Advanced Cryptography

4th class

أستاذ المادة: د. عبد المنعم صالح رحمة

Block Cipher Principles

Most symmetric block encryption algorithms in current use are based on a structure referred to as a Feistel block cipher . For that reason, it is important to examine the design principles of the Feistel cipher. We begin with a comparison of stream ciphers and block ciphers. Then we discuss the motivation for the Feistel block cipher structure. Finally, we discuss some of its implications.

Stream Ciphers and Block Ciphers

A stream cipher is one that encrypts a digital data stream one bit or one byte at a time. Examples of classical stream ciphers are the autokeyed Vigenère cipher and the Vernam cipher. A block cipher is one in which a block of plaintext is treated as a whole and used to produce a ciphertext block of equal length. Typically, a block size of 64 or 128 bits is used. a block cipher can be used to achieve the same effect as a stream cipher.

Far more effort has gone into analyzing block ciphers. In general, they seem applicable to a broader range of applications than stream ciphers. The vast majority of network-based symmetric cryptographic applications make use of block ciphers. Accordingly, the concern in this chapter, and in our discussions throughout the book of symmetric encryption, will focus on block ciphers.

Motivation for the Feistel Cipher Structure

A block cipher operates on a plaintext block of n bits to produce a ciphertext block of n bits. There are 2^n possible different plaintext blocks and, for the encryption to be reversible (i.e., for decryption to be possible), each must produce a unique ciphertext block. Such a transformation is called reversible, or nonsingular. The following examples illustrate nonsingular and singular transformation for n = 2.

Reversible Mapping

Plaintext	Ciphertext
00	11
01	10
10	00
11	01

Irreversible Mapping

Plaintext	Ciphertext
00	11
01	10
10	01
11	01

In the latter case, a ciphertext of 01 could have been produced by one of two plaintext blocks. So if we limit ourselves to reversible mappings, the number of different transformations is $2^n!$.

Figure 3.1 illustrates the logic of a general substitution cipher for n = 4. A 4-bit input produces one of 16 possible input states, which is mapped by the substitution cipher into a unique one of 16 possible output states, each of which is represented by 4 ciphertext bits. The encryption and decryption mappings can be defined by a tabulation, as shown in Table 3.1. This is the most general form of block cipher and can be used to define any reversible mapping between plaintext and ciphertext. Feistel refers to this as the ideal block cipher, because it allows for the maximum number of possible encryption mappings from the plaintext block .

Figure 3.1. General n-bit-n-bit Block Substitution (shown with n = 4)

(This item is displayed on page 65 in the print version)

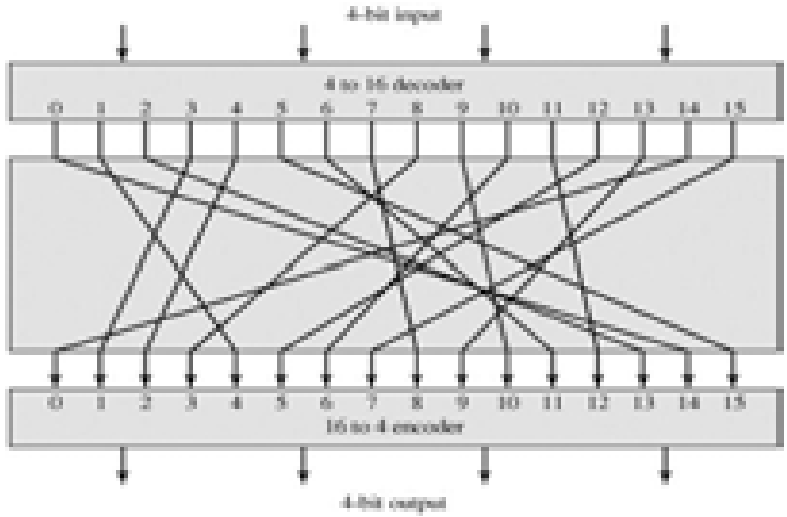


Table 3.1. Encryption and Decryption Tables for Substitution Cipher of Figure 3.4

(This item is displayed on page 65 in the print version)

Plaintext	Ciphertext
0000	1110
0001	0100
0010	1101
0011	0001
0100	0010
0101	1111
0110	1011
0111	1000
1000	0011
1001	1010
1010	0110
1011	1100

Table 3.1. Encryption and Decryption Tables for Substitution Cipher of Figure 3.4

(This item is displayed on page 65 in the print version)

Plaintext	Ciphertext
1100	0101
1101	1001
1110	0000
1111	0111
0000	1110
0001	0011
0010	0100
0011	1000
0100	0001
0101	1100
0110	1010
0111	1111
1000	0111
1001	1101
1010	1001
1011	0110
1100	1011
1101	0010
1110	0000
1111	0101

But there is a practical problem with the ideal block cipher. If a small block size, such as n = 4, is used, then the system is equivalent to a classical substitution cipher. Such systems, as we have seen, are vulnerable to a statistical analysis of the plaintext. This weakness is not inherent in the use of a substitution cipher but rather results from the use of a small block size.

If n is sufficiently large and an arbitrary reversible substitution between plaintext and ciphertext is allowed, then the statistical characteristics of the source plaintext are masked to such an extent that this type of cryptanalysis is infeasible.

An arbitrary reversible substitution cipher (the ideal block cipher) for a large block size is not practical, however, from an implementation and performance point of view. For such a transformation, the mapping itself constitutes the key. Consider again Table 3.1, which defines one particular reversible mapping from plaintext to ciphertext for n = 4. The mapping can be defined by the entries in the second column, which show the value of the ciphertext for each plaintext block. This, in essence, is the key that determines the specific mapping from among all possible mappings. In this case, using this straightforward method of defining the key, the required key length is (4 bits) x (16 rows) = 64 bits. In general, for an n-bit ideal block cipher, the length of the key defined in this fashion is n x 2^n bits. For a 64-bit block, which is a desirable length to thwart statistical attacks, the required key length is 64 x $2^{64} = 2^{70} \approx 10^{21}$bits.

In considering these difficulties, Feistel points out that what is needed is an approximation to the ideal block cipher system for large n, built up out of components that are easily realizable .But before turning to Feistel's approach, let us make one other observation. We could use the general block substitution cipher but, to make its implementation tractable, confine ourselves to a subset of the possible reversible mappings. For example, suppose we define the mapping in terms of a set of linear equations. In the case of n = 4, we have

$$y_1 = k_{11}x_1 + k_{12}x_2 + k_{13}x_3 + k_{14}x_4$$

$$y_2 = k_{21}x_1 + k_{22}x_2 + k_{23}x_3 + k_{24}x_4$$

$$y_3 = k_{31}x_1 + k_{32}x_2 + k_{33}x_3 + k_{34}x_4$$

$$y_4 = k_{41}x_1 + k_{42}x_2 + k_{43}x_3 + k_{44}x_4$$

where the x_i are the four binary digits of the plaintext block, the y_i are the four binary digits of the ciphertext block, the k_{ij} are the binary coefficients, and arithmetic is mod 2. The key size is just n^2, in this case 16 bits. The danger with this kind of formulation is that it may be vulnerable to cryptanalysis by an attacker that is aware of the structure of the algorithm.

The Feistel Cipher

Feistel proposed that we can approximate the ideal block cipher by utilizing the concept of a product cipher, which is the execution of two or more simple ciphers in sequence in such a way that the final result or product is cryptographically stronger than any of the component ciphers. The essence of the approach is to develop a block cipher with a key length of k bits and a block length of n bits, allowing a total of 2^k possible transformations, rather than the 2^n! transformations available with the ideal block cipher.

In particular, Feistel proposed the use of a cipher that alternates substitutions and permutations. In fact, this is a practical application of a proposal by Claude Shannon to develop a product cipher that alternates confusion and diffusion functions . We look next at these concepts of diffusion and confusion and then present the Feistel cipher. But first, it is worth commenting on this remarkable fact: The Feistel cipher structure, which dates back over a quarter century and which, in turn, is based on Shannon's proposal of 1945, is the structure used by many significant symmetric block ciphers currently in use.

Diffusion and Confusion

The terms diffusion and confusion were introduced by Claude Shannon to capture the two basic building blocks for any cryptographic system Shannon's concern was to thwart cryptanalysis based on statistical analysis. The reasoning is as follows. Assume the attacker has some knowledge of the statistical characteristics of the plaintext. For example, in a human-readable message in some language, the frequency distribution of the various letters may be known. Or there may be words or phrases likely to appear in the message (probable words). If these statistics are in any way reflected in the ciphertext, the cryptanalyst may be able to deduce the encryption key, or part of the key, or at least a set of keys likely to contain the exact key. In what Shannon refers to as a strongly ideal cipher, all statistics of the ciphertext are independent of the particular key used. The arbitrary substitution cipher that we discussed previously (Figure 3.1) is such a cipher, but as we have seen, is impractical.

[2] Shannon's 1949 paper appeared originally as a classified report in 1945. Shannon enjoys an amazing and unique position in the history of computer and information science. He not only developed the seminal ideas of modern

cryptography but is also responsible for inventing the discipline of information theory. In addition, he founded another discipline, the application of Boolean algebra to the study of digital circuits; this last he managed to toss off as a master's thesis.

Other than recourse to ideal systems, Shannon suggests two methods for frustrating statistical cryptanalysis: diffusion and confusion. In diffusion, the statistical structure of the plaintext is dissipated into long-range statistics of the ciphertext. This is achieved by having each plaintext digit affect the value of many ciphertext digits; generally this is equivalent to having each ciphertext digit be affected by many plaintext digits. An example of diffusion is to encrypt a message M = m_1, m_2, m_3,... of characters with an averaging operation:

$$y_n = \left(\sum_{i=1}^{k} m_{n+i} \right) \bmod 26$$

adding k successive letters to get a ciphertext letter y_n. One can show that the statistical structure of the plaintext has been dissipated. Thus, the letter frequencies in the ciphertext will be more nearly equal than in the plaintext; the digram frequencies will also be more nearly equal, and so on. In a binary block cipher, diffusion can be achieved by repeatedly performing some permutation on the data followed by applying a function to that permutation; the effect is that bits from different positions in the original plaintext contribute to a single bit of ciphertext.

[3] Some books on cryptography equate permutation with diffusion. This is incorrect. Permutation, by itself, does not change the statistics of the plaintext at the level of individual letters or permuted blocks. For example, in DES, the permutation swaps two 32-bit blocks, so statistics of strings of 32 bits or less are preserved.

Every block cipher involves a transformation of a block of plaintext into a block of ciphertext, where the transformation depends on the key. The mechanism of diffusion seeks to make the statistical relationship between the plaintext and ciphertext as complex as possible in order to thwart attempts to deduce the key. On the other hand, confusion seeks to make the relationship between the statistics of the ciphertext and the value of the encryption key as complex as possible, again to thwart attempts to discover the key. Thus, even if the attacker can get some handle on the statistics of

the ciphertext, the way in which the key was used to produce that ciphertext is so complex as to make it difficult to deduce the key. This is achieved by the use of a complex substitution algorithm. In contrast, a simple linear substitution function would add little confusion.

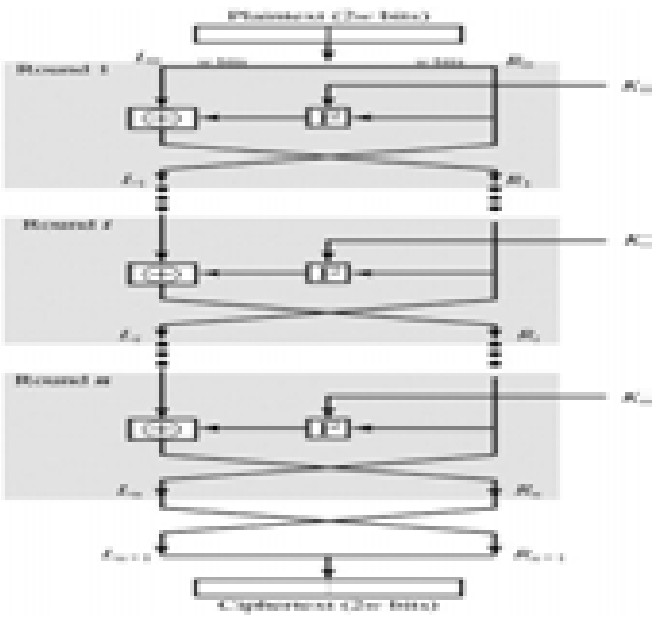
As points out, so successful are diffusion and confusion in capturing the essence of the desired attributes of a block cipher that they have become the cornerstone of modern block cipher design.

Feistel Cipher Structure

Figure 3.2 depicts the structure proposed by Feistel. The inputs to the encryption algorithm are a plaintext block of length 2w bits and a key K. The plaintext block is divided into two halves, L_0 and R_0. The two halves of the data pass through n rounds of processing and then combine to produce the ciphertext block. Each round i has as inputs L_{i-1} and R_{i-1}, derived from the previous round, as well as a subkey K_i, derived from the overall K. In general, the subkeys K_i are different from K and from each other.

Figure 3.2. Classical Feistel Network

(This item is displayed on page 69 in the print version)



All rounds have the same structure. A substitution is performed on the left half of the data. This is done by applying a *round function* F to the right half of the data and then taking the exclusive-OR of the output of that function and the left half of the data. The round function has the same general structure for each round but is parameterized by the round subkey K_i. Following this substitution, a permutation is performed that consists of the interchange of the two halves of the data. This structure is a particular form of the substitution-permutation network (SPN) proposed by Shannon.

[4] The final round is followed by an interchange that undoes the interchange that is part of the final round. One could simply leave both interchanges out of the diagram, at the sacrifice of some consistency of presentation. In any case, the effective lack of a swap in the final round is done to simplify the implementation of the decryption process, as we shall see.

The exact realization of a Feistel network depends on the choice of the following parameters and design features:

Block size: Larger block sizes mean greater security (all other things being equal) but reduced encryption/decryption speed for a given algorithm. The greater security is achieved by greater diffusion Traditionally, a block size of 64 bits has been considered a reasonable tradeoff and was nearly universal in block cipher design. However, the new AES uses a 128-bit block size.

- Key size: Larger key size means greater security but may decrease encryption/decryption speed. The greater security is achieved by greater resistance to brute-force attacks and greater confusion. Key sizes of 64 bits or less are now widely considered to be inadequate, and 128 bits has become a common size.
- Number of rounds: The essence of the Feistel cipher is that a single round offers inadequate security but that multiple rounds offer increasing security. A typical size is 16 rounds.Subkey generation algorithm: Greater complexity in this algorithm should lead to greater difficulty of cryptanalysis.
- Round function: Again, greater complexity generally means greater resistance to cryptanalysis.

There are two other considerations in the design of a Feistel cipher:

- Fast software encryption/decryption: In many cases, encryption is embedded in applications or utility functions in such a way as to preclude a hardware implementation. Accordingly, the speed of execution of the algorithm becomes a concern.
- Ease of analysis: Although we would like to make our algorithm as difficult as possible to cryptanalyze, there is great benefit in making the algorithm easy to analyze. That is, if the algorithm can be concisely and clearly explained, it is easier to analyze that algorithm for cryptanalytic vulnerabilities and therefore develop a higher level of assurance as to its strength. DES, for example, does not have an easily analyzed functionality.

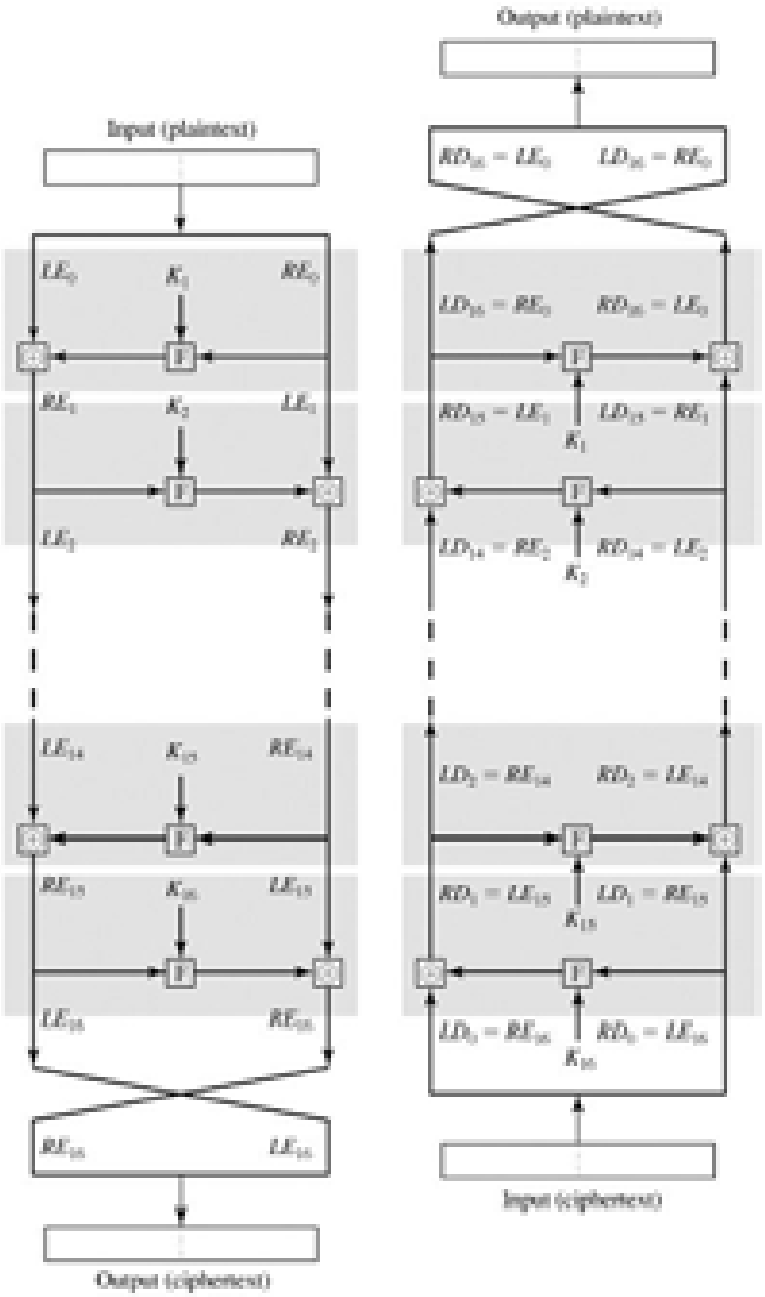
Feistel Decryption Algorithm

The process of decryption with a Feistel cipher is essentially the same as the encryption process. The rule is as follows: Use the ciphertext as input to the algorithm, but use the subkeys K_i in reverse order. That is, use K_n in the first round, K_{n-1} in the second round, and so on until K_1 is used in the last round. This is a nice feature because it means we need not implement two different algorithms, one for encryption and one for decryption.

To see that the same algorithm with a reversed key order produces the correct result, consider Figure 3.3, which shows the encryption process going down the left-hand side and the decryption process going up the right-hand side for a 16-round algorithm (the result would be the same for any number of rounds). For clarity, we use the notation LE_i and RE_i for data traveling through the encryption algorithm and LD_i and RD_i for data traveling through the decryption algorithm. The diagram indicates that, at every round, the intermediate value of the decryption process is equal to the corresponding value of the encryption process with the two halves of the value swapped. To put this another way, let the output of the ith encryption round be $LE_i||RE_i$ (L_i concatenated with R_i). Then the corresponding input to the (16 i)th decryption round is $RE_i||LE_i$ or, equivalently, $RD_{16-i}||LD_{16-i}$.

Figure 3.3. Feistel Encryption and Decryption

(This item is displayed on page 71 in the print version)



Let us walk through Figure 3.3 to demonstrate the validity of the preceding assertions. After the last iteration of the encryption process, the two halves of the output are swapped, so that the ciphertext is $RE_{16}||LE_{16}$. The output of that round is the ciphertext. Now take that ciphertext and use it as input to the same algorithm. The input to the first round is $RE_{16}||LE_{16}$, which is equal to the 32-bit swap of the output of the sixteenth round of the encryption process.

[5] To simplify the diagram, it is untwisted, not showing the swap that occurs at the end of each iteration. But please note that the intermediate result at the end of the ith stage of the encryption process is the 2w-bit quantity formed by concatenating LE_i and RE_i, and that the intermediate result at the end of the ith stage of the decryption process is the 2w-bit quantity formed by concatenating LD_i and RD_i.

Now we would like to show that the output of the first round of the decryption process is equal to a 32-bit swap of the input to the sixteenth round of the encryption process. First, consider the encryption process. We see that

$LE_{16} = RE_{15}$

$RE_{16} = LE_{15} \times F(RE_{15}, K_{16})$

On the decryption side,

$LD_1 = RD_0 = LE_{16} = RE_{15}$

$RD_1 = LD_0 \times F(RD_0, K_{16})$

$= RE_{16} \times F(RE_{15}, K_{16})$

$= [LE_{15} \times F(RE_{15}, K_{16})] \times F(RE_{15}, K_{16})$

The XOR has the following properties:

$[A \times B] \times C = A \times [B \times C]$

$D \times D = 0$

$E \times 0 = E$

Thus, we have $LD_1 = RE_{15}$ and $RD_1 = LE_{15}$. Therefore, the output of the first round of the decryption process is $LE_{15}||RE_{15}$, which is the 32-bit swap of the input to the sixteenth round of the encryption. This correspondence holds all the way through the 16 iterations, as is easily shown. We can cast this process in general terms. For the ith iteration of the encryption algorithm,

$LE_i = RE_{i-1}$

$RE_i = LE_{i-1} \times F(RE_{i-1}, K_i)$

Rearranging terms,

$RE_{i-1} = LE_i$

$LE_{i-1} = RE_i \times F(RE_{i-1}, K_{i2} = RE_i \times F(LE_i, K_i)$

Thus, we have described the inputs to the ith iteration as a function of the outputs, and these equations confirm the assignments shown in the right-hand side of Figure 3.3.

Finally, we see that the output of the last round of the decryption process is $RE_0||LE_0$. A 32-bit swap recovers the original plaintext, demonstrating the validity of the Feistel decryption process.

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 two arguments. The equations still hold.

Differential and Linear Cryptanalysis

For most of its life, the prime concern with DES has been its vulnerability to brute-force attack because of its relatively short (56 bits) key length. However, there has also been interest in finding cryptanalytic attacks on DES. With the increasing popularity of block ciphers with longer key lengths, including triple DES, brute-force attacks have become increasingly impractical. Thus, there has been increased emphasis on cryptanalytic attacks on DES and other symmetric block ciphers. In this section, we provide a brief overview of the two most powerful and promising approaches: differential cryptanalysis and linear cryptanalysis.

Differential Cryptanalysis

One of the most significant advances in cryptanalysis in recent years is differential cryptanalysis. In this section, we discuss the technique and its applicability to DES.

History

Differential cryptanalysis was not reported in the open literature until 1990. The first published effort appears to have been the cryptanalysis of a block cipher called FEAL by Murphy. This was followed by a number of papers by Biham and Shamir, who demonstrated this form of attack on a variety of encryption algorithms and hash functions; their results are summarized in .

The most publicized results for this approach have been those that have application to DES. Differential cryptanalysis is the first published attack that is capable of breaking DES in less than 2^{55} complexity. The scheme, as reported in , can successfully cryptanalyze DES with an effort on the order of 2^{47} encryptions, requiring 2^{47} chosen plaintexts. Although 2^{47} is certainly significantly less than 2^{55} the need for the adversary to find 2^{47} chosen plaintexts makes this attack of only theoretical interest.

Although differential cryptanalysis is a powerful tool, it does not do very well against DES. The reason, according to a member of the IBM team that designed DES ,is that differential cryptanalysis was known to the team as early as 1974. The need to strengthen DES against attacks using differential cryptanalysis played a large part in the design of the S-boxes and the permutation P. As evidence of the impact of these changes, consider these

comparable results reported in , Differential cryptanalysis of an eight-round LUCIFER algorithm requires only 256 chosen plaintexts, whereas an attack on an eight-round version of DES requires 2^{14} chosen plaintexts.

Differential Cryptanalysis Attack

The differential cryptanalysis attack is complex; provides a complete description. The rationale behind differential cryptanalysis is to observe the behavior of pairs of text blocks evolving along each round of the cipher, instead of observing the evolution of a single text block. Here, we provide a brief overview so that you can get the flavor of the attack.

We begin with a change in notation for DES. Consider the original plaintext block m to consist of two halves m_0,m_1. Each round of DES maps the right-hand input into the left-hand output and sets the right-hand output to be a function of the left-hand input and the subkey for this round. So, at each round, only one new 32-bit block is created. If we label each new block m_1($2 \leq I \leq 17$), then the intermediate message halves are related as follows:

$m_{i+1} = m_{i-1} \oplus f(m_i, K_i)$, i = 1, 2, ..., 16

In differential cryptanalysis, we start with two messages, m and m', with a known XOR difference $\Delta m = m \oplus m'$, and consider the difference between the intermediate message halves: $m_i = m_i \oplus m_i'$ Then we have:

$$\begin{aligned}\Delta m_{i+1} &= m_{i+1} \oplus m'_{i+1} \\ &= [m_{i-1} \oplus f(m_i, K_i)] \oplus [m'_{i-1} \oplus f(m'_i, K_i)] \\ &= \Delta m_{i-1} \oplus [f(m_i, K_i) \oplus f(m'_i, K_i)]\end{aligned}$$

Now, suppose that many pairs of inputs to f with the same difference yield the same output difference if the same subkey is used. To put this more precisely, let us say that X may cause Y with probability p, if for a fraction p of the pairs in which the input XOR is X, the output XOR equals Y. We want to suppose that there are a number of values of X that have high probability of causing a particular output difference. Therefore, if we know Δm_{i-1} and Δm_i with high probability, then we know Δm_{i+1} with high probability. Furthermore, if a number of such differences are determined, it is feasible to determine the subkey used in the function f.

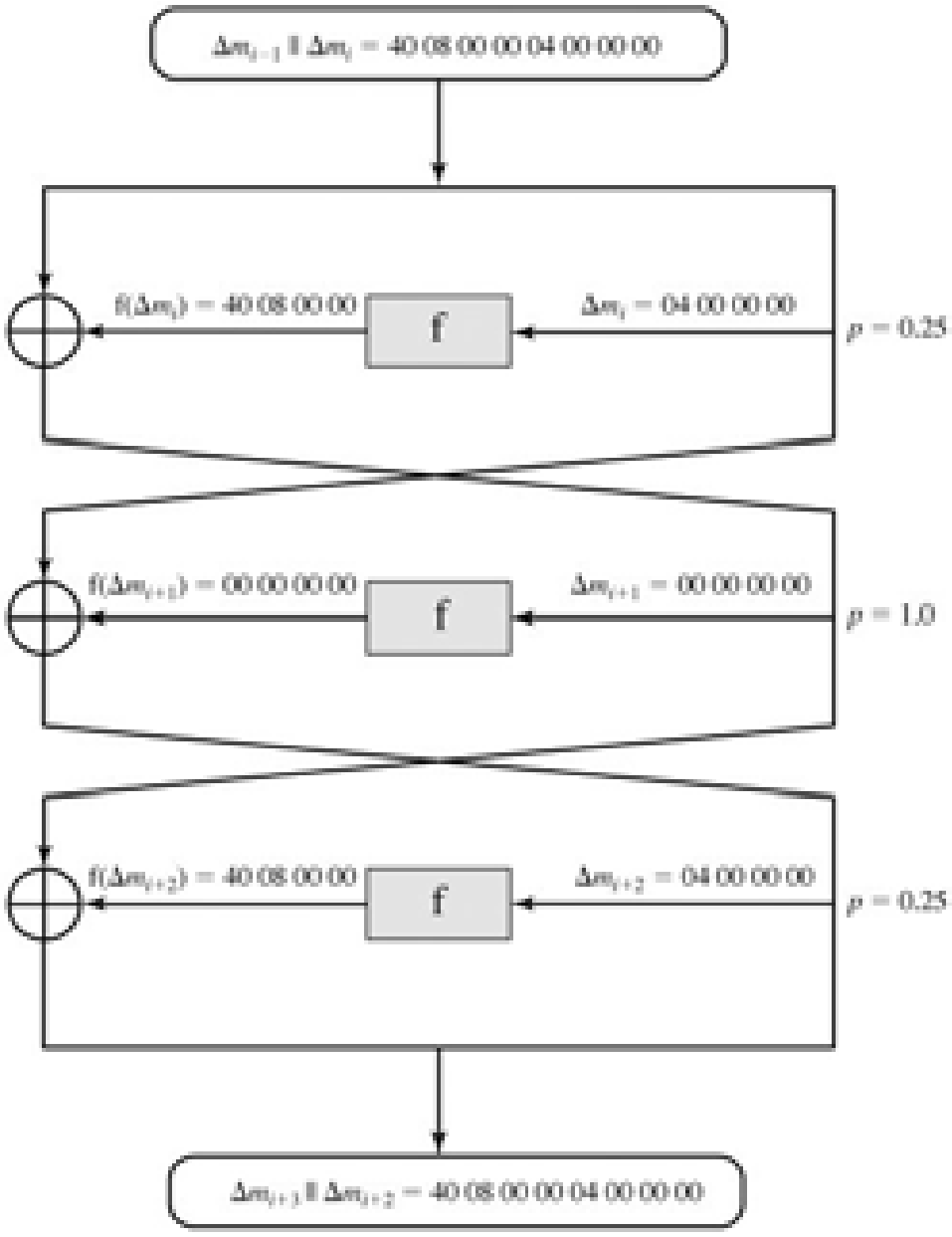
The overall strategy of differential cryptanalysis is based on these considerations for a single round. The procedure is to begin with two plaintext messages m and m' with a given difference and trace through a probable pattern of differences after each round to yield a probable difference for the ciphertext. Actually, there are two probable patterns of differences for the two 32-bit halves: ($m_{17}||m_{16}$). Next, we submit m and m' for encryption to determine the actual difference under the unknown key and compare the result to the probable difference. If there is a match,

$$E(K, m) \oplus E(K, m') = (\Delta m_{17}||m_{16})$$

then we suspect that all the probable patterns at all the intermediate rounds are correct. With that assumption, we can make some deductions about the key bits. This procedure must be repeated many times to determine all the key bits.

Figure 3.7, based on a figure in , illustrates the propagation of differences through three rounds of DES. The probabilities shown on the right refer to the probability that a given set of intermediate differences will appear as a function of the input differences. Overall, after three rounds the probability that the output difference is as shown is equal to 0.25 x 1 x 0.25 = 0.0625.

Figure 3.7. Differential Propagation through Three Round of DES (numbers in hexadecimal)



Linear Cryptanalysis

A more recent development is linear cryptanalysis, described in . This attack is based on finding linear approximations to describe the transformations performed in DES. This method can find a DES key given 2^{43} known plaintexts, as compared to 2^{47} chosen plaintexts for differential cryptanalysis. Although this is a minor improvement, because it may be easier to acquire known plaintext rather than chosen plaintext, it still leaves linear cryptanalysis infeasible as an attack on DES. So far, little work has been done by other groups to validate the linear cryptanalytic approach.

We now give a brief summary of the principle on which linear cryptanalysis is based. For a cipher with n-bit plaintext and ciphertext blocks and an m-bit key, let the plaintext block be labeled P[1], ... P[n], the cipher text block C[1], ... C[n], and the key K[1], ... K[m]. Then define

$A[i, j, ..., k] = A[i] \oplus A[j] \quad ... \quad A[k]$

The objective of linear cryptanalysis is to find an effective linear equation of the form:

$P[\alpha_1, \alpha_2, ..., \alpha_a] \quad C[\beta_1, \beta_2, ..., \beta_b] = K[\gamma_1, \gamma_2, ..., \gamma_c]$

(where x = 0 or 1; $1 \leq a, b \leq n$, $1 \leq c$ m, and where the α, β and γ terms represent fixed, unique bit locations) that holds with probability $p \neq 0.5$. The further p is from 0.5, the more effective the equation. Once a proposed relation is determined, the procedure is to compute the results of the left-hand side of the preceding equation for a large number of plaintext-ciphertext pairs. If the result is 0 more than half the time, assume $K[\gamma_1, \gamma_2, ..., \gamma_c] = 0$. If it is 1 most of the time, assume $K[\gamma_1, \gamma_2, ..., \gamma_c] = 1$. This gives us a linear equation on the key bits. Try to get more such relations so that we can solve for the key bits. Because we are dealing with linear equations, the problem can be approached one round of the cipher at a time, with the results combined.

Diffie - Hellman Key Exchange

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography and is generally referred to as Diffie-Hellman key exchange. A number of commercial products employ this key exchange technique.

[1] Williamson of Britain's CESG published the identical scheme a few months earlier in a classified document [WILL76] and claims to have discovered it several years prior to that.

The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of secret values.

The Diffie-Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms. Briefly, we can define the discrete logarithm in the following way. First, we define a primitive root of a prime number p as one whose powers modulo p generate all the integers from 1 to p 1. That is, if a is a primitive root of the prime number p, then the numbers

a mod p, a^2 mod p,..., a^{p1} mod p

are distinct and consist of the integers from 1 through p 1 in some permutation.

