From Equation (6.10), we have $a_3 = \{03\}; a_2 = \{01\}; a_1 = \{01\}$; and $a_0 = \{02\}$. For the jth column of **State**, we have the polynomial $\text{col}_j(x) = s_{3,j}x^3 + s_{2,j}x^2 + s_{1,j}x + s_{0,j}$. Substituting into Equation (6.14), we can express $d(x) = a(x) \times \text{col}_j(x)$ as

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}$$

which is equivalent to Equation (6.3).

Multiplication by x

Consider the multiplication of a polynomial in the ring by x: $c(x) = x \oplus b(x)$. We have

$$\begin{aligned} c(x) = x \oplus b(x) &= [x \times (b_3x^3 + b_2x^2 + b_1x + \text{b}_0] \bmod (x^4 + 1) \\ &= (b_3x^4 + b_2x^3 + b_1x^2 + \text{b}_0x) \bmod (x^4 + 1) \\ &= b_2x^3 + b_1x^2 + \text{b}_0x + b_3 \end{aligned}$$

Thus, multiplication by x corresponds to a 1-byte circular left shift of the 4 bytes in the word representing the polynomial. If we represent the polynomial as a 4-byte column vector, then we have

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

CHAPTER 7

Block Cipher Operation

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Analyze the security of multiple encryption schemes.
- Explain the meet-in-the-middle attack.
- Compare and contrast ECB, CBC, CFB, OFB, and counter modes of operation.
- Present an overview of the XTS-AES mode of operation.

This chapter continues our discussion of symmetric ciphers. We begin with the topic of multiple encryption, looking in particular at the most widely used multiple-encryption scheme: triple DES.

The chapter next turns to the subject of **block cipher modes of operation**. We find that there are a number of different ways to apply a block cipher to plaintext, each with its own advantages and particular applications.

7.1 MULTIPLE ENCRYPTION AND TRIPLE DES

Because of its vulnerability to brute-force attack, DES, once the most widely used symmetric cipher, has been largely replaced by stronger encryption schemes. Two approaches have been taken. One approach is to design a completely new algorithm that is resistant to both cryptanalytic and brute-force attacks, of which AES is a prime example. Another alternative, which preserves the existing investment in software and equipment, is to use multiple encryption with DES and multiple keys. We begin by examining the simplest example of this second alternative. We then look at the widely accepted **triple DES (3DES)** algorithm.

Double DES

The simplest form of multiple encryption has two encryption stages and two keys (Figure 7.1a). Given a plaintext P and two encryption keys K_1 and K_2, ciphertext C is generated as

$$C = \mathrm{E}(K_2, \mathrm{E}(K_1, P))$$

Decryption requires that the keys be applied in reverse order:

$$P = \mathrm{D}(K_1, \mathrm{D}(K_2, C))$$

For DES, this scheme appears to involve a key length of $56 \times 2 = 112$ bits, and should result in a dramatic increase in cryptographic strength. But we need to examine the algorithm more closely.

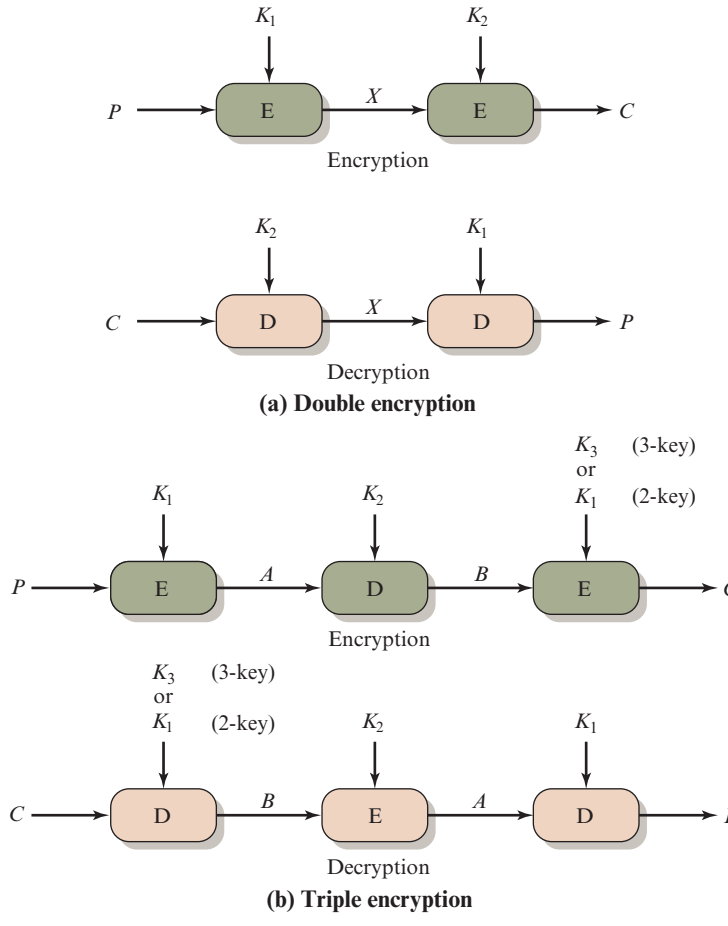


Figure 7.1 Multiple Encryption

Reduction to a Single Stage Suppose it were true for DES, for all 56-bit key values, that given any two keys K_1 and K_2, it would be possible to find a key K_3 such that

$$\mathrm{E}(K_2, \mathrm{E}(K_1, P)) = \mathrm{E}(K_3, P) \tag{7.1}$$

If this were the case, then double encryption, and indeed any number of stages of multiple encryption with DES, would be useless because the result would be equivalent to a single encryption with a single 56-bit key.

On the face of it, it does not appear that Equation (7.1) is likely to hold. Consider that encryption with DES is a mapping of 64-bit blocks to 64-bit blocks. In fact, the mapping can be viewed as a permutation. That is, if we consider all 2^{64} possible input blocks, DES encryption with a specific key will map each block into a unique 64-bit block. Otherwise, if, say, two given input blocks mapped to the same output block, then decryption to recover the original plaintext would be impossible.

With 2^{64} possible inputs, how many different mappings are there that generate a permutation of the input blocks? The value is easily seen to be

$$(2^{64})! = 10^{34738000000000000000} > (10^{10^{20}})$$

On the other hand, DES defines one mapping for each different key, for a total number of mappings:

$$2^{56} < 10^{17}$$

Therefore, it is reasonable to assume that if DES is used twice with different keys, it will produce one of the many mappings that are not defined by a single application of DES. Although there was much supporting evidence for this assumption, it was not until 1992 that the assumption was proven [CAMP92].

Meet-in-the-Middle Attack Thus, the use of double DES results in a mapping that is not equivalent to a single DES encryption. But there is a way to attack this scheme, one that does not depend on any particular property of DES but that will work against any block encryption cipher.

The algorithm, known as a **meet-in-the-middle attack**, was first described in [DIFF77]. It is based on the observation that, if we have

$$C = \mathrm{E}(K_2, \mathrm{E}(K_1, P))$$

then (see Figure 7.1a)

$$X = \mathrm{E}(K_1, P) = \mathrm{D}(K_2, C)$$

Given a known pair, (P, C), the attack proceeds as follows. First, encrypt P for all 2^{56} possible values of K_1. Store these results in a table and then sort the table by the values of X. Next, decrypt C using all 2^{56} possible values of K_2. As each decryption is produced, check the result against the table for a match. If a match occurs, then test the two resulting keys against a new known plaintext–ciphertext pair. If the two keys produce the correct ciphertext, accept them as the correct keys.

For any given plaintext P, there are 2^{64} possible ciphertext values that could be produced by double DES. Double DES uses, in effect, a 112-bit key, so that there are 2^{112} possible keys. Therefore, for a given plaintext P, the maximum number of different 112-bit keys that could produce a given ciphertext C is $2^{112}/2^{64} = 2^{48}$. Thus, the foregoing procedure can produce about 2^{48} false alarms on the first (P, C) pair. A similar argument indicates that with an additional 64 bits of known plaintext and ciphertext, the false alarm rate is reduced to $2^{48-64} = 2^{-16}$. Put another way, if the meet-in-the-middle attack is performed on two blocks of known plaintext–ciphertext, the probability that the correct keys are determined is $1 - 2^{-16}$. The result is that a known plaintext attack will succeed against double DES, which has a key size of 112 bits, with an effort on the order of 2^{56}, which is not much more than the 2^{55} required for single DES.

Triple DES with Two Keys

An obvious counter to the meet-in-the-middle attack is to use three stages of encryption with three different keys. Using DES as the underlying algorithm, this approach is commonly referred to as 3DES, or Triple Data Encryption

Algorithm (TDEA). As shown in Figure 7.1b, there are two versions of 3DES; one using two keys and one using three keys. NIST SP 800-67 (*Recommendation for the Triple Data Encryption Block Cipher,* January 2012) defines the two-key and three-key versions. We look first at the strength of the two-key version and then examine the three-key version.

Two-key triple encryption was first proposed by Tuchman [TUCH79]. The function follows an encrypt-decrypt-encrypt (EDE) sequence (Figure 7.1b):

$$C = \mathrm{E}(K_1, \mathrm{D}(K_2, \mathrm{E}(K_1, P)))$$
$$P = \mathrm{D}(K_1, \mathrm{E}(K_2, \mathrm{D}(K_1, C)))$$

There is no cryptographic significance to the use of decryption for the second stage. Its only advantage is that it allows users of 3DES to decrypt data encrypted by users of the older single DES:

$$C = \mathrm{E}(K_1, \mathrm{D}(K_1, \mathrm{E}(K_1, P))) = \mathrm{E}(K_1, P)$$
$$P = \mathrm{D}(K_1, \mathrm{E}(K_1, \mathrm{D}(K_1, C))) = \mathrm{D}(K_1, C)$$

3DES with two keys is a relatively popular alternative to DES and has been adopted for use in the key management standards ANSI X9.17 and ISO 8732.[1]

Currently, there are no practical cryptanalytic attacks on 3DES. Coppersmith [COPP94] notes that the cost of a brute-force key search on 3DES is on the order of $2^{112} \approx (5 \times 10^{33})$ and estimates that the cost of differential cryptanalysis suffers an exponential growth, compared to single DES, exceeding 10^{52}.

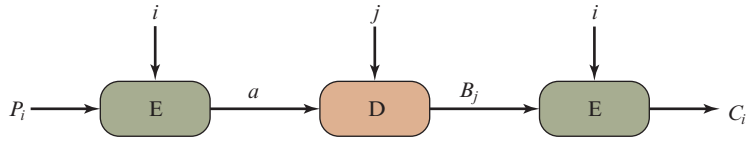
It is worth looking at several proposed attacks on 3DES that, although not practical, give a flavor for the types of attacks that have been considered and that could form the basis for more successful future attacks.

The first serious proposal came from Merkle and Hellman [MERK81]. Their plan involves finding plaintext values that produce a first intermediate value of $A = 0$ (Figure 7.1b) and then using the meet-in-the-middle attack to determine the two keys. The level of effort is 2^{56}, but the technique requires 2^{56} chosen plaintext–ciphertext pairs, which is a number unlikely to be provided by the holder of the keys.

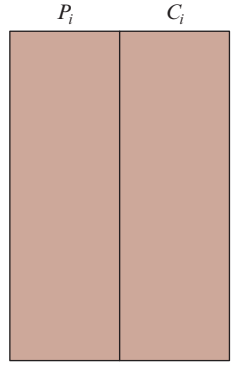
A known-plaintext attack is outlined in [VANO90]. This method is an improvement over the chosen-plaintext approach but requires more effort. The attack is based on the observation that if we know A and C (Figure 7.1b), then the problem reduces to that of an attack on double DES. Of course, the attacker does not know A, even if P and C are known, as long as the two keys are unknown. However, the attacker can choose a potential value of A and then try to find a known (P, C) pair that produces A. The attack proceeds as follows.

1. Obtain n (P, C) pairs. This is the known plaintext. Place these in a table (Table 1) sorted on the values of P (Figure 7.2b).

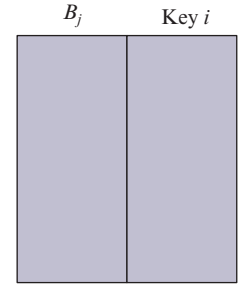
[1]American National Standards Institute (ANSI): *Financial Institution Key Management (Wholesale)*. From its title, X9.17 appears to be a somewhat obscure standard. Yet a number of techniques specified in this standard have been adopted for use in other standards and applications, as we shall see throughout this book.



(a) Two-key triple encryption with candidate pair of keys



(b) Table of *n* known plaintext–ciphertext pairs, sorted on *P*



(c) Table of intermediate values and candidate keys

Figure 7.2 Known-Plaintext Attack on Triple DES

2. Pick an arbitrary value a for A, and create a second table (Figure 7.2c) with entries defined in the following fashion. For each of the 2^{56} possible keys $K_1 = i$, calculate the plaintext value P_i such that

$$P_i = \mathrm{D}(i, a)$$

For each P_i that matches an entry in Table 1, create an entry in Table 2 consisting of the K_1 value and the value of B that is produced for the (P, C) pair from Table 1, assuming that value of K_1:

$$B = \mathrm{D}(i, C)$$

At the end of this step, sort Table 2 on the values of B.

3. We now have a number of candidate values of K_1 in Table 2 and are in a position to search for a value of K_2. For each of the 2^{56} possible keys $K_2 = j$, calculate the second intermediate value for our chosen value of a:

$$B_j = \mathrm{D}(j, a)$$

At each step, look up B_j in Table 2. If there is a match, then the corresponding key i from Table 2 plus this value of j are candidate values for the unknown keys (K_1, K_2). Why? Because we have found a pair of keys (i, j) that produce a known (P, C) pair (Figure 7.2a).

4. Test each candidate pair of keys (i, j) on a few other plaintext–ciphertext pairs. If a pair of keys produces the desired ciphertext, the task is complete. If no pair succeeds, repeat from step 1 with a new value of a.

For a given known (P, C), the probability of selecting the unique value of a that leads to success is $1/2^{64}$. Thus, given n (P, C) pairs, the probability of success for a single selected value of a is $n/2^{64}$. A basic result from probability theory is that the expected number of draws required to draw one red ball out of a bin containing n red balls and $N - n$ green balls is $(N + 1)/(n + 1)$ if the balls are not replaced. So the expected number of values of a that must be tried is, for large n,

$$\frac{2^{64} + 1}{n + 1} \approx \frac{2^{64}}{n}$$

Thus, the expected running time of the attack is on the order of

$$(2^{56})\frac{2^{64}}{n} = 2^{120 - \log_2 n}$$

Triple DES with Three Keys

Although the attacks just described appear impractical, anyone using two-key 3DES may feel some concern. Thus, many researchers now feel that three-key 3DES is the preferred alternative (e.g., [KALI96a]). In SP 800-57, Part 1 (*Recommendation for Key Management—Part 1: General,* July 2012) NIST recommends that 2-key 3DES be retired as soon as practical and replaced with 3-key 3DES.

