP are International Data Encryption Algorithm (IDEA), Triple-DES (Data Encryption Standard), and CAST (named after the inventors Carlisle Adams and Stafford Tavares [19]). A public-key algorithm is used for the encryption of each session key. The public-key algorithms offered by PGP are the RSA algorithm, described in Section 14.5.3, and the Diffie-Hellman algorithm.

Public-key algorithms are also used for the creation of digital signatures. PGP version 5.0 uses the Digital Signature Algorithm (DSA) specified in the NIST Digital Signature Standard (DSS). PGP version 2.6 uses the RSA algorithm for its digital signatures. If the available channel is insecure for key exchange, it is safest to use a public-key algorithm. If a secure channel is available, then private-key encryption is preferred, since it typically offers improved speed over public-key systems.

The technique for message encryption employed by PGP version 2.6 is illustrated in Figure 14.20. The plaintext is compressed with the ZIP algorithm prior to encryption. PGP uses the ZIP routine written by Jean-Loup Gailly, Mark Alder, and Richard B. Wales [18]. If the compressed text is shorter than the uncompressed text, the compressed text will be encrypted, otherwise the uncompressed text is encrypted.

TABLE 14.9 PGP 2.6 versus PGP 5.0 and Later

Function	PGP Version 2.6 Algorithm Used [17]	PGP Version 5.0 and Later Algorithm Used [18]
Encryption of message using private-key algorithm with private-session key	IDEA	Triple-DES, CAST, or IDEA
Encryption of private-session key with public-key algorithm	RSA	RSA or Diffie-Hellman (the Elgamal variation)
Digital Signature	RSA	RSA and NIST[1] Digital Signature Standard (DSS)[2]
Hash Function used for creating message digest for Digital Signatures	MD5	SHA-1

[1] National Institute of Standards and Technology, a division of the U.S. Department of Commerce.

[2] Digital Signature Standard selected by NIST.