For any integer b and a primitive root a of prime number p, we can find a unique exponent i such that

$b \equiv a^i \pmod{p}$ where $0 \leq i \leq (p\ 1)$

The exponent i is referred to as the discrete logarithm of b for the base a, mod p. We express this value as $dlog_{a,p}$ (b).

The Algorithm

Figure 10.7 summarizes the Diffie-Hellman key exchange algorithm. For this scheme, there are two publicly known numbers: a prime number q and an integer that is a primitive root of q. Suppose the users A and B wish to exchange a key. User A selects a random integer $X_A < q$ and computes $Y_A =$

α^{XA} mod q. Similarly, user B independently selects a random integer $X_A < q$ and computes $Y_B = \alpha^{XB}$ mod q. Each side keeps the X value private and makes the Y value available publicly to the other side. User A computes the key as $K = (Y_B)^X{}_A$ mod q and user B computes the key as $K = (Y_A)^{XB}$ mod q. These two calculations produce identical results:

$K = (Y_B)^{XA}$ mod q

$= (\alpha^{XB} \text{ mod } q)^{XA}$ mod q

$= (\alpha^{XB})^{XA}$ mod q by the rules of modular arithmetic

$= (\alpha^{XB\,XA}$ mod q

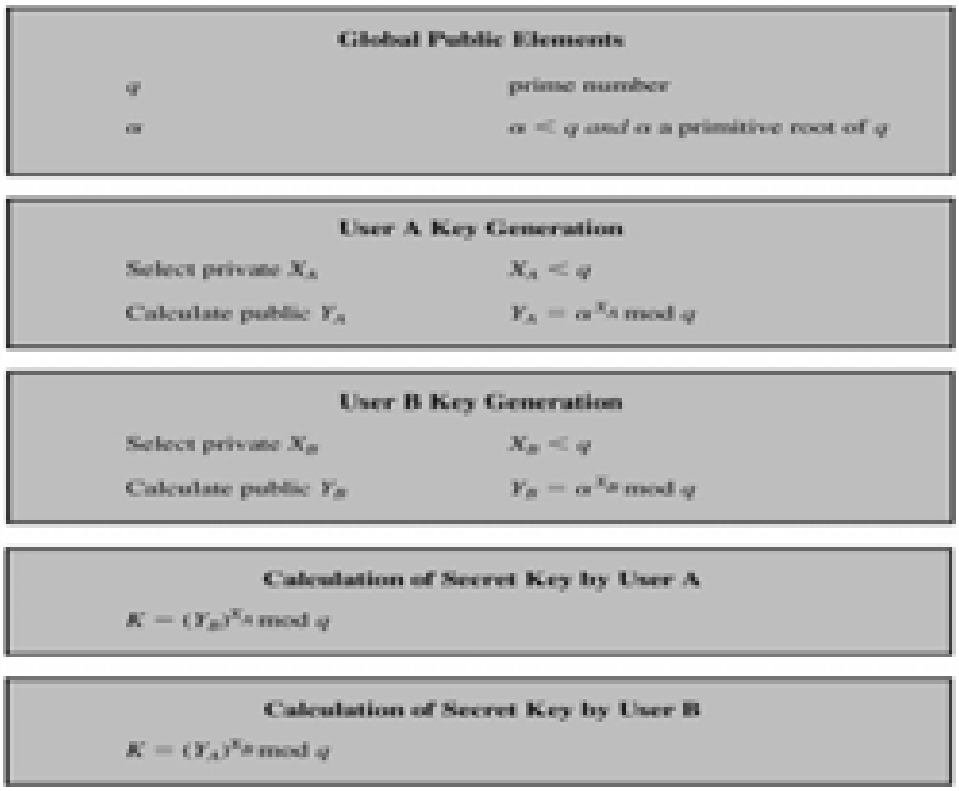
$= (\alpha^{XA})^{XB}$ mod q

$= (\alpha^{XA}$ mod q)

$= (\ ^{XA}$ mod q)^{XB} mod q

$= (Y_A)^{XB}$ mod q

Figure 10.7. The Diffie-Hellman Key Exchange Algorithm



The result is that the two sides have exchanged a secret value. Furthermore, because X_A and X_B are private, an adversary only has the following ingredients to work with: q, α,Y_A, and Y_B. Thus, the adversary is forced to take a discrete logarithm to determine the key. For example, to determine the private key of user B, an adversary must compute

$X_B = \text{dlog}_{\alpha,q}(Y_B)$

The adversary can then calculate the key K in the same manner as user B calculates it.

The security of the Diffie-Hellman key exchange lies in the fact that, while it is relatively easy to calculate exponentials modulo a prime, it is very difficult to calculate discrete logarithms. For large primes, the latter task is considered infeasible.

Here is an example. Key exchange is based on the use of the prime number q = 353 and a primitive root of 353, in this case α= 3. A and B select secret keys X_A = 97 and X_B = 233, respectively. Each computes its public key:

A computes $Y_A = 3^{97} \bmod 353 = 40$.

B computes $Y_B = 3^{233} \bmod 353 = 248$.

After they exchange public keys, each can compute the common secret key:

A computes $K = (Y_B)^{X_A} \bmod 353 = 248^{97} \bmod 353 = 160$.

B computes $K = (Y_A)^{X_E} \bmod 353 = 40^{233} \bmod 353 = 160$.

We assume an attacker would have available the following information:

q = 353; α = 3; Y_A = 40; Y_B = 248

In this simple example, it would be possible by brute force to determine the secret key 160. In particular, an attacker E can determine the common key by discovering a solution to the equation $3^a \bmod 353 = 40$ or the equation $3^b \bmod 353 = 248$. The brute-force approach is to calculate powers of 3 modulo

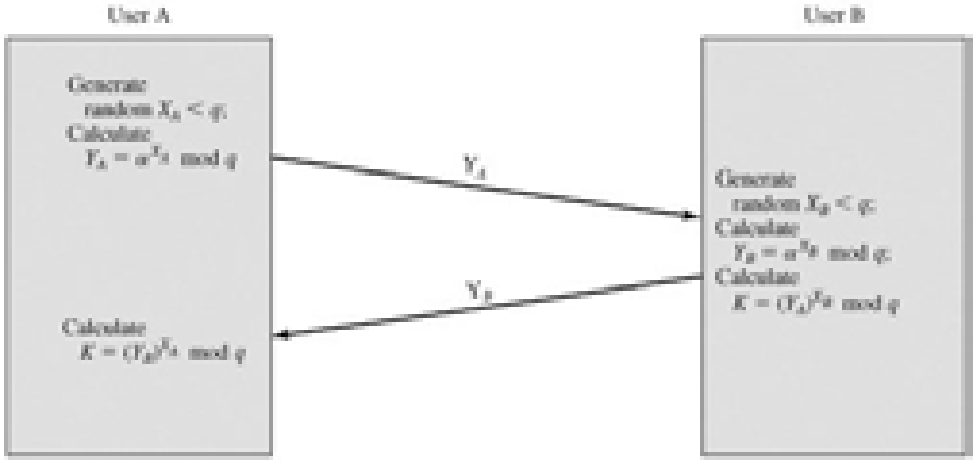
353, stopping when the result equals either 40 or 248. The desired answer is reached with the exponent value of 97, which provides $3^{97} \bmod 353 = 40$.

With larger numbers, the problem becomes impractical.

Key Exchange Protocols

Figure 10.8 shows a simple protocol that makes use of the Diffie-Hellman calculation. Suppose that user A wishes to set up a connection with user B and use a secret key to encrypt messages on that connection. User A can generate a one-time private key X_A, calculate Y_A, and send that to user B. User B responds by generating a private value X_B calculating Y_B, and sending Y_B to user A. Both users can now calculate the key. The necessary public values q and α would need to be known ahead of time. Alternatively, user A could pick values for q and α and include those in the first message.

Figure 10.8. Diffie-Hellman Key Exchange



As an example of another use of the Diffie-Hellman algorithm, suppose that a group of users (e.g., all users on a LAN) each generate a long-lasting private value X_i (for user i) and calculate a public value Y_i. These public values, together with global public values for q and α, are stored in some central directory. At any time, user j can access user i's public value, calculate a secret key, and use that to send an encrypted message to user A. If the central directory is trusted, then this form of communication provides both confidentiality and a degree of authentication. Because only i and j can determine the key, no other user can read the message (confidentiality). Recipient i knows that only user j could have created a message using this

key (authentication). However, the technique does not protect against replay attacks.

Man-in-the-Middle Attack

The protocol depicted in Figure 10.8 is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is the adversary. The attack proceeds as follows:

1. Darth prepares for the attack by generating two random private keys X_{D1} and X_{D2} and then computing the corresponding public keys Y_{D1} and Y_{D2}.
2. Alice transmits Y_A to Bob.
3. Darth intercepts Y_A and transmits Y_{D1} to Bob. Darth also calculates $K2 = (Y_A)^{X_{D2}} \bmod q$.
4. Bob receives Y_{D1} and calculates $K1 = (Y_{D1})^{X_E} \bmod q$.
5. Bob transmits X_A to Alice.
6. Darth intercepts X_A and transmits Y_{D2} to Alice. Darth calculates $K1 = (Y_B)^{X_{D1}} \bmod q$.
7. Alice receives Y_{D2} and calculates $K2 = (Y_{D2})^{X_A} \bmod q$.

At this point, Bob and Alice think that they share a secret key, but instead Bob and Darth share secret key K1 and Alice and Darth share secret key K2. All future communication between Bob and Alice is compromised in the following way:

1. Alice sends an encrypted message M: E(K2, M).
2. Darth intercepts the encrypted message and decrypts it, to recover M.
3. Darth sends Bob E(K1, M) or E(K1, M'), where M' is any message. In the first case, Darth simply wants to eavesdrop on the communication without altering it. In the second case, Darth wants to modify the message going to Bob.

The key exchange protocol is vulnerable to such an attack because it does not authenticate the participants. This vulnerability can be overcome with the use of digital signatures and public-key certificates.

Discrete Logarithms

Discrete logarithms are fundamental to a number of public-key algorithms, including Diffie-Hellman key exchange and the digital signature algorithm (DSA). This section provides a brief overview of discrete logarithms. For the interested reader.

The Powers of an Integer, Modulo n

Recall from Euler's theorem [Equation (8.4)] that, for every a and n that are relatively prime:

$a^{\phi(n)} \equiv 1 \pmod{n}$

where $\phi(n)$, Euler's totient function, is the number of positive integers less than n and relatively prime to n. Now consider the more general expression:

Equation 8-10

$$a^m \equiv 1 \pmod{n}$$

If a and n are relatively prime, then there is at least one integer m that satisfies Equation (8.10), namely, $m = \phi(n)$. The least positive exponent m for which Equation (8.10) holds is referred to in several ways:

- the order of a (mod n)
- the exponent to which a belongs (mod n)
- the length of the period generated by a

To see this last point, consider the powers of 7, modulo 19:

$$\begin{aligned} 7^1 &\equiv & &7 \pmod{19} \\ 7^2 &= 49 = 2 \times 19 + 11 \equiv & &11 \pmod{19} \\ 7^3 &= 343 = 18 \times 19 + 1 \equiv & &1 \pmod{19} \\ 7^4 &= 2401 = 126 \times 19 + 7 \equiv & &7 \pmod{19} \end{aligned}$$

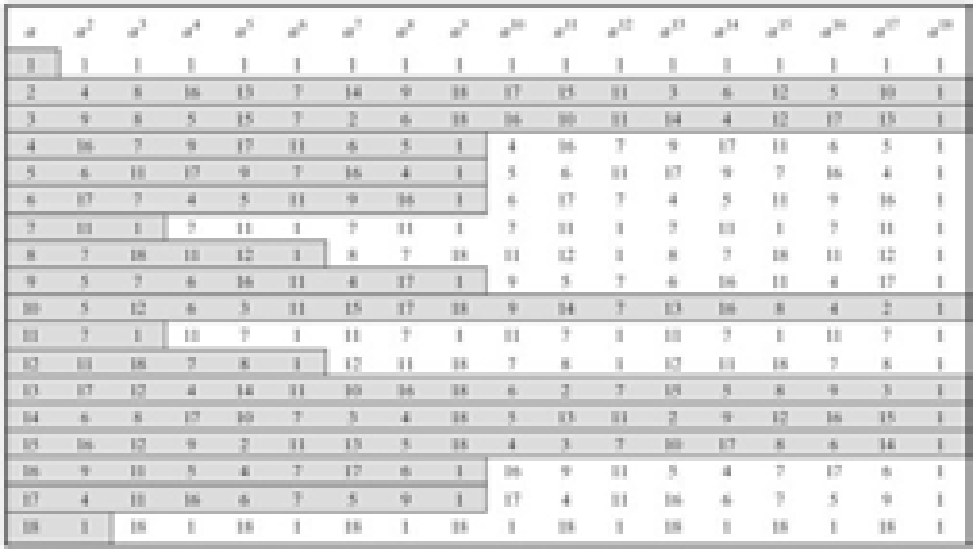
$$7^5 = 16807 = 884 \times 19 + 11 \equiv 11 \pmod{19}$$

There is no point in continuing because the sequence is repeating. This can be proven by noting that $7^3 \equiv 1 \pmod{19}$ and therefore $7^{3+j} \equiv 7^3 7^j \equiv 7^j \pmod{19}$, and hence any two powers of 7 whose exponents differ by 3 (or a multiple of 3) are congruent to each other (mod 19). In other words, the sequence is periodic, and the length of the period is the smallest positive exponent m such that $7^m \equiv 1 \pmod{19}$.

Table 8.3 shows all the powers of a, modulo 19 for all positive a < 19. The length of the sequence for each base value is indicated by shading. Note the following:

1. All sequences end in 1. This is consistent with the reasoning of the preceding few paragraphs.
2. The length of a sequence divides ϕ (19) = 18. That is, an integral number of sequences occur in each row of the table.
3. Some of the sequences are of length 18. In this case, it is said that the base integer a generates (via powers) the set of nonzero integers modulo 19. Each such integer is called a primitive root of the modulus 19.

Table 8.3. Powers of Integers, Modulo 19



More generally, we can say that the highest possible exponent to which a number can belong (mod n) is (n). If a number is of this order, it is referred to as a primitive root of n. The importance of this notion is that if a is a primitive root of n, then its powers

$a, a^2, ..., a^{\phi(n)}$

are distinct (mod n) and are all relatively prime to n. In particular, for a prime number p, if a is a primitive root of p, then

$a, a^2, ..., a^{p1}$

are distinct (mod p). For the prime number 19, its primitive roots are 2, 3, 10, 13, 14, and 15.

Not all integers have primitive roots. In fact, the only integers with primitive roots are those of the form 2, 4, p , and 2p , where p is any odd prime and ϕ is a positive integer. The proof is not simple but can be found in many number theory books.

Logarithms for Modular Arithmetic

With ordinary positive real numbers, the logarithm function is the inverse of exponentiation. An analogous function exists for modular arithmetic.

Let us briefly review the properties of ordinary logarithms. The logarithm of a number is defined to be the power to which some positive base (except 1) must be raised in order to equal the number. That is, for base x and for a value y:

$y = x^{\log x(y)}$

The properties of logarithms include the following:

$\log_x(1) = 0$

$\log_x(x) = 1$

Equation 8-11

$$\log_x(yz) = \log_x(y) + \log_x(z)$$

Equation 8-12

$$\log_x(y^r) = r \times \log_x(y)$$

Consider a primitive root a for some prime number p (the argument can be developed for nonprimes as well). Then we know that the powers of a from 1 through (p 1) produce each integer from 1 through (p 1) exactly once. We also know that any integer b satisfies

$b \equiv r \pmod{p}$ for some r, where $0 \leq r \leq (p\ 1)$

by the definition of modular arithmetic. It follows that for any integer b and a primitive root a of prime number p, we can find a unique exponent i such that

$b \equiv a^i \pmod{p}$ where $0 \leq i \leq (p\ 1)$

This exponent i is referred to as the discrete logarithm of the number b for the base a (mod p). We denote this value as $\text{dlog}_{a,p}(b)$.

Many texts refer to the discrete logarithm as the *index*. There is no generally agreed notation for this concept, much less an agreed name.

Note the following:

Equation 8-13

$$\text{dlog}_{a,p}(1) = 0, \text{ because } a^0 \bmod p = 1 \bmod p = 1$$

Equation 8-14

$\text{dlog}_{a,p}(a) = 1$, because $a^1 \bmod p = a$

Here is an example using a nonprime modulus, n = 9. Here (n) = 6 and a = 2 is a primitive root. We compute the various powers of a and find

$2^0 = 1$ $2^4 \equiv 7 \pmod 9$

$2^1 = 2$ $2^5 \equiv 5 \pmod 9$

$2^2 = 4$ $2^6 \equiv 1 \pmod 9$

$2^3 = 8$

This gives us the following table of the numbers with given discrete logarithms (mod 9) for the root a = 2:

Logarithm	0	1	2	3	4	5
Number	1	2	4	8	7	5

To make it easy to obtain the discrete logarithms of a given number, we rearrange the table:

Number	1	2	4	5	7	8
Logarithm	0	1	2	5	4	3

Now consider

$x = a^{\text{dloga,p(x)}} \bmod p \quad y = a^{\text{dloga,p(y)}} \bmod p$

$xy = a^{\text{dloga,p(xy)}} \bmod p$

Using the rules of modular multiplication,

$$xy \bmod p = [(x \bmod p)(y \bmod p)] \bmod p$$

$$a^{\text{dlog}_{a,p}(xy)} \bmod p = \left[\left(a^{\text{dlog}_{a,p}(x)} \bmod p\right)\left(a^{\text{dlog}_{a,p}(y)} \bmod p\right)\right] \bmod p$$

$$= \left(a^{\text{dlog}_{a,p}(x)+\text{dlog}_{a,p}(y)}\right) \bmod p$$

But now consider Euler's theorem, which states that, for every a and n that are relatively prime:

$a^{\phi(n)} \equiv 1 \pmod{n}$

Any positive integer z can be expressed in the form $z = q + k\phi(n)$, with $0 \leq q < \phi(n)$. Therefore, by Euler's theorem,

$a^z \equiv a^q \pmod{n}$ if $z \equiv q \bmod \phi(n)$

Applying this to the foregoing equality, we have

$\text{dlog}_{a,p}(xy) \equiv [\text{dlog}_{a,p}(x) + \text{dlog}_{a,p}(y)] \pmod{\phi(p)}$

and generalizing,

$\text{dlog}_{a,p}(y^r) \equiv [r \times \text{dlog}_{a,p}(y)] \pmod{\phi(n)}$

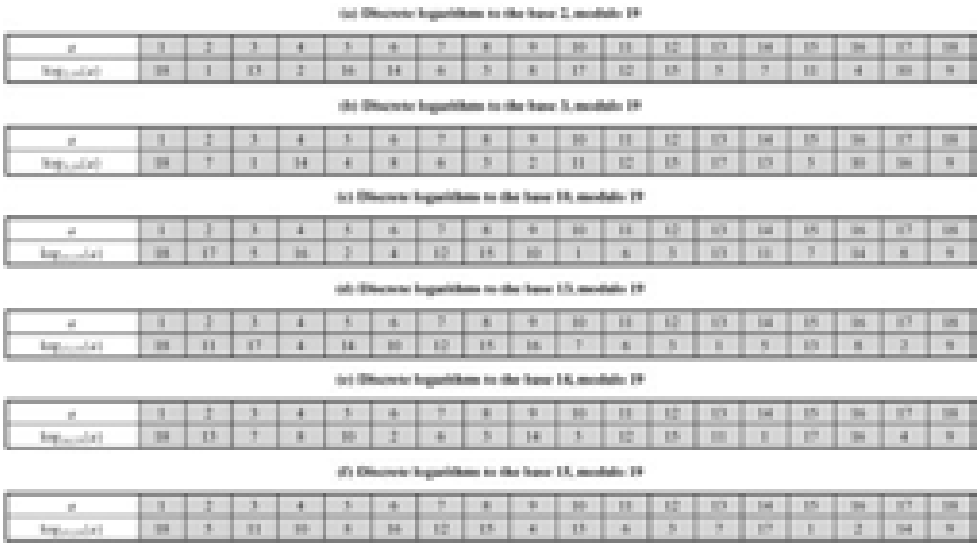
This demonstrates the analogy between true logarithms and discrete logarithms.

Keep in mind that unique discrete logarithms mod m to some base a exist only if a is a primitive root of m.

Table 8.4, which is directly derived from Table 8.3, shows the sets of discrete logarithms that can be defined for modulus 19.

Table 8.4. Tables of Discrete Logarithms, Modulo 19

(This item is displayed on page 252 in the print version)



Calculation of Discrete Logarithms

Consider the equation

$y = g^x \bmod p$

Given g, x, and p, it is a straightforward matter to calculate y. At the worst, we must perform x repeated multiplications, and algorithms exist for achieving greater efficiency.

However, given y, g, and p, it is, in general, very difficult to calculate x (take the discrete logarithm). The difficulty seems to be on the same order of magnitude as that of factoring primes required for RSA. At the time of this writing, the asymptotically fastest known algorithm for taking discrete logarithms modulo a prime number is on the order of :

$$_{e}((\ln p)^{1/3}(\ln(\ln p))^{2/3})$$

which is not feasible for large primes.

Elliptic Curve Arithmetic

Most of the products and standards that use public-key cryptography for encryption and digital signatures use RSA. As we have seen, the key length for secure RSA use has increased over recent years, and this has put a heavier processing load on applications using RSA. This burden has ramifications, especially for electronic commerce sites that conduct large numbers of secure transactions. Recently, a competing system has begun to challenge RSA: elliptic curve cryptography (ECC). Already, ECC is showing up in standardization efforts, including the IEEE P1363 Standard for Public-Key Cryptography.

The principal attraction of ECC, compared to RSA, is that it appears to offer equal security for a far smaller key size, thereby reducing processing overhead. On the other hand, although the theory of ECC has been around for some time, it is only recently that products have begun to appear and that there has been sustained cryptanalytic interest in probing for weaknesses. Accordingly, the confidence level in ECC is not yet as high as that in RSA.

ECC is fundamentally more difficult to explain than either RSA or Diffie-Hellman, and a full mathematical description is beyond the scope of this book. This section and the next give some background on elliptic curves and ECC. We begin with a brief review of the concept of abelian group. Next, we examine the concept of elliptic curves defined over the real numbers.

This is followed by a look at elliptic curves defined over finite fields. Finally, we are able to examine elliptic curve ciphers.

Abelian Groups

that an abelian group G, sometimes denoted by {G, • }, is a set of elements with a binary operation, denoted by •, that associates to each ordered pair (a, b) of elements in G an element (a • b) in G, such that the following axioms are obeyed:

The operator • is generic and can refer to addition, multiplication, or some other mathematical operation.

(A1) Closure:	If a and b belong to G, then a • b is also in G.
(A2) Associative:	a • (b • c) = (a • b) • c for all a, b, c in G.
(A3) Identity element:	There is an element e in G such that a • e = e • a = a for all a in G.
(A4) Inverse element:	For each a in G there is an element a' in G such that a • a' = a' • a = e.
(A5) Commutative:	a • b = b • a for all a, b in G.

A number of public-key ciphers are based on the use of an abelian group. For example, Diffie-Hellman key exchange involves multiplying pairs of nonzero integers modulo a prime number q. Keys are generated by exponentiation over the group, with exponentiation defined as repeated multiplication. For example, $a^k \bmod q = \underbrace{(a \times a \times \ldots \times a)}_{k \text{ times}} \bmod q$. To attack Diffie-Hellman, the attacker must determine k given a and a^k; this is the discrete log problem.

For elliptic curve cryptography, an operation over elliptic curves, called addition, is used. Multiplication is defined by repeated addition. For example, $a \times k = \underbrace{(a + a + \ldots + a)}_{k \text{ times}}$, where the addition is performed over an elliptic

curve. Cryptanalysis involves determining k given a and (a x k).

An elliptic curve is defined by an equation in two variables, with coefficients. For cryptography, the variables and coefficients are restricted to

elements in a finite field, which results in the definition of a finite abelian group. Before looking at this, we first look at elliptic curves in which the variables and coefficients are real numbers. This case is perhaps easier to visualize.

Elliptic Curves over Real Numbers

Elliptic curves are not ellipses. They are so named because they are described by cubic equations, similar to those used for calculating the circumference of an ellipse. In general, cubic equations for elliptic curves take the form

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

where a, b, c, d, and e are real numbers and x and y take on values in the real numbers. For our purpose, it is sufficient to limit ourselves to equations of the form

Equation 10-1

$$y^2 = x^3 + ax + b$$

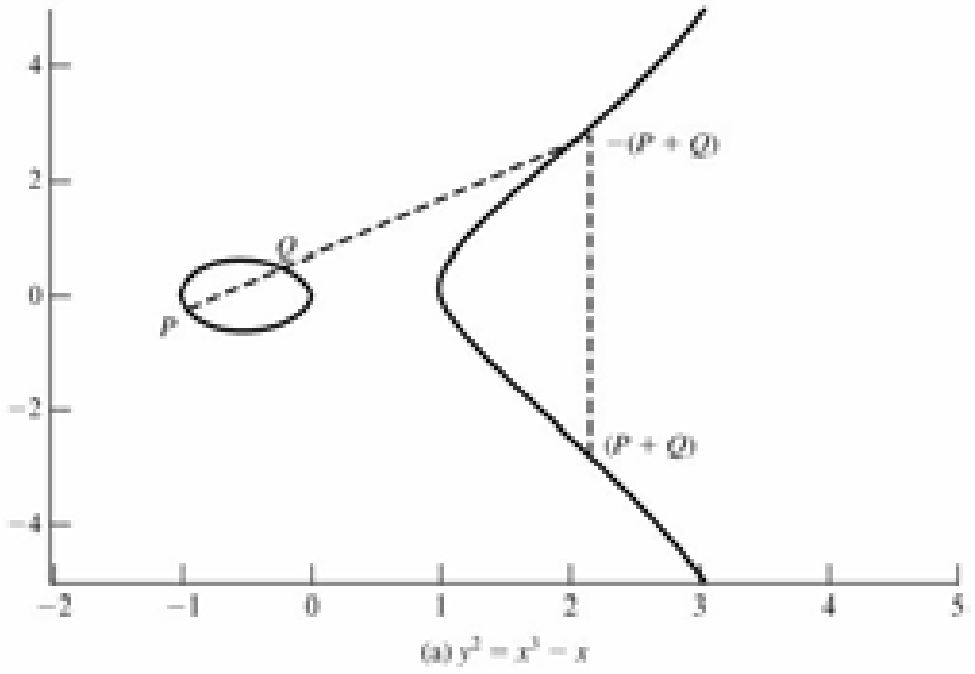
Such equations are said to be cubic, or of degree 3, because the highest exponent they contain is a 3. Also included in the definition of an elliptic curve is a single element denoted O and called the point at infinity or the *zero point*, which we discuss subsequently. To plot such a curve, we need to compute

$$y = \sqrt{x^3 + ax + b}$$

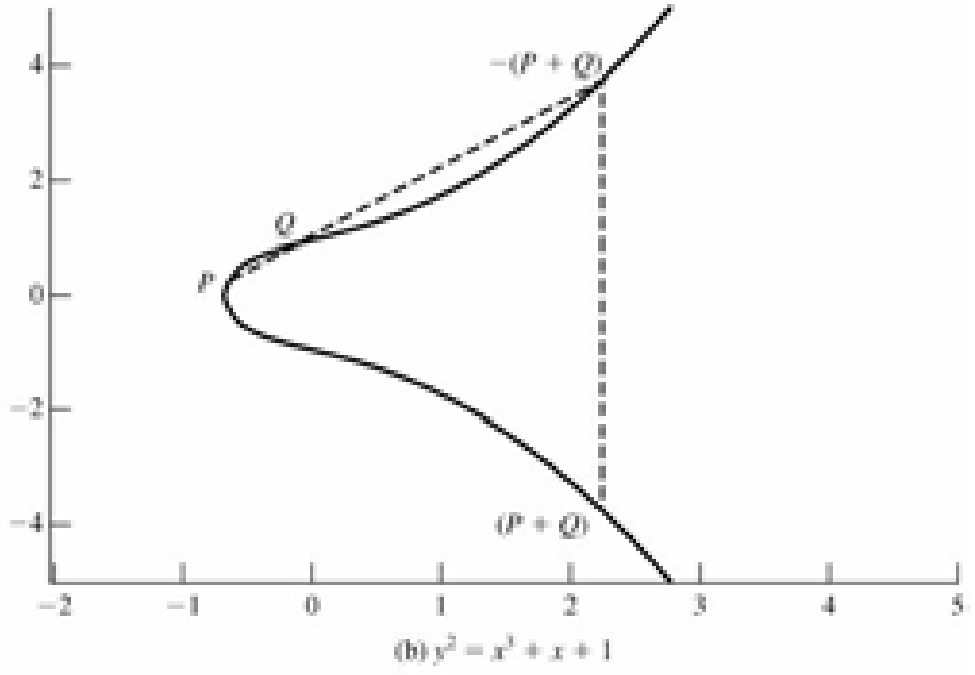
For given values of a and b, the plot consists of positive and negative values of y for each value of x. Thus each curve is symmetric about y = 0. Figure 10.9 shows two examples of elliptic curves. As you can see, the formula sometimes produces weird-looking curves.

Figure 10.9. Example of Elliptic Curves

(This item is displayed on page 304 in the print version)



(a) $y^2 = x^3 - x$



(b) $y^2 = x^3 + x + 1$

Now, consider the set of points E(a, b) consisting of all of the points (x, y) that satisfy Equation (10.1) together with the element O. Using a different value of the pair (a, b) results in a different set E(a, b). Using this

terminology, the two curves in Figure 10.9 depict the sets E(1,0) and E(1, 1), respectively.

Geometric Description of Addition

It can be shown that a group can be defined based on the set E(a, b) for specific values of a and b in Equation (10.1), provided the following condition is met:

Equation 10-2

$$4a^3 + 27b^2 \neq 0$$

To define the group, we must define an operation, called addition and denoted by +, for the set E(a, b), where a and b satisfy Equation (10.2). In geometric terms, the rules for addition can be stated as follows: If three points on an elliptic curve lie on a straight line, their sum is O. From this definition, we can define the rules of addition over an elliptic curve:

1. O serves as the additive identity. Thus O = O; for any point P on the elliptic curve, P + O = P. In what follows, we assume P ≠ O and Q ≠ O.
2. The negative of a point P is the point with the same x coordinate but the negative of the y coordinate; that is, if P = (x, y), then P = (x, y). Note that these two points can be joined by a vertical line. Note that P + (P) = P P = O.
3. To add two points P and Q with different x coordinates, draw a straight line between them and find the third point of intersection R. It is easily seen that there is a unique point R that is the point of intersection (unless the line is tangent to the curve at either P or Q, in which case we take R = P or R = Q, respectively). To form a group structure, we need to define addition on these three points as follows: P + Q = R. That is, we define P + Q to be the mirror image (with respect to the x axis) of the third point of intersection. Figure 10.9 illustrates this construction.
4. The geometric interpretation of the preceding item also applies to two points, P and P, with the same x coordinate. The points are joined by a vertical line, which can be viewed as also intersecting the curve at the

infinity point. We therefore have P + (P) = O, consistent with item (2).

5. To double a point Q, draw the tangent line and find the other point of intersection S. Then Q + Q = 2Q = S.

With the preceding list of rules, it can be shown that the set E(a, b) is an abelian group.

Algebraic Description of Addition

In this subsection we present some results that enable calculation of additions over elliptic curves. For two distinct points P = (x_P, y_P) and Q = (x_Q, y_P) that are not negatives of each other, the slope of the line l that joins them is Δ = (y_Q y_P). There is exactly one other point where l intersects the elliptic curve, and that is the negative of the sum of P and Q. After some algebraic manipulation, we can express the sum R = P + Q as follows:

Equation 10-3

$$x_R = \Delta^2 - x_P - x_Q$$
$$y_R = -y_P + \Delta(x_P - x_R)$$

We also need to be able to add a point to itself: P + P = 2P = R. When $y_P \neq 0$, the expressions are

Equation 10-4

$$x_R = \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P$$
$$y_R = \left(\frac{3x_P^2 + a}{2y_P}\right)(x_P - x_R) - y_P$$

Elliptic Curves over Z_p

Elliptic curve cryptography makes use of elliptic curves in which the variables and coefficients are all restricted to elements of a finite field. Two families of elliptic curves are used in cryptographic applications: prime curves over Z_p and binary curves over GF (2^m). For a prime curve over Z_p, we use a cubic equation in which the variables and coefficients all take on values in the set of integers from 0 through p 1 and in which calculations are performed modulo p. For a binary curve defined over GF(2^m), the variables and coefficients all take on values in GF(2^n) and in calculations are performed over GF(2^n). points out that prime curves are best for software applications, because the extended bit-fiddling operations needed by binary curves are not required; and that binary curves are best for hardware applications, where it takes remarkably few logic gates to create a powerful, fast cryptosystem. We examine these two families in this section and the next.