Three-key 3DES is defined as

$$C = \mathrm{E}(K_3, \mathrm{D}(K_2, \mathrm{E}(K_1, P)))$$

Backward compatibility with DES is provided by putting $K_3 = K_2$ or $K_1 = K_2$. One might expect that 3TDEA would provide $56 \cdot 3 = 168$ bits of strength. However, there is an attack on 3TDEA that reduces the strength to the work that would be involved in exhausting a 112-bit key [MERK81].

A number of Internet-based applications have adopted three-key 3DES, including PGP and S/MIME, both discussed in Chapter 21.

7.2 ELECTRONIC CODEBOOK

A block cipher takes a fixed-length block of text of length b bits and a key as input and produces a b-bit block of ciphertext. If the amount of plaintext to be encrypted is greater than b bits, then the block cipher can still be used by breaking the plaintext up into b-bit blocks. When multiple blocks of plaintext are encrypted using the same key, a number of security issues arise. To apply a block cipher in a variety of applications, five *modes of operation* have been defined by NIST (SP 800-38A). In essence, a mode of operation is a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application, such as applying a block cipher to a sequence of data blocks or a data stream. The five modes are intended to cover a wide variety of applications of encryption for which a block cipher could be used. These modes are intended for use with any symmetric block cipher, including triple DES and AES. The modes are summarized in Table 7.1 and described in this and the following sections.

Table 7.1 Block Cipher Modes of Operation

Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next block of plaintext and the preceding block of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed s bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding encryption output, and full blocks are used.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

The simplest mode is the electronic codebook (ECB) mode, in which plaintext is handled one block at a time and each block of plaintext is encrypted using the same key (Figure 7.3). The term *codebook* is used because, for a given key, there is a unique ciphertext for every b-bit block of plaintext. Therefore, we can imagine a gigantic codebook in which there is an entry for every possible b-bit plaintext pattern showing its corresponding ciphertext.

For a message longer than b bits, the procedure is simply to break the message into b-bit blocks, padding the last block if necessary. Decryption is performed one block at a time, always using the same key. In Figure 7.3, the plaintext (padded as necessary) consists of a sequence of b-bit blocks, $P_1, P_2, \ldots, P_N$; the corresponding sequence of ciphertext blocks is $C_1, C_2, \ldots, C_N$. We can define ECB mode as follows.

ECB	$C_j = E(K, P_j) \qquad j = 1, \ldots, N$	$P_j = D(K, C_j) \qquad j = 1, \ldots, N$

The ECB mode should be used only to secure messages shorter than a single block of underlying cipher (i.e., 64 bits for 3DES and 128 bits for AES), such as to encrypt a secret key. Because in most of the cases messages are longer than the encryption block mode, this mode has a minimum practical value.

The most significant characteristic of ECB is that if the same b-bit block of plaintext appears more than once in the message, it always produces the same ciphertext.

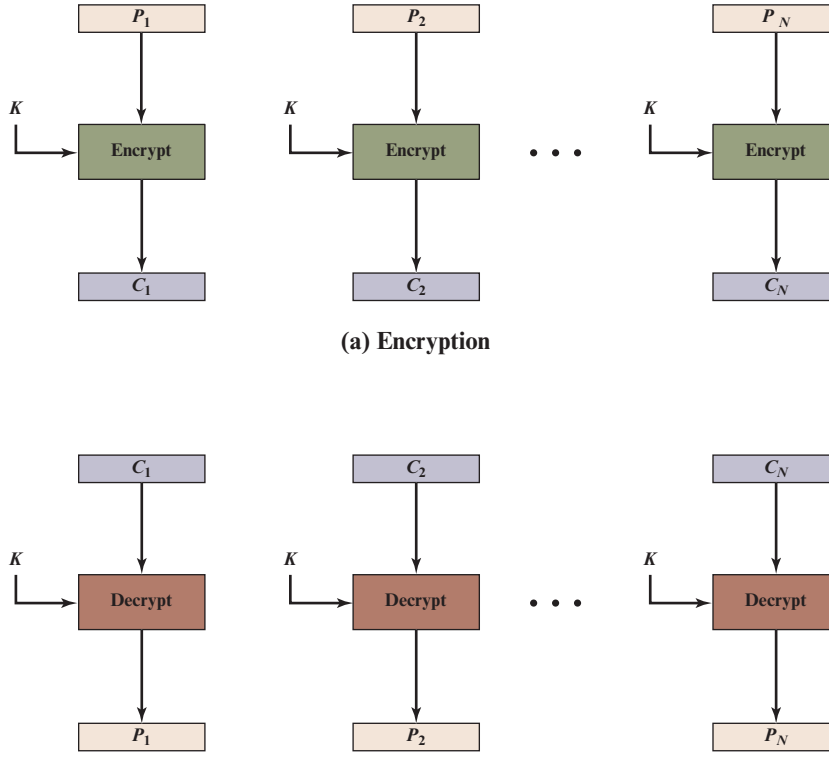


Figure 7.3 Electronic Codebook (ECB) Mode

For lengthy messages, the ECB mode may not be secure. If the message is highly structured, it may be possible for a cryptanalyst to exploit these regularities. For example, if it is known that the message always starts out with certain predefined fields, then the cryptanalyst may have a number of known plaintext–ciphertext pairs to work with. If the message has repetitive elements with a period of repetition a multiple of b bits, then these elements can be identified by the analyst. This may help in the analysis or may provide an opportunity for substituting or rearranging blocks.

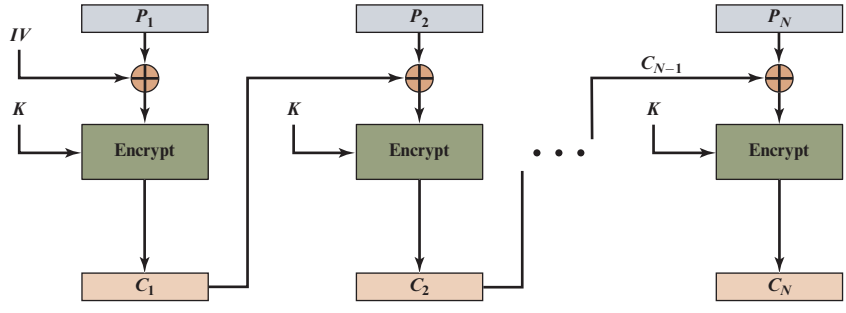
We now turn to more complex modes of operation. [KNUD00] lists the following criteria and properties for evaluating and constructing block cipher modes of operation that are superior to ECB:

- **Overhead:** The additional operations for the encryption and decryption operation when compared to encrypting and decrypting in the ECB mode.
- **Error recovery:** The property that an error in the ith ciphertext block is inherited by only a few plaintext blocks after which the mode resynchronizes.
- **Error propagation:** The property that an error in the ith ciphertext block is inherited by the ith and all subsequent plaintext blocks. What is meant here is a bit error that occurs in the transmission of a ciphertext block, not a computational error in the encryption of a plaintext block.

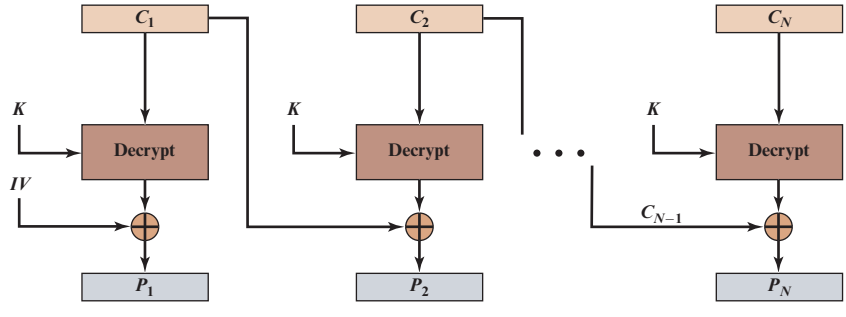
- **Diffusion:** How the plaintext statistics are reflected in the ciphertext. Low entropy plaintext blocks should not be reflected in the ciphertext blocks. Roughly, low entropy equates to predictability or lack of randomness (see Appendix B).
- **Security:** Whether or not the ciphertext blocks leak information about the plaintext blocks.

7.3 CIPHER BLOCK CHAINING MODE

To overcome the security deficiencies of ECB, we would like a technique in which the same plaintext block, if repeated, produces different ciphertext blocks. A simple way to satisfy this requirement is the cipher block chaining (CBC) mode (Figure 7.4). In this scheme, the input to the encryption algorithm is the XOR of the current plaintext block and the preceding ciphertext block; the same key is used for each block. In effect, we have chained together the processing of the sequence of plaintext blocks. The input to the encryption function for each plaintext block bears no fixed relationship to the plaintext block. Therefore, repeating patterns of b bits are not exposed. As with the ECB mode, the CBC mode requires that the last block be padded to a full b bits if it is a partial block.



(a) Encryption



(b) Decryption

Figure 7.4 Cipher Block Chaining (CBC) Mode

For decryption, each cipher block is passed through the decryption algorithm. The result is XORed with the preceding ciphertext block to produce the plaintext block. To see that this works, we can write

$$C_j = \mathrm{E}(K, [C_{j-1} \oplus P_j])$$

Then

$$\mathrm{D}(K, C_j) = \mathrm{D}(K, E(K, [C_{j-1} \oplus P_j]))$$
$$\mathrm{D}(K, C_j) = C_{j-1} \oplus P_j$$
$$C_{j-1} \oplus \mathrm{D}(K, C_j) = C_{j-1} \oplus C_{j-1} \oplus P_j = P_j$$

To produce the first block of ciphertext, an initialization vector (IV) is XORed with the first block of plaintext. On decryption, the IV is XORed with the output of the decryption algorithm to recover the first block of plaintext. The IV is a data block that is the same size as the cipher block. We can define CBC mode as

CBC	$C_1 = \mathrm{E}(K, [P_1 \oplus \mathrm{IV}])$ $C_j = \mathrm{E}(K, [P_j \oplus C_{j-1}])j = 2, \ldots, N$	$P_1 = \mathrm{D}(K, C_1) \oplus \mathrm{IV}$ $P_j = \mathrm{D}(K, C_j) \oplus C_{j-1}\, j = 2, \ldots, N$

The IV must be known to both the sender and receiver but be unpredictable by a third party. In particular, for any given plaintext, it must not be possible to predict the IV that will be associated to the plaintext in advance of the generation of the IV. For maximum security, the IV should be protected against unauthorized changes. This could be done by sending the IV using ECB encryption. One reason for protecting the IV is as follows: If an opponent is able to fool the receiver into using a different value for IV, then the opponent is able to invert selected bits in the first block of plaintext. To see this, consider

$$C_1 = \mathrm{E}(K, [\mathrm{IV} \oplus P_1])$$
$$P_1 = \mathrm{IV} \oplus \mathrm{D}(K, C_1)$$

Now use the notation that $X[i]$ denotes the ith bit of the b-bit quantity X. Then

$$P_1[i] = \mathrm{IV}[i] \oplus \mathrm{D}(K, C_1)[i]$$

Then, using the properties of XOR, we can state

$$P_1[i]' = \mathrm{IV}[i]' \oplus \mathrm{D}(K, C_1)[i]$$

where the prime notation denotes bit complementation. This means that if an opponent can predictably change bits in IV, the corresponding bits of the received value of P_1 can be changed.

For other possible attacks based on prior knowledge of IV, see [VOYD83].

So long as it is unpredictable, the specific choice of IV is unimportant. SP 800-38A recommends two possible methods: The first method is to apply the encryption function, under the same key that is used for the encryption of the plaintext, to a **nonce**.[2] The nonce must be a data block that is unique to each execution of

[2]NIST SP 800-90 (*Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) defines nonce as follows: A time-varying value that has at most a negligible chance of repeating, for example, a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.

the encryption operation. For example, the nonce may be a counter, a timestamp, or a message number. The second method is to generate a random data block using a random number generator.

In conclusion, because of the chaining mechanism of CBC, it is an appropriate mode for encrypting messages of length greater than b bits.

In addition to its use to achieve confidentiality, the CBC mode can be used for authentication. This use is described in Chapter 12.

7.4 CIPHER FEEDBACK MODE

For AES, DES, or any block cipher, encryption is performed on a block of b bits. In the case of DES, $b = 64$ and in the case of AES, $b = 128$. However, it is possible to convert a block cipher into a stream cipher, using one of the three modes to be discussed in this and the next two sections: cipher feedback (CFB) mode, output feedback (OFB) mode, and counter (CTR) mode. A stream cipher eliminates the need to pad a message to be an integral number of blocks. It also can operate in real time. Thus, if a character stream is being transmitted, each character can be encrypted and transmitted immediately using a character-oriented stream cipher.

One desirable property of a stream cipher is that the ciphertext be of the same length as the plaintext. Thus, if 8-bit characters are being transmitted, each character should be encrypted to produce a ciphertext output of 8 bits. If more than 8 bits are produced, transmission capacity is wasted.

Figure 7.5 depicts the CFB scheme. In the figure, it is assumed that the unit of transmission is s bits; a common value is $s = 8$. As with CBC, the units of plaintext are chained together, so that the ciphertext of any plaintext unit is a function of all the preceding plaintext. In this case, rather than blocks of b bits, the plaintext is divided into *segments* of s bits.

First, consider encryption. The input to the encryption function is a b-bit shift register that is initially set to some initialization vector (IV). The leftmost (most significant) s bits of the output of the encryption function are XORed with the first segment of plaintext P_1 to produce the first unit of ciphertext C_1, which is then transmitted. In addition, the contents of the shift register are shifted left by s bits, and C_1 is placed in the rightmost (least significant) s bits of the shift register. This process continues until all plaintext units have been encrypted.

For decryption, the same scheme is used, except that the received ciphertext unit is XORed with the output of the encryption function to produce the plaintext unit. Note that it is the *encryption* function that is used, not the decryption function. This is easily explained. Let $\text{MSB}_s(X)$ be defined as the most significant s bits of X. Then

$$C_1 = P_1 \oplus \text{MSB}_s[\text{E}(K, \text{IV})]$$

Therefore, by rearranging terms:

$$P_1 = C_1 \oplus \text{MSB}_s[\text{E}(K, \text{IV})]$$

The same reasoning holds for subsequent steps in the process.