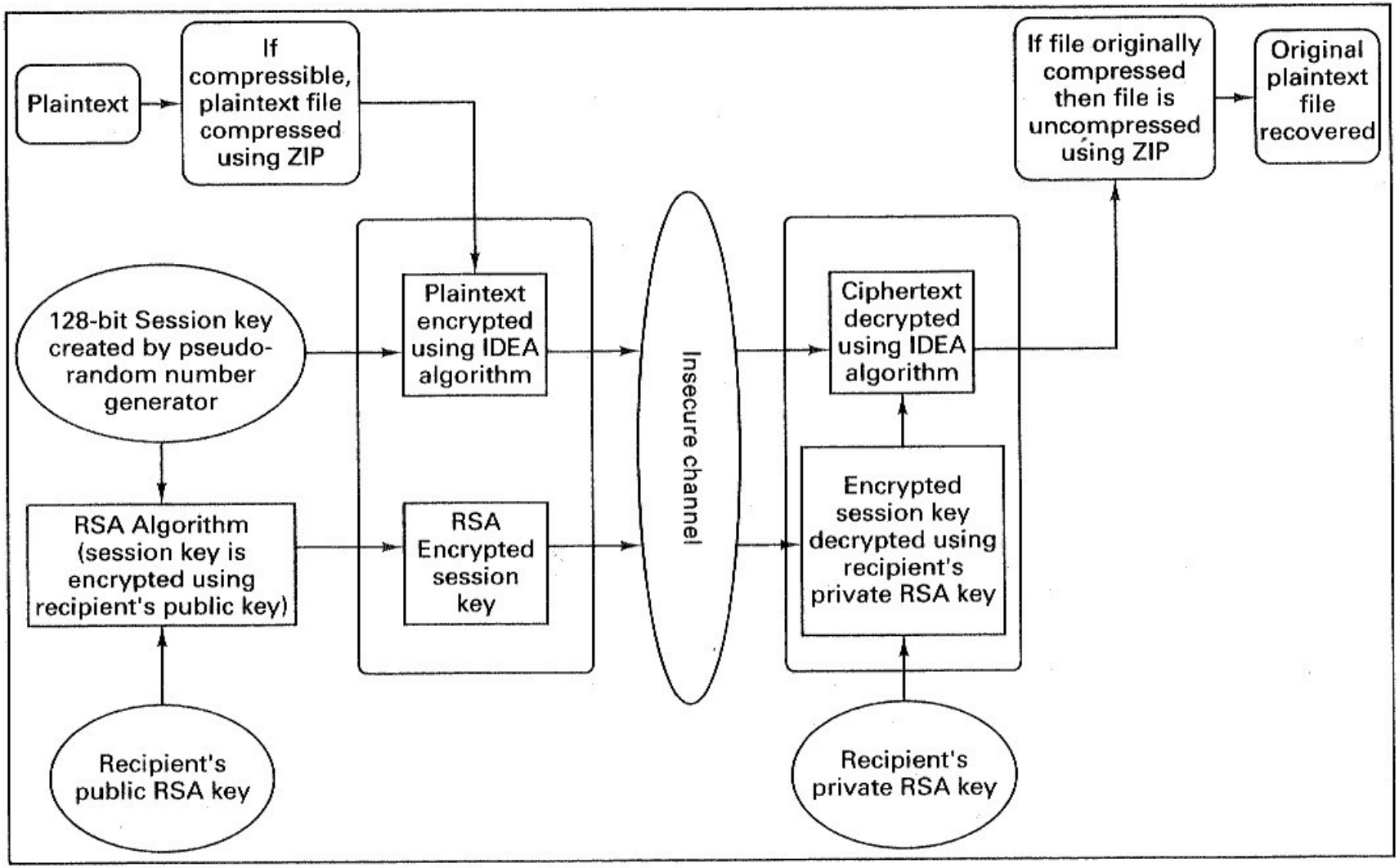

Figure 14.20 The PGP technique.

Small files (approximately 30 characters for ASCII files) will not benefit from compression. Additionally, PGP recognizes files previously compressed by popular compression routines, such as PKZIP, and will not attempt to compress them. Data compression removes redundant character strings in a file and produces a more uniform distribution of characters. Compression provides a shorter file to encrypt and decrypt (which reduces the time needed to encrypt, decrypt, and transmit a file), but compression is also advantageous because it can hinder some cryptanalytic attacks that exploit redundancy. If compression is performed on a file, it should occur *prior to* encryption (never afterwards). Why is that a good rule to follow? Because a *good* encryption algorithm yields ciphertext with a nearly statistically uniform distribution of characters; therefore, if a data compression algorithm came after such encryption, it should result in no compression at all. If any ciphertext can be compressed, then the encryption algorithm that formed that ciphertext was a poor algorithm. A compression algorithm should be *unable* to find redundant patterns in text that was encrypted by a good encryption algorithm.

As shown in Figure 14.20, PGP Version 2.6 begins file encryption by creating a 128-bit session key using a pseudo-random number generator. The compressed plaintext file is then encrypted with the IDEA private-key algorithm using this random session key. The random session key is then encrypted by the RSA public-key algorithm using the *recipient's public key*. The RSA-encrypted session key and the IDEA-encrypted file are sent to the recipient. When the recipient needs to read the file, the encrypted session key is first decrypted with RSA using the *recipient's private key*. The ciphertext file is then decrypted with IDEA using the decrypted session key. After uncompression, the recipient can read the plaintext file.

14.6.1 Triple-DES, CAST, and IDEA

As listed in Table 14.9, PGP offers three block ciphers for message encryption, Triple-DES, CAST, and IDEA. All three ciphers operate on 64-bit blocks of plaintext and ciphertext. Triple-DES has a key size of 168-bits while CAST and IDEA use key lengths of 128 bits.

14.6.1.1 Description of Triple-DES

The Data Encryption Standard (DES) described in Section 14.3.5 has been used since the late 1970s, but some have worried about its security because of its relatively small key size (56 bits). With Triple-DES, the message to be encrypted is run through the DES algorithm 3 times (the second DES operation is run in decrypt mode); each operation is performed with a different 56-bit key. As illustrated in Figure 14.21, this gives the effect of a 168-bit key length.