There is no obvious geometric interpretation of elliptic curve arithmetic over finite fields. The algebraic interpretation used for elliptic curve arithmetic over real numbers does readily carry over, and this is the approach we take.

For elliptic curves over Z_p, as with real numbers, we limit ourselves to equations of the form of Equation (10.1), but in this case with coefficients and variables limited to Z_p:

Equation 10-5

$$y^2 \bmod p = (x^3 + ax + b) \bmod p$$

For example, Equation (10.5) is satisfied for a = 1, b = 1, x = 9, y = 9, y = 7, p = 23:

$7^2 \bmod 23 = (9^3 + 9 + 1) \bmod 23$

$49 \bmod 23 = 739 \bmod 23$

$3 = 3$

Now consider the set E_p (a, b) consisting of all pairs of integers (x, y) that satisfy Equation (10.5), together with a point at infinity O. The coefficients a and b and the variables x and y are all elements of Z_p.

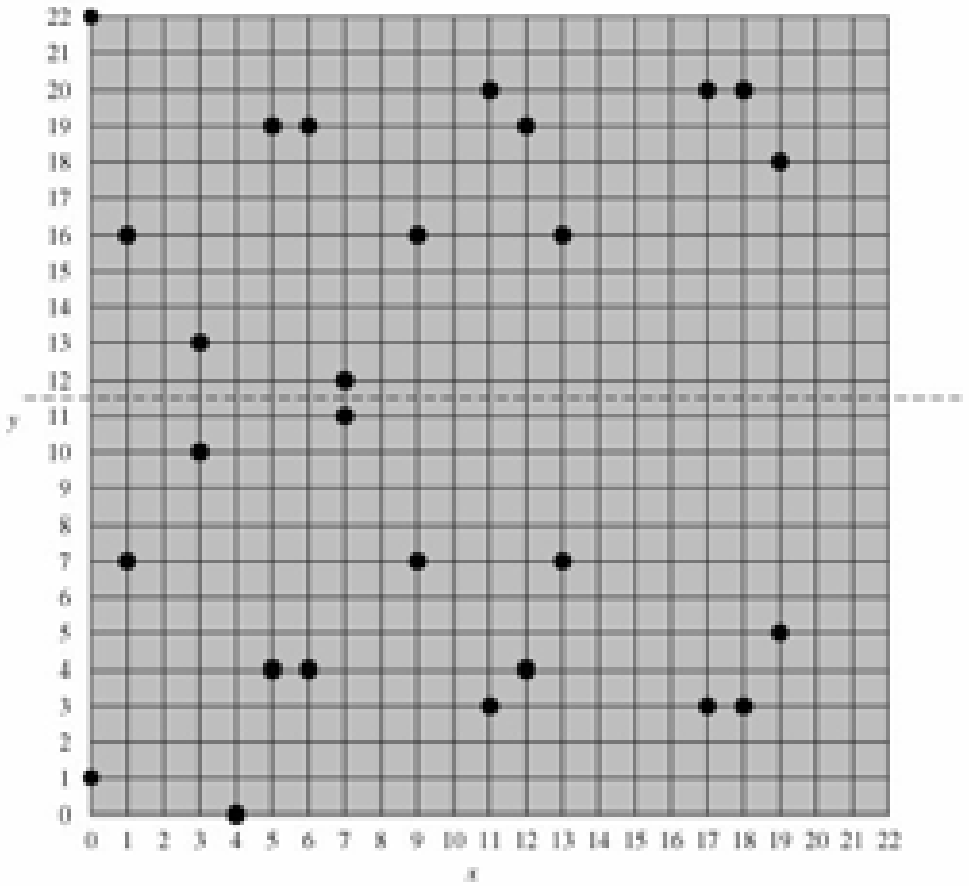
For example, let p = 23 and consider the elliptic curve $y^2 = x^3 + x + 1$. In this case, a = b = 1. Note that this equation is the same as that of Figure 10.9b. The figure shows a continuous curve with all of the real points that satisfy the equation. For the set $E_{23}(1, 1)$, we are only interested in the nonnegative integers in the quadrant from (0, 0) through (p 1, p 1) that satisfy the equation mod p. Table 10.1 lists the points (other than O) that are part of $E_{23}(1,1)$. Figure 10.10 plots the points of $E_{23}(1,1)$; note that the points, with one exception, are symmetric about y = 11.5.

Table 10.1. Points on the Elliptic Curve $E_{23}(1,1)$

(0, 1)	(6, 4)	(12, 19)
(0, 22)	(6, 19)	(13, 7)
(1, 7)	(7, 11)	(13, 16)
(1, 16)	(7, 12)	(17, 3)
(3, 10)	(9, 7)	(17, 20)
(3, 13)	(9, 16)	(18, 3)
(4, 0)	(11, 3)	(18, 20)
(5, 4)	(11, 20)	(19, 5)
(5, 19)	(12, 4)	(19, 18)

Figure 10.10. The Elliptic Curve E_{23} (1, 1)

(This item is displayed on page 307 in the print version)



It can be shown that a finite abelian group can be defined based on the set $E_p(a, b)$ provided that $(x^3 + ax + b) \bmod p$ has no repeated factors. This is equivalent to the condition

Equation 10-6

$$(4a^3 + 27b^2) \bmod p \neq 0 \bmod p$$

Note that Equation (10.6) has the same form as Equation (10.2).

The rules for addition over $E_p(a, b)$ correspond to the algebraic technique described for elliptic curves defined over real number. For all points P, Q

Ep(a, b);

1. P + O = P.

   If P = (x_P, y_P) then P + (x_P, y_P) = O. The point (x_P, y_P) is the negative of P, denoted as P. For example, in E_{23} (1, 1), for P = (13, 7), we have P = (13, 7). But 7 mod 23 = 16. Therefore, P = (13, 16), which is also in E_{23}(1,1).

2. If P = (x_P, y_Q) and Q = (x_Q, y_Q) with P ≠Q, then R = P + Q = (x_R, y_R) is determined by the following rules:

   $x_R = (\lambda^2 x_P x_Q) \bmod p$

   $y_R = (\lambda (x_P x_R) y_P) \bmod p$

   where

$$\lambda = \begin{cases} \left(\frac{y_Q - y_P}{x_Q - x_P}\right) \bmod p & \text{if } P \neq Q \\ \left(\frac{3x_P^2 + a}{2y_P}\right) \bmod p & \text{if } P = Q \end{cases}$$

3. Multiplication is defined as repeated addition; for example, 4P = P + P + P + P.

   For example, let P = (3,10) and Q = (9,7) in E_{23}(1,1). Then

$$\lambda = \left(\frac{7 - 10}{9 - 3}\right) \bmod 23 = \left(\frac{-3}{6}\right) \bmod 23 = \left(\frac{-1}{2}\right) \bmod 23 = 11$$

   $x_R = (11^2\ 3\ 9) \bmod 23 = 17$

   $y_R = (11(3\ 17)\ 10) \bmod 23 = 164 \bmod 23 = 20$

So P + Q = (17, 20). To find 2P,

$$\lambda = \left(\frac{3(3^2) + 1}{2 \times 10}\right) \bmod 23 = \left(\frac{5}{20}\right) \bmod 23 = \left(\frac{1}{4}\right) \bmod 23 = 6$$

The last step in the preceding equation involves taking the multiplicative inverse of 4 in Z_{23}. To confirm, note that (6 x 4) mod 23 = 24 mod 23 = 1.

$x_R = (6^2 \ 3 \ 3) \bmod 23 = 30 \bmod 23 = 7$

$y_R = (6(3 \ 7) \ 10) \bmod 23 = (\ 34) \bmod 23 = 12$

and 2P = (7, 12).

For determining the security of various elliptic curve ciphers, it is of some interest to know the number the number of points in a finite abelian group defined over an elliptic curve. In the case of the finite group $E_p(a,b)$, the number of points N is bounded by

$$p + 1 - 2\sqrt{p} \leq N \leq p + 1 + 2\sqrt{p}$$

Note that the number of points in $E_p(a, b)$ is approximately equal to the number of elements in Zp, namely p elements.

Elliptic Curves over GF(2^m)

a finite field GF(2^m) consists of 2^m elements, together with addition and multiplication operations that can be defined over polynomials. For elliptic curves over GF(2^m), we use a cubic equation in which the variables and coefficients all take on values in GF(2^m), for some number m, and in which calculations are performed using the rules of arithmetic in GF(2^m).

It turns out that the form of cubic equation appropriate for cryptographic applications for elliptic curves is somewhat different for GF(2^m) than for Z_p. The form is

Equation 10-7

$$y^2 + xy = x^3 + ax^2 + b$$

where it is understood that the variables x and y and the coefficients a and b are elements of GF(2^m) of and that calculations are performed in GF(2^m).

Now consider the set $E_2^m(a, b)$ consisting of all pairs of integers (x, y) that satisfy Equation (10.7), together with a point at infinity O.

For example, let us use the finite field GF(2^4) with the irreducible polynomial $f(x) = x^4 + x + 1$. This yields a generator that satisfies $f(g) = 0$, with a value of $g^4 = g + 1$, or in binary 0010. We can develop the powers of g as follows:

$g^0 = 0001$	$g^4 = 0011$	$g^8 = 0101$	$g^{12} = 1111$
$g^1 = 0010$	$g^5 = 0110$	$g^9 = 1010$	$g^{13} = 1101$
$g^2 = 0100$	$g^6 = 1100$	$g^{10} = 0111$	$g^{14} = 1001$
$g^3 = 1000$	$g^7 = 1011$	$g^{11} = 1110$	$g^{15} = 0001$

For example, $g^5 = (g^4)(g) = g^2 + g = 0110$.

Now consider the elliptic curve $y^2 + xy = x^3 + g^4x^2 + 1$. In this case $a = g^4$ and $b = g^0 = 1$. One point that satisfies this equation is (g^5, g^3):

$(g^3)^2 + (g^5)(g^3) = (g^5)^3 + (g^4)(g^5)^2 + 1$

$g^6 + g^8 = g^{15} + g^{14} + 1$

$1100 + 0101 = 0001 + 1001 + 0001$

$1001 = 1001$

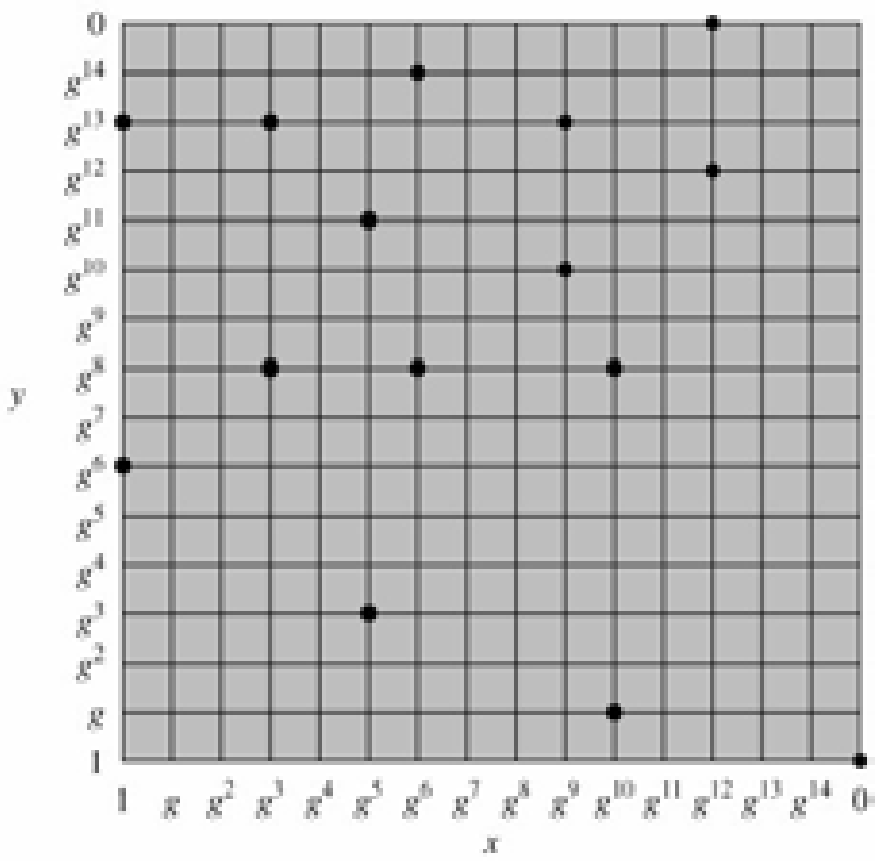
Table 10.2 lists the points (other than O) that are part of $E_2^4(g^4, 1)$. Figure 10.11 plots the points of $E_2^4(g^4, 1)$.

Table 10.2. Points on the Elliptic Curve $E_2^4(g^4, 1)$

(0, 1)	(g^5, g^3)	(g^9, g^{13})
$(1, g^6)$	(g^5, g^{11})	(g^{10}, g)
$(1, g^{13})$	$g^6, g^8)$	(g^{10}, g^8)
(g^3, g^8)	(g^6, g^{14})	$(g^{12},0)$

Table 10.2. Points on the Elliptic Curve E_2^4 (g^4, 1)		
(0, 1)	(g^5, g^3)	(g^9, g^{13})
(g^3, g^{13})	(g^9, g^{10})	(g^{12}, g^{12})

Figure 10.11. The Elliptic Curve E_2^4(g^4, 1)



It can be shown that a finite abelian group can be defined based on the set $E_{2m}(a, b)$, provided that $b \neq 0$. The rules for addition can be stated as follows. For all points P, Q $\neq E_2^m(a, b)$:

1. P + O = P.

   If P = (x_P, y_P), then P + (x_P, $x_p + y_P$) = O. The point (x_P, $x_P + y_P$) is the negative of P, denoted as P.

2. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq Q$ and $P \neq Q$, then $R = P + Q = (x_R, y_R)$ is determined by the following rules:

   $x_R = \lambda^2 + \lambda + x_P + x_Q + a$

   $y_R = \lambda (x_P + x_R) + x_R + y_P$

   where

   $$\lambda = \frac{y_Q + y_P}{x_Q + x_P}$$

3. If $= (x_P, y_P)$ then $R = 2P = (x_R, y_R)$ is determined by the following rules:

   $$x_R = \lambda^2 + \lambda + a$$
   $$y_R = x_P^2 + (\lambda + 1)x_R$$

   where

   $$\lambda = x_P + \frac{y_P}{x_P}$$

Elliptic Curve Cryptography

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple additions are the counterpart of modular exponentiation. To form a cryptographic system using elliptic curves, we need to find a "hard problem" corresponding to factoring the product of two primes or taking the discrete logarithm.

Consider the equation Q = kP where Q, P $E_p(a, b)$ and k < p. It is relatively easy to calculate Q given k and P, but it is relatively hard to determine k given Q and P. This is called the discrete logarithm problem for elliptic curves.

Consider the group $E_{23}(9, 17)$. This is the group defined by the equation $y^2 \bmod 23 = (x^3 + 9x + 17) \bmod 23$. What is the discrete logarithm k of Q = (4, 5) to the base P = (16.5)? The brute-force method is to compute multiples of P until Q is found.

Thus

P = (16, 5); 2P = (20, 20); 3P = (14, 14); 4P = (19, 20); 5P = (13, 10); 6P = (7, 3); 7P = (8, 7); 8P (12, 17); 9P = (4, 5).

Because 9P = (4, 5) = Q, the discrete logarithm Q = (4, 5) to the base P = (16, 5) is k = 9. In a real application, k would be so large as to make the brute-force approach infeasible.

In the remainder of this section, we show two approaches to ECC that give the flavor of this technique.

Analog of Diffie-Hellman Key Exchange

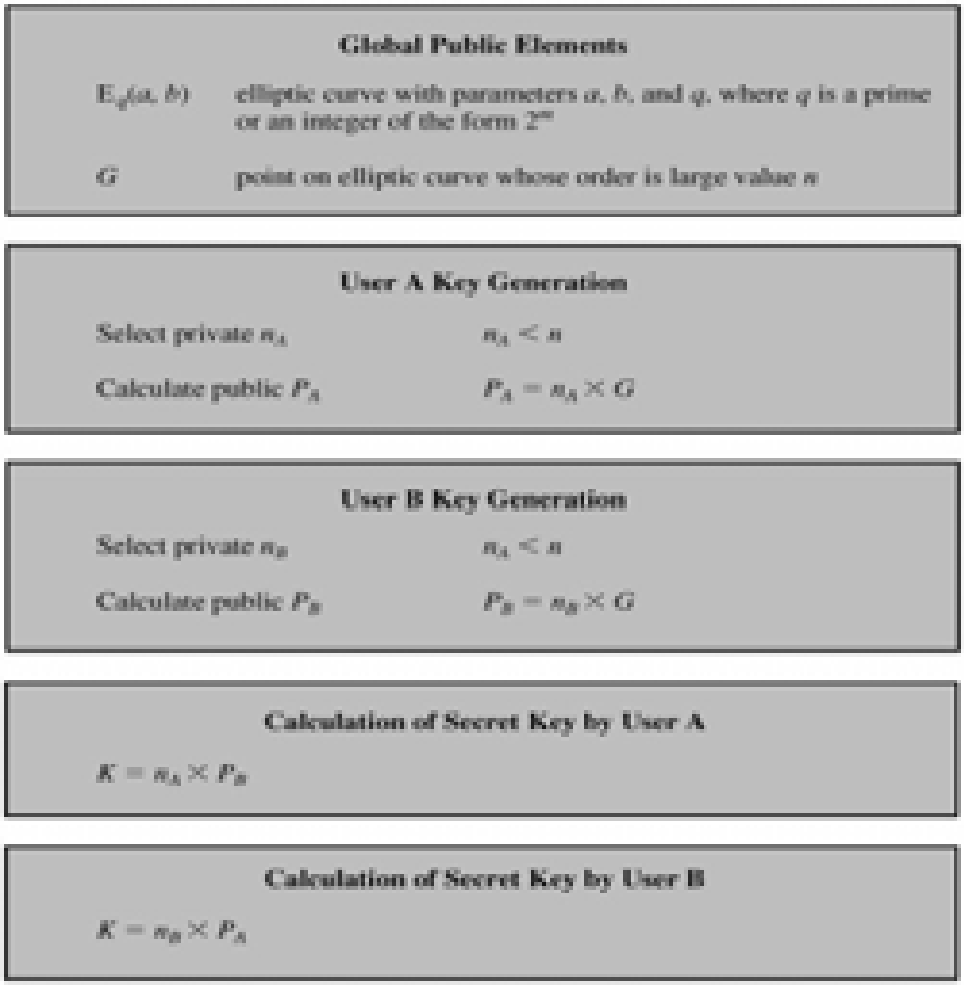
Key exchange using elliptic curves can be done in the following manner. First pick a large integer q, which is either a prime number p or an integer of the form 2^m and elliptic curve parameters a and b for Equation (10.5) or Equation (10.7). This defines the elliptic group of points $E_q(a, b)$. Next, pick a base point $G = (x_1, y_1)$ in $E_p(a, b)$ whose order is a very large value n. The order n of a point G on an elliptic curve is the smallest positive integer n

such that nG = O. $E_q(a, b)$ and G are parameters of the cryptosystem known to all participants.

A key exchange between users A and B can be accomplished as follows (Figure 10.12):

1. A selects an integer n_A less than n. This is A's private key. A then generates a public key $P_A = n_A \times G$; the public key is a point in Eq(a, b).
2. B similarly selects a private key n_B and computes a public key P_B.
3. A generates the secret key $K = n_A \times P_B$. B generates the secret key $K = n_B \times P_A$.

Figure 10.12. ECC Diffie-Hellman Key Exchange



The two calculations in step 3 produce the same result because

$n_A \times P_B = n_A \times (n_B \times G) = n_B \times (n_A \times G) = n_B \times P_A$

To break this scheme, an attacker would need to be able to compute k given G and kG, which is assumed hard.

As an example, take p = 211; Ep(0, 4), which is equivalent to the curve $y^2 = x^3$ 4; and G = (2, 2). One can calculate that 240G = O. A's private key is n_A = 121, so A's public key is P_A = 121(2, 2) = (115, 48). B's private key is n_B = 203, so B's public key is 203(2, 2) = (130, 203). The shared secret key is 121(130, 203) = 203(115, 48) = (161, 69).

Note that the secret key is a pair of numbers. If this key is to be used as a session key for conventional encryption, then a single number must be generated. We could simply use the x coordinates or some simple function of the x coordinate.

Elliptic Curve Encryption/Decryption

Several approaches to encryption/decryption using elliptic curves have been analyzed in the literature. In this subsection we look at perhaps the simplest. The first task in this system is to encode the plaintext message m to be sent as an x-y point P_m. It is the point P_m that will be encrypted as a ciphertext and subsequently decrypted. Note that we cannot simply encode the message as the x or y coordinate of a point, because not all such coordinates are in Eq(a, b); for example, see Table 10.1. Again, there are several approaches to this encoding, which we will not address here, but suffice it to say that there are relatively straightforward techniques that can be used.

As with the key exchange system, an encryption/decryption system requires a point G and an elliptic group Eq(a, b) as parameters. Each user A selects a private key n_A and generates a public key $P_A = n_A \times G$.

To encrypt and send a message P_m to B, A chooses a random positive integer k and produces the ciphertext C_m consisting of the pair of points:

$C_m = \{kG, P_m + kP_B\}$

Note that A has used B's public key P_B. To decrypt the ciphertext, B multiplies the first point in the pair by B's secret key and subtracts the result from the second point:

$$P_m + kP_B \; n_B(kG) = P_m + k(n_BG) \; n_B(kG) = P_m$$

A has masked the message P_m by adding kP_B to it. Nobody but A knows the value of k, so even though P_B is a public key, nobody can remove the mask kP_B. However, A also includes a "clue," which is enough to remove the mask if one knows the private key n_B. For an attacker to recover the message, the attacker would have to compute k given G and kG, which is assumed hard.

As an example of the encryption process (taken from [KOBL94]), take p = 751; $E_p(1, 188)$, which is equivalent to the curve $y^2 = x^3 x + 188$; and G = (0, 376). Suppose that A wishes to send a message to B that is encoded in the elliptic point P_m = (562, 201) and that A selects the random number k = 386. B's public key is P_B = (201, 5). We have 386(0, 376) = (676, 558), and (562, 201) + 386(201, 5) = (385, 328). Thus A sends the cipher text {(676, 558), (385, 328)}.

Security of Elliptic Curve Cryptography

The security of ECC depends on how difficult it is to determine k given kP and P. This is referred to as the elliptic curve logarithm problem. The fastest known technique for taking the elliptic curve logarithm is known as the Pollard rho method. Table 10.3 compares various algorithms by showing comparable key sizes in terms of computational effort for cryptanalysis. As can be seen, a considerably smaller key size can be used for ECC compared to RSA. Furthermore, for equal key lengths, the computational effort required for ECC and RSA is comparable . Thus, there is a computational advantage to using ECC with a shorter key length than a comparably secure RSA.

Table 10.3. Comparable Key Sizes in Terms of Computational Effort for Cryptanalysis

Symmetric Scheme (key size in bits)	ECC-Based Scheme (size of n in bits)	RSA/DSA (modulus size in bits)
56	112	512

Table 10.3. Comparable Key Sizes in Terms of Computational Effort for Cryptanalysis		
Symmetric Scheme (key size in bits)	ECC-Based Scheme (size of n in bits)	RSA/DSA (modulus size in bits)
80	160	1024
112	224	2048
128	256	3072
92	384	7680
256	512	15360
Source: Certicom		

Evaluation Criteria For AES

The Origins of AES

in 1999, NIST issued a new version of its DES standard (FIPS PUB 46-3) that indicated that DES should only be used for legacy systems and that triple DES (3DES) be used. 3DES has two attractions that assure its widespread use over the next few years. First, with its 168-bit key length, it overcomes the vulnerability to brute-force attack of DES. Second, the underlying encryption algorithm in 3DES is the same as in DES. This algorithm has been subjected to more scrutiny than any other encryption algorithm over a longer period of time, and no effective cryptanalytic attack based on the algorithm rather than brute force has been found. Accordingly, there is a high level of confidence that 3DES is very resistant to cryptanalysis. If security were the only consideration, then 3DES would be an appropriate choice for a standardized encryption algorithm for decades to come.

The principal drawback of 3DES is that the algorithm is relatively sluggish in software. The original DES was designed for mid-1970s hardware implementation and does not produce efficient software code. 3DES, which has three times as many rounds as DES, is correspondingly slower. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable.

Because of these drawbacks, 3DES is not a reasonable candidate for long-term use. As a replacement, NIST in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have a security strength equal to or better than 3DES and significantly improved efficiency. In addition to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits.

In a first round of evaluation, 15 proposed algorithms were accepted. A second round narrowed the field to 5 algorithms. NIST completed its evaluation process and published a final standard (FIPS PUB 197) in November of 2001. NIST selected Rijndael as the proposed AES algorithm. The two researchers who developed and submitted Rijndael for the AES are both cryptographers from Belgium: Dr. Joan Daemen and Dr. Vincent Rijmen.

Ultimately, AES is intended to replace 3DES, but this process will take a number of years. NIST anticipates that 3DES will remain an approved algorithm (for U.S. government use) for the foreseeable future.

AES Evaluation

It is worth examining the criteria used by NIST to evaluate potential candidates. These criteria span the range of concerns for the practical application of modern symmetric block ciphers. In fact, two set of criteria evolved. When NIST issued its original request for candidate algorithm nominations in 1997 , the request stated that candidate algorithms would be compared based on the factors shown in Table 5.1 (ranked in descending order of relative importance). The three categories of criteria were as follows:

- Security: This refers to the effort required to cryptanalyze an algorithm. The emphasis in the evaluation was on the practicality of the attack. Because the minimum key size for AES is 128 bits, brute-force attacks with current and projected technology were considered impractical. Therefore, the emphasis, with respect to this point, is cryptanalysis other than a brute-force attack.

  Cost: NIST intends AES to be practical in a wide range of applications. Accordingly, AES must have high computational

efficiency, so as to be usable in high-speed applications, such as broadband links.

- Algorithm and implementation characteristics: This category includes a variety of considerations, including flexibility; suitability for a variety of hardware and software implementations; and simplicity, which will make an analysis of security more straightforward.

Table 5.1. NIST Evaluation Criteria for AES (September 12, 1997)

SECURITY
• Actual security: compared to other submitted algorithms (at the same key and block size). • Randomness: the extent to which the algorithm output is indistinguishable from a random permutation on the input block. • Soundness: of the mathematical basis for the algorithm's security. • Other security factors: raised by the public during the evaluation process, including any attacks which demonstrate that the actual security of the algorithm is less than the strength claimed by the submitter.
COST
• Licensing requirements: NIST intends that when the AES is issued, the algorithm(s) specified in the AES shall be available on a worldwide, non-exclusive, royalty-free basis. • Computational efficiency: The evaluation of computational efficiency will be applicable to both hardware and software implementations. Round 1 analysis by NIST will focus primarily on software implementations and specifically on one key-block size combination (128-128); more attention will be paid to hardware implementations and other supported key-block size combinations during Round 2 analysis. Computational efficiency essentially refers to the speed of the algorithm. Public comments on each algorithm's efficiency (particularly for various platforms and applications) will also be taken into consideration by NIST. • Memory requirements: The memory required to implement a candidate algorithmfor both hardware and software implementations of the algorithmwill also be considered during the evaluation process. Round 1 analysis by NIST will focus primarily on software implementations; more attention will be paid to hardware implementations during Round 2. Memory requirements will include such factors as gate counts for hardware implementations, and code size and RAM requirements for software implementations.
ALGORITHM AND IMPLEMENTATION CHARACTERISTICS
• Flexibility: Candidate algorithms with greater flexibility will meet the needs of

Table 5.1. NIST Evaluation Criteria for AES (September 12, 1997)

SECURITY
more users than less flexible ones, and therefore, inter alia, are preferable. However, some extremes of functionality are of little practical application (e.g., extremely short key lengths); for those cases, preference will not be given. Some examples of flexibility may include (but are not limited to) the following: a. The algorithm can accommodate additional key- and block-sizes (e.g., 64-bit block sizes, key sizes other than those specified in the Minimum Acceptability Requirements section, [e.g., keys between 128 and 256 that are multiples of 32 bits, etc.]) b. The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g., 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.). c. The algorithm can be implemented as a stream cipher, message authentication code (MAC) generator, pseudorandom number generator, hashing algorithm, etc. • Hardware and software suitability: A candidate algorithm shall not be restrictive in the sense that it can only be implemented in hardware. If one can also implement the algorithm efficiently in firmware, then this will be an advantage in the area of flexibility. • Simplicity: A candidate algorithm shall be judged according to relative simplicity of design.

Using these criteria, the initial field of 21 candidate algorithms was reduced first to 15 candidates and then to 5 candidates. By the time that a final evaluation had been done the evaluation criteria, as described in , had evolved. The following criteria were used in the final evaluation:

- General security: To assess general security, NIST relied on the public security analysis conducted by the cryptographic community. During the course of the three-year evaluation process, a number of cryptographers published their analyses of the strengths and weaknesses of the various candidates. There was particular emphasis on analyzing the candidates with respect to known attacks, such as differential and linear cryptanalysis. However, compared to the analysis of DES, the amount of time and the number of cryptographers devoted to analyzing Rijndael are quite limited. Now that a single AES cipher has been chosen, we can expect to see a more extensive security analysis by the cryptographic community.

- Software implementations: The principal concerns in this category are execution speed, performance across a variety of platforms, and variation of speed with key size.
- Restricted-space environments: In some applications, such as smart cards, relatively small amounts of random-access memory (RAM) and/or read-only memory (ROM) are available for such purposes as code storage (generally in ROM); representation of data objects such as S-boxes (which could be stored in ROM or RAM, depending on whether pre-computation or Boolean representation is used); and subkey storage (in RAM).
- Hardware implementations: Like software, hardware implementations can be optimized for speed or for size. However, in the case of hardware, size translates much more directly into cost than is usually the case for software implementations. Doubling the size of an encryption program may make little difference on a general-purpose computer with a large memory, but doubling the area used in a hardware device typically more than doubles the cost of the device.
- Attacks on implementations: The criterion of general security, discussed in the first bullet, is concerned with cryptanalytic attacks that exploit mathematical properties of the algorithms. There is another class of attacks that use physical measurements conducted during algorithm execution to gather information about quantities such as keys. Such attacks exploit a combination of intrinsic algorithm characteristics and implementation-dependent features. Examples of such attacks are timing attacks and power analysis. The basic idea behind power analysis is the observation that the power consumed by a smart card at any particular time during the cryptographic operation is related to the instruction being executed and to the data being processed. For example, multiplication consumes more power than addition, and writing 1s consumes more power than writing 0s.

  Encryption versus decryption: This criterion deals with several issues related to considerations of both encryption and decryption. If the encryption and decryption algorithms differ, then extra space is needed for the decryption. Also, whether the two algorithms are the same or not, there may be timing differences between encryption and decryption.

- Key agility: Key agility refers to the ability to change keys quickly and with a minimum of resources. This includes both subkey

computation and the ability to switch between different ongoing security associations when subkeys may already be available.

- Other versatility and flexibility: indicates two areas that fall into this category. Parameter flexibility includes ease of support for other key and block sizes and ease of increasing the number of rounds in order to cope with newly discovered attacks. Implementation flexibility refers to the possibility of optimizing cipher elements for particular environments.
- Potential for instruction-level parallelism: This criterion refers to the ability to exploit ILP features in current and future processors.

Table 5.2 shows the assessment that NIST provided for Rijndael based on these criteria.