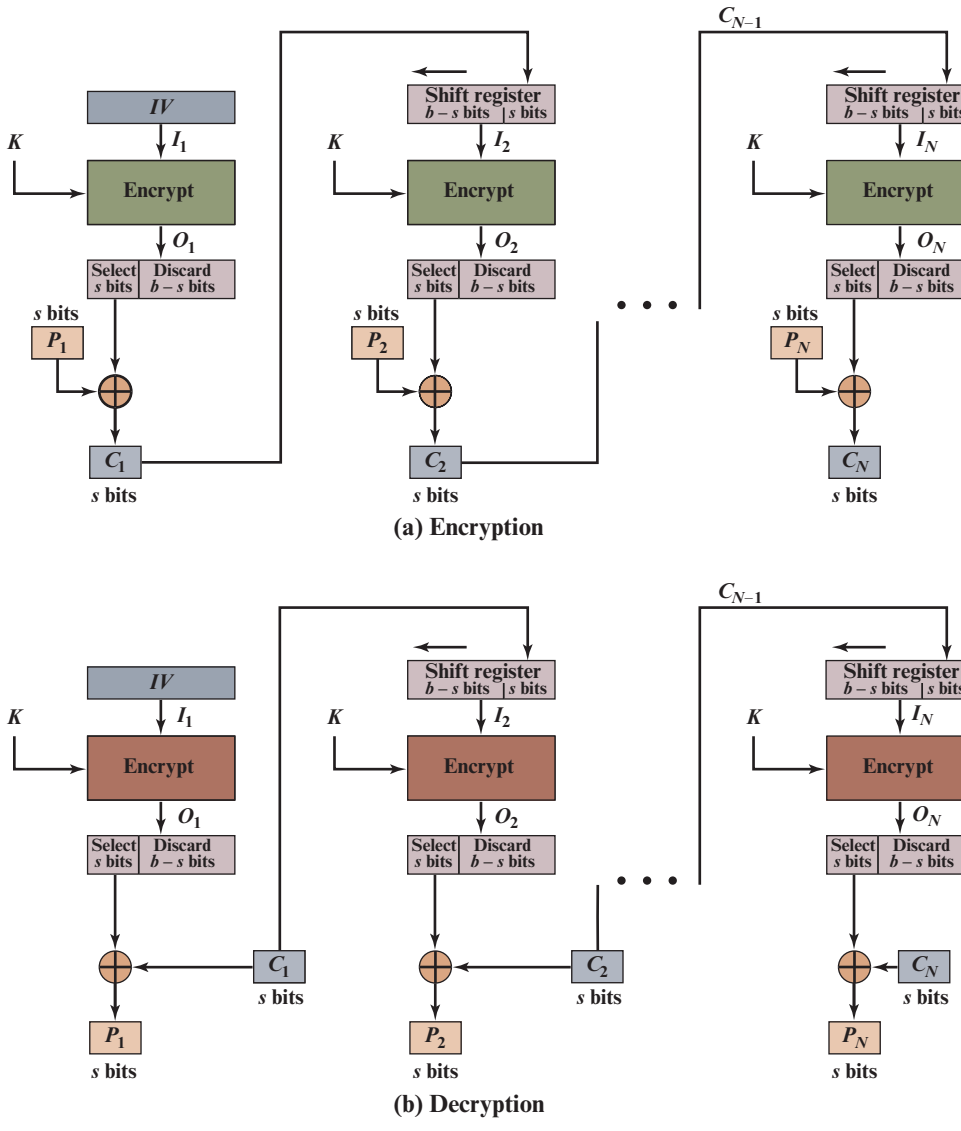


Figure 7.5 *s*-bit Cipher Feedback (CFB) Mode

We can define CFB mode as follows.

CFB	$I_1 = IV$ $I_j = \text{LSB}_{b-s}(I_{j-1}) \| C_{j-1}$ $\quad j = 2, \ldots, N$ $O_j = \text{E}(K, I_j)$ $\quad j = 1, \ldots, N$ $C_j = P_j \oplus \text{MSB}_s(O_j)$ $\quad j = 1, \ldots, N$	$I_1 = IV$ $I_j = \text{LSB}_{b-s}(I_{j-1}) \| C_{j-1}$ $\quad j = 2, \ldots, N$ $O_j = \text{E}(K, I_j)$ $\quad j = 1, \ldots, N$ $P_j = C_j \oplus \text{MSB}_s(O_j)$ $\quad j = 1, \ldots, N$

Although CFB can be viewed as a stream cipher, it does not conform to the typical construction of a stream cipher. In a typical stream cipher, the cipher takes

as input some initial value and a key and generates a stream of bits, which is then XORed with the plaintext bits (see Figure 4.1). In the case of CFB, the stream of bits that is XORed with the plaintext also depends on the plaintext.

In CFB encryption, like CBC encryption, the input block to each forward cipher function (except the first) depends on the result of the previous forward cipher function; therefore, multiple forward cipher operations cannot be performed in parallel. In CFB decryption, the required forward cipher operations can be performed in parallel if the input blocks are first constructed (in series) from the IV and the ciphertext.

7.5 OUTPUT FEEDBACK MODE

The **output feedback (OFB) mode** is similar in structure to that of CFB. For OFB, the output of the encryption function is fed back to become the input for encrypting the next block of plaintext (Figure 7.6). In CFB, the output of the XOR unit is fed back to become input for encrypting the next block. The other difference is that the OFB mode operates on full blocks of plaintext and ciphertext, whereas CFB operates on an s-bit subset. OFB encryption can be expressed as

$$C_j = P_j \oplus \mathrm{E}(K, O_{j-1})$$

where

$$O_{j-1} = \mathrm{E}(K, O_{j-2})$$

Some thought should convince you that we can rewrite the encryption expression as:

$$C_j = P_j \oplus \mathrm{E}(K, [C_{j-1} \oplus P_{j-1}])$$

By rearranging terms, we can demonstrate that decryption works.

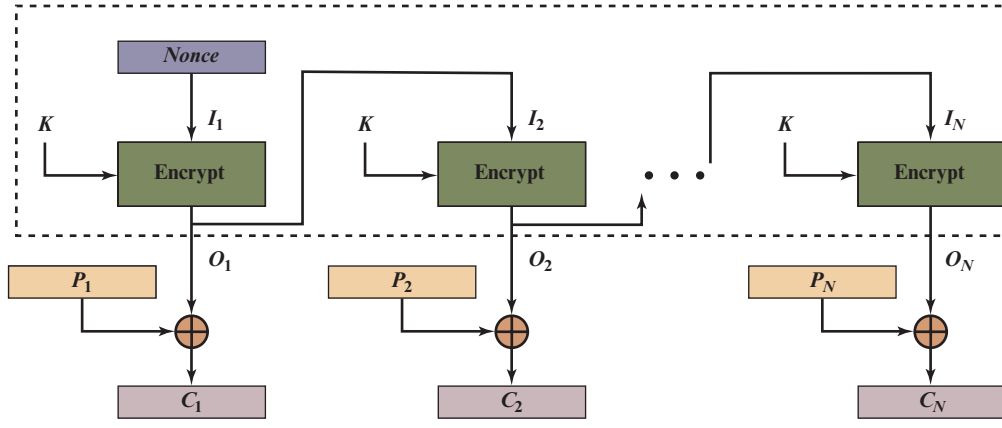
$$P_j = C_j \oplus \mathrm{E}(K, [C_{j-1} \oplus P_{j-1}])$$

We can define OFB mode as follows.

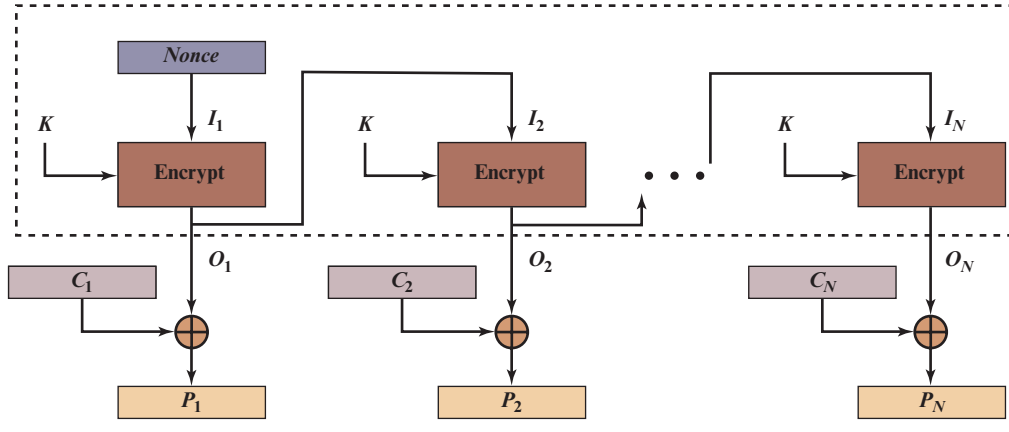
OFB	$I_1 = Nonce$ $I_j = O_{j-1} \quad j = 2, \ldots, N$ $O_j = \mathrm{E}(K, I_j) \quad j = 1, \ldots, N$ $C_j = P_j \oplus O_j \quad j = 1, \ldots, N-1$ $C_N^* = P_N^* \oplus \mathrm{MSB}_u(O_N)$	$I_1 = Nonce$ $I_j = O_{j-1} \quad j = 2, \ldots, N$ $O_j = \mathrm{E}(K, I_j) \quad j = 1, \ldots, N$ $P_j = C_j \oplus O_j \quad j = 1, \ldots, N-1$ $P_N^* = C_N^* \oplus \mathrm{MSB}_u(O_N)$

Let the size of a block be b. If the last block of plaintext contains u bits (indicated by *), with $u < b$, the most significant u bits of the last output block O_N are used for the XOR operation; the remaining $b - u$ bits of the last output block are discarded.

As with CBC and CFB, the OFB mode requires an initialization vector. In the case of OFB, the IV must be a nonce; that is, the IV must be unique to each execution of the encryption operation. The reason for this is that the sequence of



(a) Encryption



(b) Decryption

Figure 7.6 Output Feedback (OFB) Mode

encryption output blocks, O_i, depends only on the key and the IV and does not depend on the plaintext. Therefore, for a given key and IV, the stream of output bits used to XOR with the stream of plaintext bits is fixed. If two different messages had an identical block of plaintext in the identical position, then an attacker would be able to determine that portion of the O_i stream.

One advantage of the OFB method is that bit errors in transmission do not propagate. For example, if a bit error occurs in C_1, only the recovered value of P_1 is affected; subsequent plaintext units are not corrupted. With CFB, C_1 also serves as input to the shift register and therefore causes additional corruption downstream.

The disadvantage of OFB is that it is more vulnerable to a message stream modification attack than is CFB. Consider that complementing a bit in the ciphertext complements the corresponding bit in the recovered plaintext. Thus, controlled

changes to the recovered plaintext can be made. This may make it possible for an opponent, by making the necessary changes to the checksum portion of the message as well as to the data portion, to alter the ciphertext in such a way that it is not detected by an error-correcting code. For a further discussion, see [VOYD83].

OFB has the structure of a typical stream cipher, because the cipher generates a stream of bits as a function of an initial value and a key, and that stream of bits is XORed with the plaintext bits (see Figure 4.1). The generated stream that is XORed with the plaintext is itself independent of the plaintext; this is highlighted by dashed boxes in Figure 7.6. One distinction from the stream ciphers we discuss in Chapter 8 is that OFB encrypts plaintext a full block at a time, where typically a block is 64 or 128 bits. Many stream ciphers encrypt one byte at a time.

7.6 COUNTER MODE

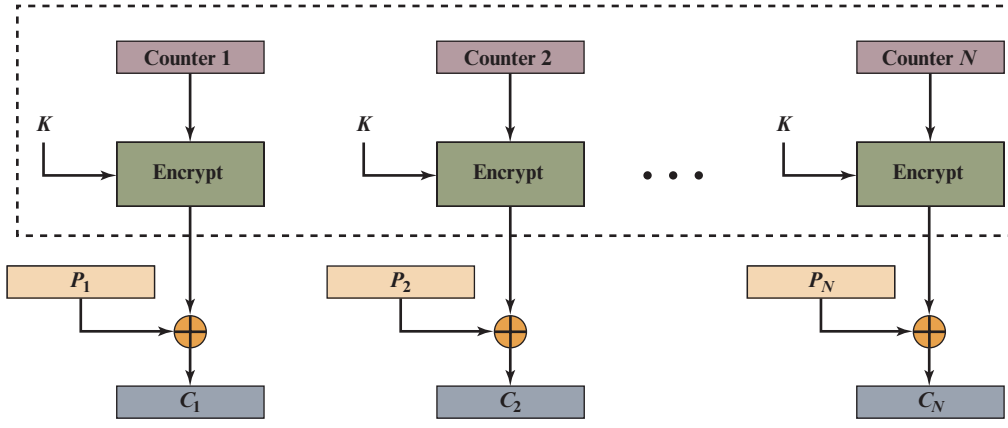
Although interest in the **counter (CTR) mode** has increased recently with applications to ATM (asynchronous transfer mode) network security and IPsec (IP security), this mode was proposed in 1979 (e.g., [DIFF79]).

Figure 7.7 depicts the CTR mode. A counter equal to the plaintext block size is used. The only requirement stated in SP 800-38A is that the counter value must be different for each plaintext block that is encrypted. Typically, the counter is initialized to some value and then incremented by 1 for each subsequent block (modulo 2^b, where b is the block size). For encryption, the counter is encrypted and then XORed with the plaintext block to produce the ciphertext block; there is no chaining. For decryption, the same sequence of counter values is used, with each encrypted counter XORed with a ciphertext block to recover the corresponding plaintext block. Thus, the initial counter value must be made available for decryption. Given a sequence of counters $T_1, T_2, \ldots, T_N$, we can define CTR mode as follows.