14.6.1.2 Description of CAST

CAST is a family of block ciphers developed by Adams and Tavares [19]. PGP version 5.0 uses a version of CAST known as CAST5, or CAST-128. This version has a block size of 64-bits and a key length of 128-bits. The CAST algorithm uses six *S*-boxes with an 8-bit input and a 32-bit output. By comparison, DES uses

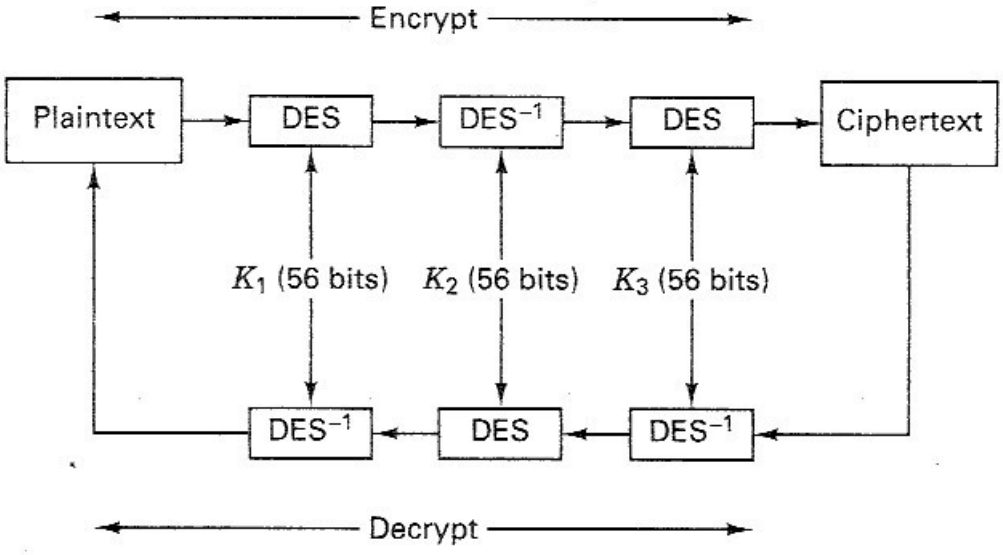

Figure 14.21 Encryption/decryption with triple-DES.

eight *S*-boxes with a 6-bit input and a 4-bit output. The *S*-boxes in Cast-128 were designed to provide highly nonlinear transformations, making this algorithm particularly resistant to cryptanalysis [11].

14.6.1.3 Description of IDEA

The International Data Encryption Algorithm (IDEA) is a block cipher designed by Xuejia Lai and James Massey [19]. It is a 64-bit iterative block cipher (involving eight iterations or rounds) with a 128-bit key. The security of IDEA relies on the use of three types of arithmetic operations on 16-bit words. The operations are addition modulo 2^{16}, multiplication modulo $2^{16} + 1$, and bit-wise exclusive-OR (XOR). The 128-bit key is used for the iterated encryption and decryption in a reordered fashion. As shown in Table 14.10, the original key K_0 is divided into eight 16-bit subkeys $Z_x^{(R)}$, where x is the subkey number of the round R. Six of these subkeys are used in round 1, and the remaining two are used in round 2. K_0 is then rotated 25 bits to the left yielding K_1, which is in turn divided into eight subkeys; the first 4 of these subkeys are used in round 2, and the last four in round 3. The process continues, as shown in Table 14.10, yielding a total of 52 subkeys.

The subkey schedule for each round is listed in Table 14.11 for both encryption and decryption rounds. Decryption is carried out in the same manner as encryption. The decryption subkeys are calculated from the encryption subkeys, as shown in Table 14.11, where it is seen that the decryption subkeys are either the additive or multiplicative inverses of the encryption subkeys.

The message is divided into 64-bit data blocks. These blocks are then divided into four 16-bit subblocks: M_1, M_2, M_3, and M_4. A sequence of such four subblocks becomes the input to the first round of IDEA algorithm. This data is manipulated for a total of eight rounds. Each round uses a different set of six subkeys as specified in Table 14.11. After a round, the second and third 16-bit data subblocks are