Table 5.2. Final NIST Evaluation of Rijndael (October 2, 2000)

General Security
Rijndael has no known security attacks. Rijndael uses S-boxes as nonlinear components. Rijndael appears to have an adequate security margin, but has received some criticism suggesting that its mathematical structure may lead to attacks. On the other hand, the simple structure may have facilitated its security analysis during the timeframe of the AES development process.
Software Implementations
Rijndael performs encryption and decryption very well across a variety of platforms, including 8-bit and 64-bit platforms, and DSPs. However, there is a decrease in performance with the higher key sizes because of the increased number of rounds that are performed. Rijndael's high inherent parallelism facilitates the efficient use of processor resources, resulting in very good software performance even when implemented in a mode not capable of interleaving. Rijndael's key setup time is fast.
Restricted-Space Environments
In general, Rijndael is very well suited for restricted-space environments where either encryption or decryption is implemented (but not both). It has very low RAM and ROM requirements. A drawback is that ROM requirements will increase if both encryption and decryption are implemented simultaneously, although it appears to remain suitable for these environments. The key schedule for decryption is separate from encryption.
Hardware Implementations
Rijndael has the highest throughput of any of the finalists for feedback modes and second

Table 5.2. Final NIST Evaluation of Rijndael (October 2, 2000)
General Security
highest for non-feedback modes. For the 192 and 256-bit key sizes, throughput falls in standard and unrolled implementations because of the additional number of rounds. For fully pipelined implementations, the area requirement increases, but the throughput is unaffected.
Attacks on Implementations
The operations used by Rijndael are among the easiest to defend against power and timing attacks. The use of masking techniques to provide Rijndael with some defense against these attacks does not cause significant performance degradation relative to the other finalists, and its RAM requirement remains reasonable. Rijndael appears to gain a major speed advantage over its competitors when such protections are considered.
Encryption vs. Decryption
The encryption and decryption functions in Rijndael differ. One FPGA study reports that the implementation of both encryption and decryption takes about 60% more space than the implementation of encryption alone. Rijndael's speed does not vary significantly between encryption and decryption, although the key setup performance is slower for decryption than for encryption.
Key Agility
Rijndael supports on-the-fly subkey computation for encryption. Rijndael requires a one-time execution of the key schedule to generate all subkeys prior to the first decryption with a specific key. This places a slight resource burden on the key agility of Rijndael.
Other Versatility and Flexibility
Rijndael fully supports block sizes and key sizes of 128 bits, 192 bits and 256 bits, in any combination. In principle, the Rijndael structure can accommodate any block sizes and key sizes that are multiples of 32, as well as changes in the number of rounds that are specified.
Potential for Instruction-Level Parallelism
Rijndael has an excellent potential for parallelism for a single block encryption.

Fermat's and Euler's Theorems

Two theorems that play important roles in public-key cryptography are Fermat's theorem and Euler's theorem.

Fermat's Theorem

Fermat's theorem states the following: If p is prime and a is a positive integer not divisible by p, then

Equation 8-2

$$a^{p-1} \equiv 1 (\text{mod } p)$$

Proof: Consider the set of positive integers less than p:{1,2,..., p 1} and multiply each element by a, modulo p, to get the set X = {a mod p, 2a mod p, . . . (p 1)a mod p}. None of the elements of X is equal to zero because p does not divide a. Furthermore no two of the integers in X are equal. To see this, assume that ja ≡ ka (mod p) where 1 ≤ j < k ≤ p 1. Because a is relatively prime to p, we can eliminate a from both sides of the equation [see Equation (4.3)] resulting in: j ≡ k(mode p). This last equality is impossible because j and k are both positive integers less than p. Therefore, we know that the (p 1) elements of X are all positive integers, with no two elements equal. We can conclude the X consists of the set of integers {1,2,..., p 1} in some order. Multiplying the numbers in both sets and taking the result mod p yields

that two numbers are relatively prime if they have no prime factors in common; that is, their only common divisor is 1. This is equivalent to saying that two numbers are relatively prime if their greatest common divisor is 1.

a x 2a x ... x (p 1) ≡ [(1 x 2 x ... x (p 1)](mode p)

$a^{p1(p\ 1)!} \equiv (p\ 1)!(\text{mod } p)$

We can cancel the (p 1)! term because it is relatively prime to p [see Equation (4.3)]. This yields Equation (8.2).

a = 7, p = 19
$7^2 = 49 \equiv 11 \pmod{19}$
$7^4 \equiv 121 \equiv 7 \pmod{19}$
$7^8 \equiv 49 \equiv 7 \pmod{19}$
$7^{16} \equiv 121 \quad 7 \pmod{19}$
$a^{p1} = 7^{18} = 7^{16} \times 7^2 \equiv 7 \times 11 \equiv 1 \pmod{19}$

An alternative form of Fermat's theorem is also useful: If p is prime and a is a positive integer, then

Equation 8-3

$$a^p = a \pmod{p}$$

Note that the first form of the theorem [Equation (8.2)] requires that a be relatively prime to p, but this form does not.

p = 5,a = 3	$a^p = 3^5 = 243 \equiv 3 \pmod{5} = a \pmod{p}$
p = 5, a = 10	$a^p = 10^5 = 100000 \equiv 10 \pmod{5} = 0 \pmod{5} = a \pmod{p}$

Euler's Totient Function

Before presenting Euler's theorem, we need to introduce an important quantity in number theory, referred to as Euler's totient function and written $\phi(n)$, defined as the number of positive integers less than n and relatively prime to n. By convention, $\phi(1) = 1$.

Determine ϕ (37) and ϕ (35).

Because 37 is prime, all of the positive integers from 1 through 36 are relatively prime to 37. Thus ϕ (37) = 36.

To determine ϕ (35), we list all of the positive integers less than 35 that are relatively prime to it:

1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18,

19, 22, 23, 24, 26, 27, 29, 31, 32, 33, 34.

There are 24 numbers on the list, so ϕ (35) = 24.

Table 8.2 lists the first 30 values of ϕ (n). The value ϕ (1) is without meaning but is defined to have the value 1.

Table 8.2. Some Values of Euler's Totient Function ϕ (n)

n	ϕ (n)
1	1
2	1
3	2
4	2
5	4
6	2
7	6
8	4
9	6
10	4
11	10
12	4
13	12

Table 8.2. Some Values of Euler's Totient Function ϕ (n)	
n	ϕ (n)
14	6
15	8
16	8
17	16
18	6
19	18
20	8
21	12
22	10
23	22
24	8
25	20
26	12
27	18
28	12
29	28
30	8

It should be clear that for a prime number p,

ϕ (p) = p 1

Now suppose that we have two prime numbers p and q, with p ≠ q. Then we can show that for n = pq,

ϕ (n) = ϕ (pq) = ϕ (p) x ϕ (q) = (p 1) x (q x 1)

To see that ϕ (n) = ϕ (p) x ϕ (q), consider that the set of positive integers less that n is the set {1,..., (pq 1)}. The integers in this set that are not relatively prime to n are the set {p,2 p,..., (q 1)p} and the set {q,2q,..., (p 1)q} Accordingly,

ϕ (n) = (pq 1) [(q 1) + (p 1)]

= pq (p + q) + 1

= (p 1) x (q 1)

= ϕ (p) x ϕ (q)

(21) = (3) x (7) = (3 1) x (7 1) = 2 x 6 = 12
where the 12 integers are {1,2,4,5,8,10,11,13,16,17,19,20}

Euler's Theorem

Euler's theorem states that for every a and n those are relatively prime:

Equation 8-4

$$a^{\phi(n)} \equiv 1 (\text{mod } n)$$

a = 3; n = 10; ϕ (10) = 4	$a^{\phi(n)} = 3^4 = 81 \equiv 1 (\text{mod } 10) = 1 (\text{mod } n)$
a = 2; n = 11; ϕ (11) = 10	$a^{\phi(n)} = 2^{10} = 1024 \equiv 1 (\text{mod } 11) = 1 (\text{mod } n)$

Proof: Equation (8.4) is true if n is prime, because in that case ϕ (n) = (n 1) and Fermat's theorem holds. However, it also holds for any integer n. Recall that ϕ(n) is the number of positive integers less than n that are relatively prime to n. Consider the set of such integers, labeled as follows:

R $\{x_1, x_2, ..., x_{\phi(n)}\}$

That is, each element x_i of R is a unique positive integer less than n with gcd(x_i, n) = 1. Now multiply each element by a, modulo n:

S = $\{(ax_1 \text{ mod } n), (ax_2 \text{ mod } n), ..., (ax_{\phi(n)} \text{ mod } n)\}$

The set S is a permutation of R, by the following line of reasoning:

1. Because a is relatively prime to n and x_i is relatively prime to n, ax_i must also be relatively prime to n. Thus, all the members of S are integers that are less than n and that are relatively prime to n.
2. There are no duplicates in S. Refer to Equation (4.3). If $ax_i \bmod n = ax_j \bmod n$ then $x_i = x_j$.

Therefore,

$$\prod_{i=1}^{\phi(n)} (ax_i \bmod n) = \prod_{i=1}^{\phi(n)} x_i$$

$$\prod_{i=1}^{\phi(n)} ax_i = \prod_{i=1}^{\phi(n)} x_i \pmod n$$

$$a^{\phi(n)} \times \left[\prod_{i=1}^{\phi(n)} x_i\right] = \prod_{i=1}^{\phi(n)} x_i \pmod n$$

$$a^{\phi(n)} = 1 \pmod n$$

This is the same line of reasoning applied to the proof of Fermat's theorem. As is the case for Fermat's theorem, an alternative form of the theorem is also useful:

Equation 8-5

$$a^{\phi(n)+1} = a \pmod n$$

Again, similar to the case with Fermat's theorem, the first form of Euler's theorem [Equation (8.4)] requires that a be relatively prime to n, but this form does not.

Finite Fields Of the Form GF(2^n)

Earlier in this chapter, we mentioned that the order of a finite field must be of the form p^n where p is a prime and n is a positive integer. we looked at the special case of finite fields with order p. We found that, using modular arithmetic in Z_p, all of the axioms for a field (Figure 4.1) are satisfied. For polynomials over p^n, with n > 1, operations modulo p^n do not produce a field. In this section, we show what structure satisfies the axioms for a field in a set with p^n elements, and concentrate on GF (2^n).

Motivation

Virtually all encryption algorithms, both symmetric and public key, involve arithmetic operations on integers. If one of the operations that is used in the algorithm is division, then we need to work in arithmetic defined over a field. For convenience and for implementation efficiency, we would also like to work with integers that fit exactly into a given number of bits, with no wasted bit patterns. That is, we wish to work with integers in the range 0 through 2^n 1, which fit into an n-bit word.

Suppose we wish to define a conventional encryption algorithm that operates on data 8 bits at a time and we wish to perform division. With 8 bits, we can represent integers in the range 0 through 255. However, 256 is not a prime number, so that if arithmetic is performed in Z_{256} (arithmetic modulo 256), this set of integers will not be a field. The closest prime number less than 256 is 251. Thus, the set Z_{251}, using arithmetic modulo 251, is a field. However, in this case the 8-bit patterns representing the integers 251 through 255 would not be used, resulting in inefficient use of storage.

As the preceding example points out, if all arithmetic operations are to be used, and we wish to represent a full range of integers in n bits, then arithmetic modulo will not work; equivalently, the set of integers modulo 2^n, for n > 1, is not a field. Furthermore, even if the encryption algorithm uses only addition and multiplication, but not division, the use of the set Z_2^n is questionable, as the following example illustrates.

Suppose we wish to use 3-bit blocks in our encryption algorithm, and use only the operations of addition and multiplication. Then arithmetic modulo 8 is well defined, as shown in Table 4.1. However, note that in the multiplication table, the nonzero integers do not appear an equal number of

times. For example, there are only four occurrences of 3, but twelve occurrences of 4. On the other hand, as was mentioned, there are finite fields of the form GF (2^n) so there is in particular a finite field of order $2^3 = 8$. Arithmetic for this field is shown in Table 4.5. In this case, the number of occurrences of the nonzero integers is uniform for multiplication. To summarize,

Integer	1	2	3	4	5	6	7
Occurrences in Z_8	4	8	4	12	4	8	4
Occurrences in GF(2^3)	7	7	7	7	7	7	7

Table 4.5. Arithmetic in GF(2^3)

(This item is displayed on page 121 in the print version)

	+	000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
000	0	0	1	2	3	4	5	6	7
001	1	1	0	3	2	5	4	7	6
010	2	2	3	0	1	6	7	4	5
011	3	3	2	1	0	7	6	5	4
100	4	4	5	6	7	0	1	2	3
101	5	5	4	7	6	1	0	3	2
110	6	6	7	4	5	2	3	0	1
111	7	7	6	5	4	3	2	1	0

(a) Addition

	×	000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
000	0	0	0	0	0	0	0	0	0
001	1	0	1	2	3	4	5	6	7
010	2	0	2	4	6	3	1	7	5
011	3	0	3	6	5	7	4	1	2
100	4	0	4	3	7	6	2	5	1
101	5	0	5	1	4	2	7	3	6
110	6	0	6	7	1	5	3	2	4
111	7	0	7	5	2	1	6	4	3

(b) Multiplication

w	$-w$	w^{-1}
0	0	—
1	1	1
2	2	5
3	3	6
4	4	7
5	5	2
6	6	3
7	7	4

(c) Additive and multiplicative inverses

For the moment, let us set aside the question of how the matrices of Table 4.5 were constructed and instead make some observations.

1. The addition and multiplication tables are symmetric about the main diagonal, in conformance to the commutative property of addition and multiplication. This property is also exhibited in Table 4.1, which uses mod 8 arithmetic.
2. All the nonzero elements defined by Table 4.5 have a multiplicative inverse, unlike the case with Table 4.1.
3. The scheme defined by Table 4.5 satisfies all the requirements for a finite field. Thus, we can refer to this scheme as GF(2^3).

For convenience, we show the 3-bit assignment used for each of the elements of GF (2^3).

Intuitively, it would seem that an algorithm that maps the integers unevenly onto themselves might be cryptographically weaker than one that provides a uniform mapping. Thus, the finite fields of the form GF(2^n) are attractive for cryptographic algorithms.

To summarize, we are looking for a set consisting of 2^n elements, together with a definition of addition and multiplication over the set that define a field. We can assign a unique integer in the range 0 through 2^n 1 to each element of the set. Keep in mind that we will not use modular arithmetic, as we have seen that this does not result in a field. Instead, we will show how polynomial arithmetic provides a means for constructing the desired field.

Modular Polynomial Arithmetic

Consider the set S of all polynomials of degree n 1 or less over the field Z_p. Thus, each polynomial has the form

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$

where each a_i takes on a value in the set {0, 1,..., p 1}. There are a total of p^n different polynomials in S.

For p = 3 and n = 2, the $3^2 = 9$ polynomials in the set are		
0	x	2x
1	x + 1	2x + 1
2	x + 2	2x + 2
For p = 2 and n = 3, the $2^3 = 8$ the polynomials in the set are		
0	x + 1	$x^2 + x$
1	x^2	$x^2 + x + 1$
x	$x^2 + 1$	

With the appropriate definition of arithmetic operations, each such set S is a finite field. The definition consists of the following elements:

1. Arithmetic follows the ordinary rules of polynomial arithmetic using the basic rules of algebra, with the following two refinements.

   Arithmetic on the coefficients is performed modulo p. That is, we use the rules of arithmetic for the finite field Z_p.

2. If multiplication results in a polynomial of degree greater than n 1, then the polynomial is reduced modulo some irreducible polynomial m(x) of degree n. That is, we divide by m(x) and keep the remainder. For a polynomial f(x), the remainder is expressed as r(x) = f(x) mod m(x).

The Advanced Encryption Standard (AES) uses arithmetic in the finite field GF (2^8), with the irreducible polynomial $m(x) = x^8 + x^4 x^3 + x + 1$. Consider the two polynomials $f(x) = x^6 + x^4 + x^2 + x + 1$ and $g(x) = x^7 + x + 1$. Then

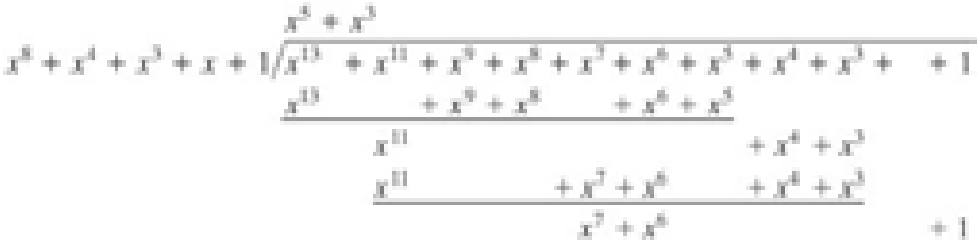
$f(x) + g(x) = x^6 + x^4 x^2 + x + 1 + x^7 + x + 1$

$f(x) \times g(x) = x^{13} + x^{11} + x^9 + x^8 + x^7 +$

$x^7 + x^5 + x^3 + x^2 + x +$

$x^6 + x^4 + x^2 + x + 1$

$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$



Therefore, f(x) x g(x) mod m(x) = $x^7 + x^6 + 1$

As with ordinary modular arithmetic, we have the notion of a set of residues in modular polynomial arithmetic. The set of residues modulo m(x), an nth-degree polynomial, consists of p^n elements. Each of these elements is represented by one of the p^n polynomials of degree m < n.

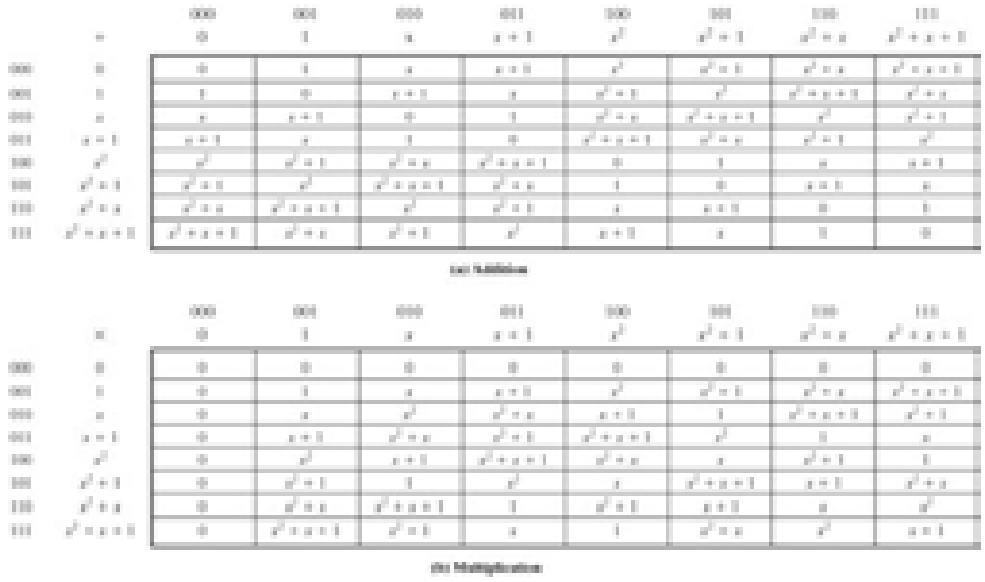
The residue class [x + 1], modulo m(x), consists of all polynomials a(x) such that a(x) (x + 1) (mod m(x)). Equivalently, the residue class [x + 1] consists of all polynomials a(x) that satisfy the equality a(x) mod m(x) = x + 1.

It can be shown that the set of all polynomials modulo an irreducible nth-degree polynomial m(x) satisfies the axioms in Figure 4.1, and thus forms a finite field. Furthermore, all finite fields of a given order are isomorphic; that is, any two finite-field structures of a given order have the same structure, but the representation, or labels, of the elements may be different.

To construct the finite field GF (2^3), we need to choose an irreducible polynomial of degree 3. There are only two such polynomials: ($x^3 + x^2 + 1$) and ($x^3 + x + 1$). Using the latter, Table 4.6 shows the addition and multiplication tables for GF (2^3). Note that this set of tables has the identical structure to those of Table 4.5. Thus, we have succeeded in finding a way to define a field of order 2^3.

Table 4.6. Polynomial Arithmetic Modulo ($x^3 + x + 1$)

(This item is displayed on page 124 in the print version)



Finding the Multiplicative Inverse

Just as the Euclidean algorithm can be adapted to find the greatest common divisor of two polynomials, the extended Euclidean algorithm can be adapted to find the multiplicative inverse of a polynomial. Specifically, the algorithm will find the multiplicative inverse of b(x) modulo m(x) if the degree of b(x) is less than the degree of m(x) and gcd[m(x), b(x)] = 1. If m(x) is an irreducible polynomial, then it has no factor other than itself or 1, so that gcd[m(x), b(x)] = 1. The algorithm is as follows:

EXTENDED EUCLID [m(x), b(x)]
1. [A1(x), A2(x), A3(x)] ←[1, 0, m(x)]; [B1(x), B2(x), B3(x)] ← [0, 1, b(x)]
2. if B3(x) = 0 return A3(x) = gcd[m(x), b(x)]; no Inverse
3. if B3(x) = 1 return B3(x) = gcd[m(x), b(x)]; B2(x) = b(x)^{1} mod m(x)
4. Q(x) = quotient of A3(x)/B3(x)
5. [T1(x), T2(x), T3(x)] ← [A1(x) Q(x)B1(x), A2(x) Q(x)B2(x), A3(x) QB3(x)]
6. [A1(x), A2(x), A3(x)] ← [B1(x), B2(x), B3(x)]

7. [B1(x), B2(x), B3(x)] ← [T1(x), T2(x), T3(x)]
8. goto 2

Table 4.7 shows the calculation of the multiplicative inverse of $(x^7 + x + 1)$ mod $(x^8 + x^4 + x^3 + x + 1)$. The result is that $(x^7 + x + 1)^1 = (x^7)$. That is, $(x^7 + x + 1)(x^7) \equiv 1 \pmod{(x^8 + x^4 + x^3 + x + 1)}$.

Table 4.7. Extended Euclid $[(x^8 + x^4 + x^3 + x + 1), (x^7 + x + 1)]$

(This item is displayed on page 125 in the print version)

Initialization	A1(x) = 1; A2(x) = 0; A3(x) = $x^8 + x^4 + x^3 + x + 1$ B1(x) = 0; B2(x) = 1; B3(x) = $x^7 + x + 1$
Iteration 1	Q(x) = x A1(x) = 0; A2(x) = 1; A3(x) = $x^7 + x + 1$ B1(x) = 1; B2(x) = x; B3(x) = $x^4 + x^3 + x^2 + 1$
Iteration 2	Q(x) = $x^3 + x^2 + 1$ A1(x) = 1; A2(x) = x; A3(x) = $x^4 + x^3 + x^2 + 1$ B1(x) = $x^3 + x^2 + 1$; B2(x) = $x^4 + x^3 + x + 1$; B3(x) = x
Iteration 3	Q(x) = $x^3 + x^2 + x$ A1(x) = $x^3 + x^2 + 1$; A2(x) = $x^4 + x^3 + x + 1$; A3(x) = x B1(x) = $x^6 + x^2 + x + 1$; B2(x) = x^7; B3(x) = 1
Iteration 4	B3(x) = gcd[$(x^7 + x + 1)$, $(x^8 + x^4 + x^3 + x + 1)$] = 1 B2(x) = $(x^7 + x + 1)^1$ mod $(x^8 + x^4 + x^3 + x + 1) = x^7$

Computational Considerations

A polynomial f(x) in $GF(2^n)$

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$

can be uniquely represented by its n binary coefficients $(a_{n1}a_{n2}...a_0)$. Thus, every polynomial in $GF(2^n)$ can be represented by an n-bit number.

Tables 4.5 and 4.6 show the addition and multiplication tables for GF(2^3) modulo m(x) = ($x^3 + x + 1$). Table 4.5 uses the binary representation, and Table 4.6 uses the polynomial representation.

Addition

We have seen that addition of polynomials is performed by adding corresponding coefficients and, in the case of polynomials over Z_2 addition is just the XOR operation. So, addition of two polynomials in GF(2^n) corresponds to a bitwise XOR operation.

Consider the two polynomials in GF(2^8) from our earlier example: $f(x) = x^6 + x^4 + x^2 + x + 1$ and $g(x) = x^7 + x + 1$.

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1)$	$= x^7 + x^6 + x^6 + x^4 + x^2$	(polynomial notation)
(01010111) ⊕ (10000011)	= (11010100)	(binary notation)
{57} ⊕ {83}	= {D4}	(hexadecimal notation)[7]

[7] A basic refresher on number systems (decimal, binary, hexadecimal) can be found at the Computer Science Student Resource Site at WilliamStallings.com/StudentSupport.html. Here each of two groups of 4 bits in a byte is denoted by a single hexadecimal character, the two characters enclosed in brackets.

Multiplication

There is no simple XOR operation that will accomplish multiplication in GF(2^n) However, a reasonably straightforward, easily implemented technique is available. We will discuss the technique with reference to GF(2^8) using $m(x) = x^8 + x^4 + x^3 + x + 1$, which is the finite field used in AES. The technique readily generalizes to GF (2^n).

The technique is based on the observation that

Equation 4-8

$$x^8 \bmod m(x) = [m(x) - x^8] = (x^4 + x^3 + x + 1)$$

A moment's thought should convince you that Equation (4.8) is true; if not, divide it out. In general, in GF (2^n) with an nth-degree polynomial p(x), we have $x^n \bmod p(x) = [p(x)\ x^n]$.

Now, consider a polynomial in GF (2^8), which has the form $f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. If we multiply by x, we have

Equation 4-9

$$x \times f(x) = (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod m(x)$$

If $b_7 = 0$, then the result is a polynomial of degree less than 8, which is already in reduced form, and no further computation is necessary. If $b_7 = 1$, then reduction modulo m(x) is achieved using Equation (4.8):

$x \times f(x) = (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 +$

$b_1x^2 + b_0x) + (x^4 + x^3 + x + 1)$

It follows that multiplication by x (i.e., 00000010) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (00011011), which represents $(x^4 + x^3 + x + 1)$. To summarize,

Equation 4-10

$$x \times f(x) = \begin{cases} (b_6b_5b_4b_3b_2b_1b_00) & \text{if } b_7 = 0 \\ (b_6b_5b_4b_3b_2b_1b_00) \oplus (00011011) & \text{if } b_7 = 1 \end{cases}$$

Multiplication by a higher power of x can be achieved by repeated application of Equation (4.10). By adding intermediate results, multiplication by any constant in GF(2^8) can be achieved.

In an earlier example, we showed that for $f(x) = x^6 + x^4 + x^2 + x + 1$, $g(x) = x^7 + x + 1$, and $m(x) = x^8 + x^4 + x^3 + x + 1$, $f(x) \times g(x) \bmod m(x) = x^7 + x^6 + 1$. Redoing this in binary arithmetic, we need to compute (01010111) x (10000011). First, we determine the results of multiplication by powers of x:

(01010111) x (00000001) = (10101110)

(01010111) x (00000100) = (01011100) ⊕ (00011011) = (01000111)

(01010111) x (00001000) = (10001110)

(01010111) x (00010000) = (00011100) ⊕ (00011011) = (00000111)

(01010111) x (00100000) = (00001110)

(01010111) x (01000000) = (00011100)

(01010111) x (10000000) = (00111000)

So,

(01010111) x (10000011) = (01010111) x [(00000001) x (00000010) x (10000000)]

= (01010111) ⊕ (10101110) ⊕ (00111000) = (11000001)

which is equivalent to $x^7 + x^6 + 1$.

Using a Generator

An equivalent technique for defining a finite field of the form GF(2^n) using the same irreducible polynomial, is sometimes more convenient. To begin, we need two definitions: A generator g of a finite field F of order q (contains q elements) is an element whose first q 1 powers generate all the nonzero elements of F. That is, the elements of F consist of 0, g^0, g^1,..., g^{q2}. Consider a field F defined by a polynomial f(x). An element b contained in F is called a root of the polynomial if f(b) = 0. Finally, it can be shown that a root g of

an irreducible polynomial is a generator of the finite field defined on that polynomial.

Let us consider the finite field GF (2^3), defined over the irreducible polynomial $x^3 + x + 1$, discussed previously. Thus, the generator g must satisfy $f(x) = g^3 + g + 1 = 0$. Keep in mind, as discussed previously, that we need not find a numerical solution to this equality. Rather, we deal with polynomial arithmetic in which arithmetic on the coefficients is performed modulo 2. Therefore, the solution to the preceding equality is $g^3 = g\ 1 = g + 1$. We now show that g in fact generates all of the polynomials of degree less than 3. We have the following:

$$g^4 = g(g^3) = g(g + 1) = g^2 + g$$

$$g^5 = g(g^4) = g(g^2 + g) = g^3 + g^2 = g^2 + g + 1$$

$$g^6 = g(g^5) = g(g^2 + g + 1) = g^3 + g^2 + g = g^2 + g + g + 1 = g^2 + 1$$

$$g^7 = g(g^6) = g(g^2 + 1) = g^3 + g = g + g + 1 = 1 = g^0$$

We see that the powers of g generate all the nonzero polynomials in GF(2^3). Also, it should be clear that $g^k = g^{k \bmod 7}$ for any integer k. Table 4.8 shows the power representation, as well as the polynomial and binary representations.

Table 4.8. Generator for GF(2^3) using $x^3 + x + 1$

Power Representation	Polynomial Representation	Binary Representation	Decimal (Hex) Representation
0	0	000	0
g^0 (= g^7)	1	001	1
g^1	g	010	2
g^2	g^2	100	4
g^3	g + 1	011	3
g^4	g^2 + g	110	6
g^5	g^2 + g + 1	111	7
g^6	g^2 + 1	101	5

This power representation makes multiplication easy. To multiply in the power notation, add exponents modulo 7. For example, $g^4 \times g^6 = g^{(10 \bmod 7)} = g^3 = g + 1$. The same result is achieved using polynomial arithmetic, as follows: we have $g^4 = g^2 + g$ and $g^6 = g^2 + 1$. Then, $(g^2 + g) \times (g^2 + 1) = g^4 + g^3 + g^2 + 1$. Next, we need to determine $(g^4 + g^3 + g^2 + 1) \bmod (g^3 + g + 1)$ by division:

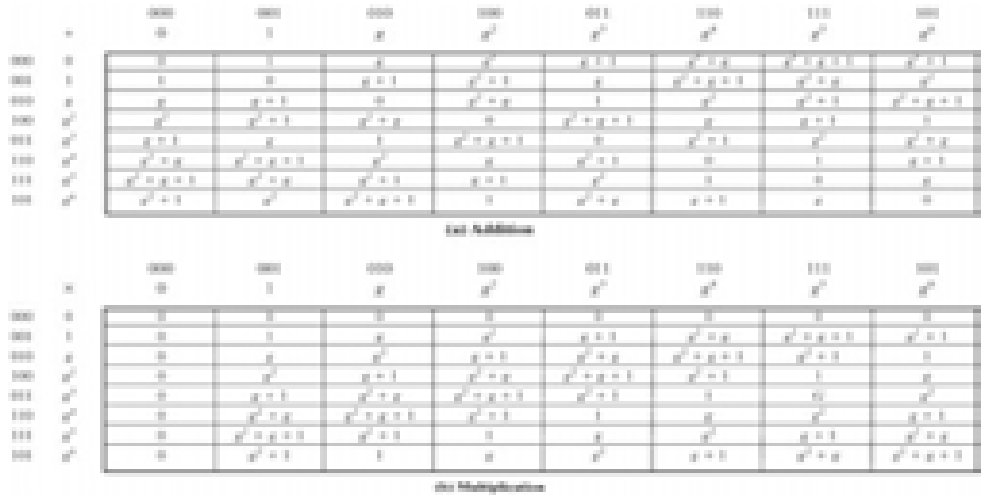
$$
\begin{array}{r}
g + 1 \\
g^3 + g^2 + 1 \overline{)\, g^4 + g^3 + g^2 + g} \\
\underline{g^4 \quad + g^2 + g} \\
g^3 \\
\underline{g^3 \quad + g + 1} \\
g + 1
\end{array}
$$

We get a result of g + 1, which agrees with the result obtained using the power representation.

Table 4.9 shows the addition and multiplication tables for GF(2^3) using the power represenation. Note that this yields the identical results to the polynomial representation (Table 4.6) with some of the rows and columns interchanged.

Table 4.9. GF(2^3) Arithmetic Using Generator for the Polynomial ($x^3 + x + 1$)

(This item is displayed on page 128 in the print version)



In general, for GF(2^n) with irreducible polynomial f(x), determine g^n = f(x) g^n. Then calculate all of the powers of g from g^{n+1} through g^{2n2}. The elements of the field correspond to the powers of g from through g^{2n2}, plus the value 0. For multiplication of two elements in the field, use the equality $g^k = g^{k \bmod (2n1)}$ for any integer k.

Finite Fields of the Form GF (p)

we defined a field as a set that obeys all of the axioms of Figure 4.1 and gave some examples of infinite fields. Infinite fields are not of particular interest in the context of cryptography. However, finite fields play a crucial role in many cryptographic algorithms. It can be shown that the order of a finite field (number of elements in the field) must be a power of a prime p^n, where n is a positive integer. a prime number is an integer whose only positive integer factors are itself and 1. That is, the only positive integers that are divisors of p are p and 1.

The finite field of order p^n is generally written GF(p^n); stands for Galois field, in honor of the mathematician who first studied finite fields. Two special cases are of interest for our purposes. For n = 1, we have the finite field GF(p); this finite field has a different structure than that for finite fields with n > 1 and is studied in this section. .

Finite Fields of Order p

For a given prime, p, the finite field of order p, GF(p) is defined as the set Z_p of integers {0, 1,..., p 1}, together with the arithmetic operations modulo p.

that the set Z_n of integers {0,1,...,n 1}, together with the arithmetic operations modulo n, is a commutative ring (Table 4.2). We further observed that any integer in Z_n has a multiplicative inverse if and only if that integer is relatively prime to n

If n is prime, then all of the nonzero integers in Z_n are relatively prime to n, and therefore there exists a multiplicative inverse for all of the nonzero integers in Z_n. Thus, we can add the following properties to those listed in Table 4.2 for Z_p:

[4] As stated in the discussion of Equation (4.3), two integers are relatively prime if their only common positive integer factor is 1.