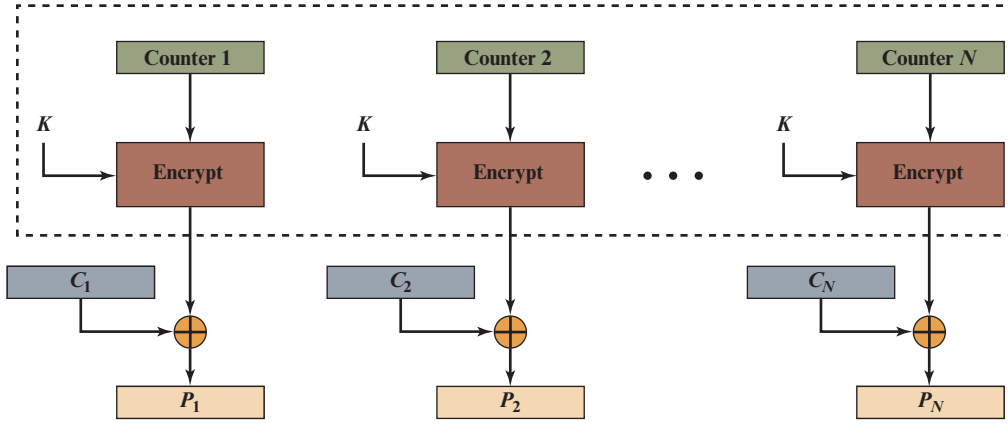
CTR	$C_j = P_j \oplus \mathrm{E}(K, T_j) \quad j = 1, \ldots, N-1$ $C_N^* = P_N^* \oplus \mathrm{MSB}_u[\mathrm{E}(K, T_N)]$	$P_j = C_j \oplus \mathrm{E}(K, T_j) \quad j = 1, \ldots, N-1$ $P_N^* = C_N^* \oplus \mathrm{MSB}_u[\mathrm{E}(K, T_N)]$

For the last plaintext block, which may be a partial block of u bits, the most significant u bits of the last output block are used for the XOR operation; the remaining $b - u$ bits are discarded. Unlike the ECB, CBC, and CFB modes, we do not need to use padding because of the structure of the CTR mode.

As with the OFB mode, the initial counter value must be a nonce; that is, T_1 must be different for all of the messages encrypted using the same key. Further, all T_i values across all messages must be unique. If, contrary to this requirement, a counter value is used multiple times, then the confidentiality of all of the plaintext blocks corresponding to that counter value may be compromised. In particular, if any plaintext block that is encrypted using a given counter value is known, then the output of the encryption function can be determined easily from the associated ciphertext block. This output allows any other plaintext blocks that are encrypted using the same counter value to be easily recovered from their associated ciphertext blocks.



(a) Encryption



(b) Decryption

Figure 7.7 Counter (CTR) Mode

One way to ensure the uniqueness of counter values is to continue to increment the counter value by 1 across messages. That is, the first counter value of the each message is one more than the last counter value of the preceding message.

[LIPM00] lists the following advantages of CTR mode.

- **Hardware efficiency:** Unlike the three chaining modes, encryption (or decryption) in CTR mode can be done in parallel on multiple blocks of plaintext or ciphertext. For the chaining modes, the algorithm must complete the computation on one block before beginning on the next block. This limits the maximum throughput of the algorithm to the reciprocal of the time for one execution of block encryption or decryption. In CTR mode, the throughput is only limited by the amount of parallelism that is achieved.

- **Software efficiency:** Similarly, because of the opportunities for parallel execution in CTR mode, processors that support parallel features, such as aggressive pipelining, multiple instruction dispatch per clock cycle, a large number of registers, and SIMD instructions, can be effectively utilized.
- **Preprocessing:** The execution of the underlying encryption algorithm does not depend on input of the plaintext or ciphertext. Therefore, if sufficient memory is available and security is maintained, preprocessing can be used to prepare the output of the encryption boxes that feed into the XOR functions, as in Figure 7.7. When the plaintext or ciphertext input is presented, then the only computation is a series of XORs. Such a strategy greatly enhances throughput.
- **Random access:** The ith block of plaintext or ciphertext can be processed in random-access fashion. With the chaining modes, block C_i cannot be computed until the $i - 1$ prior blocks are computed. There may be applications in which a ciphertext is stored and it is desired to decrypt just one block; for such applications, the random access feature is attractive.
- **Provable security:** It can be shown that CTR is at least as secure as the other modes discussed in this chapter.
- **Simplicity:** Unlike ECB and CBC modes, CTR mode requires only the implementation of the encryption algorithm and not the decryption algorithm. This matters most when the decryption algorithm differs substantially from the encryption algorithm, as it does for AES. In addition, the decryption key scheduling need not be implemented.

Note that, with the exception of ECB, all of the NIST-approved block cipher modes of operation involve feedback. This is clearly seen in Figure 7.8. To highlight the feedback mechanism, it is useful to think of the encryption function as taking input from an input register whose length equals the encryption block length and with output stored in an output register. The input register is updated one block at a time by the feedback mechanism. After each update, the encryption algorithm is executed, producing a result in the output register. Meanwhile, a block of plaintext is accessed. Note that both OFB and CTR produce output that is independent of both the plaintext and the ciphertext. Thus, they are natural candidates for stream ciphers that encrypt plaintext by XOR one full block at a time.

7.7 XTS-AES MODE FOR BLOCK-ORIENTED STORAGE DEVICES

In 2010, NIST approved an additional block cipher mode of operation, XTS-AES. This mode is also an IEEE standard, IEEE Std 1619-2007, which was developed by the IEEE Security in Storage Working Group (P1619). The standard describes a method of encryption for data stored in sector-based devices where the threat model includes possible access to stored data by the adversary. The standard has received widespread industry support.

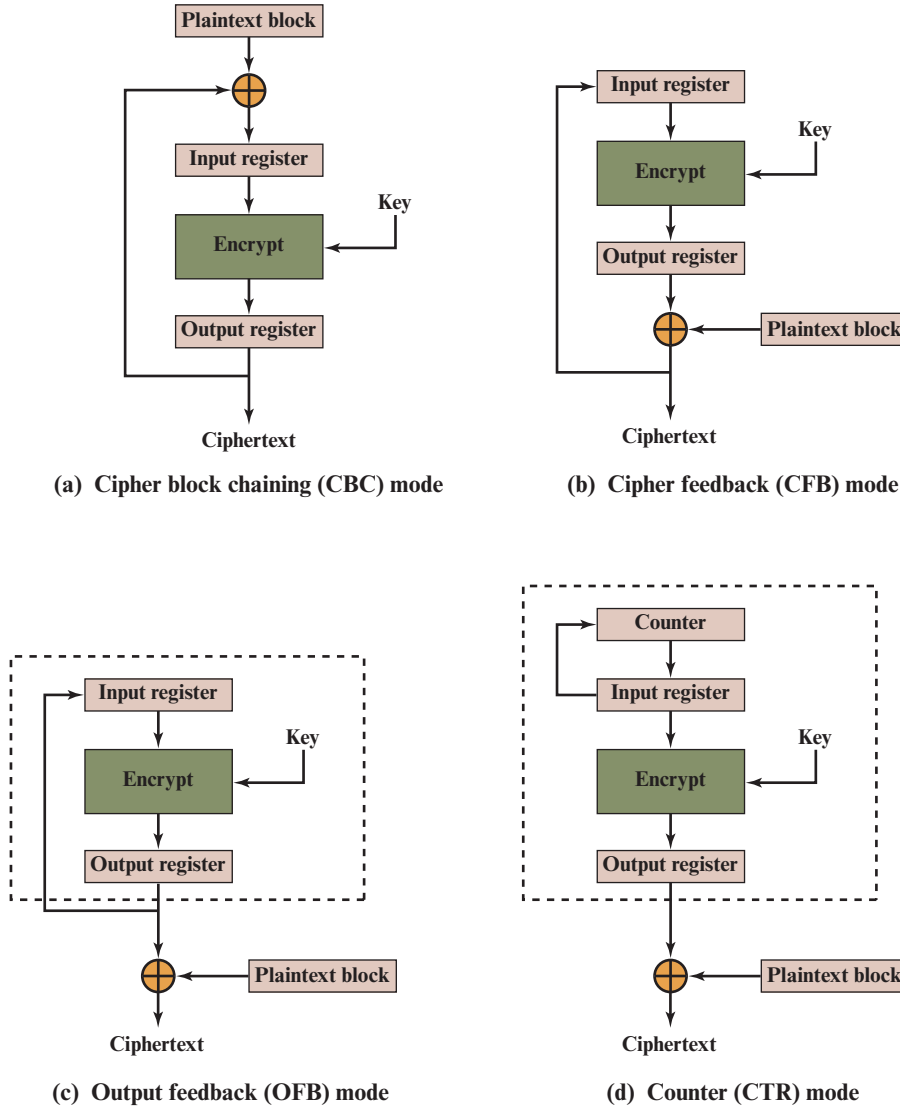
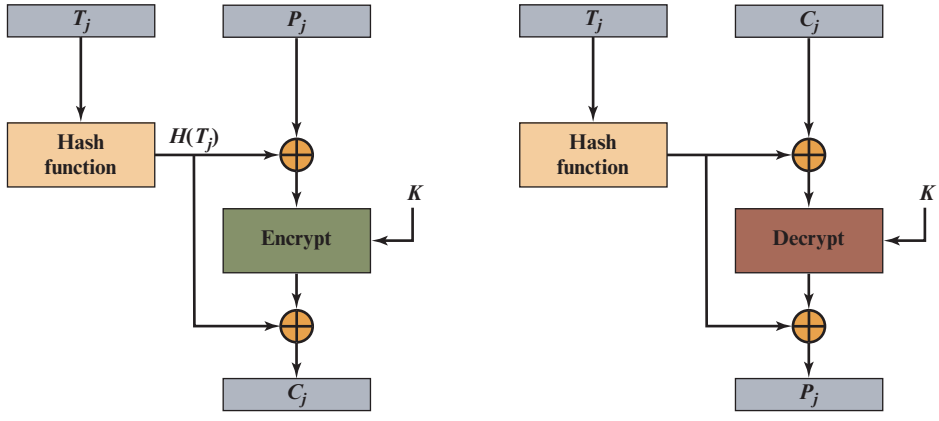


Figure 7.8 Feedback Characteristic of Modes of Operation

Tweakable Block Ciphers

The XTS-AES mode is based on the concept of a **tweakable block cipher**, introduced in [LISK02]. The form of this concept used in XTS-AES was first described in [ROGA04a].

Before examining XTS-AES, let us consider the general structure of a tweakable block cipher. A tweakable block cipher is one that has three inputs: a plaintext P, a symmetric key K, and a tweak T; and produces a ciphertext output C. We can write this as $C = \mathrm{E}(K, T, P)$. The tweak need not be kept secret. Whereas the purpose of the key is to provide security, the purpose of the tweak is to provide



(a) Encryption **(b) Decryption**

Figure 7.9 Tweakable Block Cipher

variability. That is, the use of different tweaks with the same plaintext and same key produces different outputs. The basic structure of several tweakable block ciphers that have been implemented is shown in Figure 7.9. Encryption can be expressed as:

$$C = \mathrm{H}(T) \oplus \mathrm{E}(K, \mathrm{H}(T) \oplus P)$$

where H is a hash function. For decryption, the same structure is used with the plaintext as input and decryption as the function instead of encryption. To see that this works, we can write

$$\mathrm{H}(T) \oplus C = \mathrm{E}(K, \mathrm{H}(T) \oplus P)$$
$$\mathrm{D}[K, H(T) \oplus C] = \mathrm{H}(T) \oplus P$$
$$\mathrm{H}(T) \oplus \mathrm{D}(K, \mathrm{H}(T) \oplus C) = P$$

It is now easy to construct a block cipher mode of operation by using a different tweak value on each block. In essence, the ECB mode is used but for each block the tweak is changed. This overcomes the principal security weakness of ECB, which is that two encryptions of the same block yield the same ciphertext.

Storage Encryption Requirements

The requirements for encrypting stored data, also referred to as "data at rest" differ somewhat from those for transmitted data. The P1619 standard was designed to have the following characteristics:

1. The ciphertext is freely available for an attacker. Among the circumstances that lead to this situation:
   a. A group of users has authorized access to a database. Some of the records in the database are encrypted so that only specific users can successfully read/

write them. Other users can retrieve an encrypted record but are unable to read it without the key.

b. An unauthorized user manages to gain access to encrypted records.

c. A data disk or laptop is stolen, giving the adversary access to the encrypted data.

2. The data layout is not changed on the storage medium and in transit. The encrypted data must be the same size as the plaintext data.

3. Data are accessed in fixed sized blocks, independently from each other. That is, an authorized user may access one or more blocks in any order.

4. Encryption is performed in 16-byte blocks, independently from other blocks (except the last two plaintext blocks of a sector, if its size is not a multiple of 16 bytes).

5. There are no other metadata used, except the location of the data blocks within the whole data set.

6. The same plaintext is encrypted to different ciphertexts at different locations, but always to the same ciphertext when written to the same location again.

7. A standard conformant device can be constructed for decryption of data encrypted by another standard conformant device.

The P1619 group considered some of the existing modes of operation for use with stored data. For CTR mode, an adversary with write access to the encrypted media can flip any bit of the plaintext simply by flipping the corresponding ciphertext bit.

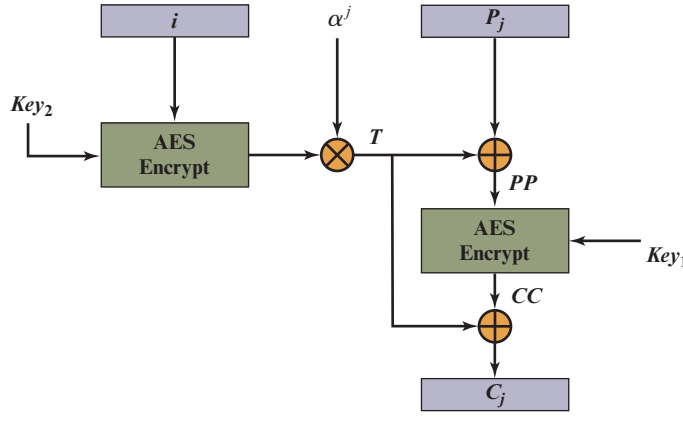
Next, consider requirement 6 and the use of CBC. To enforce the requirement that the same plaintext encrypts to different ciphertext in different locations, the IV could be derived from the sector number. Each sector contains multiple blocks. An adversary with read/write access to the encrypted disk can copy a ciphertext sector from one position to another within the same block, and an application reading the sector off the new location will still get the same plaintext sector (except perhaps the first 128 bits). Another weakness is that an adversary can flip any bit of the plaintext by flipping the corresponding ciphertext bit of the previous block, with the side-effect of "randomizing" the previous block.

Operation on a Single Block

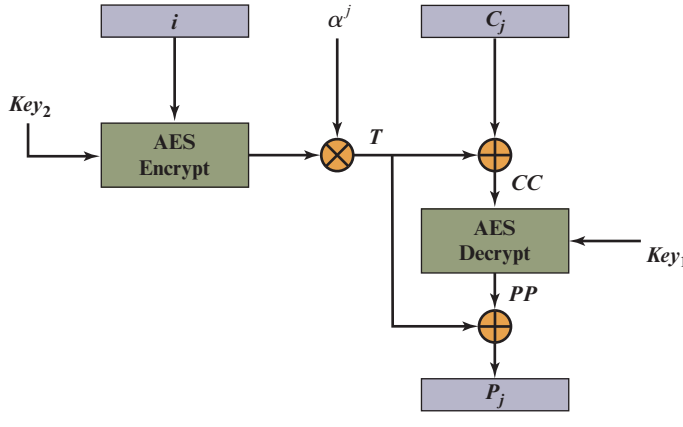
Figure 7.10 shows the encryption and decryption of a single block. The operation involves two instances of the AES algorithm with two keys. The following parameters are associated with the algorithm.