Multiplicative inverse (w^{-1})	For each $w \in Z_p$, $w \neq 0$, there exists a $z \in Z_p$ such that $w \times z \equiv 1 \pmod{p}$

Because w is relatively prime to p, if we multiply all the elements of Z_p by w, the resulting residues are all of the elements of Z_p permuted. Thus, exactly one of the residues has the value 1. Therefore, there is some integer Z_p in that, when multiplied by w, yields the residue 1. That integer is the multiplicative inverse of w, designated w^{-1}. Therefore, Z_p is in fact a finite field. Further, Equation (4.3) is consistent with the existence of a multiplicative inverse and can be rewritten without the condition:

Equation 4-5

$$\textbf{if } (a \times b) \equiv (a \times c) \pmod{p} \textbf{ then } b \equiv c \pmod{p}$$

Multiplying both sides of Equation (4.5) by the multiplicative inverse of a, we have:

$((a^{-1}) \times a \times b) \equiv ((a^{-1}) \times a \times c) \pmod{p}$

$b \equiv c \pmod{p}$

The simplest finite field is GF(2). Its arithmetic operations are easily summarized:

Addition

+	0	1
0	0	1
1	1	0

Multiplication

×	0	1
0	0	0
1	0	1

Inverses

w	$-w$	w^{-1}
0	0	—
1	1	1

In this case, addition is equivalent to the exclusive-OR (XOR) operation, and multiplication is equivalent to the logical AND operation.

Table 4.3 shows GF (7). This is a field of order 7 using modular arithmetic modulo 7. As can be seen, it satisfies all of the properties required of a field (Figure 4.1). Compare this table with Table 4.1. In the latter case, we see that the set Z_8 using modular arithmetic modulo 8, is not a field. Later in this chapter, we show how to define addition and multiplication operations on Z_8 in such a way as to form a finite field.

Table 4.3. Arithmetic in GF (7)

(This item is displayed on page 111 in the print version)

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

(a) Addition modulo 7

×	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

(b) Multiplication modulo 7

w	$-w$	w^{-1}
0	0	—
1	6	1
2	5	4
3	4	5
4	3	2
5	2	3
6	1	6

(c) Additive and multiplicative inverses modulo 7

Finding the Multiplicative Inverse in GF (p)

It is easy to find the multiplicative inverse of an element in GF(p) for small values of p. You simply construct a multiplication table, such as shown in

Table 4.3b, and the desired result can be read directly. However, for large values of p, this approach is not practical.

If gcd(m, b) = 1, then b has a multiplicative inverse modulo m. That is, for positive integer b < m, there exists a b^1 < m such that bb^1 = 1 mod m. The Euclidean algorithm can be extended so that, in addition to finding gcd(m, b), if the gcd is 1, the algorithm returns the multiplicative inverse of b.

EXTENDED EUCLID (m, b)
1. (A1, A2, A3) ← (1, 0, m); (B1, B2, B3) ← (0, 1, b)
2. if B3 = 0 return A3 = gcd(m, b); no inverse
3. if B3 = 1 return B3 = gcd(m, b); B2 = b^1 mod m

4. $Q = \left\lfloor \frac{A3}{B3} \right\rfloor$

5. (T1, T2, T3) ← (A1 QB1, A2 QB2, A3 QB3)
6. (A1, A2, A3) ← (B1, B2, B3)
7. (B1, B2, B3) ← (T1, T2, T3)
8. goto 2

Throughout the computation, the following relationships hold:

$$mT1 + bT2 = T3 \quad mA1 + bA2 = A3 \quad mB1 + bB2 = B3$$

To see that this algorithm correctly returns gcd(m, b), note that if we equate A and B in the Euclidean algorithm with A3 and B3 in the extended Euclidean algorithm, then the treatment of the two variables is identical. At each iteration of the Euclidean algorithm, A is set equal to the previous value of B and B is set equal to the previous value of A mod B. Similarly, at each step of the extended Euclidean algorithm, A3 is set equal to the previous value of B3, and B3 is set equal to the previous value of A3 minus the integer quotient of A3 multiplied by B3. This latter value is simply the remainder of A3 divided by B3, which is A3 mod B3.

Note also that if gcd(m, b) = 1, then on the final step we would have B3 = 0 and A3 = 1. Therefore, on the preceding step, B3 = 1. But if B3 = 1, then we can say the following:

$mB1 + bB2 = B3$

$mB1 + bB2 = 1$

$bB2 = 1\ mB1$

$bB2 \equiv 1 \pmod{m}$

And B2 is the multiplicative inverse of b, modulo m.

Table 4.4 is an example of the execution of the algorithm. It shows that gcd(1759, 550) = 1 and that the multiplicative inverse of 550 is 355; that is, $550 \times 335 \equiv 1 \pmod{1759}$.

Table 4.4. Finding the Multiplicative Inverse of 550 in GF(1759)

Q	A1	A2	A3	B1	B2	B3
	1	0	1759	0	1	550
3	0	1	550	1	3	109
5	1	3	109	5	16	5
21	5	16	5	106	339	4
1	106	339	4	111	355	1

Groups, Rings, and Fields

Groups, rings, and fields are the fundamental elements of a branch of mathematics known as abstract algebra, or modern algebra. In abstract algebra, we are concerned with sets on whose elements we can operate algebraically; that is, we can combine two elements of the set, perhaps in several ways, to obtain a third element of the set. These operations are subject to specific rules, which define the nature of the set. By convention, the notation for the two principal classes of operations on set elements is usually the same as the notation for addition and multiplication on ordinary numbers. However, it is important to note that, in abstract algebra, we are

not limited to ordinary arithmetical operations. All this should become clear as we proceed.

Groups

A group G, sometimes denoted by {G, ·} is a set of elements with a binary operation, denoted by ·, that associates to each ordered pair (a, b) of elements in G an element (a · b) in G, such that the following axioms are obeyed:

The operator · is generic and can refer to addition, multiplication, or some other mathematical operation.

(A1) Closure:	If a and b belong to G, then a · b is also in G.
(A2) Associative:	a · (b · c) = (a · b) · c for all a, b, c in G.
(A3) Identity element:	There is an element e in G such that a · e = e · a = a for all a in G.
(A4) Inverse element:	For each a in G there is an element a' in G such that a · a' = a' · a = e.

Let N_n denote a set of n distinct symbols that, for convenience, we represent as {1,2,...,n}. A permutation of n distinct symbols is a one-to-one mapping from N_n to N_n. Define S_n to be the set of all permutations of n distinct symbols. Each element of S_n is represented by a permutation of the integers in {1,2,...,n}. It is easy to demonstrate that S_n is a group:

A1: If $\pi, \rho \in S_n$, then the composite mapping $\pi \cdot \rho$ is formed by permuting the elements of ρ according to the permutation π. For example, {3,2,1} · {1,3,2} = {2,3,1}. Clearly, $\pi \cdot \rho \in S_n$.

A2: The composition of mappings is also easily seen to be associative.

A3: The identity mapping is the permutation that does not alter the order of

the n elements. For S_n, the identity element is {1,2,...,n}.

A4: For any $\pi \in S_n$, the mapping that undoes the permutation defined by π is the inverse element for π .There will always be such an inverse. For example $\{2,3,1\} \cdot \{3,1,2\} = \{1,2,3\}$

If a group has a finite number of elements, it is referred to as a finite group, and the order of the group is equal to the number of elements in the group. Otherwise, the group is an infinite group.

A group is said to be abelian if it satisfies the following additional condition:

(A5) Commutative: $a \cdot b = b \cdot a$ for all a, b in G.

The set of integers (positive, negative, and 0) under addition is an abelian group. The set of nonzero real numbers under multiplication is an abelian group. The set S_n from the preceding example is a group but not an abelian group for $n > 2$.

When the group operation is addition, the identity element is 0; the inverse element of a is a; and subtraction is defined with the following rule: a b = a + (b).

Cyclic Group

We define exponentiation within a group as repeated application of the group operator, so that $a^3 = a \cdot a \cdot a$. Further, we define $a^0 = e$, the identity element; and $a^{-n} = (a')^n$. A group G is cyclic if every element of G is a power

a^k (k is an integer) of a fixed element a G. The element a is said to generate the group G, or to be a generator of G. A cyclic group is always abelian, and may be finite or infinite.

The additive group of integers is an infinite cyclic group generated by the element 1. In this case, powers are interpreted additively, so that n is the nth power of 1.

Rings

A ring R, sometimes denoted by {R, +, x}, is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in R the following axioms are obeyed:

Generally, we do not use the multiplication symbol, x, but denote multiplication by the concatenation of two elements.

(A1-A5) R is an abelian group with respect to addition; that is, R satisfies axioms A1 through A5. For the case of an additive group, we denote the identity element as 0 and the inverse of a as a.

(M1) Closure under multiplication:	If a and b belong to R, then ab is also in R.
(M2) Associativity of multiplication:	a(bc) = (ab)c for all a, b, c in R.
(M3) Distributive laws:	a(b + c) = ab + ac for all a, b, c in R. (a + b)c = ac + bc for all a, b, c in R.

In essence, a ring is a set in which we can do addition, subtraction [a b = a + (-b)], and multiplication without leaving the set.

With respect to addition and multiplication, the set of all n-square matrices over the real numbers is a ring.

A ring is said to be commutative if it satisfies the following additional condition:

(M4) Commutativity of multiplication: ab = ba for all a, b in R.

Let S be the set of even integers (positive, negative, and 0) under the usual operations of addition and multiplication. S is a commutative ring. The set of all n-square matrices defined in the preceding example is not a commutative ring.

Next, we define an integral domain, which is a commutative ring that obeys the following axioms:

(M5) Multiplicative identity:	There is an element 1 in R such that $a1 = 1a = a$ for all a in R.
(M6) No zero divisors:	If a, b in R and $ab = 0$, then either $a = 0$ or $b = 0$.

Let S be the set of integers, positive, negative, and 0, under the usual operations of addition and multiplication. S is an integral domain.

Fields

A field F, sometimes denoted by {F, +, x}, is a set of elements with two binary operations, called addition and multiplication, such that for all a, b, c in F the following axioms are obeyed:

(A1M6) F is an integral domain; that is, F satisfies axioms A1 through A5 and M1 through M6.

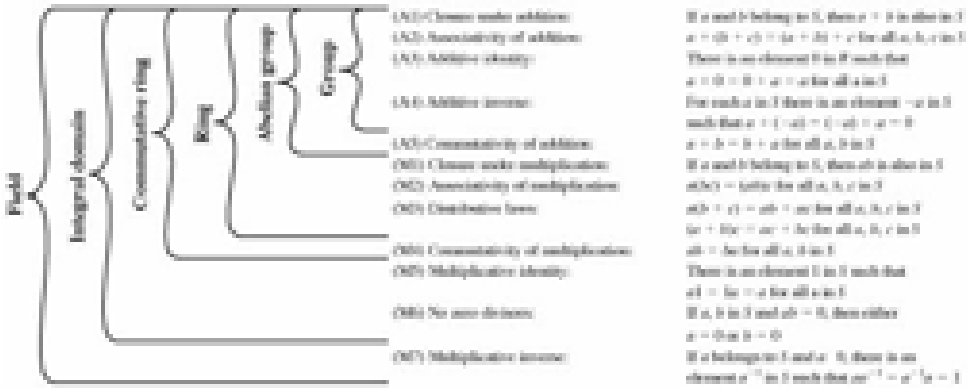
(M7) Multiplicative inverse:	For each a in F, except 0, there is an element a^{-1} in F such that $aa^{-1} = (a^{-1})a = 1$.

In essence, a field is a set in which we can do addition, subtraction, multiplication, and division without leaving the set. Division is defined with the following rule: $a/b = a(b^{-1})$.

Familiar examples of fields are the rational numbers, the real numbers, and the complex numbers. Note that the set of all integers is not a field, because not every element of the set has a multiplicative inverse; in fact, only the elements 1 and -1 have multiplicative inverses in the integers.

Figure 4.1 summarizes the axioms that define groups, rings, and fields.

Figure 4.1. Group, Ring, and Field



Key Distribution

For symmetric encryption to work, the two parties to an exchange must share the same key, and that key must be protected from access by others. Furthermore, frequent key changes are usually desirable to limit the amount of data compromised if an attacker learns the key. Therefore, the strength of any cryptographic system rests with the key distribution technique, a term that refers to the means of delivering a key to two parties who wish to exchange data, without allowing others to see the key. For two parties A and B, key distribution can be achieved in a number of ways, as follows:

1. A can select a key and physically deliver it to B.
2. A third party can select the key and physically deliver it to A and B.
3. If A and B have previously and recently used a key, one party can transmit the new key to the other, encrypted using the old key.
4. If A and B each has an encrypted connection to a third party C, C can deliver a key on the encrypted links to A and B.

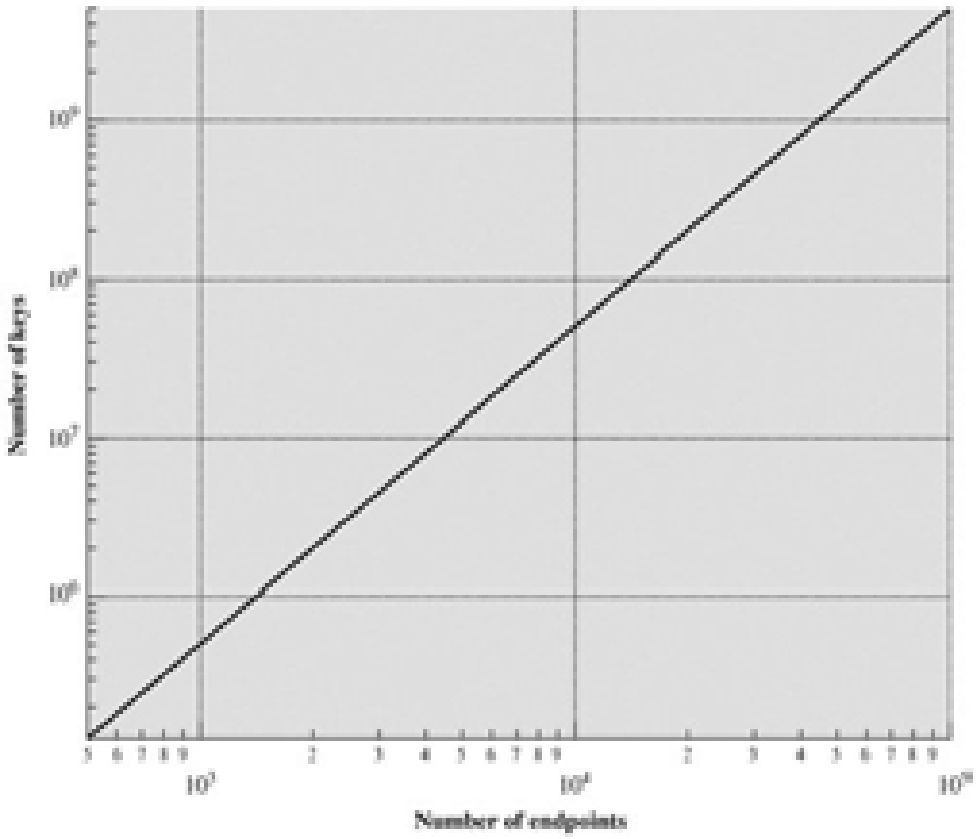
Options 1 and 2 call for manual delivery of a key. For link encryption, this is a reasonable requirement, because each link encryption device is going to be exchanging data only with its partner on the other end of the link. However, for end-to-end encryption, manual delivery is awkward. In a distributed system, any given host or terminal may need to engage in exchanges with many other hosts and terminals over time. Thus, each device

needs a number of keys supplied dynamically. The problem is especially difficult in a wide area distributed system.

The scale of the problem depends on the number of communicating pairs that must be supported. If end-to-end encryption is done at a network or IP level, then a key is needed for each pair of hosts on the network that wish to communicate. Thus, if there are N hosts, the number of required keys is [N (N 1)]/2. If encryption is done at the application level, then a key is needed for every pair of users or processes that require communication. Thus, a network may have hundreds of hosts but thousands of users and processes. Figure 7.7 illustrates the magnitude of the key distribution task for end-to-end encryption. A network using node-level encryption with 1000 nodes would conceivably need to distribute as many as half a million keys. If that same network supported 10,000 applications, then as many as 50 million keys may be required for application-level encryption.

Note that this figure uses a log-log scale, so that a linear graph indicates exponential growth.

Figure 7.7. Number of Keys Required to Support Arbitrary Connections between Endpoints



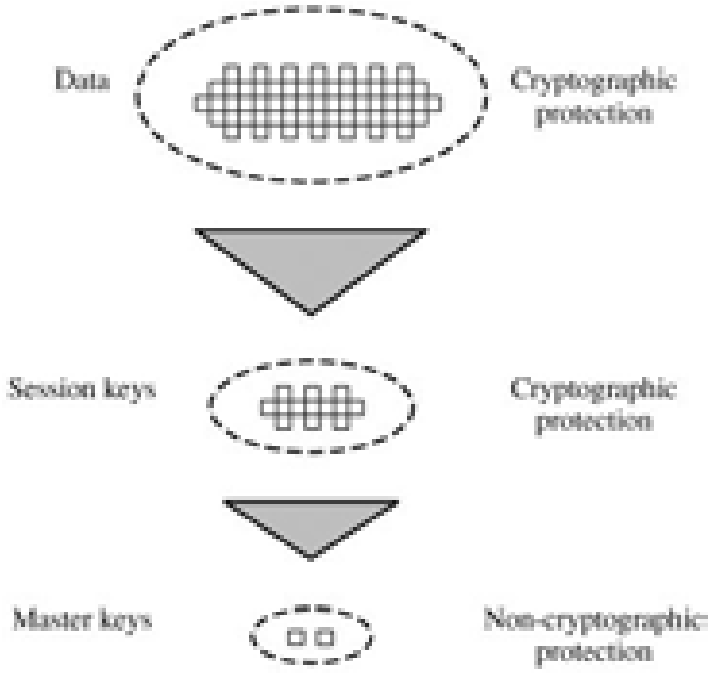
Returning to our list, option 3 is a possibility for either link encryption or end-to-end encryption, but if an attacker ever succeeds in gaining access to one key, then all subsequent keys will be revealed. Furthermore, the initial distribution of potentially millions of keys must still be made.

For end-to-end encryption, some variation on option 4 has been widely adopted. In this scheme, a key distribution center is responsible for distributing keys to pairs of users (hosts, processes, applications) as needed. Each user must share a unique key with the key distribution center for purposes of key distribution.

The use of a key distribution center is based on the use of a hierarchy of keys. At a minimum, two levels of keys are used (Figure 7.8). Communication between end systems is encrypted using a temporary key, often referred to as a session key. Typically, the session key is used for the duration of a logical connection, such as a frame relay connection or transport connection, and then discarded. Each session key is obtained from the key distribution center over the same networking facilities used for end-

user communication. Accordingly, session keys are transmitted in encrypted form, using a master key that is shared by the key distribution center and an end system or user.

Figure 7.8. The Use of a Key Hierarchy



For each end system or user, there is a unique master key that it shares with the key distribution center. Of course, these master keys must be distributed in some fashion. However, the scale of the problem is vastly reduced. If there are N entities that wish to communicate in pairs, then, as was mentioned, as many as [N(N 1)]/2 session keys are needed at any one time. However, only N master keys are required, one for each entity. Thus, master keys can be distributed in some noncryptographic way, such as physical delivery.

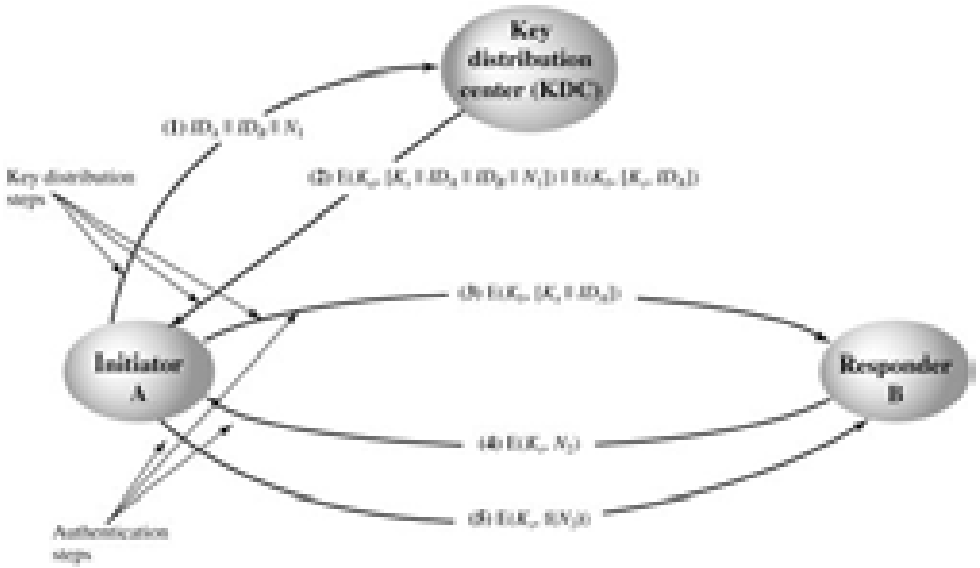
A Key Distribution Scenario

The key distribution concept can be deployed in a number of ways. A typical scenario is illustrated in Figure 7.9.

The scenario assumes that each user shares a unique master key with the key distribution center (KDC).

Figure 7.9. Key Distribution Scenario

(This item is displayed on page 213 in the print version)



Let us assume that user A wishes to establish a logical connection with B and requires a one-time session key to protect the data transmitted over the connection. A has a master key, K_a, known only to itself and the KDC; similarly, B shares the master key K_b with the KDC. The following steps occur:

1. A issues a request to the KDC for a session key to protect a logical connection to B. The message includes the identity of A and B and a unique identifier, N_1, for this transaction, which we refer to as a nonce. The nonce may be a timestamp, a counter, or a random number; the minimum requirement is that it differs with each request. Also, to prevent masquerade, it should be difficult for an opponent to guess the nonce. Thus, a random number is a good choice for a nonce.

   The following definitions are useful in understanding the purpose of the nonce component. Nonce: The present or particular occasion. Nonce word: A word occurring, invented, or used just for a particular occasion. From the American Heritage Dictionary of the English Language, 3rd ed.

2. The KDC responds with a message encrypted using K_a Thus, A is the only one who can successfully read the message, and A knows that it originated at the KDC. The message includes two items intended for A:

    - The one-time session key, K_s, to be used for the session
    - The original request message, including the nonce, to enable A to match this response with the appropriate request

    Thus, A can verify that its original request was not altered before reception by the KDC and, because of the nonce, that this is not a replay of some previous request.

    In addition, the message includes two items intended for B:

    - The one-time session key, K_s to be used for the session
    - An identifier of A (e.g., its network address), ID_A

    These last two items are encrypted with K_b (the master key that the KDC shares with B). They are to be sent to B to establish the connection and prove A's identity.

3. A stores the session key for use in the upcoming session and forwards to B the information that originated at the KDC for B, namely, $E(K_b, [K_s \| ID_A])$. Because this information is encrypted with K_b, it is protected from eavesdropping. B now knows the session key (K_s), knows that the other party is A (from ID_A), and knows that the information originated at the KDC (because it is encrypted using K_b).

    At this point, a session key has been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

4. Using the newly minted session key for encryption, B sends a nonce, N_2, to A.

5. Also using K_s, A responds with $f(N_2)$, where f is a function that performs some transformation on N_2 (e.g., adding one).

These steps assure B that the original message it received (step 3) was not a replay.

Note that the actual key distribution involves only steps 1 through 3 but that steps 4 and 5, as well as 3, perform an authentication function.

Hierarchical Key Control

It is not necessary to limit the key distribution function to a single KDC. Indeed, for very large networks, it may not be practical to do so. As an alternative, a hierarchy of KDCs can be established. For example, there can be local KDCs, each responsible for a small domain of the overall internetwork, such as a single LAN or a single building. For communication among entities within the same local domain, the local KDC is responsible for key distribution. If two entities in different domains desire a shared key, then the corresponding local KDCs can communicate through a global KDC. In this case, any one of the three KDCs involved can actually select the key. The hierarchical concept can be extended to three or even more layers, depending on the size of the user population and the geographic scope of the internetwork.

A hierarchical scheme minimizes the effort involved in master key distribution, because most master keys are those shared by a local KDC with its local entities. Furthermore, such a scheme limits the damage of a faulty or subverted KDC to its local area only.

Session Key Lifetime

The more frequently session keys are exchanged, the more secure they are, because the opponent has less cipher text to work with for any given session key. On the other hand, the distribution of session keys delays the start of any exchange and places a burden on network capacity. A security manager must try to balance these competing considerations in determining the lifetime of a particular session key.

For connection-oriented protocols, one obvious choice is to use the same session key for the length of time that the connection is open, using a new session key for each new session. If a logical connection has a very long lifetime, then it would be prudent to change the session key periodically, perhaps every time the PDU (protocol data unit) sequence number cycles.

For a connectionless protocol, such as a transaction-oriented protocol, there is no explicit connection initiation or termination. Thus, it is not obvious how often one needs to change the session key. The most secure approach is to use a new session key for each exchange. However, this negates one of the principal benefits of connectionless protocols, which is minimum overhead and delay for each transaction. A better strategy is to use a given

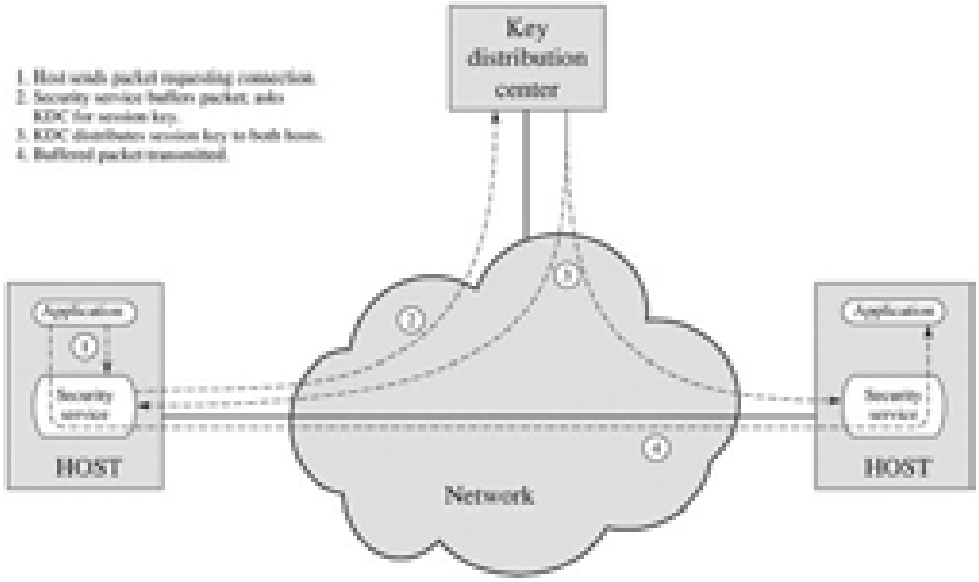
session key for a certain fixed period only or for a certain number of transactions.

A Transparent Key Control Scheme

The approach suggested in Figure 7.9 has many variations, one of which is described in this subsection. The scheme (Figure 7.10) is useful for providing end-to-end encryption at a network or transport level in a way that is transparent to the end users. The approach assumes that communication makes use of a connection-oriented end-to-end protocol, such as TCP. The noteworthy element of this approach is a session security module (SSM), which may consists of functionality at one protocol layer, that performs end-to-end encryption and obtains session keys on behalf of its host or terminal.

Figure 7.10. Automatic Key Distribution for Connection-Oriented Protocol

(This item is displayed on page 216 in the print version)



The steps involved in establishing a connection are shown in the figure. When one host wishes to set up a connection to another host, it transmits a connection-request packet (step 1). The SSM saves that packet and applies to the KDC for permission to establish the connection (step 2). The communication between the SSM and the KDC is encrypted using a master key shared only by this SSM and the KDC. If the KDC approves the connection request, it generates the session key and delivers it to the two

appropriate SSMs, using a unique permanent key for each SSM (step 3). The requesting SSM can now release the connection request packet, and a connection is set up between the two end systems (step 4). All user data exchanged between the two end systems are encrypted by their respective SSMs using the one-time session key.

The automated key distribution approach provides the flexibility and dynamic characteristics needed to allow a number of terminal users to access a number of hosts and for the hosts to exchange data with each other.

Decentralized Key Control

The use of a key distribution center imposes the requirement that the KDC be trusted and be protected from subversion. This requirement can be avoided if key distribution is fully decentralized. Although full decentralization is not practical for larger networks using symmetric encryption only, it may be useful within a local context.

A decentralized approach requires that each end system be able to communicate in a secure manner with all potential partner end systems for purposes of session key distribution. Thus, there may need to be as many as [n(n 1)]/2 master keys for a configuration with n end systems.

A session key may be established with the following sequence of steps (Figure 7.11):

1. A issues a request to B for a session key and includes a nonce, N_1

2. B responds with a message that is encrypted using the shared master key. The response includes the session key selected by B, an identifier of B, the value $f(N_1)$, and another nonce, N_2.

3. Using the new session key, A returns $f(N_2)$ to B.

Figure 7.11. Decentralized Key Distribution

Thus, although each node must maintain at most (n 1) master keys, as many session keys as required may be generated and used. Because the messages transferred using the master key are short, cryptanalysis is difficult. As before, session keys are used for only a limited time to protect them.

Controlling Key Usage

The concept of a key hierarchy and the use of automated key distribution techniques greatly reduce the number of keys that must be manually managed and distributed. It may also be desirable to impose some control on the way in which automatically distributed keys are used. For example, in addition to separating master keys from session keys, we may wish to define different types of session keys on the basis of use, such as

- Data-encrypting key, for general communication across a network
- PIN-encrypting key, for personal identification numbers (PINs) used in electronic funds transfer and point-of-sale applications
- File-encrypting key, for encrypting files stored in publicly accessible locations

To illustrate the value of separating keys by type, consider the risk that a master key is imported as a data-encrypting key into a device. Normally, the master key is physically secured within the cryptographic hardware of the key distribution center and of the end systems. Session keys encrypted with this master key are available to application programs, as are the data encrypted with such session keys. However, if a master key is treated as a session key, it may be possible for an unauthorized application to obtain plaintext of session keys encrypted with that master key.

Thus, it may be desirable to institute controls in systems that limit the ways in which keys are used, based on characteristics associated with those keys. One simple plan is to associate a tag with each key .The proposed technique

is for use with DES and makes use of the extra 8 bits in each 64-bit DES key. That is, the 8 nonkey bits ordinarily reserved for parity checking form the key tag. The bits have the following interpretation:

- One bit indicates whether the key is a session key or a master key.
- One bit indicates whether the key can be used for encryption.
- One bit indicates whether the key can be used for decryption.
- The remaining bits are spares for future use.

Because the tag is embedded in the key, it is encrypted along with the key when that key is distributed, thus providing protection. The drawbacks of this scheme are that (1) the tag length is limited to 8 bits, limiting its flexibility and functionality; and (2) because the tag is not transmitted in clear form, it can be used only at the point of decryption, limiting the ways in which key use can be controlled.

A more flexible scheme, referred to as the control vector, is described in .

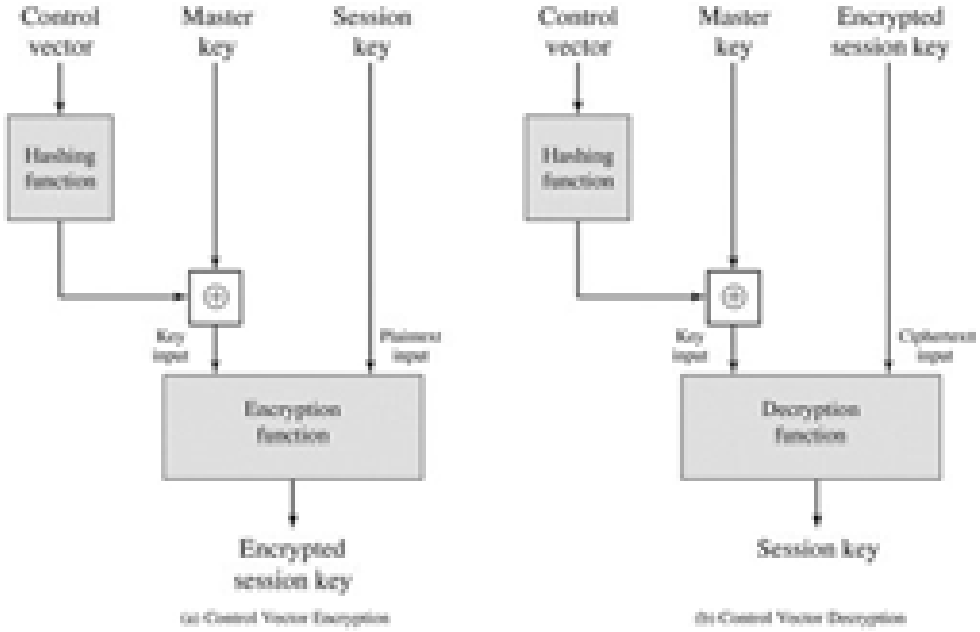
In this scheme, each session key has an associated control vector consisting of a number of fields that specify the uses and restrictions for that session key. The length of the control vector may vary.