Key	The 256 or 512 bit XTS-AES key; this is parsed as a concatenation of two fields of equal size called Key_1 and Key_2, such that $Key = Key_1 \| Key_2$.
P_j	The jth block of plaintext. All blocks except possibly the final block have a length of 128 bits. A plaintext data unit, typically a disk sector, consists of a sequence of plaintext blocks $P_1, P_2, \ldots, P_m$.
C_j	The jth block of ciphertext. All blocks except possibly the final block have a length of 128 bits.

j	The sequential number of the 128-bit block inside the data unit.
i	The value of the 128-bit tweak. Each data unit (sector) is assigned a tweak value that is a nonnegative integer. The tweak values are assigned consecutively, starting from an arbitrary nonnegative integer.
α	A primitive element of GF(2^{128}) that corresponds to polynomial x (i.e., $0000 \ldots 010_2$).
α^j	α multiplied by itself j times, in GF(2^{128}).
$\oplus$	Bitwise XOR.
$\otimes$	Modular multiplication of two polynomials with binary coefficients modulo $x^{128} + x^7 + x^2 + x + 1$. Thus, this is multiplication in GF(2^{128}).



(a) Encryption



(b) Decryption

Figure 7.10 XTS-AES Operation on Single Block

In essence, the parameter j functions much like the counter in CTR mode. It assures that if the same plaintext block appears at two different positions within a data unit, it will encrypt to two different ciphertext blocks. The parameter i functions much like a nonce at the data unit level. It assures that, if the same plaintext block appears at the same position in two different data units, it will encrypt to two different ciphertext blocks. More generally, it assures that the same plaintext data unit will encrypt to two different ciphertext data units for two different data unit positions.

The encryption and decryption of a single block can be described as

XTS-AES block operation	$T = \mathrm{E}(K_2, i) \otimes \alpha^j$ $PP = P \oplus T$ $CC = \mathrm{E}(K_1, PP)$ $C = CC \oplus T$	$T = \mathrm{E}(K_2, i) \otimes \alpha^j$ $CC = C \oplus T$ $PP = \mathrm{D}(K_1, \mathrm{CC})$ $P = PP \oplus T$

To see that decryption recovers the plaintext, let us expand the last line of both encryption and decryption. For encryption, we have

$$C = CC \oplus T = \mathrm{E}(K_1, \mathrm{PP}) \oplus T = \mathrm{E}(K_1, P \oplus T) \oplus T$$

and for decryption, we have

$$P = PP \oplus T = \mathrm{D}(K_1, CC) \oplus T = \mathrm{D}(K_1, C \oplus T) \oplus T$$

Now, we substitute for C:

$$\begin{aligned} P &= \mathrm{D}(K_1, C \oplus T) \oplus T \\ &= \mathrm{D}(K_1, [E(K_1, P \oplus T) \oplus T] \oplus T) \oplus T \\ &= \mathrm{D}(K_1, E(K_1, P \oplus T)) \oplus T \\ &= (P \oplus T) \oplus T = P \end{aligned}$$

Operation on a Sector

The plaintext of a sector or data unit is organized into blocks of 128 bits. Blocks are labeled $P_0, P_1, \ldots, P_m$. The last block my be null or may contain from 1 to 127 bits. In other words, the input to the XTS-AES algorithm consists of m 128-bit blocks and possibly a final partial block.

For encryption and decryption, each block is treated independently and encrypted/decrypted as shown in Figure 7.10. The only exception occurs when the last block has less than 128 bits. In that case, the last two blocks are encrypted/decrypted using a **ciphertext-stealing** technique instead of padding. Figure 7.11 shows the scheme. P_{m-1} is the last full plaintext block, and P_m is the final plaintext block, which contains s bits with $1 \leq s \leq 127$. C_{m-1} is the last full ciphertext block, and C_m is the final ciphertext block, which contains s bits. This technique is commonly called ciphertext stealing because the processing of the last block "steals" a temporary ciphertext of the penultimate block to complete the cipher block.

Let us label the block encryption and decryption algorithms of Figure 7.10 as

Block encryption: XTS-AES-blockEnc(K, P_j, i, j)

Block decryption: XTS-AES-blockDec(K, C_j, i, j)

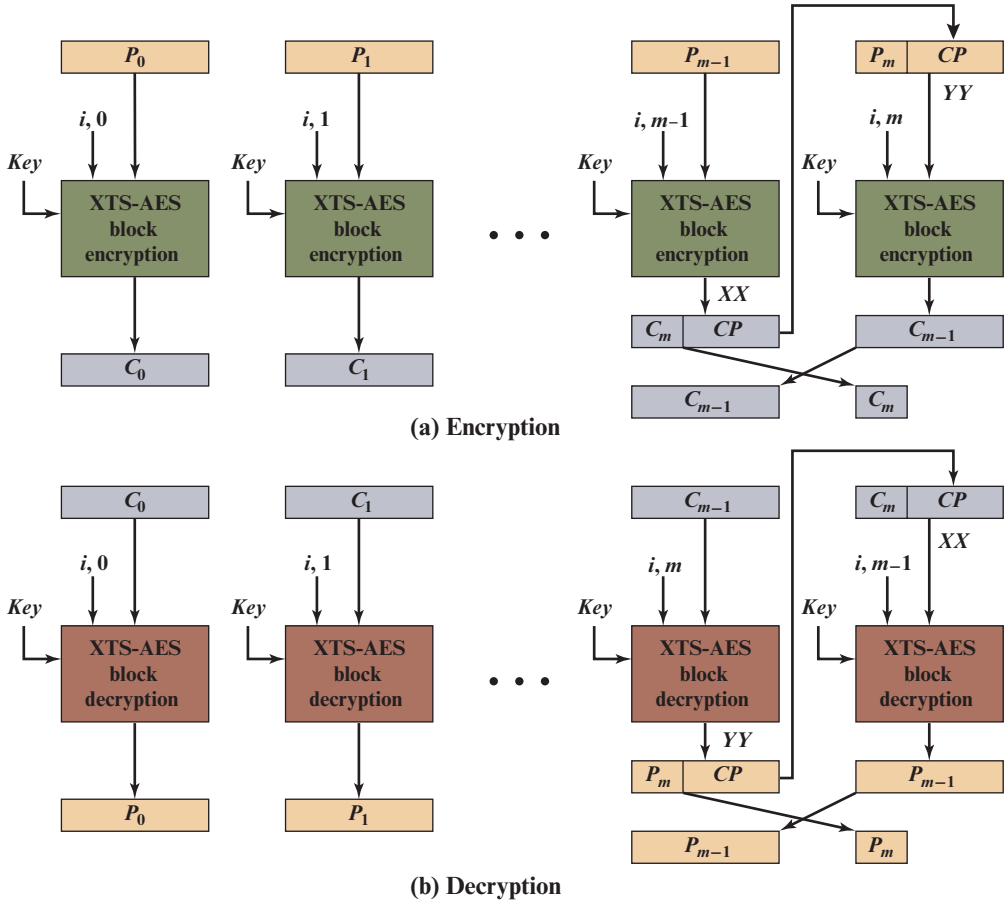


Figure 7.11 XTS-AES Mode

Then, XTS-AES mode is defined as follows:

XTS-AES mode with null final block	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j) \quad j = 0, \ldots, m-1$
	$P_j = \text{XTS-AES-blockEnc}(K, C_j, i, j) \quad j = 0, \ldots, m-1$
XTS-AES mode with final block containing s bits	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j)\ j = 0, \ldots, m-2$ $XX = \text{XTS-AES-blockEnc}(K, P_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(XX)$ $YY = P_m \Vert CP$ $C_{m-1} = \text{XTS-AES-blockEnc}(K, YY, i, m)$ $C_m = \text{MSB}_s(XX)$
	$P_j = \text{XTS-AES-blockDec}(K, C_j, i, j)\ j = 0, \ldots, m-2$ $YY = \text{XTS-AES-blockDec}(K, C_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(YY)$ $XX = C_m \Vert \text{CP}$ $P_{m-1} = \text{XTS-AES-blockDec}(K, XX, i, m)$ $P_m = \text{MSB}_s(YY)$

As can be seen, XTS-AES mode, like CTR mode, is suitable for parallel operation. Because there is no chaining, multiple blocks can be encrypted or decrypted simultaneously. Unlike CTR mode, XTS-AES mode includes a nonce (the parameter i) as well as a counter (parameter j).

7.8 FORMAT-PRESERVING ENCRYPTION

Format-preserving encryption (FPE) refers to any encryption technique that takes a plaintext in a given format and produces a ciphertext in the same format. For example, credit cards consist of 16 decimal digits. An FPE that can accept this type of input would produce a ciphertext output of 16 decimal digits. Note that the ciphertext need not be, and in fact is unlikely to be, a valid credit card number. But it will have the same format and can be stored in the same way as credit card number plaintext.

A simple encryption algorithm is not format preserving, with the exception that it preserves the format of binary strings. For example, Table 7.2 shows three types of plaintext for which it might be desired to perform FPE. The third row shows examples of what might be generated by an FPE algorithm. The fourth row shows (in hexadecimal) what is produced by AES with a given key.

Motivation

FPE facilitates the retrofitting of encryption technology to legacy applications, where a conventional encryption mode might not be feasible because it would disrupt data fields/pathways. FPE has emerged as a useful cryptographic tool, whose applications include financial-information security, data sanitization, and transparent encryption of fields in legacy databases.

The principal benefit of FPE is that it enables protection of particular data elements in a legacy database that did not provide encryption of those data elements, while still enabling workflows that were in place before FPE was in use. With FPE, as opposed to ordinary AES encryption or 3DES encryption, no database schema changes and minimal application changes are required. Only applications that need to see the plaintext of a data element need to be modified and generally these modifications will be minimal.

Some examples of legacy applications where FPE is desirable:

- COBOL data-processing applications: Any changes in the structure of a record requires corresponding changes in all code that references that record structure. Typical code sizes involve hundreds of modules, each containing around 5,000–10,000 lines on average.

Table 7.2 Comparison of Format-Preserving Encryption and AES

	Credit Card	Tax ID	Bank Account Number
Plaintext	8123 4512 3456 6780	219-09-9999	800N2982K-22
FPE	8123 4521 7292 6780	078-05-1120	709G9242H-35
AES (hex)	af411326466add24 c86abd8aa525db7a	7b9af4f3f218ab25 07c7376869313afa	9720ec7f793096ff d37141242e1c51bd

- Database applications: Fields that are specified to take only character strings cannot be used to store conventionally encrypted binary ciphertext. Base64 encoding of such binary ciphertext is not always feasible without increase in data lengths, requiring augmentation of corresponding field lengths.
- FPE-encrypted characters can be significantly compressed for efficient transmission. This cannot be said about AES-encrypted binary ciphertext.

Difficulties in Designing an FPE

A general-purpose standardized FPE should meet a number of requirements:

1. The ciphertext is of the same length and format as the plaintext.
2. It should be adaptable to work with a variety of character and number types. Examples include decimal digits, lowercase alphabetic characters, and the full character set of a standard keyboard or international keyboard.
3. It should work with variable plaintext lengths.
4. Security strength should be comparable to that achieved with AES.
5. Security should be strong even for very small plaintext lengths.

Meeting the first requirement is not at all straightforward. As illustrated in Table 7.2, a straightforward encryption with AES yields a 128-bit binary block that does not resemble the required format. Also, a standard symmetric block cipher is not easily adaptable to produce an FPE.

Consider a simple example. Assume that we want an algorithm that can encrypt decimal digit strings of maximum length of 32 digits. The input to the algorithm can be stored in 16 bytes (128 bits) by encoding each digit as four bits and using the corresponding binary value for each digit (e.g., 6 is encoded as 0101). Next, we use AES to encrypt the 128-bit block, in the following fashion:

1. The plaintext input X is represented by the string of 4-bit decimal digits $X[1] \ldots X[16]$. If the plaintext is less than 16 digits long, it is padded out to the left (most significant) with zeros.
2. Treating X as a 128-bit binary string and using key K, form ciphertext $Y = \text{AES}_K(X)$.
3. Treat Y as a string of length 16 of 4-bit elements.
4. Some of the entries in Y may have values greater than 9 (e.g., 1100). To generate ciphertext Z in the required format, calculate

$$Z[i] = Y[i] \bmod 10, \qquad 1 \leq i \leq 16$$

This generates a ciphertext of 16 decimal digits, which conforms to the desired format. However, this algorithm does not meet the basic requirement of any encryption algorithm of reversibility. It is impossible to decrypt Z to recover the original plaintext X because the operation is one-way; that is, it is a many-to-one function. For example, $12 \bmod 10 = 2 \bmod 10 = 2$. Thus, we need to design a reversible function that is both a secure encryption algorithm and format preserving.

A second difficulty in designing an FPE is that some of the input strings are quite short. For example, consider the 16-digit credit card number (CCN). The first six digits provide the issuer identification number (IIN), which identifies the institution that issued the card. The final digit is a check digit to catch typographical errors or other mistakes. The remaining nine digits are the user's account number. However, a number of applications require that the last four digits be in the clear (the check digit plus three account digits) for applications such as credit card receipts, which leaves only six digits for encryption. Now suppose that an adversary is able to obtain a number of plaintext/ciphertext pairs. Each such pair corresponds to not just one CCN, but multiple CCNs that have the same middle six digits. In a large database of credit card numbers, there may be multiple card numbers with the same middle six digits. An adversary may be able to assemble a large dictionary mapping known as six-digit plaintexts to their corresponding ciphertexts. This could be used to decrypt unknown ciphertexts from the database. As pointed out in [BELL10a], in a database of 100 million entries, on average about 100 CCNs will share any given middle-six digits. Thus, if the adversary has learned k CCNs and gains access to such a database, the adversary can decrypt approximately $100k$ CCNs.

The solution to this second difficulty is to use a tweakable block cipher; this concept is described in Section 7.7. For example, the tweak for CCNs could be the first two and last four digits of the CCN. Prior to encryption, the tweak is added, digit-by-digit mod 10, to the middle six-digit plaintext, and the result is then encrypted. Two different CCNs with identical middle six digits will yield different tweaked inputs and therefore different ciphertexts. Consider the following:

CCN	Tweak	Plaintext	Plaintext + Tweak
4012 8812 3456 1884	401884	123456	524230
5105 1012 3456 6782	516782	123456	639138

Two CCNs with the same middle six digits have different tweaks and therefore different values to the middle six digits after the tweak is added.

Feistel Structure for Format-Preserving Encryption

As the preceding discussion shows, the challenge with FPE is to design an algorithm for scrambling the plaintext that is secure, preserves format, and is reversible. A number of approaches have been proposed in recent years [ROGA10, BELL09] for FPE algorithms. The majority of these proposals use a Feistel structure. Although IBM introduced this structure with their Lucifer cipher [SMIT71] almost half a century ago, it remains a powerful basis for implementing ciphers.

This section provides a general description of how the Feistel structure can be used to implement an FPE. In the following section, we look at three specific Feistel-based algorithms that are in the process of receiving NIST approval.

Encryption and Decryption Figure 7.12 shows the Feistel structure used in all of the NIST algorithms, with encryption shown on the left-hand side and decryption on the right-hand side. The structure in Figure 7.12 is the same as that shown in

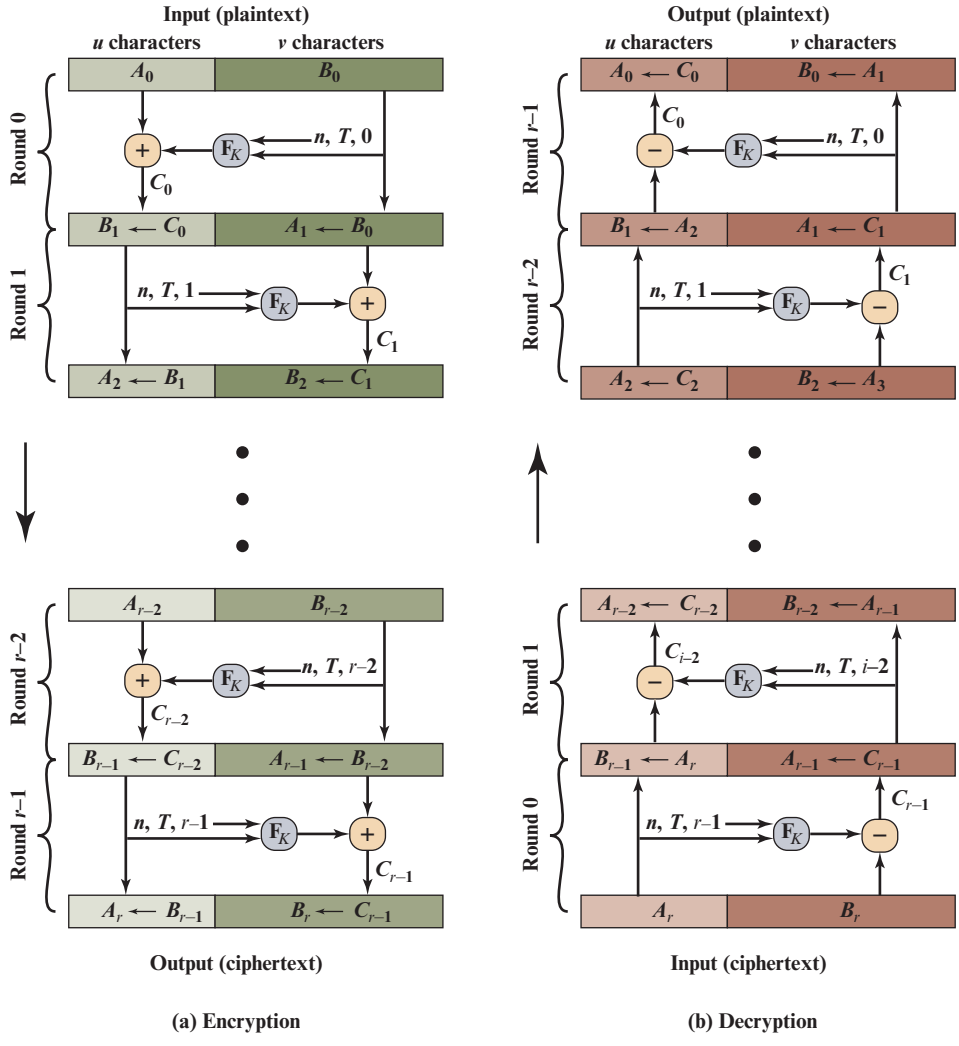


Figure 7.12 Feistel Structure for Format-Preserving Encryption

Figure 4.3 but, to simplify the presentation, it is untwisted, not illustrating the swap that occurs at the end of each round.

The input to the encryption algorithm is a plaintext character string of $n = u + v$ characters. If n is even, then $u = v$, otherwise u and v differ by 1. The two parts of the string pass through an even number of rounds of processing to produce a ciphertext block of n characters and the same format as the plaintext. Each round i has inputs A_i and B_i, derived from the preceding round (or plaintext for round 0).

All rounds have the same structure. On even-numbered rounds, a substitution is performed on the left part (length u) of the data, A_i. This is done by applying the round function F_K to the right part (length v) of the data, B_i, and then performing

a modular addition of the output of F_K with A_i. The modular addition function and the selection of modulus are described subsequently. On odd-numbered rounds, the substitution is done on the right part of the data. F_K is a one-way function that converts the input into a binary string, performs a scrambling transformation on the string, and then converts the result back into a character string of suitable format and length. The function has as parameters the secret key K, the plaintext length n, a tweak T, and the round number i.

Note that on even-numbered rounds, F_K has an input of v characters, and that the modular addition produces a result of u characters, whereas on odd-numbered rounds, F_K has an input of u characters, and that the modular addition produces a result of v characters. The total number of rounds is even, so that the output consists of an A portion of length u concatenated with a B portion of length v, matching the partition of the plaintext.

The process of decryption is essentially the same as the encryption process. The differences are: (1) the addition function is replaced by a subtraction function that is its inverse; and (2) the order of the round indices is reversed.

To demonstrate that the decryption produces the correct result, Figure 7.12b shows the encryption process going down the left-hand side and the decryption process going up the right-hand side. The diagram indicates that, at every round, the intermediate value of the decryption process is equal to the corresponding value of the encryption process. We can walk through the figure to validate this, starting at the bottom. The ciphertext is produced at the end of round $r - 1$ as a string of the form $A_r \| B_r$, with A_r and B_r having string lengths u and v, respectively. Encryption round $r - 1$ can be described with the following equations:

$$A_r = B_{r-1}$$
$$B_r = A_{r-1} + F_K[B_{r-1}]$$

Because we define the subtraction function to be the inverse of the addition function, these equations can be rewritten:

$$B_{r-1} = A_r$$
$$A_{r-1} = B_r - F_K[B_{r-1}]$$

It can be seen that the last two equations describe the action of round 0 of the decryption function, so that the output of round 0 of decryption equals the input of round $r - 1$ of encryption. This correspondence holds all the way through the r iterations, as is easily shown.

Note that the derivation does not require that F be a reversible function. To see this, take a limiting case in which F produces a constant output (e.g., all ones) regardless of the values of its input. The equations still hold.

Character Strings The NIST algorithms, and the other FPE algorithms that have been proposed, are used with plaintext consisting of a string of elements, called characters. Specifically, a finite set of two or more symbols is called an *alphabet*, and the elements of an alphabet are called *characters*. A *character string* is a finite sequence of characters from an alphabet. Individual characters may repeat in the string. The number of different characters in an alphabet is called the *base*, also

referred to as the *radix* of the alphabet. For example, the lowercase English alphabet a, b, c, . . . has a radix, or base, of 26. For purposes of encryption and decryption, the plaintext alphabet must be converted to numerals, where a *numeral* is a nonnegative integer that is less than the base. For example, for the lowercase alphabet, the assignment could be characters a, b, c, . . . , z map into 0, 1, 2, . . . , 25.

A limitation of this approach is that all of the elements in a plaintext format must have the same radix. So, for example, an identification number that consists of an alphabetic character followed by nine numeric digits cannot be handled in format-preserving fashion by the FPEs that have been implemented so far.

The NIST document defines notation for specifying these conversions (Table 7.3a). To begin, it is assumed that the character string is represented by a numeral string. To convert a numeral string X into a number x, the function $\text{NUM}_{radix}(\text{X})$ is used. Viewing X as the string $X[1] \ldots X[m]$ with the most significant numeral first, the function is defined as

$$\text{NUM}_{radix}(X) = \sum_{i=1}^{m} X[i]\, radix^{m-i} = \sum_{i=0}^{m-1} X[m-i]\, radix^{i}$$

Observe that $0 \leq \text{NUM}_{radix}(X) < radix^m$ and that $0 \leq X[i] < radix$.

Table 7.3 Notation and Parameters Used in FPE Algorithms

$[x]^s$	Converts an integer into a byte string; it is the string of s bytes that encodes the number x, with $0 \leq x < 2^{8s}$. The equivalent notation is $\text{STR}_2^{8s}(x)$.
$\text{LEN}(X)$	Length of the character string X.
$\text{NUM}_{radix}(X)$	Converts strings to numbers. The number that the numeral string X represents in base *radix*, with the most significant character first. In other words, it is the nonnegative integer less than $radix^{\text{LEN}(X)}$ whose most-significant-character-first representation in base *radix* is X.
$\text{PRF}_K(X)$	A pseudorandom function that produces a 128-bit output with X as the input, using encryption key K.
$\text{STR}_{radix}^{m}(x)$	Given a nonnegative integer x less than $radix^m$, this function produces a representation of x as a string of m characters in base *radix*, with the most significant character first.
$[i \,..\, j]$	The set of integers between two integers i and j, including i and j.
$X[i \,..\, j]$	The substring of characters of a string X from $X[i]$ to $X[j]$, including $X[i]$ and $X[j]$.
$\text{REV}(X)$	Given a bit string, X, the string that consists of the bits of X in reverse order.

(a) Notation

radix	The base, or number of characters, in a given plaintext alphabet.
tweak	Input parameter to the encryption and decryption functions whose confidentiality is not protected by the mode.
tweakradix	The base for tweak strings
minlen	Minimum message length, in characters.
maxlen	Maximum message length, in characters.
maxTlen	Maximum tweak length

(b) Parameters

For example, consider the string zaby in radix 26, which converts to the numeral string 25 0 1 24. This converts to the number $x = (25 \times 26^3) + (1 \times 26^1) + 24 = 439450$. To go in the opposite direction and convert from a number $x < radix^m$ to a numeral string X of length m, the function $\text{STR}^m_{radix}(x)$ is used:

$$\text{STR}^m_{radix}(x) = X[1] \ldots X[m], \text{ where}$$
$$X[i] = \left\lfloor \frac{x}{radix^{m-i}} \right\rfloor \bmod radix, \quad i = 1, \ldots, m$$

With the mapping of characters to numerals and the use of the NUM function, a plaintext character string can be mapped to a number and stored as an unsigned integer. We would like to treat this unsigned integer as a bit string that can be input to a bit-scrambling algorithm in F_K. However, different platforms store unsigned integers differently, some in little-endian and some in big-endian fashion. So one more step is needed. By the definition of the STR function, $\text{STR}^{8s}_2(x)$ will generate a bit string of length $8s$, equivalently a byte string of length s, which is a binary integer with the most significant bit first, regardless of how x is stored as an unsigned integer. For convenience the following notation is used: $[x]^s = \text{STR}^{8s}_2(x)$. Thus, $[\text{NUM}_{radix}(X)]^s$ will convert the character string X into an unsigned integer and then convert that to a byte string of length s bytes with the most significant bit first.

Continuing, the preceding example should help clarify the issues involved.

Character string	**"zaby"**
Numeral string X representation of character string	`25 0 1 24`
Convert X to number $x = \text{NUM}_{26}(X)$	decimal: `439450` hex: `6B49A` binary: `1101011010010011010`
x stored on big-endian byte order machine as a 32-bit unsigned integer	hex: `00 06 B4 9A` binary: `00000000000001101011010010011010`
x stored on little-endian byte order machine as a 32-bit unsigned integer	hex: `9A B4 06 00` binary: `10011010101101000000011000000000`
Convert x, regardless of endian format, to a bit string of length 32 bits (4 bytes), expressed as $[x]^4$	`00000000000001101011010010011010`

The Function F_K We can now define in general terms the function F_K. The core of F_K is some type of randomizing function whose input and output are bit strings. For convenience, the strings should be multiples of 8 bits, forming byte strings. Define m to be u for even rounds and v for odd rounds; this specifies the desired output character string length. Define b to be the number of bytes needed to store the number representing a character string of m bytes. Then the round,

including F_K, consists of the following general steps (A and B refer to A_i and B_i for round i):

1. $Q \leftarrow [\text{NUM}_{radix}(X)]^b E$	Converts numeral string X into byte string Q of length b bytes.
2. $Y \leftarrow \text{RAN}[Q]$	A pseudorandom function PRNF that produces a pseudorandom byte string Y as a function of the bits of Q.
3. $y \leftarrow \text{NUM}_2(Y)$	Converts Y into unsigned integer.
4. $c \leftarrow (\text{NUM}_{radix}(A) + y) \bmod radix^m$	Converts numeral string A into an integer and adds to y, modulo $radix^m$.
5. $C \leftarrow \text{STR}^m_{radix}(c)$	Converts c into a numeral string C of length m.
6. $A \leftarrow B$; $B \leftarrow C$	Completes the round by placing the unchanged value of B from the preceding round into A, and placing C into B.

Steps 1 through 3 constitute the round function F_K. Step 3 is presented with Y, which is an unstructured bit string. Because different platforms may store unsigned integers using different word lengths and endian conventions, it is necessary to perform $\text{NUM}_2(Y)$ to get an unsigned integer y. The stored bit sequence for y may or may not be identical to the bit sequence for Y.

As mentioned, the pseudorandom function in step 2 need not be reversible. Its purpose is to provide a randomized, scrambled bit string. For DES, this is achieved by using fixed S-boxes, as described in Appendix C. Virtually all FPE schemes that use the Feistel structure use AES as the basis for the scrambling function to achieve stronger security.

Relationship Between Radix, Message Length, and Bit Length Consider a numeral string X of length *len* and base *radix*. If we convert this to a number $x = \text{NUM}_{radix}(X)$, then the maximum value of x is $radix^{len} - 1$. The number of bits needed to encode x is

$$bitlen = \lceil \text{LOG}_2(radix^{len}) \rceil = \lceil len \text{LOG}_2(radix) \rceil$$

Observe that an increase in either *radix* or *len* increases *bitlen*. Often, we want to limit the value of *bitlen* to some fixed upper limit, for example, 128 bits, which is the size of the input to AES encryption. We also want the FPE to handle a variety of radix values. The typical FPE, and all of those discussed subsequently, allow a given range of radix values and then define a maximum character string length in order to provide the algorithm with a fixed value of *bitlen*. Let the range of radix values be from 2 to *maxradix*, and the maximum allowable character string value be *maxlen*. Then the following relationship holds:

$$maxlen \leq \lfloor bitlen/\text{LOG}_2(radix) \rfloor, \text{ or equivalently}$$
$$maxlen \leq \lfloor bitlen \times \text{LOG}_{radix}(2) \rfloor$$

For example, for a radix of 10, $maxlen \leq \lfloor 0.3 \times bitlen \rfloor$; for a radix of 26, $maxlen \leq \lfloor 0.21 \times bitlen \rfloor$. The larger the radix, the smaller the maximum character length for a given bit length.