The control vector is cryptographically coupled with the key at the time of key generation at the KDC. The coupling and decoupling processes are illustrated in Figure 7.12. As a first step, the control vector is passed through a hash function that produces a value whose length is equal to the encryption key length.

In essence, a hash function maps values from a larger range into a smaller range, with a reasonably uniform spread. Thus, for example, if numbers in the range 1 to 100 are hashed into numbers in the range 1 to 10, approximately 10% of the source values should map into each of the target values.

Figure 7.12. Control Vector Encryption and Decryption

(This item is displayed on page 219 in the print version)



The hash value is then XORed with the master key to produce an output that is used as the key input for encrypting the session key. Thus,

Hash value = H = h(CV)

Key input = $K_m \oplus H$

Ciphertext = $E([K_m \oplus H], K_s)$

where K_m is the master key and K_s is the session key. The session key is recovered in plaintext by the reverse operation:

$D([K_m \oplus H], E([K_m \oplus H], K_s))$

When a session key is delivered to a user from the KDC, it is accompanied by the control vector in clear form. The session key can be recovered only by using both the master key that the user shares with the KDC and the control vector. Thus, the linkage between the session key and its control vector is maintained.

Use of the control vector has two advantages over use of an 8-bit tag. First, there is no restriction on length of the control vector, which enables arbitrarily complex controls to be imposed on key use. Second, the control

vector is available in clear form at all stages of operation. Thus, control of key use can be exercised in multiple locations.

Key Management

we examined the problem of the distribution of secret keys. One of the major roles of public-key encryption has been to address the problem of key distribution. There are actually two distinct aspects to the use of public-key cryptography in this regard:

- The distribution of public keys
- The use of public-key encryption to distribute secret keys

We examine each of these areas in turn.

Distribution of Public Keys

Several techniques have been proposed for the distribution of public keys. Virtually all these proposals can be grouped into the following general schemes:

- Public announcement
- Publicly available directory
- Public-key authority
- Public-key certificates

Public Announcement of Public Keys

On the face of it, the point of public-key encryption is that the public key is public. Thus, if there is some broadly accepted public-key algorithm, such as RSA, any participant can send his or her public key to any other participant or broadcast the key to the community at large (Figure 10.1). For example, because of the growing popularity of PGP, which makes use of RSA, many PGP users have adopted the practice of appending their public key to messages that they send to public forums, such as USENET newsgroups and Internet mailing lists.

Figure 10.1. Uncontrolled Public-Key Distribution

Although this approach is convenient, it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be user A and send a public key to another participant or broadcast such a public key. Until such time as user A discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for A and can use the forged keys for authentication (see Figure 9.3).

Publicly Available Directory

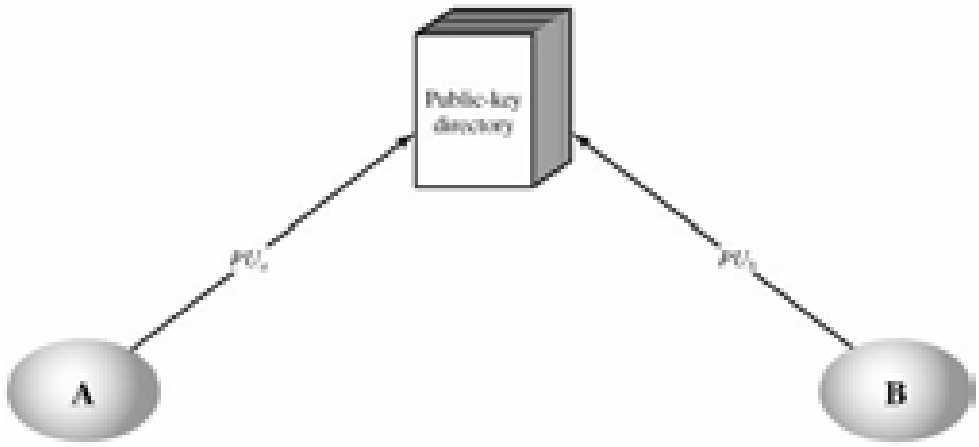
A greater degree of security can be achieved by maintaining a publicly available dynamic directory of public keys. Maintenance and distribution of the public directory would have to be the responsibility of some trusted entity or organization (Figure 10.2). Such a scheme would include the following elements:

1. The authority maintains a directory with a {name, public key} entry for each participant.

Each participant registers a public key with the directory authority. Registration would have to be in person or by some form of secure authenticated communication.

2. A participant may replace the existing key with a new one at any time, either because of the desire to replace a public key that has already been used for a large amount of data, or because the corresponding private key has been compromised in some way.
3. Participants could also access the directory electronically. For this purpose, secure, authenticated communication from the authority to the participant is mandatory.

Figure 10.2. Public-Key Publication



This scheme is clearly more secure than individual public announcements but still has vulnerabilities. If an adversary succeeds in obtaining or computing the private key of the directory authority, the adversary could authoritatively pass out counterfeit public keys and subsequently impersonate any participant and eavesdrop on messages sent to any participant. Another way to achieve the same end is for the adversary to tamper with the records kept by the authority.

Public-Key Authority

Stronger security for public-key distribution can be achieved by providing tighter control over the distribution of public keys from the directory. A typical scenario is illustrated in Figure 10.3.As before, the scenario assumes that a central authority maintains a dynamic directory of public keys of all

participants. In addition, each participant reliably knows a public key for the authority, with only the authority knowing the corresponding private key. The following steps (matched by number to Figure 10.3) occur:

1. A sends a timestamped message to the public-key authority containing a request for the current public key of B.

2. The authority responds with a message that is encrypted using the authority's private key, PR_{auth} Thus, A is able to decrypt the message using the authority's public key. Therefore, A is assured that the message originated with the authority. The message includes the following:

    - B's public key, PU_b which A can use to encrypt messages destined for B
    - The original request, to enable A to match this response with the corresponding earlier request and to verify that the original request was not altered before reception by the authority

    ---

    - The original timestamp, so A can determine that this is not an old message from the authority containing a key other than B's current public key

3. A stores B's public key and also uses it to encrypt a message to B containing an identifier of A (ID_A) and a nonce (N_1), which is used to identify this transaction uniquely.

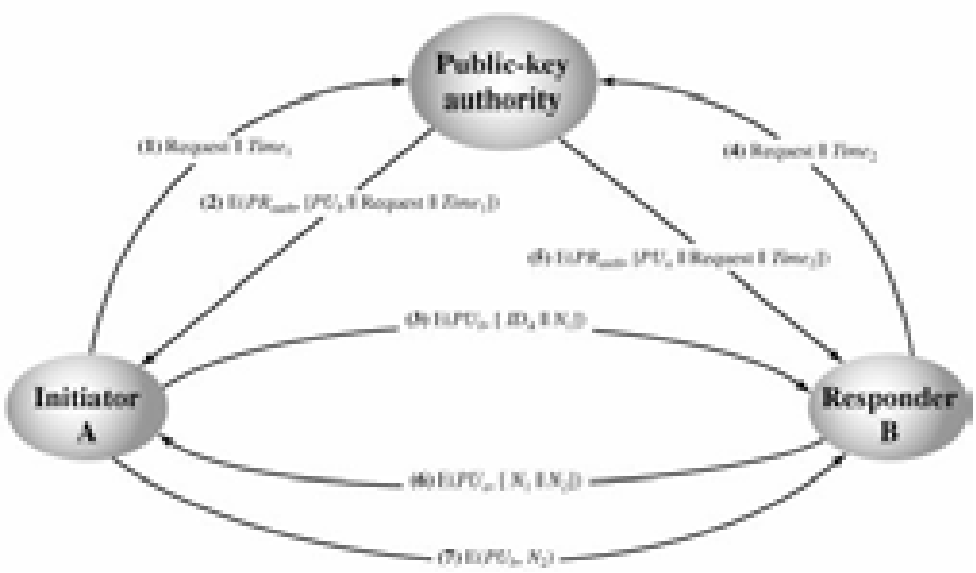
4, 5. B retrieves A's public key from the authority in the same manner as A retrieved B's public key.

At this point, public keys have been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

6. B sends a message to A encrypted with PU_a and containing A's nonce (N_1) as well as a new nonce generated by B (N_2) Because only B could have decrypted message (3), the presence of N_1 in message (6) assures A that the correspondent is B.

7. A returns N_2, encrypted using B's public key, to assure B that its correspondent is A.

Figure 10.3. Public-Key Distribution Scenario



Thus, a total of seven messages are required. However, the initial four messages need be used only infrequently because both A and B can save the other's public key for future use, a technique known as caching. Periodically, a user should request fresh copies of the public keys of its correspondents to ensure currency.

Public-Key Certificates

The scenario of Figure 10.3 is attractive, yet it has some drawbacks. The public-key authority could be somewhat of a bottleneck in the system, for a user must appeal to the authority for a public key for every other user that it wishes to contact. As before, the directory of names and public keys maintained by the authority is vulnerable to tampering.

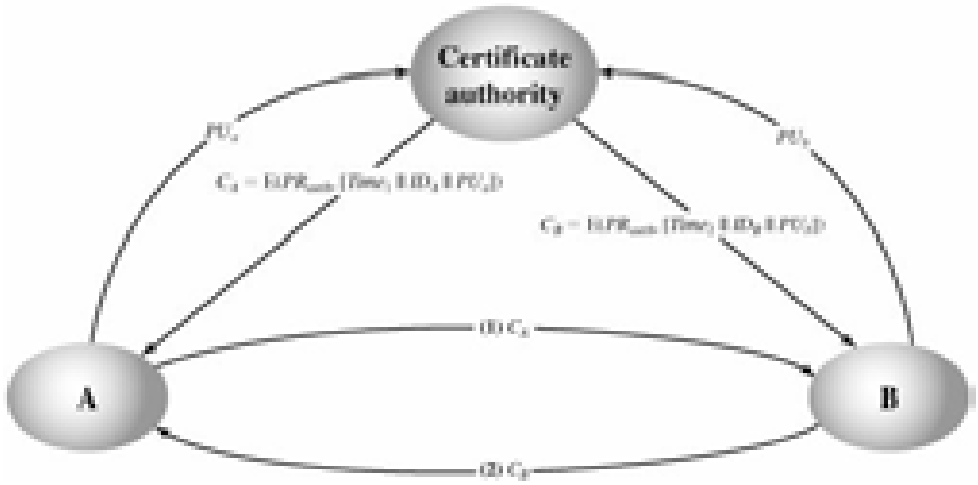
An alternative approach, first suggested by Kohnfelder , is to use certificates that can be used by participants to exchange keys without contacting a public-key authority, in a way that is as reliable as if the keys were obtained directly from a public-key authority. In essence, a certificate consists of a public key plus an identifier of the key owner, with the whole block signed by a trusted third party. Typically, the third party is a certificate authority, such as a government agency or a financial institution, that is trusted by the user community. A user can present his or her public key to the authority in a secure manner, and obtain a certificate. The user can then publish the certificate. Anyone needed this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signature. A participant can also convey its key information to another by transmitting its certificate. Other participants can verify that the certificate was created by the authority. We can place the following requirements on this scheme:

1. Any participant can read a certificate to determine the name and public key of the certificate's owner.
2. Any participant can verify that the certificate originated from the certificate authority and is not counterfeit.
3. Only the certificate authority can create and update certificates.

4. Any participant can verify the currency of the certificate.

A certificate scheme is illustrated in Figure 10.4. Each participant applies to the certificate authority, supplying a public key and requesting a certificate.

Figure 10.4. Exchange of Public-Key Certificates



Application must be in person or by some form of secure authenticated communication. For participant A, the authority provides a certificate of the form

$C_A = E(PR_{auth}, [T||ID_A||PU_a])$

where PR_{auth} is the private key used by the authority and T is a timestamp. A may then pass this certificate on to any other participant, who reads and verifies the certificate as follows:

$D(PU_{auth}, C_A) = D(PU_{auth}, E(PR_{auth}, [T||ID_A||PU_a])) = (T||ID_A||PU_a)$

The recipient uses the authority's public key, PU_{auth} to decrypt the certificate. Because the certificate is readable only using the authority's public key, this verifies that the certificate came from the certificate authority. The elements ID_A and PU_a provide the recipient with the name and public key of the certificate's holder. The timestamp T validates the currency of the certificate. The timestamp counters the following scenario. A's private key is learned by an adversary. A generates a new private/public key pair and applies to the certificate authority for a new certificate. Meanwhile, the adversary replays the old certificate to B. If B then encrypts messages using the compromised old public key, the adversary can read those messages.

In this context, the compromise of a private key is comparable to the loss of a credit card. The owner cancels the credit card number but is at risk until all possible communicants are aware that the old credit card is obsolete. Thus, the timestamp serves as something like an expiration date. If a certificate is sufficiently old, it is assumed to be expired.

One scheme has become universally accepted for formatting public-key certificates: the X.509 standard. X.509 certificates are used in most network security applications, including IP security, secure sockets layer (SSL), secure electronic transactions (SET), and S/MIME.

Distribution of Secret Keys Using Public-Key Cryptography

Once public keys have been distributed or have become accessible, secure communication that thwarts eavesdropping (Figure 9.2), tampering (Figure 9.3), or both (Figure 9.4) is possible. However, few users will wish to make exclusive use of public-key encryption for communication because of the relatively slow data rates that can be achieved. Accordingly, public-key encryption provides for the distribution of secret keys to be used for conventional encryption.

Simple Secret Key Distribution

An extremely simple scheme was put forward by Merkle, as illustrated in Figure 10.5. If A wishes to communicate with B, the following procedure is employed:

1. A generates a public/private key pair $\{PU_a, PR_a\}$ and transmits a message to B consisting of PU_a and an identifier of A, ID_A.

2. B generates a secret key, K_s, and transmits it to A, encrypted with A's public key.

3. A computes $D(PR_a, E(PU_a, K_s))$ to recover the secret key. Because only A can decrypt the message, only A and B will know the identity of K_s.

4. A discards PU_a and PR_a and B discards PU_a.

Figure 10.5. Simple Use of Public-Key Encryption to Establish a Session Key

A and B can now securely communicate using conventional encryption and the session key K_s. At the completion of the exchange, both A and B discard K_s. Despite its simplicity, this is an attractive protocol. No keys exist before the start of the communication and none exist after the completion of communication. Thus, the risk of compromise of the keys is minimal. At the same time, the communication is secure from eavesdropping.

The protocol depicted in Figure 10.5 is insecure against an adversary who can intercept messages and then either relay the intercepted message or substitute another message (see Figure 1.4c). Such an attack is known as a man-in-the-middle attack . In this case, if an adversary, E, has control of the intervening communication channel, then E can compromise the communication in the following fashion without being detected:

1. A generates a public/private key pair $\{PU_a, PR_a\}$ and transmits a message intended for B consisting of PU_a and an identifier of A, ID_A.
2. E intercepts the message, creates its own public/private key pair $\{PU_e, PR_e\}$ and transmits $PU_e||ID_A$ to B.
3. B generates a secret key, K_s, and transmits $E(PU_e, K_s)$.
4. E intercepts the message, and learns K_s by computing $D(PR_e, E(PU_e, K_s))$.
5. E transmits $E(PU_a, K_s)$ to A.

The result is that both A and B know K_s and are unaware that K_s has also been revealed to E. A and B can now exchange messages using K_s E no longer actively interferes with the communications channel but simply eavesdrops. Knowing K_s E can decrypt all messages, and both A and B are unaware of the problem. Thus, this simple protocol is only useful in an environment where the only threat is eavesdropping.

Secret Key Distribution with Confidentiality and Authentication

Figure 10.6, provides protection against both active and passive attacks. We begin at a point when it is assumed that A and B have exchanged public keys by one of the schemes described earlier in this section. Then the following steps occur:

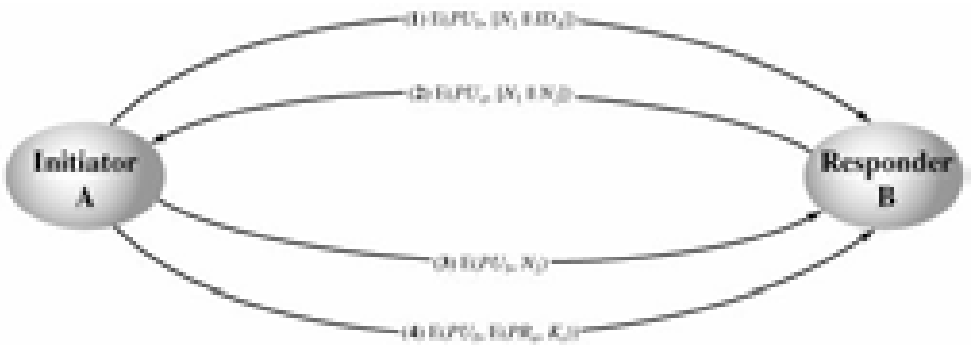
1. A uses B's public key to encrypt a message to B containing an identifier of A (ID_A) and a nonce (N_1), which is used to identify this transaction uniquely.

2. B sends a message to A encrypted with PU_a and containing A's nonce (N_1) as well as a new nonce generated by B (N_2) Because only B could have decrypted message (1), the presence of N_1 in message (2) assures A that the correspondent is B.
3. A returns N_2 encrypted using B's public key, to assure B that its correspondent is A.

4. A selects a secret key K_s and sends $M = E(PU_b, E(PR_a, K_s))$ to B. Encryption of this message with B's public key ensures that only B can read it; encryption with A's private key ensures that only A could have sent it.

5. B computes $D(PU_a, D(PR_b, M))$ to recover the secret key.

Figure 10.6. Public-Key Distribution of Secret Keys



Notice that the first three steps of this scheme are the same as the last three steps of Figure 10.3. The result is that this scheme ensures both confidentiality and authentication in the exchange of a secret key.

A Hybrid Scheme

Yet another way to use public-key encryption to distribute secret keys is a hybrid approach in use on IBM mainframes . This scheme retains the use of a key distribution center (KDC) that shares a secret master key with each user and distributes secret session keys encrypted with the master key. A public key scheme is used to distribute the master keys. The following rationale is provided for using this three-level approach:

- Performance: There are many applications, especially transaction-oriented applications, in which the session keys change frequently.

Distribution of session keys by public-key encryption could degrade overall system performance because of the relatively high computational load of public-key encryption and decryption. With a three-level hierarchy, public-key encryption is used only occasionally to update the master key between a user and the KDC.

- Backward compatibility: The hybrid scheme is easily overlaid on an existing KDC scheme, with minimal disruption or software changes.

The addition of a public-key layer provides a secure, efficient means of distributing master keys. This is an advantage in a configuration in which a single KDC serves a widely distributed set of users.

Modular Arithmetic

Given any positive integer n and any nonnegative integer a, if we divide a by n, we get an integer quotient q and an integer remainder r that obey the following relationship:

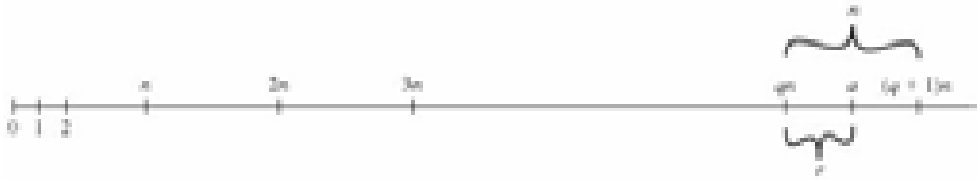
Equation 4-1

$$a = qn + r \qquad 0 \leq r < n; q = \lfloor a/n \rfloor$$

where $\lfloor x \rfloor$ is the largest integer less than or equal to x.

Figure 4.2 demonstrates that, given a and positive n, it is always possible to find q and r that satisfy the preceding relationship. Represent the integers on the number line; a will fall somewhere on that line (positive a is shown, a similar demonstration can be made for negative a). Starting at 0, proceed to n, 2n, up to qn such that $qn \leq a$ and $(q + 1)n > a$. The distance from qn to a is r, and we have found the unique values of q and r. The remainder r is often referred to as a residue.

Figure 4.2. The Relationship $a = qn + r$, $0 \leq r < n$



a = 11;	n = 7;	11 = 1 x 7 + 4;	r = 4	q = 1
a = -11;	n = 7;	-11 = (-2) x 7 + 3;	r = 3	q = -2

If a is an integer and n is a positive integer, we define a mod n to be the remainder when a is divided by n. The integer n is called the modulus. Thus, for any integer a, we can always write:

$a = \lfloor a/n \rfloor x\ n + (a \bmod n)$

11 mod 7 = 4;	-11 mod 7 = 3

Two integers a and b are said to be congruent modulo n, if (a mod n) = (b mod n). This is written as $a \equiv b \pmod{n}$.

We have just used the operator mod in two different ways: first as a binary operator that produces a remainder, as in the expression a mod b; second as a congruence relation that shows the equivalence of two integers, as in the expression To distinguish the two uses, the mod term is enclosed in parentheses for a congruence relation; this is common but not universal in the literature. See Appendix D for a further discussion.

$73 \equiv 4 \pmod{23}$;	$21 \equiv -9 \pmod{10}$

Divisors

We say that a nonzero b divides a if a = mb for some m, where a, b, and m are integers. That is, b divides a if there is no remainder on division. The notation is commonly used to mean b divides a. Also, if b|a, we say that b is a divisor of a.

The positive divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

The following relations hold:

- If a|1, then a = ±1.
- If a|b and b|a, then a = ±b.
- Any b ≠ 0 divides 0.
- If b|g and b|h, then b|(mg + nh) for arbitrary integers m and n.

To see this last point, note that

If b|g, then g is of the form g = b x g_1 for some integers g_1.

If b|h, then h is of the form h = b x h_1 for some integers h_1.

So

mg + nh = mbg_1 + nbh_1 = b x (mg_1 + nh_1)

and therefore b divides mg + nh.

b = 7; g = 14; h = 63; m = 3; n = 2. 7|14 and 7|63. To show: 7|(3 x 14 + 2 x 63) We have (3 x 14 + 2 x 63) = 7(3 x 2 + 2 x 9) And it is obvious that 7|(7(3 x 2 + 2 x 9))

Note that if a ≡ 0 (mod n), then n|a.

Properties of Congruences

Congruences have the following properties:

1. a ≡ b (mod n) if n|(a b).
2. a ≡ b (mod n) implies b ≡ a (mod n)..
3. a ≡ b (mod n) and b ≡ c (mod n) imply a ≡ c (mod n).

To demonstrate the first point, if n|(a b), then (a b) = kn for some k. So we can write a = b + kn. Therefore, (a mod n) = (reminder when b + kn is divided by n) = (reminder when b is divided by n) = (b mod n)

23 ≡ 8 (mod 5)	because	23 8 = 15 = 5 3
11 ≡ 5 (mod 8)	because	11 5 = 16 = 8 x (2)
81 ≡ 0 (mod 27)	because	81 0 = 81 = 27 x 3

The remaining points are as easily proved.

Modular Arithmetic Operations

Note that, by definition (Figure 4.2), the (mod n) operator maps all integers into the set of integers {0, 1,... (n 1)}. This suggests the question: Can we perform arithmetic operations within the confines of this set? It turns out that we can; this technique is known as modular arithmetic.

Modular arithmetic exhibits the following properties:

1. [(a mod n) + (b mod n)] mod n = (a + b) mod n
2. [(a mod n) (b mod n)] mod n = (a b) mod n
3. [(a mod n) x (b mod n)] mod n = (a x b) mod n

We demonstrate the first property. Define (a mod n) = r_a and (b mod n) = r_b. Then we can write a = r_a + jn for some integer j and b = r_b + kn for some integer k. Then

(a + b) mod n = (r_a + jn + r_b +kn) mod n

= (r_a + r_b (k + j)n) mod n

= (r_a + r_b) mod n

= [(a mod n] + (b mod n)] mod n

The remaining properties are as easily proved. Here are examples of the three properties:

11 mod 8 = 3; 15 mod 8 = 7
[(11 mod 8) + (15 mod 8)] mod 8 = 10 mod 8 = 2 (11 + 15) mod 8 = 26 mod 8 = 2
[(11 mod 8) (15 mod 8)] mod 8 = 4 mod 8 = 4 (11 15) mod 8 = 4 mod 8 = 4
[(11 mod 8) x (15 mod 8)] mod 8 = 21 mod 8 = 5 (11 x 15) mod 8 = 165 mod 8 = 5

Exponentiation is performed by repeated multiplication, as in ordinary arithmetic.

To find 11^7 mod 13, we can proceed as follows:
$11^2 = 121 \equiv 4 \pmod{13}$
$11^4 = (11^2)^2 \equiv 4^2 \equiv 3 \pmod{13}$
$11^7 \equiv 11 \times 4 \times 3 \equiv 132 \equiv 2 \pmod{13}$

Thus, the rules for ordinary arithmetic involving addition, subtraction, and multiplication carry over into modular arithmetic.

Table 4.1 provides an illustration of modular addition and multiplication modulo 8. Looking at addition, the results are straightforward and there is a regular pattern to the matrix. Both matrices are symmetric about the main diagonal, in conformance to the commutative property of addition and multiplication. As in ordinary addition, there is an additive inverse, or negative, to each integer in modular arithmetic. In this case, the negative of an integer x is the integer y such that (x + y) mod 8 = 0. To find the additive inverse of an integer in the left-hand column, scan across the corresponding row of the matrix to find the value 0; the integer at the top of that column is the additive inverse; thus (2 + 6) mod 8 = 0. Similarly, the entries in the multiplication table are straightforward. In ordinary arithmetic, there is a multiplicative inverse, or reciprocal, to each integer. In modular arithmetic mod 8, the multiplicative inverse of x is the integer y such that (x x y) mod 8 = 1 mod 8. Now, to find the multiplicative inverse of an integer from the multiplication table, scan across the matrix in the row for that integer to find the value 1; the integer at the top of that column is the multiplicative inverse;

thus (3 x 3) mod 8 = 1. Note that not all integers mod 8 have a multiplicative inverse; more about that later.

Table 4.1. Arithmetic Modulo 8

+	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

(a) Addition modulo 8

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

(b) Multiplication modulo 8

w	$-w$	w^{-1}
0	0	—
1	7	1
2	6	—
3	5	3
4	4	—
5	3	5
6	2	—
7	1	7

(c) Additive and multiplicative inverses modulo 8

Properties of Modular Arithmetic

Define the set Z_n as the set of nonnegative integers less than n:

$Z_n = \{0, 1,...,(n\ 1)\}$

This is referred to as the set of residues, or residue classes modulo n. To be more precise, each integer in Z_n represents a residue class. We can label the residue classes modulo n as [0], [1], [2],...,[n 1], where

$[r] = \{a: a \text{ is an integer}, a \equiv r \pmod n\}$

The residue classes modulo 4 are

The residue classes modulo 4 are	
	[0] = { ..., 16, 12, 8, 4, 0, 4, 8, 12, 16,... }
	[1] = { ..., 15, 11, 7, 3, 1, 5, 9, 13, 17,... }
	[2] = { ..., 14, 10, 6, 2, 2, 6, 10, 14, 18,... }
	[3] = { ..., 13, 9, 5, 1, 3, 7, 11, 15, 19,... }

Of all the integers in a residue class, the smallest nonnegative integer is the one usually used to represent the residue class. Finding the smallest nonnegative integer to which k is congruent modulo n is called reducing k modulo n.

If we perform modular arithmetic within Z_n, the properties shown in Table 4.2 hold for integers in Z_n. Thus, Z_n is a commutative ring with a multiplicative identity element (Figure 4.1).

Table 4.2. Properties of Modular Arithmetic for Integers in Z_n

Property	Expression
Commutative laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive laws	$[w + (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$ $[w + (x \times y)] \bmod n = [(w + x) \times (w + y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 + w) \bmod n = w \bmod n$
Additive inverse (-w)	For each $w \equiv Z_n$, there exists a z such that $w + z$ 0 mod n

There is one peculiarity of modular arithmetic that sets it apart from ordinary arithmetic. First, observe that, as in ordinary arithmetic, we can write the following:

Equation 4-2

$$\text{if } (a + b) \equiv (a + c) \pmod{n} \quad \text{then} \quad b \equiv c \pmod{n}$$

(5 + 23) ≡ (5 + 7)(mod 8}; 23 ≡ 7 (mod 8)

Equation (4.2) is consistent with the existence of an additive inverse. Adding the additive inverse of a to both sides of Equation (4.2), we have:

((a) + a + b) ≡ ((a) + a + c)(mod n)

b ≡ c (mod n)

However, the following statement is true only with the attached condition:

Equation 4-3

$$\text{if } (a \times b) \equiv (a \times c) \pmod{n} \text{ then } b \equiv c \pmod{n} \quad \text{if } a \text{ is relatively prime to } n$$

where the term relatively prime is defined as follows: two integers are relatively prime if their only common positive integer factor is 1. Similar to the case of Equation (4.2), we can say that Equation (4.3) is consistent with the existence of a multiplicative inverse. Applying the multiplicative inverse of a to both sides of Equation (4.2), we have:

$((a^1)ab) \equiv ((a^1)ac)(\text{mod } n)$

b ≡ c (mod n)

To see this, consider an example in which the condition of Equation (4.3) does not hold. The integers 6 and 8 are not relatively prime, since they have the common factor 2. We have the following:

6 x 3 = 18 ≡ 2 (mod 8)

6 x 7 = 42 ≡ 2 (mod 8)

Yet 3 ≢7 (mod 8).

The reason for this strange result is that for any general modulus n, a multiplier a that is applied in turn to the integers 0 through (n 1) will fail to produce a complete set of residues if a and n have any factors in common.

With a = 6 and n = 8,

Z_8	0	1	2	3	4	5	6	7
Multiply by 6	0	6	12	18	24	30	36	42
Residues	0	6	4	2	0	6	4	2

Because we do not have a complete set of residues when multiplying by 6, more than one integer in Z_8 maps into the same residue. Specifically, 6 x 0 mod 8 = 6 x 4 mod 8; 6 x 1 mod 8 = 6 x 5 mod 8; and so on. Because this is a many-to-one mapping, there is not a unique inverse to the multiply operation.

However, if we take a = 5 and n = 8, whose only common factor is 1,

Z_8	0	1	2	3	4	5	6	7
Multiply by 6	0	5	10	15	20	25	30	35
Residues	0	5	2	7	4	1	6	3

The line of residues contains all the integers in Z_8, in a different order.

In general, an integer has a multiplicative inverse in Z_n if that integer is relatively prime to n. Table 4.1c shows that the integers 1, 3, 5, and 7 have a multiplicative inverse in Z_8, but 2, 4, and 6 do not.

Multiple Encryption and Triple DES

Given the potential vulnerability of DES to a brute-force attack, there has been considerable interest in finding an alternative. One approach is to design a completely new algorithm, of which AES is a prime example. Another alternative, which would preserve the existing investment in software and equipment, is to use multiple encryption with DES and

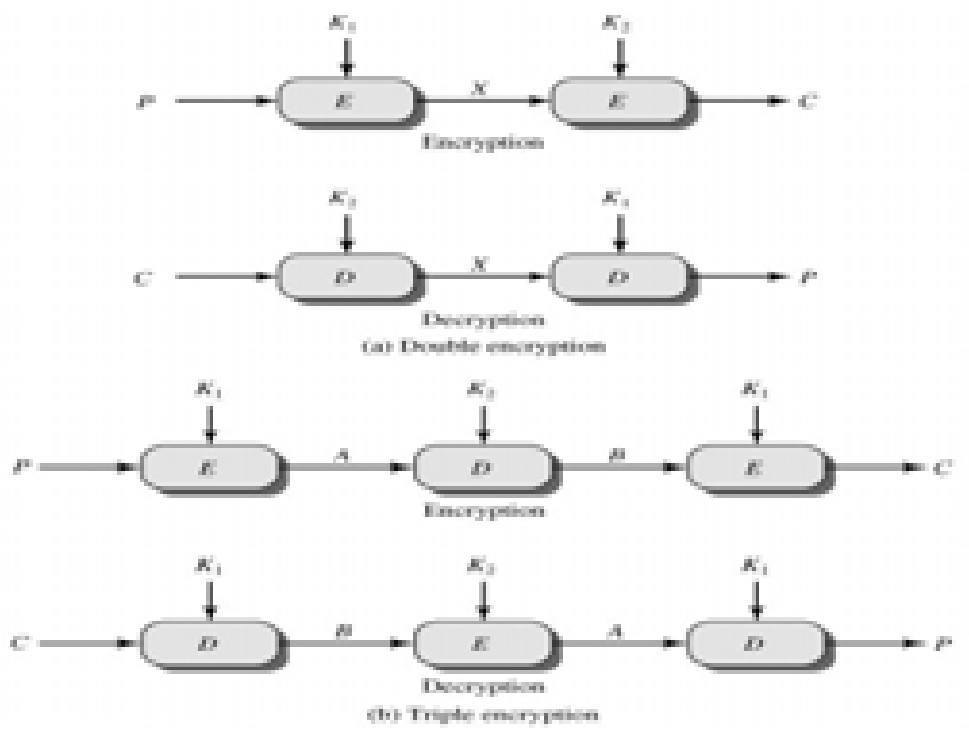
multiple keys. We begin by examining the simplest example of this second alternative. We then look at the widely accepted triple DES (3DES) approach.