NIST Methods for Format-Preserving Encryption

In 2013, NIST issued SP 800-38G: *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*. This Recommendation specifies three methods for format-preserving encryption, called FF1, FF2, and FF3. The three methods all use the Feistel structure shown in Figure 7.12. They employ somewhat different round functions F_K, which are built using AES. Important differences are the following:

- FF1 supports the greatest range of lengths for the plaintext character string and the tweak. To achieve this, the round function uses a cipher-block-chaining (CBC) style of encryption, whereas FF2 and FF3 employ simple electronic codebook (ECB) encryption.
- FF2 uses a subkey generated from the encryption key and the tweak, whereas FF1 and FF3 use the encryption key directly. The use of a subkey may help protect the original key from side-channel analysis, which is an attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or cryptanalysis. Examples of such attacks are attempts to deduce key bits based on power consumption or execution time.
- FF3 offers the lowest round count, eight, compared to ten for FF1 and FF2, and is the least flexible in the tweaks that it supports.

Algorithm FF1 Algorithm FF1 was submitted to NIST as a proposed FPE mode [BELL10a, BELL10b] with the name FFX[Radix]. FF1 uses a pseudorandom function $\text{PRF}_K(X)$ that produces a 128-bit output with inputs X that is a multiple of 128 bits and encryption key K (Figure 7.13). In essence, $PRF_K(X)$ use CBC encryption (Figure 7.4) with X as the plaintext input, encryption key K, and an initial vector (IV) of all zeros. The output is the last block of ciphertext produced. This is also

Prerequisites:

Approved, 128-bit block cipher, CIPH;
Key, K, for the block cipher;

Input:

Nonempty bit string, X, such that LEN(X) is a multiple of 128.

Output:

128-bit block, Y

Steps:

1. Let $m = \text{LEN}(X)/128$.
2. Partition X into m 128-bit blocks $X_1, \ldots, X_m$, so that $X = X_1 \| \ldots \| X_m$
3. Let $Y_0 = [0]^{16}$
4. For j from 1 to m:
5. let $Y_j = \text{CIPH}_K(Y_{j-1} \oplus X_j)$.
6. Return Y_m.

Figure 7.13 Algorithm PRF(X)

equivalent to the message authentication code known as CBC-MAC, or CMAC, described in Chapter 12.

The FF1 encryption algorithm is illustrated in Figure 7.14. The shaded lines correspond to the function F_K. The algorithm has 10 rounds and the following parameters (Table 7.3b):

- $radix \in [2 .. 2^{16}]$
- $radix^{minlen} \geq 100$
- $minlen \geq 2$
- $maxlen < 2^{32}$. For the maximum radix value of 2^{16}, the maximum bit length to store the integer value of X is 16×2^{32} bits; for the minimum radix value of 2, the maximum bit length to store the integer value of X is 2^{32} bits.
- $maxTlen < 2^{32}$

The inputs to the encryption algorithm are a character string X of length n and a tweak T of length t. The tweak is optional in that it may be the empty string.

Prerequisites:

Approved, 128-bit block cipher, CIPH;
Key, K, for the block cipher;
Base, *radix*, for the character alphabet;
Range of supported message lengths, [minlen .. maxlen];
Maximum byte length for tweaks, maxTlen.

Inputs:

Character string, X, in base *radix* of length n such that $n \in$ [minlen .. maxlen];
Tweak T, a byte string of byte length t, such that $t \in$ [0 .. maxTlen].

Output:

Character string, Y, such that $\text{LEN}(Y) = n$.

Steps:

1. Let $u = \lfloor n/2 \rfloor$; $v = n - u$.
2. Let $A = X[1 .. u]$; $B = X[u + 1 .. n]$.
3. Let $b = \lceil \lceil v \, \text{LOG}_2(radix) \rceil / 8 \rceil$; $d = 4\lceil b/4 \rceil + 4$
4. Let $P = [1]^1 \| [2]^1 \| [1]^1 \| [radix]^3 \| [10]^1 \| [u \bmod 256]^1 \| [n]^4 \| [t]^4$.
5. For **i** from 0 to 9:

   i. Let $Q = T \| [0]^{(-t-b-1) \bmod 16} \| [i]^1 \| [\text{NUM}_{radix}(B)]^b$.
   ii. Let $R = \text{PRF}_K(P \| Q)$.
   iii. Let S be the first d bytes of the following string of $\lceil d/16 \rceil$ 128-bit blocks:
   $R \| \text{CIPH}_K(R \oplus [1]^{16}) \| \text{CIPH}_K(R \oplus [2]^{16}) \| \ldots \| \text{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16})$.
   iv. Let $y = \text{NUM}_2(S)$.
   v. If i is even, let $m = u$; else, let $m = v$.

   vi. Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$.
   vii. Let $C = \text{STR}^m_{radix}(c)$.
   viii. Let $A = B$.
   ix. Let $B = C$.

6. Return $Y = A \| B$.

Figure 7.14 Algorithm FF1 (FFX[Radix])

The output is the encrypted character string Y of length n. What follows is a step-by-step description of the algorithm.

1., 2. The input X is split into two substrings A and B. If n is even, A and B are of equal length. Otherwise, B is one character longer than A.

3. The expression $\lceil v \, \text{LOG}_2(\text{radix}) \rceil$ equals the number of bits needed to encode B, which is v characters long. Encoding B as a byte string, b is the number of bytes in the encoding. The definition of d ensures that the output of the Feistel round function is at least 4 bytes longer than this encoding of B, which minimizes any bias in the modular reduction in step 5.vi, as explained subsequently.

4. P is a 128-bit (16-byte) block that is a function of *radix*, u, n, and t. It serves as the first block of plaintext input to the CBC encryption mode used in 5.ii, and is intended to increase security.

5. The loop through the 10 rounds of encryption.

5.i The tweak, T, the substring, B, and the round number, i, are encoded as a binary string, Q, which is one or more 128-bit blocks in length. To understand this step, first note that the value $\text{NUM}_{radix}(B)$ produces a numeral string that represents B in base *radix*. How this numeral string is formatted and stored is outside the scope of the standard. Then, the value $[\text{NUM}_{radix}(B)]^b$ produces the representation of the numerical value of B as a binary number in a string of b bytes. We also have the length of T is t bytes, and the round number is stored in a single byte. This yields a length of $(t + b + 1)$ bytes. This is padded out with $z = (-t - b - 1) \bmod 16$ bytes. From the rules of modular arithmetic, we know that $(z + t + b + 1) \bmod 16 = 0$. Thus the length of Q is one or more 128-bit blocks.

5.ii The concatenation of P and Q is input to the pseudorandom function PRF to produce a 128-bit output R. This function is the pseudorandom core of the Feistel round function. It scrambles the bits of B_i (Figure 7.12).

5.iii This step either truncates or expands R to a byte string S of length d bytes. That is, if $d \leq 16$ bytes, then R is the first d bytes of R. Otherwise the 16-byte R is concatenated with successive encryptions of R XORed with successive constants to produce the shortest string of 16-byte blocks whose length is greater than or equal to d bytes.

5.iv This step begins the process of converting the results of the scrambling of B_i into a form suitable for combining with A_i. In this step, the d-byte string S is converted into a numeral string in base 2 that represents S. That is, S is represented as a binary string y.

5.v This step determines the length m of the character string output that is required to match the length of the B portion of the round output. For even-numbered rounds, the length is u characters, and for odd-numbered rounds it is v characters, as shown in Figure 7.12.

5.vi The numerical values of A and y are added modulo $radix^m$. This truncates the value of the sum to a value c that can be stored in m characters.

5.vii This step converts the c into the proper representation C as a string of m characters.

5.viii, 5.ix These steps complete the round by placing the unchanged value of B from the preceding round into A, and placing C into B.

6. After the final round, the result is returned as the concatenation of A and B.

It may be worthwhile to clarify the various uses of the NUM function in FF1. NUM converts strings with a given radix into integers. In step 5.i, B is a character string in base radix, so $\text{NUM}_{radix}(B)$ converts this into an integer, which is stored as a byte string, suitable for encryption in step 5.ii. For step 5.iv, S is a byte string output of an encryption function, which can be viewed a bit string, so $\text{NUM}_2(S)$ converts this into an integer.

Finally, a brief explanation of the variable d is in order, which is best explained by example. Let $radix = 26$ and $v = 30$ characters. Then $b = 18$ bytes, and $d = 24$ bytes. Step 5.ii produces an output R of 16 bytes. We desire a scrambled output of b bytes to match the input, and so R needs to be padded out. Rather than padding with a constant value such as all zeros, step 5.iii pads out with random bits. The result, in step 5.iv is a number greater than $radix^m$ of fully randomized bits. The use of randomized padding avoids a potential security risk of using a fixed padding.

Algorithm FF2 Algorithm FF2 was submitted to NIST as a proposed FPE mode with the name VAES3 [VANC11]. The encryption algorithm is defined in Figure 7.15. The shaded lines correspond to the function F_K. The algorithm has the following parameters:

- $radix \in [2 .. 2^8]$
- $tweakradix \in [2 .. 2^8]$
- $radix^{minlen} \geq 100$
- $minlen \geq 2$
- $maxlen \leq 2\lfloor 120/\text{LOG}_2(radix) \rfloor$ if $radix$ is a power of 2. For the maximum radix value of 2^8, $maxlen \leq 30$; for the minimum radix value of 2, $maxlen \leq 240$. In both cases, the maximum bit length to store the integer value of X is 240 bits, or 30 bytes.
- $maxlen \leq 2\lfloor 98/\text{LOG}_2(radix) \rfloor$ if $radix$ is a not a power of 2. For the maximum radix value of 255, $maxlen \leq 24$; for the minimum radix value of 3, $maxlen \leq 124$.
- $maxTlen \leq \lfloor 104/\text{LOG}_2(tweakradix) \rfloor$. For the maximum $tweakradix$ value of 2^8, $maxTlen \leq 13$.

For FF2, the plaintext character alphabet and that of the tweak may be different.

The first two steps of FF2 are the same as FF1, setting values for v, u, A, and B. FF2 proceeds with the following steps:

3. P is a 128-bit (16-byte) block. If there is a tweak, then P is a function of $radix$, t, n, and the 13-byte numerical value of the tweak. If there is no tweak ($t = 0$), then P is a function of $radix$ and n. P is used to form an encryption key in step 4.

4. J is the encryption of P using the input key K.

Approved, 128-bit block cipher, CIPH;
Key, K, for the block cipher;
Base, *tweakradix*, for the tweak character alphabet;
Range of supported message lengths, [*minlen* .. *maxlen*];
Maximum supported tweak length, *maxTlen*.

Inputs:

Numeral string, X, in base *radix*, of length n such that $n \in [minlen \,..\, maxlen]$;
Tweak numeral string, T, in base *tweakradix*, of length t such that $t \in [0 \,..\, maxTlen]$.

Output:
Numeral string, Y, such that $\text{LEN}(Y) = n$.

Steps:

1. Let $u = \lfloor n/2 \rfloor$; $v = n - u$.
2. Let $A = X[1 \,..\, u]$; $B = X[u + 1 \,..\, n]$.
3. If $t > 0$, $P = [radix]^1 \| [t]^1 \| [n]^1 \| [\text{NUM}_{tweakradix}(T)]^{13}$; else $P = [radix]^1 \| [0]^1 \| [n]^1 \| [0]^{13}$.
4. Let $J = \text{CIPH}_K(P)$.
5. For i from 0 to 9:
   i. Let $Q \leftarrow [i]^1 \| [\text{NUM}_{radix}(B)]^{15}$
   ii. Let $Y \leftarrow \text{CIPH}_J(Q)$.
   iii. Let $y \leftarrow \text{NUM}_2(Y)$.
   iv. If i is even, let $m = u$; else, let $m = v$.
   v. Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$.
   vi. Let $C = \text{STR}^m_{radix}(c)$.
   vii. Let $A = B$.
   viii. Let $B = C$.
6. Return $Y = A \| B$.

Figure 7.15 Algorithm FF2 (VAES3)

5. The loop through the 10 rounds of encryption.

5.i B is converted into a 15-byte number, prepended by the round number to form a 16-byte block Q.

5.ii Q is encrypted using the encryption key J to yield Y.

The remaining steps are the same as for FF1. The essential difference is in the way in which all of the parameters are incorporated into the encryption that takes place in the block F_K. In both cases, the encryption is not simply an encryption of B using key K. For FF1, B is combined with the tweak, the round number, t, n, u, and *radix* to form a string of multiple 16-byte blocks. Then CBC encryption is used with K to produce a 16-byte output. For FF2, all of the parameters besides B are combined to form a 16-byte block, which is then encrypted with K to form the key value J. J is then used as the key for the one-block encryption of B.

The structure of FF2 explains the maximum length restrictions. In step 3, P incorporates the radix, tweak length, the numeral string length, and the tweak into the calculation. As input to AES, P is restricted to 16 bytes. With a maximum radix value of 2^8, the radix value can be stored in one byte (byte value 0 corresponds to 256). The string length n and tweak length t each easily fits into one byte. This leaves a restriction that the value of the tweak should be stored in at most 13 bytes,

or 104 bits. The number of bits to store the tweak is $\text{LOG}_2(tweakradix^{Tlen})$. This leads to the restriction $maxTlen \geq \lfloor 104/\text{LOG}_2(tweakradix) \rfloor$. Similarly step 5i incorporates B and the round number into a 16-byte input to AES, leaving 15 bytes to encode B, or 120 bits, so that the length must be less than or equal to $\lfloor 120/\text{LOG}_2(radix) \rfloor$. The parameter *maxlen* refers to the entire block, consisting of partitions A and B, thus $maxlen \geq 2\lfloor 120/\text{LOG}_2(radix) \rfloor$.

There is a further restriction on *maxlen* for a radix that is not a power of 2. As explained in [VANC11], when the radix is not a power of 2, modular arithmetic causes the value (y mod $radix^m$) to not have uniform distribution in the output space, which can result in a cryptographic weakness.