Double DES

The simplest form of multiple encryptions has two encryption stages and two keys (Figure 6.1a). Given a plaintext P and two encryption keys K_1 and K_2, ciphertext C is generated as

$C = E(K_2, E(K_1, P))$

Figure 6.1. Multiple Encryption



Decryption requires that the keys be applied in reverse order:

$P = D(K_1, D(K_2, C))$

For DES, this scheme apparently involves a key length of 56 x 2 = 112 bits, of resulting in a dramatic increase in cryptographic strength. But we need to examine the algorithm more closely.

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

Equation 6-1

$$E(K_2, E(K_1, P)) = E(K_3, P)$$

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 (6.1) 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

$$\left(2^{64}\right)! = 10^{3473800000000000000} > \left(10^{10^{20}}\right)$$

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 proved .

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, It is based on the observation that, if we have

$C = E(K_2, E(K_1, P))$

then (see Figure 6.1a)

$X = E(K_1, P) = D(K_2, P)$

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, on average, for a given plaintext P, the number of different 112-bit keys that will produce a given ciphertext C is $2^{112}/2^{64} = 2^{48}$. Thus, the foregoing procedure will 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}, 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. This raises the cost of the known-plaintext attack to 2^{112}, which is beyond what is practical now and far into the future. However, it has the drawback of requiring a key length of 56 x 3 = 168 bits, which may be somewhat unwieldy.

As an alternative, Tuchman proposed a triple encryption method that uses only two keys [TUCH79]. The function follows an encrypt-decrypt-encrypt (EDE) sequence (Figure 6.1b):

$$C = E(K_1, D(K_2, E(K_1, P)))$$

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:

$$C = E(K_1, D(K_1, E(K_1, P))) = E(K_1, P)$$

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

[1] (ANS) American National Standard: 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.

Currently, there are no practical cryptanalytic attacks on 3DES. Coppersmith 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}.

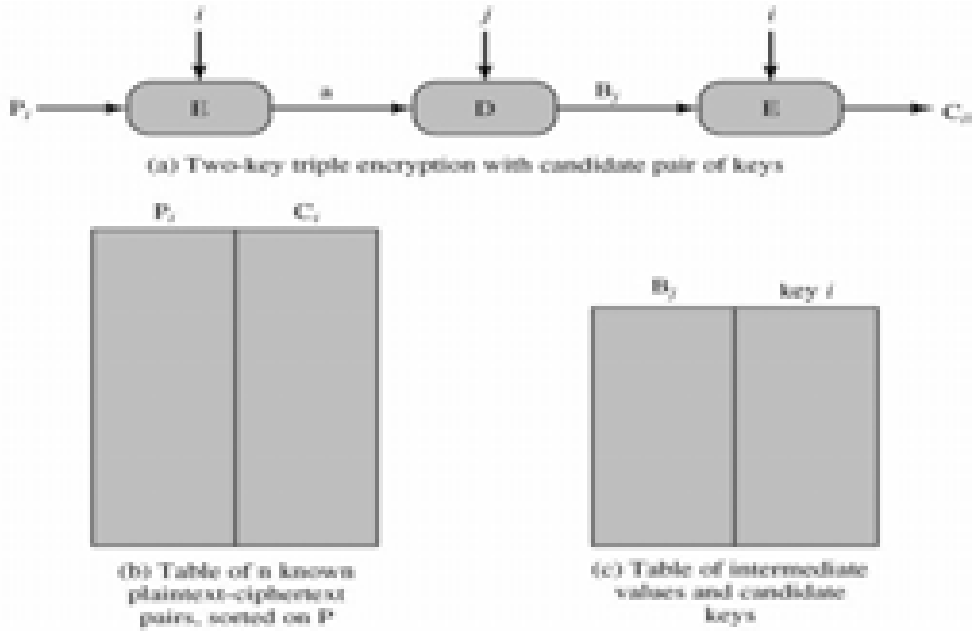
It is worth looking at several proposed attacks on 3DES that, although not practical, give a flavor for the types of attacks that have been considered and that could form the basis for more successful future attacks.

The first serious proposal came from Merkle and Hellman . Their plan involves finding plaintext values that produce a first intermediate value of A = 0 (Figure 6.1b) 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, a number unlikely to be provided by the holder of the keys.

A known-plaintext attack is outlined in . 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 6.1b), 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 6.2b).

Figure 6.2. Known-Plaintext Attack on Triple DES



2. Pick an arbitrary value a for A, and create a second table (Figure 6.2c) with entries defined in the following fashion. For each of the 2^{56} possible keys $K_1 = i$, calculate the plaintext value P_i that produces a:

   $P_i = D(i, a)$

   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:

   $B = D(i, C)$

   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:

   $B_j = D(j, a)$

   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 6.2a).

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

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

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 .Three-key 3DES has an effective key length of 168 bits and is defined as follows:

$C = E(K_3, D(K_2, E(K_1, P)))$

Backward compatibility with DES is provided by putting $K_3 = K_2$ or $K_1 = K_2$.

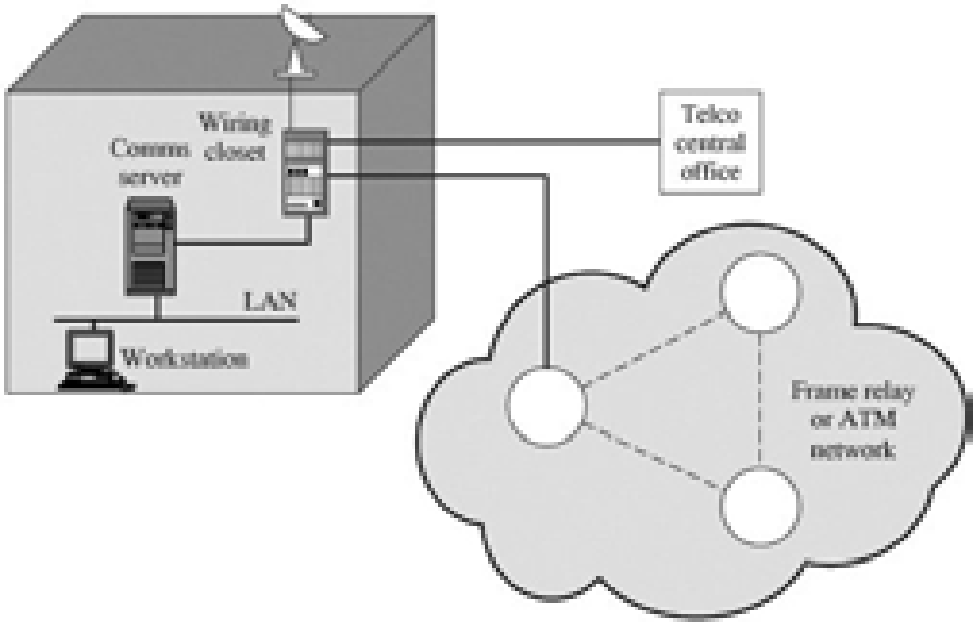
Placement of Encryption Function

If encryption is to be used to counter attacks on confidentiality, we need to decide what to encrypt and where the encryption function should be located. To begin, this section examines the potential locations of security attacks and then looks at the two major approaches to encryption placement: link and end to end.

Potential Locations for Confidentiality Attacks

As an example, consider a user workstation in a typical business organization. Figure 7.1 suggests the types of communications facilities that might be employed by such a workstation and therefore gives an indication of the points of vulnerability.

Figure 7.1. Points of Vulnerability



In most organizations, workstations are attached to local area networks (LANs). Typically, the user can reach other workstations, hosts, and servers directly on the LAN or on other LANs in the same building that are interconnected with bridges and routers. Here, then, is the first point of vulnerability. In this case, the main concern is eavesdropping by another

employee. Typically, a LAN is a broadcast network: Transmission from any station to any other station is visible on the LAN medium to all stations. Data are transmitted in the form of frames, with each frame containing the source and destination address. An eavesdropper can monitor the traffic on the LAN and capture any traffic desired on the basis of source and destination addresses. If part or all of the LAN is wireless, then the potential for eavesdropping is greater.

Furthermore, the eavesdropper need not necessarily be an employee in the building. If the LAN, through a communications server or one of the hosts on the LAN, offers a dial-in capability, then it is possible for an intruder to gain access to the LAN and monitor traffic.

Access to the outside world from the LAN is almost always available in the form of a router that connects to the Internet, a bank of dial-out modems, or some other type of communications server. From the communications server, there is a line leading to a wiring closet. The wiring closet serves as a patch panel for interconnecting internal data and phone lines and for providing a staging point for external communications.

The wiring closet itself is vulnerable. If an intruder can penetrate to the closet, he or she can tap into each wire to determine which are used for data transmission. After isolating one or more lines, the intruder can attach a low-power radio transmitter. The resulting signals can be picked up from a nearby location (e.g., a parked van or a nearby building).

Several routes out of the wiring closet are possible. A standard configuration provides access to the nearest central office of the local telephone company. Wires in the closet are gathered into a cable, which is usually consolidated with other cables in the basement of the building. From there, a larger cable runs underground to the central office.

In addition, the wiring closet may provide a link to a microwave antenna, either an earth station for a satellite link or a point-to-point terrestrial microwave link. The antenna link can be part of a private network, or it can be a local bypass to hook in to a long-distance carrier.

The wiring closet may also provide a link to a node of a packet-switching network. This link can be a leased line, a direct private line, or a switched connection through a public telecommunications network. Inside the

network, data pass through a number of nodes and links between nodes until the data arrive at the node to which the destination end system is connected.

An attack can take place on any of the communications links. For active attacks, the attacker needs to gain physical control of a portion of the link and be able to insert and capture transmissions. For a passive attack, the attacker merely needs to be able to observe transmissions. The communications links involved can be cable (telephone twisted pair, coaxial cable, or optical fiber), microwave links, or satellite channels. Twisted pair and coaxial cable can be attacked using either invasive taps or inductive devices that monitor electromagnetic emanations. Invasive taps allow both active and passive attacks, whereas inductive taps are useful for passive attacks. Neither type of tap is as effective with optical fiber, which is one of the advantages of this medium. The fiber does not generate electromagnetic emanations and hence is not vulnerable to inductive taps. Physically breaking the cable seriously degrades signal quality and is therefore detectable. Microwave and satellite transmissions can be intercepted with little risk to the attacker. This is especially true of satellite transmissions, which cover a broad geographic area. Active attacks on microwave and satellite are also possible, although they are more difficult technically and can be quite expensive.

In addition to the potential vulnerability of the various communications links, the various processors along the path are themselves subject to attack. An attack can take the form of attempts to modify the hardware or software, to gain access to the memory of the processor, or to monitor the electromagnetic emanations. These attacks are less likely than those involving communications links but are nevertheless a source of risk.

Thus, there are a large number of locations at which an attack can occur. Furthermore, for wide area communications, many of these locations are not under the physical control of the end user. Even in the case of local area networks, in which physical security measures are possible, there is always the threat of the disgruntled employee.

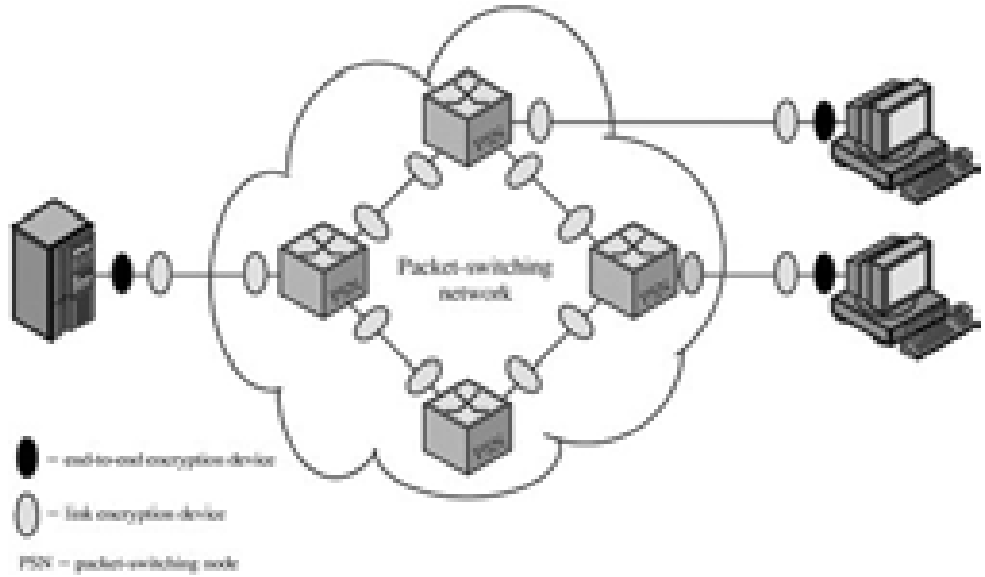
Link versus End-to-End Encryption

The most powerful and most common approach to securing the points of vulnerability highlighted in the preceding section is encryption. If encryption is to be used to counter these attacks, then we need to decide what to encrypt

and where the encryption gear should be located. As Figure 7.2 indicates, there are two fundamental alternatives: link encryption and end-to-end encryption.

Figure 7.2. Encryption Across a Packet-Switching Network

(This item is displayed on page 204 in the print version)



Basic Approaches

With link encryption, each vulnerable communications link is equipped on both ends with an encryption device. Thus, all traffic over all communications links is secured. Although this recourse requires a lot of encryption devices in a large network, its value is clear. One of its disadvantages is that the message must be decrypted each time it enters a switch (such as a frame relay switch) because the switch must read the address (logical connection number) in the packet header in order to route the frame. Thus, the message is vulnerable at each switch. If working with a public network, the user has no control over the security of the nodes.

Several implications of link encryption should be noted. For this strategy to be effective, all the potential links in a path from source to destination must use link encryption. Each pair of nodes that share a link should share a

unique key, with a different key used on each link. Thus, many keys must be provided.

With end-to-end encryption, the encryption process is carried out at the two end systems. The source host or terminal encrypts the data. The data in encrypted form are then transmitted unaltered across the network to the destination terminal or host. The destination shares a key with the source and so is able to decrypt the data. This plan seems to secure the transmission against attacks on the network links or switches. Thus, end-to-end encryption relieves the end user of concerns about the degree of security of networks and links that support the communication. There is, however, still a weak spot.

Consider the following situation. A host connects to a frame relay or ATM network, sets up a logical connection to another host, and is prepared to transfer data to that other host by using end-to-end encryption. Data are transmitted over such a network in the form of packets that consist of a header and some user data. What part of each packet will the host encrypt? Suppose that the host encrypts the entire packet, including the header. This will not work because, remember, only the other host can perform the decryption. The frame relay or ATM switch will receive an encrypted packet and be unable to read the header. Therefore, it will not be able to route the packet. It follows that the host may encrypt only the user data portion of the packet and must leave the header in the clear.

Thus, with end-to-end encryption, the user data are secure. However, the traffic pattern is not, because packet headers are transmitted in the clear. On the other hand, end-to-end encryption does provide a degree of authentication. If two end systems share an encryption key, then a recipient is assured that any message that it receives comes from the alleged sender, because only that sender shares the relevant key. Such authentication is not inherent in a link encryption scheme.

To achieve greater security, both link and end-to-end encryption are needed, as is shown in Figure 7.2. When both forms of encryption are employed, the host encrypts the user data portion of a packet using an end-to-end encryption key. The entire packet is then encrypted using a link encryption

key. As the packet traverses the network, each switch decrypts the packet, using a link encryption key to read the header, and then encrypts the entire packet again for sending it out on the next link. Now the entire packet is secure except for the time that the packet is actually in the memory of a packet switch, at which time the packet header is in the clear.

Table 7.1 summarizes the key characteristics of the two encryption strategies.

Table 7.1. Characteristics of Link and End-to-End Encryption

Link Encryption	End-to-End Encryption
Security within End Systems and Intermediate Systems	
Message exposed in sending host	Message encrypted in sending host
Message exposed in intermediate nodes	Message encrypted in intermediate nodes
Role of User	
Applied by sending host	Applied by sending process
Transparent to user	User applies encryption
Host maintains encryption facility	User must determine algorithm
One facility for all users	Users selects encryption scheme
Can be done in hardware	Software implementation
All or no messages encrypted	User chooses to encrypt, or not, for each message
Implementation Concerns	
Requires one key per (host-intermediate node) pair and (intermediate node-intermediate node) pair	Requires one key per user pair
Provides host authentication	Provides user authentication

Logical Placement of End-to-End Encryption Function

With link encryption, the encryption function is performed at a low level of the communications hierarchy. In terms of the Open Systems Interconnection (OSI) model, link encryption occurs at either the physical or link layers.

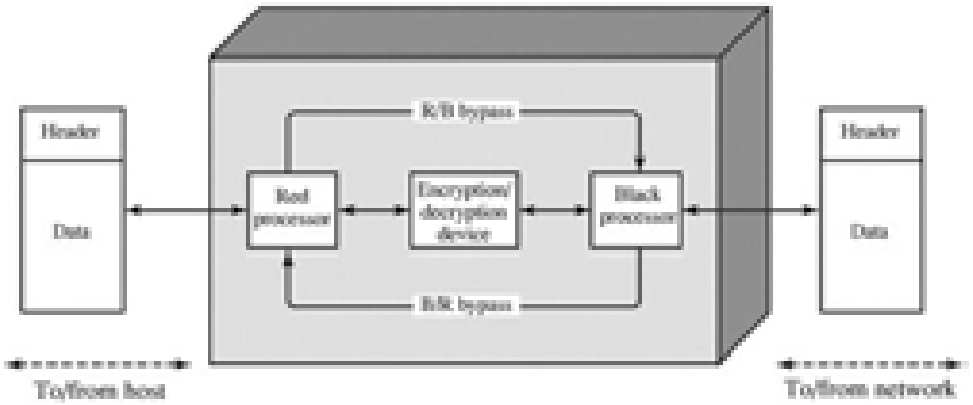
For end-to-end encryption, several choices are possible for the logical placement of the encryption function. At the lowest practical level, the encryption function could be performed at the network layer. Thus, for example, encryption could be associated with the frame relay or ATM protocol, so that the user data portion of all frames or ATM cells is encrypted.

With network-layer encryption, the number of identifiable and separately protected entities corresponds to the number of end systems in the network. Each end system can engage in an encrypted exchange with another end system if the two share a secret key. All the user processes and applications within each end system would employ the same encryption scheme with the same key to reach a particular target end system. With this arrangement, it might be desirable to off-load the encryption function to some sort of front-end processor (typically a communications board in the end system).

Figure 7.3 shows the encryption function of the front-end processor (FEP). On the host side, the FEP accepts packets. The user data portion of the packet is encrypted, while the packet header bypasses the encryption process.[1] The resulting packet is delivered to the network. In the opposite direction, for packets arriving from the network, the user data portion is decrypted and the entire packet is delivered to the host. If the transport layer functionality (e.g., TCP) is implemented in the front end, then the transport-layer header would also be left in the clear and the user data portion of the transport protocol data unit is encrypted.

[1] The terms red and black are frequently used. Red data are sensitive or classified data in the clear. Black data are encrypted data.

Figure 7.3. Front-End Processor Function

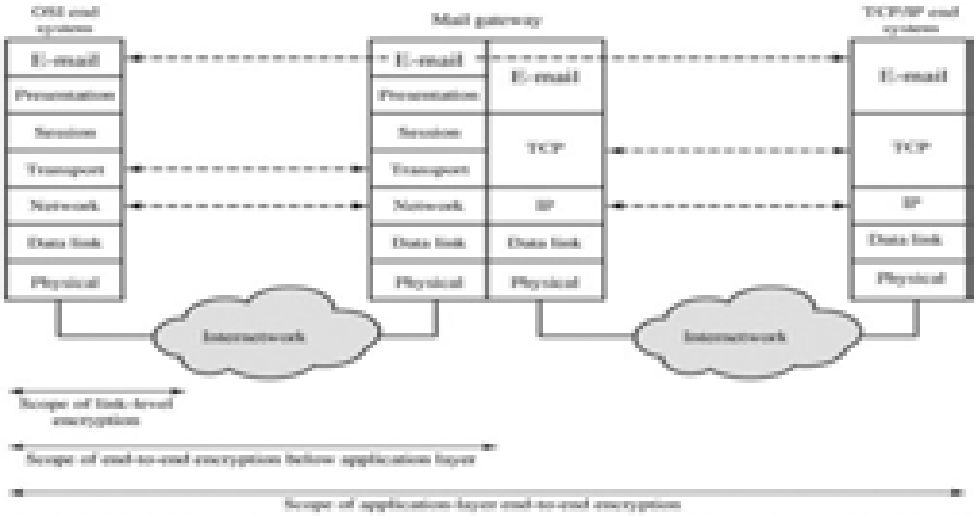


Deployment of encryption services on end-to-end protocols, such as a network-layer frame relay or TCP, provides end-to-end security for traffic within a fully integrated internetwork. However, such a scheme cannot deliver the necessary service for traffic that crosses internetwork boundaries, such as electronic mail, electronic data interchange (EDI), and file transfers.

Figure 7.4 illustrates the issues involved. In this example, an electronic mail gateway is used to interconnect an internetwork that uses an OSI-based architecture with one that uses a TCP/IP-based architecture.[2] In such a configuration, there is no end-to-end protocol below the application layer. The transport and network connections from each end system terminate at the mail gateway, which sets up new transport and network connections to link to the other end system. Furthermore, such a scenario is not limited to the case of a gateway between two different architectures. Even if both end systems use TCP/IP or OSI, there are plenty of instances in actual configurations in which mail gateways sit between otherwise isolated internetworks. Thus, for applications like electronic mail that have a store-and-forward capability, the only place to achieve end-to-end encryption is at the application layer.

[2] Appendix H provides a brief overview of the OSI and TCP/IP protocol architectures.

Figure 7.4. Encryption Coverage Implications of Store-and-Forward Communications

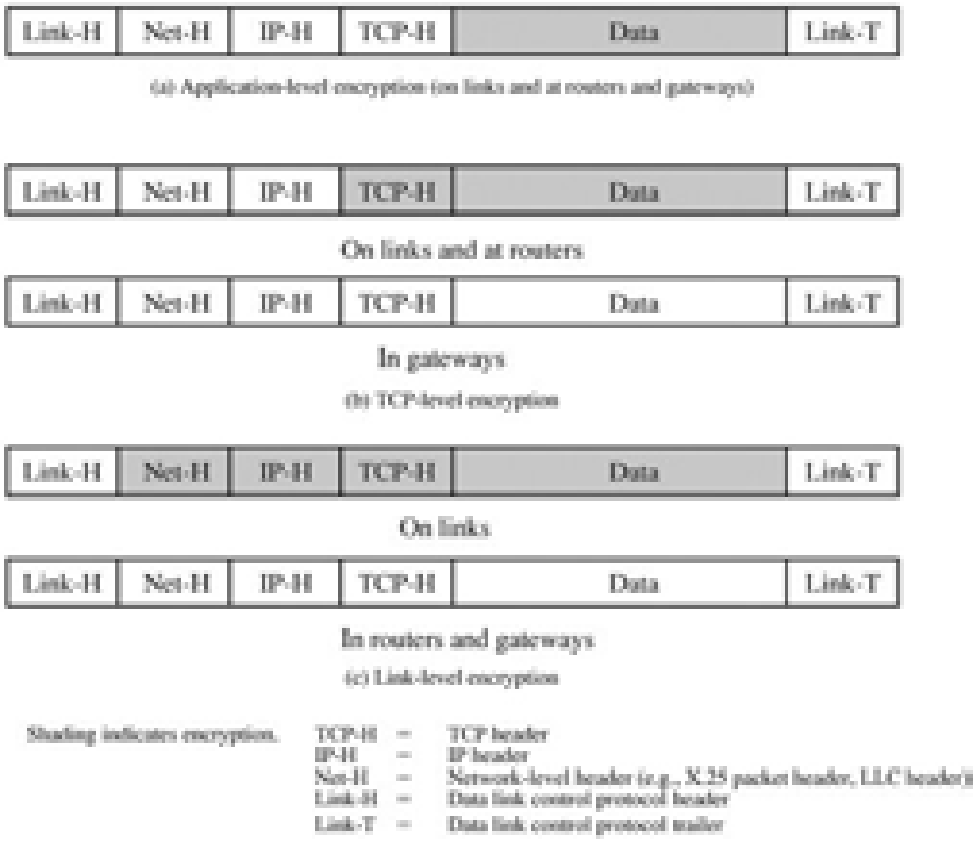


A drawback of application-layer encryption is that the number of entities to consider increases dramatically. A network that supports hundreds of hosts may support thousands of users and processes. Thus, many more secret keys need to be generated and distributed.

An interesting way of viewing the alternatives is to note that as we move up the communications hierarchy, less information is encrypted but it is more secure. Figure 7.5 highlights this point, using the TCP/IP architecture as an example. In the figure, an application-level gateway refers to a store-and-forward device that operates at the application level.

Unfortunately, most TCP/IP documents use the term gateway to refer to what is more commonly referred to as a router.

Figure 7.5. Relationship between Encryption and Protocol Levels



With application-level encryption (Figure 7.5a), only the user data portion of a TCP segment is encrypted. The TCP, IP, network-level, and link-level headers and link-level trailer are in the clear. By contrast, if encryption is performed at the TCP level (Figure 7.5b), then, on a single end-to-end connection, the user data and the TCP header are encrypted. The IP header remains in the clear because it is needed by routers to route the IP datagram from source to destination. Note, however, that if a message passes through a gateway, the TCP connection is terminated and a new transport connection is opened for the next hop. Furthermore, the gateway is treated as a destination by the underlying IP. Thus, the encrypted portions of the data unit are decrypted at the gateway. If the next hop is over a TCP/IP network, then the user data and TCP header are encrypted again before transmission. However, in the gateway itself the data unit is buffered entirely in the clear. Finally, for link-level encryption (Figure 7.5c), the entire data unit except for the link header and trailer is encrypted on each link, but the entire data unit is in the clear at each router and gateway.

The figure actually shows but one alternative. It is also possible to encrypt part or even all of the link header and trailer except for the starting and ending frame flags.

Polynomial Arithmetic

Before pursuing our discussion of finite fields, we need to introduce the interesting subject of polynomial arithmetic. We are concerned with polynomials in a single variable x, and we can distinguish three classes of polynomial arithmetic:

- Ordinary polynomial arithmetic, using the basic rules of algebra
- Polynomial arithmetic in which the arithmetic on the coefficients is performed modulo p; that is, the coefficients are in GF(p)
- Polynomial arithmetic in which the coefficients are in GF(p), and the polynomials are defined modulo a polynomial m(x) whose highest power is some integer n

This section examines the first two classes, and the next section covers the last class.

Ordinary Polynomial Arithmetic

A **polynomial** of degree n (integer $n \geq 0$) is an expression of the form

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i$$

where the a_i are elements of some designated set of numbers S, called the **coefficient set**, and $a_n \neq 0$. We say that such polynomials are defined over the coefficient set S.

A zeroth-degree polynomial is called a constant polynomial and is simply an element of the set of coefficients. An nth-degree polynomial is said to be a **monic polynomial** if $a_n = 1$.

In the context of abstract algebra, we are usually not interested in evaluating a polynomial for a particular value of x [e.g., f(7)]. To emphasize this point, the variable x is sometimes referred to as the indeterminate.

Polynomial arithmetic includes the operations of addition, subtraction, and multiplication. These operations are defined in a natural way as though the variable x was an element of S. Division is similarly defined, but requires that S be a field. Examples of fields include the real numbers, rational numbers, and Z_p for p prime. Note that the set of all integers is not a field and does not support polynomial division.

Addition and subtraction are performed by adding or subtracting corresponding coefficients. Thus, if

$$f(x) = \sum_{i=0}^{n} a_i x^i; \quad g(x) = \sum_{i=0}^{m} b_i x^i; \quad n \geq m$$

Then addition is defined as

$$f(x) + g(x) = \sum_{i=0}^{m} (a_i + b_i) x^i + \sum_{i=m+1}^{n} a_i x^i$$

and multiplication is defined as

$$f(x) \times g(x) = \sum_{i=0}^{n+m} c_i x^i$$

where

$c_k = a_0 b_{k1} + a_1 b_{k1} + ... + a_{k1} b_1 + a_k b_0$

In the last formula, we treat a_i as zero for i > n and b_i as zero for i > m. Note that the degree of the product is equal to the sum of the degrees of the two polynomials.

As an example, let $f(x) = x^3 + x^2 + 2$ and $g(x) = x^2$ x + 1, where S is the set of integers. Then

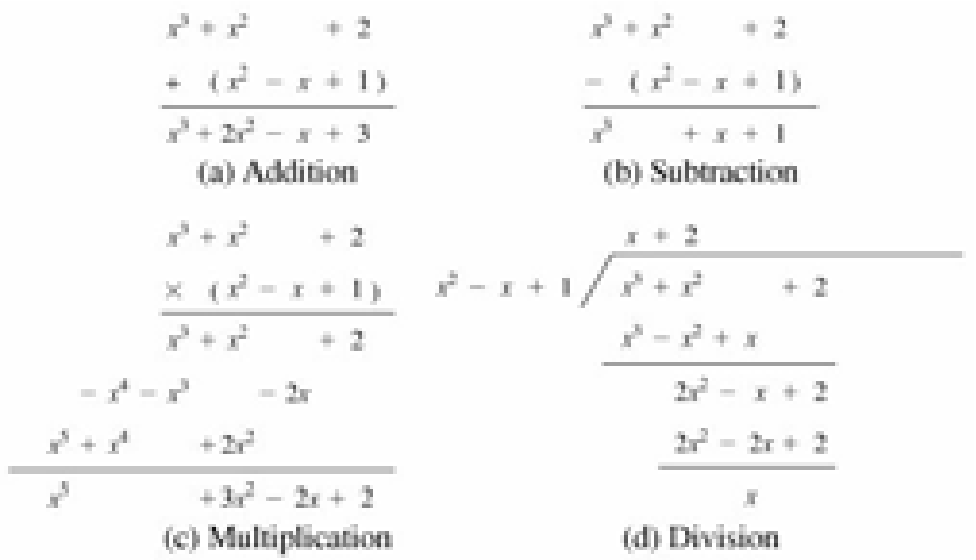
$f(x) + g(x) = x^3 + 2x^2$ x + 3

f(x) g(x) = x^3 + x + 1

f(x) x g(x) = $x^5 + 3x^2$ 2x + 2

Figures 4.3a through 4.3c show the manual calculations. We comment on division subsequently.

Figure 4.3. Examples of Polynomial Arithmetic



Polynomial Arithmetic with Coefficients in Zp

Let us now consider polynomials in which the coefficients are elements of some field F. We refer to this as a polynomial over the field F. In that case, it is easy to show that the set of such polynomials is a ring, referred to as a **polynomial ring**. That is, if we consider each distinct polynomial to be an element of the set, then that set is a ring.

In fact, the set of polynomials whose coefficients are elements of a commutative ring forms a polynomial ring, but that is of no interest in the present context.

When polynomial arithmetic is performed on polynomials over a field, then division is possible. Note that this does not mean that exact division is possible. Let us clarify this distinction. Within a field, given two elements a and b, the quotient a/b is also an element of the field. However, given a ring R that is not a field, in general division will result in both a quotient and a remainder; this is not exact division.

Consider the division 5/3 within a set S. If S is the set of rational numbers, which is a field, then the result is simply expressed as 5/3 and is an element of S. Now suppose that S is the field Z_7. In this case, we calculate (using Table 4.3c):

$5/3 = (5 \times 3^{1}) \bmod 7 = (5 \times 5) \bmod 7 = 4$

which is an exact solution. Finally, suppose that S is the set of integers, which is a ring but not a field. Then 5/3 produces a quotient of 1 and a remainder of 2:

$5/3 = 1 + 2/3$

$5 = 1 \times 3 + 2$

Thus, division is not exact over the set of integers.

Now, if we attempt to perform polynomial division over a coefficient set that is not a field, we find that division is not always defined.

If the coefficient set is the integers, then $(5x^2)/(3x)$ does not have a solution, because it would require a coefficient with a value of 5/3, which is not in the coefficient set. Suppose that we perform the same polynomial division over Z_7. Then we have $(5x^2)/(3x) = 4x$ which is a valid polynomial over Z_7.