ALGORITHM FF3 Algorithm FF3 was submitted to NIST as a proposed FPE mode with the name BPS-BC [BRIE10]. The encryption algorithm is illustrated in Figure 7.16. The shaded lines correspond to the function F_K. The algorithm has the following parameters:

- $radix \in [2 .. 2^{16}]$
- $radix^{minlen} \geq 100$
- $minlen \geq 2$

Approved, 128-bit block cipher, CIPH;
Key, K, for the block cipher;
Base, *radix*, for the character alphabet such that $radix \in [2..2^{16}]$;
Range of supported message lengths, [*minlen* .. *maxlen*], such that $minlen \geq 2$ and $maxlen \leq 2\lfloor \log_{radix}(2^{96}) \rfloor$.

Inputs:

Numeral string, X, in base *radix* of length n such that $n \in$ [minlen .. maxlen];
Tweak bit string, T, such that $\text{LEN}(T) = 64$.

Output:
Numeral string, Y, such that $\text{LEN}(Y) = n$.

Steps:

1. Let $u = \lceil n/2 \rceil$; $v = n - u$.
2. Let $A = X[1 .. u]$; $B = X[u + 1 .. n]$.
3. Let $T_L = T[0 .. 31]$ and $T_R = T[32 .. 63]$.
4. For i from 0 to 7:

   i. If i is even, let $m = u$ and $W = T_R$, else let $m = v$ and $W = T_L$.
   ii. Let $P = \text{REV}([\text{NUM}_{radix}(\text{REV}(B))]^{12}) \| [W \oplus \text{REV}([i]^4])$.
   iii. Let $Y = \text{CIPH}_K(P)$.
   iv. Let $y = \text{NUM}_2(\text{REV}(Y))$.

   v. Let $c = (\text{NUM}_{radix}(\text{REV}(A)) + y) \bmod radix^m$.
   vi. Let $C = \text{REV}(\text{STR}^m_{radix}(c))$.
   vii. Let $A = B$.
   viii. Let $B = C$.

5. Return $A \| B$.

Figure 7.16 Algorithm FF3 (BPS-BC)

- $maxlen \leq 2\lfloor \text{LOG}_{radix}(2^{96}) \rfloor$. For the maximum radix value of 2^{16}, $maxlen \leq 12$; for the minimum radix value of 2, $maxlen \leq 192$. In both cases, the maximum bit length to store the integer value of X is 192 bits, or 24 bytes.
- Tweak length = 64 bits

FF3 proceeds with the following steps:

1., 2. The input X is split into two substrings A and B. If n is even, A and B are of equal length. Otherwise, A is one character longer than B, in contrast to FF1 and FF2, where B is one character longer than A.

3. The tweak is partitioned into a 32-bit left tweak T_L and a 32-bit right tweak T_R.

4. The loop through the 8 rounds of encryption.

4.i As in FF1 and FF2, this step determines the length m of the character string output that is required to match the length of the B portion of the round output. The step also determines whether T_L or T_R will be used as W in step 4ii.

4.ii The bits of B are reversed, then $\text{NUM}_{radix}(B)$ produces a 12-byte numeral string in base *radix*; the results are again reversed. A 32-bit encoding of the round number i is stored in a 4-byte unit, which is reversed and then XORed with W. P is formed by concatenating these two results to form a 16-byte block.

4.iii P is encrypted using the encryption key K to yield Y.

4.iv This is similar to step 5.iv in FF1, except that Y is reversed before converting it into a numeral string in base 2.

4.v The numerical values of the reverse of A and y are added modulo $radix^m$. This truncates the value of the sum to a value c that can be stored in m characters.

4.vi This step converts c to a numeral string C.

The remaining steps are the same as for FF1.

7.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

block cipher modes of operation	meet-in-the-middle attack	Tweakable block cipher
ciphertext stealing	nonce	
	triple DES (3DES)	

Review Questions

7.1 What is a tweakable block cipher?

7.2 Why does double encryption of a plaintext with DES, with two different keys of 56 bits, not provide 112 bits of security?

7.3 Why is double encryption in DES with two different keys not likely to be equivalent to single encryption with a different key?

7.4 List and briefly define the block cipher modes of operation.

7.5 Why is the ECB mode not secure for encrypting large amounts of data or structured data?

Problems

7.1 You want to build a hardware device to do block encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Figure 7.17 shows two possibilities, both of which follow from the definition of CBC. Which of the two would you choose:

a. For security?

b. For performance?

7.2 Can you suggest a security improvement to either option in Figure 7.17, using only three DES chips and some number of XOR functions? Assume you are still limited to two keys.

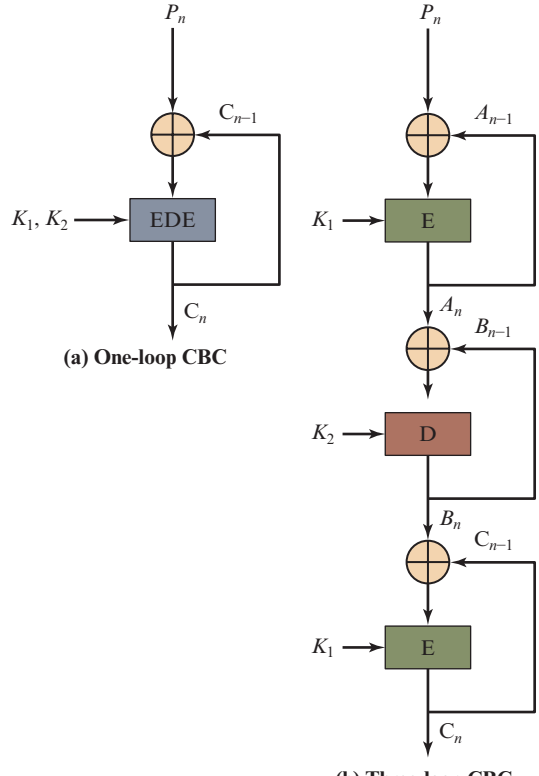


Figure 7.17 Use of Triple DES in CBC Mode

7.3 The Merkle–Hellman attack on 3DES begins by assuming a value of $A = 0$ (Figure 7.1b). Then, for each of the 2^{56} possible values of K_1, the plaintext P that produces $A = 0$ is determined. Describe the rest of the algorithm.

7.4 With the ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error in the transmitted C_1 (Figure 7.4) obviously corrupts P_1 and P_2.

a. Are any blocks beyond P_2 affected?

b. Suppose that there is a bit error in the source version of P_1. Through how many ciphertext blocks is this error propagated? What is the effect at the receiver?

7.5 Why should the initialization vector be protected against unauthorised use in the CBC mode of encryption?

7.6 By the end of the 1970s, it was already realized by practitioners of cryptography that a *DES* key size of 56 bits was small enough to permit brute-force key search attacks by adversaries having enough hardware. A proposal to increase the key size of a modified version of *DES* (potentially to 184 bits) was made by Rivest in 1984. The method is known as *DESX* and encrypts a message m with keys k_1, k_2, and k_3 of size 64, 56, and 64 bits, respectively, as follows: $DES\text{-}X(k_1, k_2, k_3, m) = k_1 \oplus DES(k_2, m \oplus k_3)$.

Show that it only gives a security of 120 bits against key search when the attacker has a few pairs of plaintext-ciphertext available. In fact, due to this attack, Rivest suggested keeping $k_1 = k_3$ with a security level of 120 bits.

7.7 Given n blocks of message, the ECB mode produces exactly n blocks of ciphertext. However, the CBC mode produces $(n + 1)$ blocks, due to the use of IV. Other modes such as CFB, OFB, and CTR also use an IV to encrypt messages. Show that secure encryption of multiple blocks of plaintext necessarily requires the use of IV (or some other form of randomization in the encryption process).

7.8 If a block of ciphertext gets corrupted during transmission in the OFB mode, how does it affect the decryption?

7.9 Is it possible to parallelize encryption in the CFB mode? What about decryption?

7.10 What are the advantages of CTR mode over the CBC mode? Explain in terms of the implementation benefits in software, hardware, and decryption throughput.

7.11 Padding may not always be appropriate. For example, one might wish to store the encrypted data in the same memory buffer that originally contained the plaintext. In that case, the ciphertext must be the same length as the original plaintext. We saw the use of ciphertext stealing in the case of XTS-AES to deal with partial blocks. Figure 7.18a shows the use of ciphertext stealing to modify CBC mode, called CBC-CTS.

a. Explain how it works.

b. Describe how to decrypt C_{n-1} and C_n.

7.12 Figure 7.18b shows an alternative to CBC-CTS for producing ciphertext of equal length to the plaintext when the plaintext is not an integer multiple of the block size.

a. Explain the algorithm.

b. Explain why CBC-CTS is preferable to this approach illustrated in Figure 7.18b.

7.13 Draw a figure similar to those of Figure 7.8 for XTS-AES mode.

7.14 Work out the following problems from first principles without converting to binary and counting the bits. Then, compare with the formulae presented for encoding a

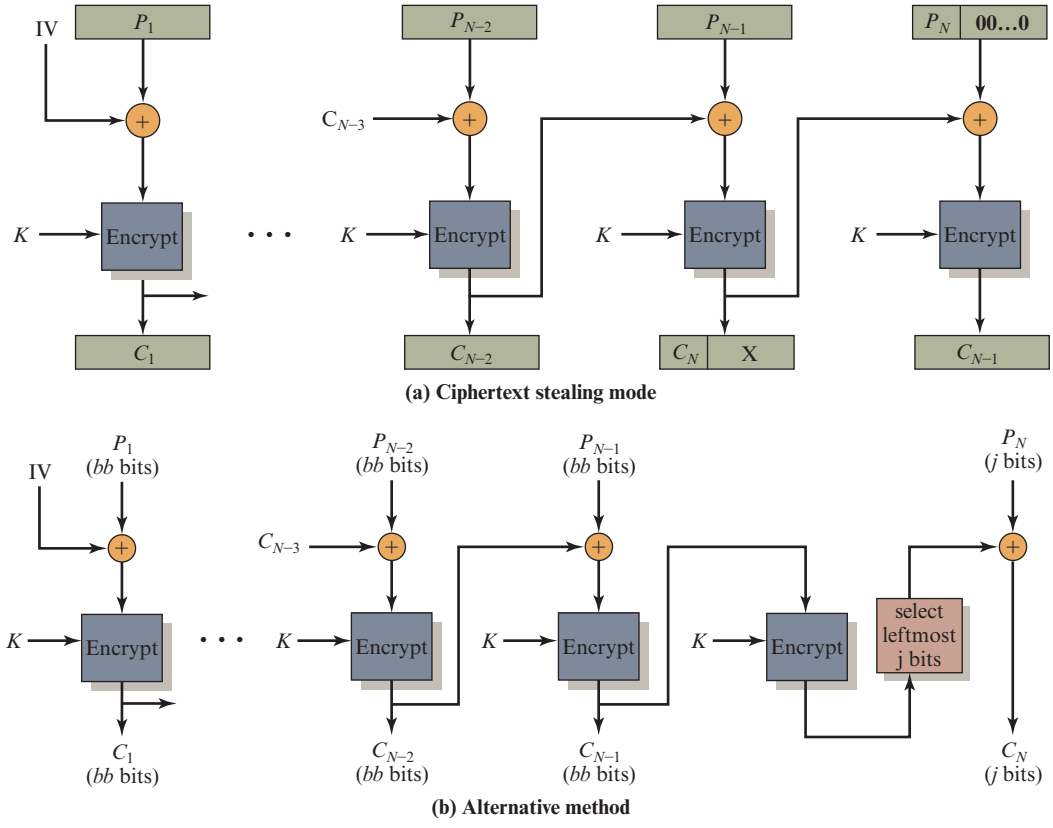


Figure 7.18 Block Cipher Modes for Plaintext not a Multiple of Block Size

character string into an integer, and vice-versa, in the specified radix. (*Hint*: Consider the next-lower and next-higher power of two for each integer.)

a. How many bits are exactly required to encode the following integers? (The number shown as an integer's subscript refers to the radix of that integer.)

i. 2047_{10}
ii. 2048_{10}
iii. 32767_{10}
iv. 32768_{10}
v. 32767_{16}
vi. 32768_{16}
vii. $537F_{16}$
viii. 29431_{10}

b. Exactly how many bytes are required to represent the numbers in (a) above?

7.15 **a.** In radix-26, write down the numeral string X for each of the following character strings, followed by the number of "digits" (i.e., the length of the numeral string) in each case.

i. "hex"
ii. "cipher"
iii. "not"
iv. "symbol"

b. For each case of problem (a), determine the number $x = \text{NUM}_{26}(X)$
c. Determine the byte form $[\boldsymbol{x}]$ for each number x computed in problem (b).
d. What is the smallest power of the radix (26) that is greater than each of the numerical strings determined in (b)?
e. Is it related to the length of the numeral string in each case, in problem (a)? If so, what is this relationship?

7.16 Refer to algorithms FF1 and FF2.
a. For step 1, for each algorithm, $u \leftarrow \lfloor n/2 \rfloor$ and $v \leftarrow \lceil n - u \rceil$. Show that for any three integers x, y, and n:
if $x = \lceil n/2 \rceil$ and $y = \lceil n - x \rceil$, then:
i. Either $x = n/2$, or $x = (n - 1)/2$.
ii. Either $y = n/2$, or $y = (n + 1)/2$.
iii. $x \leq y$. (Under what condition is $x = y$?)
b. What is the significance of result in the previous sub-problem (iii), in terms of the lengths u and v of the left and right half-strings, respectively?

7.17 In step 3 of Algorithm FF1, what do b and d represent? What is the unit of measurement (bits, bytes, digits, characters) of each of these quantities?

7.18 In the inputs to algorithms FF1, FF2, and FF3, why are the specified radix ranges important? For example, why should $radix \in [0..2^8]$ for Algorithm FF2, or $radix \in [2..2^{16}]$ in the case of Algorithm FF3?

Programming Problems

7.1 Create software that can encrypt and decrypt in cipher block chaining mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES.

Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.

7.2 Create software that can encrypt and decrypt in 4-bit cipher feedback mode using one of the following ciphers: additive modulo 256, affine modulo 256, S-DES;

or

8-bit cipher feedback mode using one of the following ciphers: 2×2 Hill modulo 256. Test data for S-DES using a binary initialization vector of 1010 1011. A binary plaintext of 0001 0010 0011 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1110 1100 1111 1010. Decryption should work correspondingly.

7.3 Create software that can encrypt and decrypt in counter mode using one of the following ciphers: affine modulo 256, Hill modulo 256, S-DES.

Test data for S-DES using a counter starting at 0000 0000. A binary plaintext of 0000 0001 0000 0010 0000 0100 encrypted with a binary key of 01111 11101 should give a binary plaintext of 0011 1000 0100 1111 0011 0010. Decryption should work correspondingly.

7.4 Implement a differential cryptanalysis attack on 3-round S-DES.