However, as we demonstrate presently, even if the coefficient set is a field, polynomial division is not necessarily exact. In general, division will produce a quotient and a remainder:

Equation 4-6

$$\frac{f(x)}{g(x)} = q(x) + \frac{r(x)}{g(x)}$$
$$f(x) = q(x)g(x) + r(x)$$

If the degree of f(x) is n and the degree of g(x) is m, ($m \geq n$), then the degree of the quotient q(x) m n is and the degree of the remainder is at most m - 1.

With the understanding that remainders are allowed, we can say that polynomial division is possible if the coefficient set is a field.

In an analogy to integer arithmetic, we can write f(x) mod g(x) for the remainder r(x) in Equation (4.6). That is, r(x) = f(x) mod g(x). If there is no remainder [i.e., r(x) = 0], then we can say g(x) divides f(x), written as g(x)|f(x); equivalently, we can say that g(x) is a factor of f(x) or g(x) is a divisor of f(x).

For the preceding example and [$f(x) = x^3 + x^2 + 2$ and $g(x) = x^2 x + 1$], f(x)/g(x) produces a quotient of q(x) = x + 2 and a remainder r(x) = x as shown in Figure 4.3d. This is easily verified by noting that

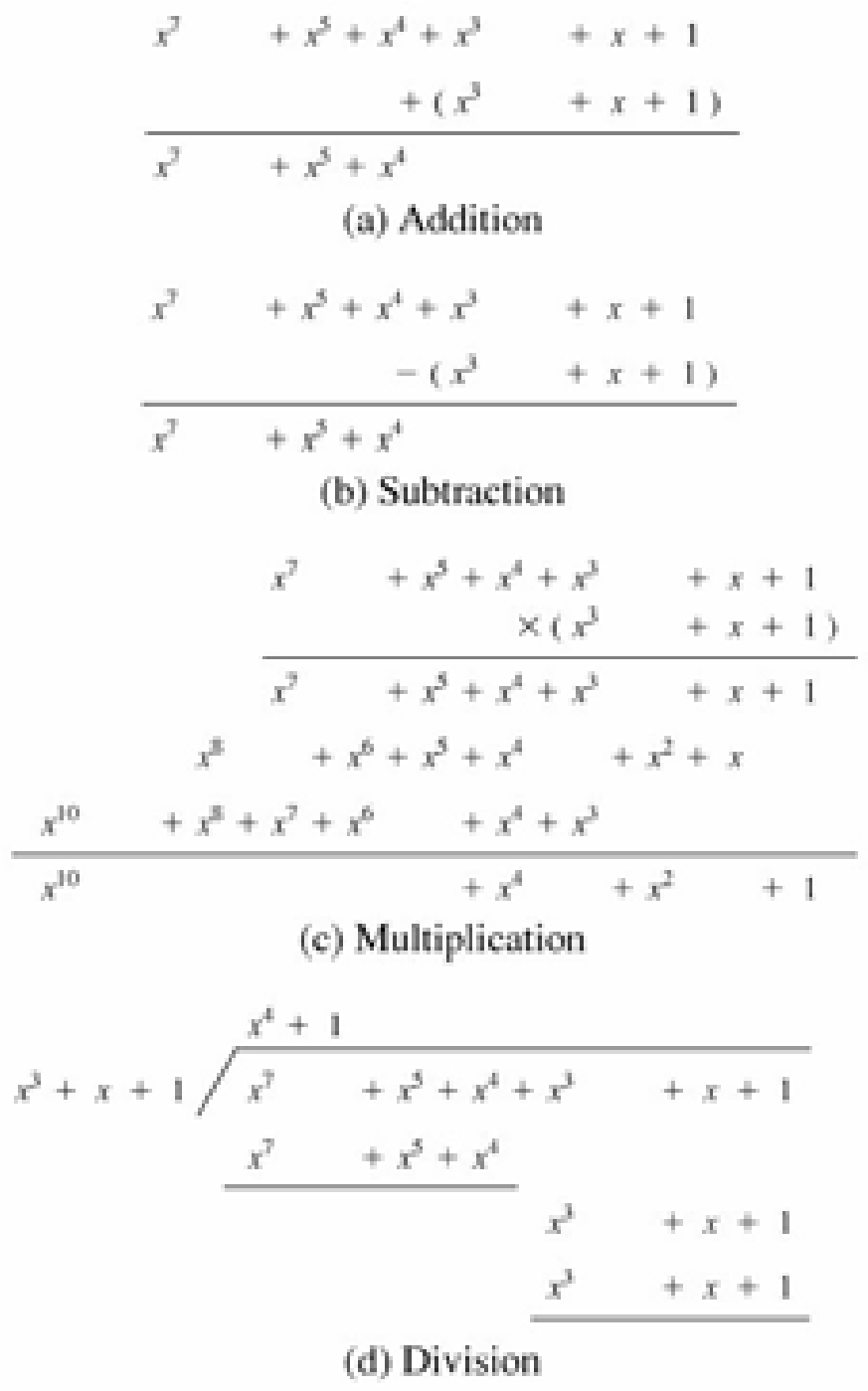
$q(x)g(x) + r(x) = (x + 2)(x^2 x + 1) + x = (x^3 + x^2 x + 2) + x$

$= x^3 + x^2 + 2 = f(x)$

For our purposes, polynomials over GF (2) are of most interest. That in GF(2), addition is equivalent to the XOR operation, and multiplication is equivalent to the logical AND operation. Further, addition and subtraction are equivalent mod 2: 1 + 1 = 1 1 = 0; 1 + 0 = 1 0 = 1; 0 + 1 = 0 1 = 1.

Figure 4.4 shows an example of polynomial arithmetic over GF(2). For $f(x) = (x^7 + x^5 + x^4 + x^3 + x + 1)$ and $g(x) = (x^3 + x + 1)$, the figure shows f(x) + g(x); f(x) g(x); f(x) x g(x); and f(x)/g(x). Note that g(x)|f(x)

Figure 4.4. Examples of Polynomial Arithmetic over GF(2)



A polynomial f(x) over a field F is called irreducible if and only if f(x) cannot be expressed as a product of two polynomials, both over F, and both

of degree lower than that of f(x). By analogy to integers, an irreducible polynomial is also called a **prime polynomial**.

The polynomial $f(x) = x^4 + 1$ over GF(2) is reducible, because $x^4 + 1 = (x + 1)(x^3 + x^2 + x + 1)$

Consider the polynomial $f(x) = x^3 + x + 1$. It is clear by inspection that x is not a factor of f(x). We easily show that x + 1 is not a factor of f(x):

$$
\begin{array}{r}
x^2 + x \\
x + 1 \overline{)\, x^3 \quad + x + 1} \\
\underline{x^3 + x^2} \\
\quad x^2 + x \\
\quad \underline{x^2 + x} \\
\qquad\quad 1
\end{array}
$$

Thus f(x) has no factors of degree 1. But it is clear by inspection that if f(x) is reducible, it must have one factor of degree 2 and one factor of degree 1. Therefore, f(x) is irreducible.

Finding the Greatest Common Divisor

We can extend the analogy between polynomial arithmetic over a field and integer arithmetic by defining the greatest common divisor as follows. The polynomial c(x) is said to be the greatest common divisor of a(x) and b(x) if

1. c(x) divides both a(x) and b(x);
2. any divisor of a(x) and b(x) is a divisor of c(x).

An equivalent definition is the following: gcd[a(x), b(x)] is the polynomial of maximum degree that divides both a(x) and b(x).

We can adapt the Euclidean algorithm to compute the greatest common divisor of two polynomials. The equality in Equation (4.4) can be rewritten as the following theorem:

Equation 4-7

$$\gcd[a(x), b(x)] = \gcd[b(x), a(x) \bmod b(x)]$$

The Euclidean algorithm for polynomials can be stated as follows. The algorithm assumes that the degree of a(x) is greater than the degree of b(x). Then, to find gcd[a(x), b(x)],

EUCLID[a(x), b(x)]
1. A(x)← a(x); B(x) ← b(x)
2. if B(x) = 0 return A(x) = gcd[a(x), b(x)]
3. R(x) = A(x) mod B(x)
4. A(x) ← B(x)
5. B(x) ← R(x)
6. goto 2

Find gcd[a(x), b(x)] for $a(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ and $b(x) = x^4 + x^2 + x + 1$.

$$
\begin{array}{r}
x^2 + x \\
x^4 + x^2 + x + 1 \overline{)\, x^6 + x^5 + x^4 + x^3 + x^2 + x + 1} \\
\underline{x^6 \quad + x^4 + x^3 + x^2} \\
x^5 \quad + x + 1 \\
\underline{x^5 \quad + x^3 + x^2 + x} \\
x^3 + x^2 \quad + 1
\end{array}
$$

A(x) = a(x); B(x) = b(x)

R(x) = A(x) mod B(x) = $x^3 + x^2 + 1$

A(x) = $x^4 + x^2 + x + 1$; B(x) = $x^3 + x^2 + 1$

Summary

We began this section with a discussion of arithmetic with ordinary polynomials. In ordinary polynomial arithmetic, the variable is not evaluated; that is, we do not plug a value in for the variable of the polynomials. Instead, arithmetic operations are performed on polynomials (addition, subtraction, multiplication, division) using the ordinary rules of algebra. Polynomial division is not allowed unless the coefficients are elements of a field.

Next, we discussed polynomial arithmetic in which the coefficients are elements of GF(p). In this case, polynomial addition, subtraction, multiplication, and division are allowed. However, division is not exact; that is, in general division results in a quotient and a remainder.

Finally, we showed that the Euclidean algorithm can be extended to find the greatest common divisor of two polynomials whose coefficients are elements of a field.

All of the material in this section provides a foundation for the following section, in which polynomials are used to define finite fields of order p^n.

Polynomials with Coefficients in GF (2^8)

we discussed polynomial arithmetic in which the coefficients are in Z_p and the polynomials are defined modulo a polynomial M(x) whose highest power is some integer n. In this case, addition and multiplication of coefficients occurred within the field Z_p; that is, addition and multiplication were performed modulo p.

The AES document defines polynomial arithmetic for polynomials of degree 3 or less with coefficients in GF(2^8). The following rules apply:

1. Addition is performed by adding corresponding coefficients in GF(2^8). if we treat the elements of GF(2^8) as 8-bit strings, then addition is equivalent to the XOR operation. So, if we have

   Equation 5-8

   $$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

   Equation 5-9

   $$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$$

   then

$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a^2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$

2. Multiplication is performed as in ordinary polynomial multiplication, with two refinements:
    a. Coefficients are multiplied in $GF(2^8)$.
    b. The resulting polynomial is reduced mod $(x^4 + 1)$.

We need to keep straight which polynomial we are talking about. that each element of $GF(2^8)$ is a polynomial of degree 7 or less with binary coefficients, and multiplication is carried out modulo a polynomial of degree 8. Equivalently, each element of $GF(2^8)$ can be viewed as an 8-bit byte whose bit values correspond to the binary coefficients of the corresponding polynomial. For the sets defined in this section, we are defining a polynomial ring in which each element of this ring is a polynomial of degree 3 or less with coefficients in $GF(2^8)$, and multiplication is carried out modulo a polynomial of degree 4. Equivalently, each element of this ring can be viewed as a 4-byte word whose byte values are elements of $GF(2^8)$ that correspond to the 8-bit coefficients of the corresponding polynomial.

We denote the modular product of a(x) and b(x) by $a(x) \oplus b(x)$. To compute $d(x) = a(x) \oplus b(x)$, the first step is to perform a multiplication without the modulo operation and to collect coefficients of like powers. Let us express this as $c(x) = a(x) \times b(x)$ Then

Equation 5-10

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

where

$c_0 = a_0 \cdot b_0$

$c_1 = (a1 \cdot b_0) \oplus (a0 \cdot b_1)$

$c_2 = (a2 \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_2)$

$c_3 = (a3 \cdot b_0) \oplus (a_2 \cdot b_1) \oplus (a_1 \cdot b_2) \qquad (a_0 \cdot b_3)$

$c_4 = (a_3 \cdot b_1) \oplus (a_2 \cdot b_2) \oplus (a_1 \cdot b_3)$

$c_5 = (a_3 \cdot b_2) \oplus (a_2 \cdot b_3)$

$c_6 = (a_3 \cdot b_3)$

The final step is to perform the modulo operation:

$d(x) = c(x) \bmod (x^4 + 1)$

That is, d(x) must satisfy the equation

$c(x) = [(x^4 + 1) \times q(x)] \oplus d(x)$

such that the degree of d(x) is 3 or less.

A practical technique for performing multiplication over this polynomial ring is based on the observation that

Equation 5-11

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4}$$

If we now combine Equations (5.10) and (5.11), we end up with

$d(x) = c(x) \bmod (x^4 + 1) = [c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0] \bmod (x^4 + 1)$

$= c_3x^3 + (c_2 \oplus c_6)x^2 + (c_1 \oplus c_5)x + (c_0 \oplus c_4)$

Expanding the c_i coefficients, we have the following equations for the coefficients of d(x):

$d_0 = (a_0 \cdot b_0) \oplus (a_3 \cdot b_1) \oplus (a) \cdot b_2) \oplus (a_1 \cdot b_3)$

$d_1 = (a_1 \cdot b_0) \oplus (a_0 \cdot b_1) \oplus (a_3 \cdot b_2) \oplus (a) \cdot b_3)$

$d_2 = (a_2) \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_2) \oplus (a_3 \cdot b_3)$

$d_3 = (a_3 \cdot b_0) \oplus (a) \cdot b_1) \oplus (a_1 \cdot b_2) \oplus (a_0 \cdot b_3)$

This can be written in matrix form:

Equation 5-12

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

MixColumns Transformation

In the discussion of MixColumns, it was stated that there were two equivalent ways of defining the transformation. The first is the matrix multiplication shown in Equation (5.3), repeated here:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

The second method is to treat each column of State as a four-term polynomial with coefficients in GF(2^8). Each column is multiplied modulo ($x^4 + 1$) by the fixed polynomial a(x), given by

$a(x = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$

From Equation (5.8), we have $a_3 = \{03\}$; $a_2 = \{01\}$; $a_0 = \{02\}$. For the jth column of State, we have the polynomial $col_j(x) = s_{3,j}x^3 + s_{2,j}x^2 + s_{1,j}x + s_{0,j}$. Substituting into Equation (5.12), we can express $d(x) = a(x) \times col_j(x)$ 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 (5.3).

Multiplication by x

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

$$c(x) = x \oplus b(x) = [x \times (b_3x^3) + b_2x^2 + b_1x + b_0)] \bmod (x^4 + 1)$$

$$= (b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod (x^4 + 1)$$

$$= b_2x^3 + b_1x^2 + b_0x + b_3$$

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

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

Prime Numbers

[1] In this section, unless otherwise noted, we deal only with the nonnegative integers. The use of negative integers would introduce no essential differences.

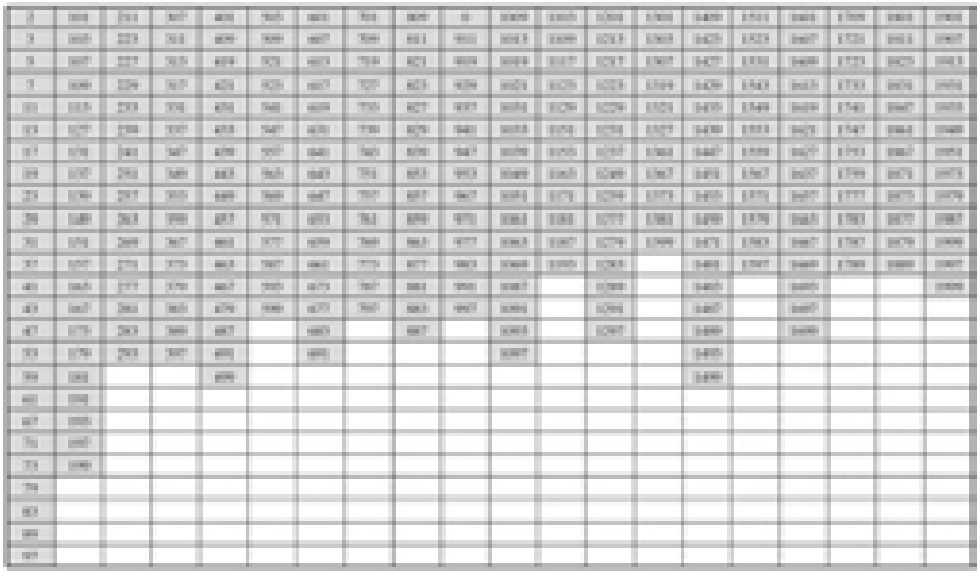
A central concern of number theory is the study of prime numbers. Indeed, whole books have been written on the subject .

An integer p > 1 is a prime number if and only if its only divisors are ± 1 and ±p. Prime numbers play a critical role in number theory and in the techniques discussed in this chapter. Table 8.1 shows the primes less than 2000. Note the way the primes are distributed. In particular, note the number of primes in each range of 100 numbers.

that integer a is said to be a divisor of integer b if there is no remainder on division. Equivalently, we say that a divides b.

Table 8.1. Primes under 2000

(This item is displayed on page 237 in the print version)



Any integer a > 1 can be factored in a unique way as

Equation 8-1

$$a = p_1^{a_1} p_2^{a_2} \ldots p_t^{a_t}$$

where p1 < p_2 < ... < p_t are prime numbers and where each is a positive integer. This is known as the fundamental theorem of arithmetic; a proof can be found in any text on number theory.

91	= 7 x 13
3600	= 2^4 x 3^2 x 5^2
11011	= 7 x 11^2 x 13

It is useful for what follows to express this another way. If P is the set of all prime numbers, then any positive integer a can be written uniquely in the following form:

$$a = \prod_{p \in P} p^{a_p} \quad \text{where each } a_p \geq 0$$

The right-hand side is the product over all possible prime numbers p; for any particular value of a, most of the exponents a_p will be 0.

The value of any given positive integer can be specified by simply listing all the nonzero exponents in the foregoing formulation.

The integer 12 is represented by {a_2 = 2, a_3 = 1}.
The integer 18 is represented by {a_2 = 1, a_3 = 2}.
The integer 91 is represented by {a_7 = 2, a_{13} = 1}.

Multiplication of two numbers is equivalent to adding the corresponding exponents. Given a $a = \prod_{p \in P} p^{a_p} b = \prod_{p \in P} p^{b_p}$ Define k = ab We know that the integer k can be expressed as the product of powers of primes: $k = \prod_{p \in P} p^{k_p}$. It follows that $k_p = a_p + b_p$ for all p P.

$k = 12 \times 18 = (2^2 \times 3) \times (2 \times 3^2) = 216$
$k_2 = 2 + 1 = 3$; $k_3 = 1 + 2 = 3$
$216 = 2^3 \times 3^3 = 8 \times 27$

What does it mean, in terms of the prime factors of a and b, to say that a divides b? Any integer of the form can be divided only by an integer that is of a lesser or equal power of the same prime number, p^j with $j \leq n$. Thus, we can say the following:

Given $a = \prod_{p \in P} p^{a_p}$, $b = \prod_{p \in P} p^{b_p}$ If a|b, then $a_p \leq b_p$ then for all p.

a	= 12;b = 36; 12|36
12	$= 2^2 \times 3$; $36 = 2^2 \times 3^2$
a_2	$= 2 = b_2$
a_3	$= 1 \leq 2 = b_3$
Thus, the inequality $a_p \leq b_p$ is satisfied for all prime numbers.	

It is easy to determine the greatest common divisor of two positive integers if we express each integer as the product of primes.

that the greatest common divisor of integers a and b, expressed gcd(a, b), is an integer c that divides both a and b without remainder and that any divisor of a and b is a divisor of c.

300	$= 2^2 \times 3^1 \times 5^2$
18	$= 2^1 \times 3^2$
gcd(18,300)	$= 2^1 \times 3^1 \times 5^0 = 6$

The following relationship always holds:

If k = gcd(a,b) then $k_p = \min(a_p, b_p)$ for all p

Determining the prime factors of a large number is no easy task, so the preceding relationship does not directly lead to a practical method of calculating the greatest common divisor.

Principles of Public-Key Cryptosystems

The concept of public-key cryptography evolved from an attempt to attack two of the most difficult problems associated with symmetric encryption. The first problem is that of key distribution.

As we have seen, key distribution under symmetric encryption requires either (1) that two communicants already share a key, which somehow has been distributed to them; or (2) the use of a key distribution center. Whitfield Diffie, one of the discoverers of public-key encryption (along with Martin Hellman, both at Stanford University at the time), reasoned that this second requirement negated the very essence of cryptography: the ability to maintain total secrecy over your own communication. As Diffie put it , "what good would it do after all to develop impenetrable cryptosystems, if their users were forced to share their keys with a KDC that could be compromised by either burglary or subpoena?"

The second problem that Diffie pondered, and one that was apparently unrelated to the first was that of "digital signatures." If the use of cryptography was to become widespread, not just in military situations but for commercial and private purposes, then electronic messages and documents would need the equivalent of signatures used in paper documents. That is, could a method be devised that would stipulate, to the satisfaction of all parties, that a digital message had been sent by a particular person?

Diffie and Hellman achieved an astounding breakthrough in by coming up with a method that addressed both problems and that was radically different from all previous approaches to cryptography, going back over four millennia.

Diffie and Hellman first publicly introduced the concepts of public-key cryptography in 1976. However, this is not the true beginning. Admiral Bobby Inman, while director of the National Security Agency (NSA),

claimed that public-key cryptography had been discovered at NSA in the mid-1960s. The first documented introduction of these concepts came in 1970, from the Communications-Electronics Security Group, Britain's counterpart to NSA, in a classified report by James Ellis .

In the next subsection, we look at the overall framework for public-key cryptography. Then we examine the requirements for the encryption/decryption algorithm that is at the heart of the scheme.

Public-Key Cryptosystems

Asymmetric algorithms rely on one key for encryption and a different but related key for decryption. These algorithms have the following important characteristic:

- It is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key.

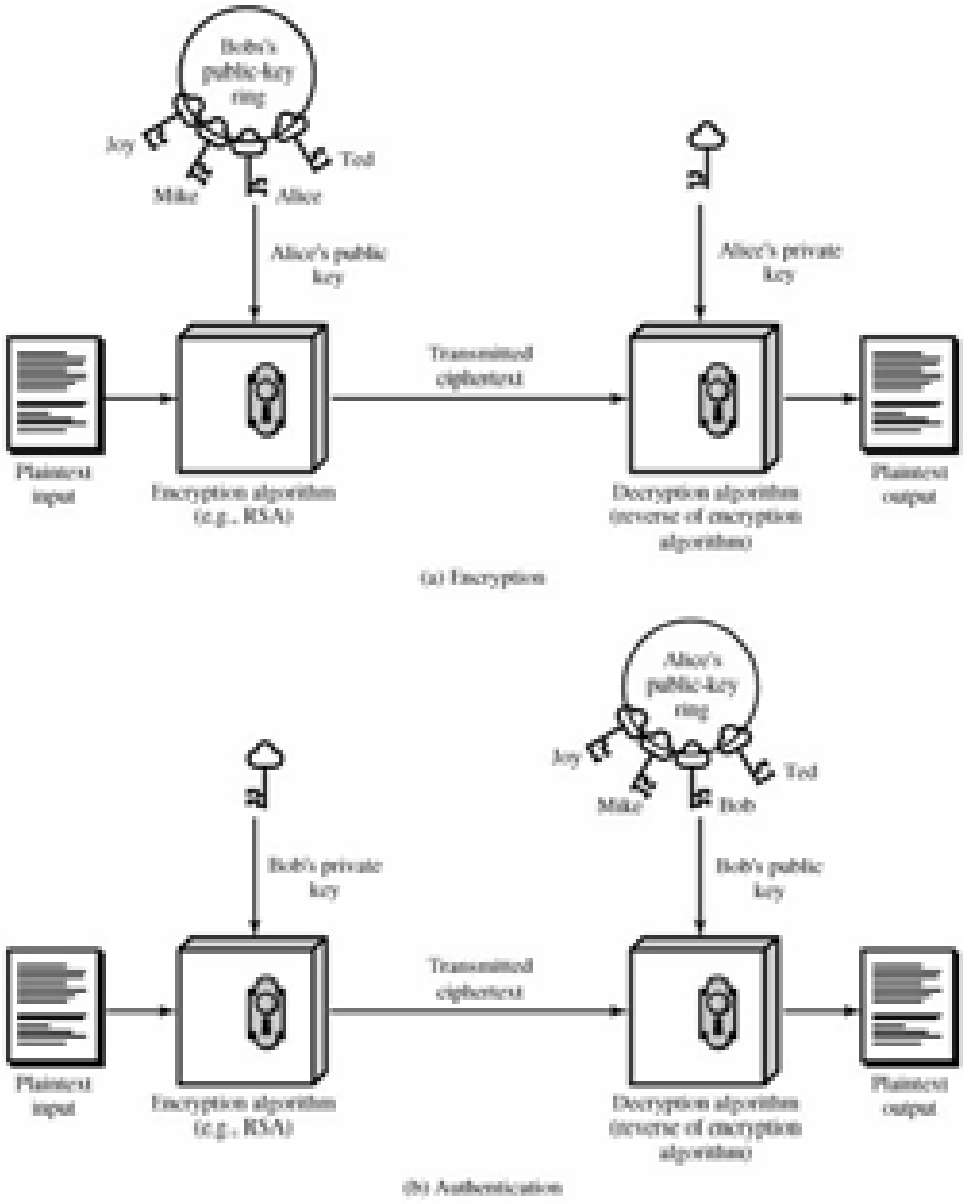
In addition, some algorithms, such as RSA, also exhibit the following characteristic:

- Either of the two related keys can be used for encryption, with the other used for decryption.

A public-key encryption scheme has six ingredients (Figure 9.1a; compare with Figure 2.1):

- Plaintext: This is the readable message or data that is fed into the algorithm as input.
- Encryption algorithm: The encryption algorithm performs various transformations on the plaintext.
- Public and private keys: This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.
- Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
- Decryption algorithm: This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

Figure 9.1. Public-Key Cryptography



The essential steps are the following:

1. Each user generates a pair of keys to be used for the encryption and decryption of

messages.

2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure 9.1a suggests, each user maintains a collection of public keys obtained from others.

3. If Bob wishes to send a confidential message to Alice, Bob encrypts the message using Alice's public key.

4. When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows Alice's private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed. As long as a user's private key remains protected and secret, incoming communication is secure. At any time, a system can change its private key and publish the companion public key to replace its old public key.

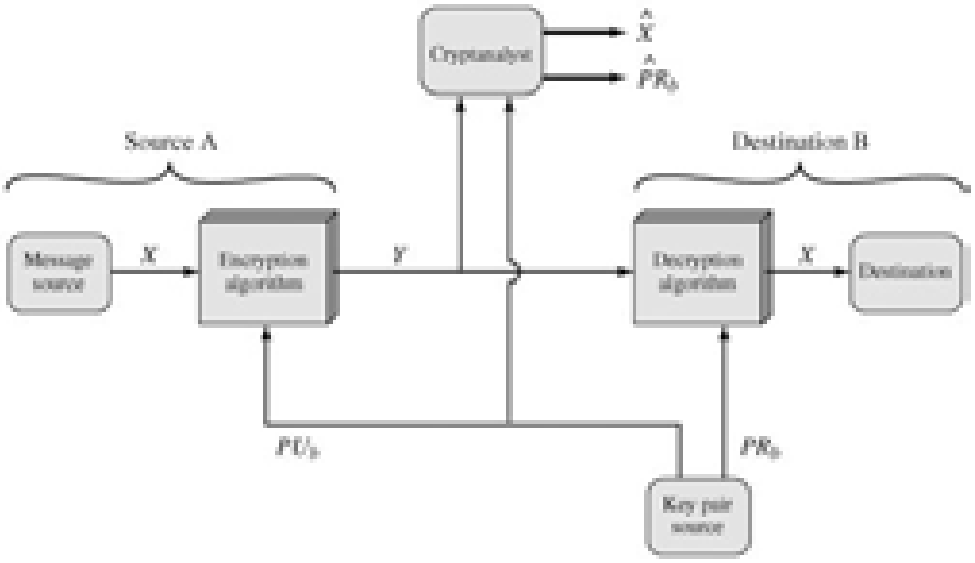
Table 9.1 summarizes some of the important aspects of symmetric and public-key encryption. To discriminate between the two, we refer to the key used in symmetric encryption as a secret key. The two keys used for asymmetric encryption are referred to as the public key and the private key. Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with symmetric encryption.

The following notation is used consistently throughout. A secret key is represented by K_m, where m is some modifier; for example, K_a is a secret key owned by user A. A public key is represented by PU_a, for user A, and the corresponding private key is PR_a, Encryption of plaintext X can be performed with a secret key, a public key, or a private key, denoted by $E(K_a, X)$, $E(PU_a, X)$, and $E(PR_a, X)$, respectively. Similarly, decryption of ciphertext C can be performed with a secret key, a public key, or a private key, denoted by $D(K_a, X)$, $D(PU_a, X)$, and $D(PR_a, X)$, respectively.

Table 9.1. Conventional and Public-Key Encryption	
Conventional Encryption	Public-Key Encryption
Needed to Work:	Needed to Work:
1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key.	1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one).
Needed for Security:	Needed for Security:
1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Let us take a closer look at the essential elements of a public-key encryption scheme, using Figure 9.2 (compare with Figure 2.2). There is some source A that produces a message in plaintext, $X = [X_1, X_2, ..., X_M,]$. The M elements of X are letters in some finite alphabet. The message is intended for destination B. B generates a related pair of keys: a public key, PU_b, and a private key, PU_b. PU_b is known only to B, whereas PU_b is publicly available and therefore accessible by A.

Figure 9.2. Public-Key Cryptosystem: Secrecy



With the message X and the encryption key PU_b as input, A forms the ciphertext $Y = [Y_1, Y_2,..., Y_N]$:

$Y = E(PU_b, X)$

The intended receiver, in possession of the matching private key, is able to invert the transformation:

$X = D(PR_b, Y)$

An adversary, observing Y and having access to PU_b but not having access to PR_b or X, must attempt to recover X and/or PR_b. It is assumed that the adversary does have knowledge of the encryption (E) and decryption (D) algorithms. If the adversary is interested only in this particular message, then the focus of effort is to recover X, by generating a plaintext estimate $\hat{X}$ Often, however, the adversary is interested in being able to read future

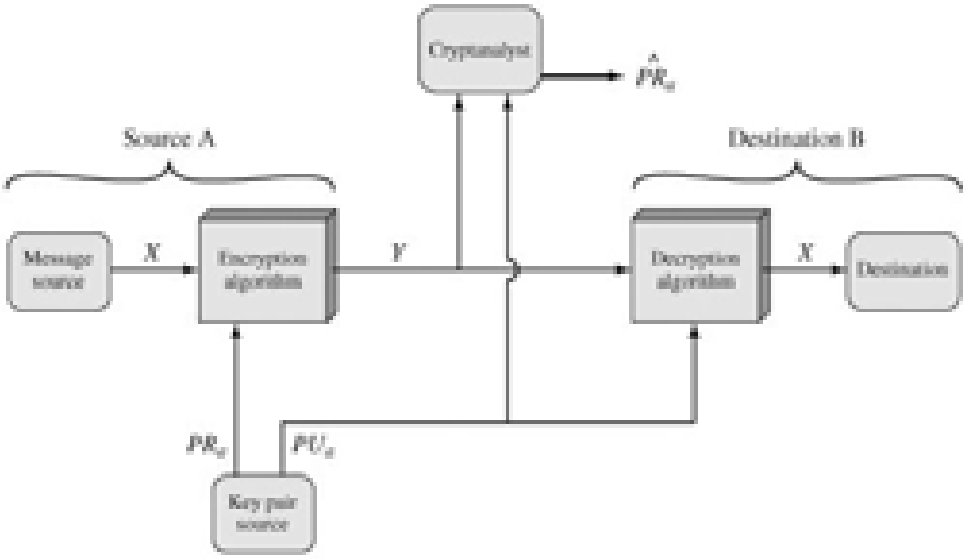
messages as well, in which case an attempt is made to recover PR_b by generating an estimate $\tilde{PR}_b$.

We mentioned earlier that either of the two related keys can be used for encryption, with the other being used for decryption. This enables a rather different cryptographic scheme to be implemented. Whereas the scheme illustrated in Figure 9.2 provides confidentiality, Figures 9.1b and 9.3 show the use of public-key encryption to provide authentication:

$Y = E(PR_a, X)$

$Y = E(PU_a, Y)$

Figure 9.3. Public-Key Cryptosystem: Authentication



In this case, A prepares a message to B and encrypts it using A's private key before transmitting it. B can decrypt the message using A's public key. Because the message was encrypted using A's private key, only A could have prepared the message. Therefore, the entire encrypted message serves as a *digital signature*. In addition, it is impossible to alter the message without access to A's private key, so the message is authenticated both in terms of source and in terms of data integrity.

In the preceding scheme, the entire message is encrypted, which, although validating both author and contents, requires a great deal of storage. Each

document must be kept in plaintext to be used for practical purposes. A copy also must be stored in ciphertext so that the origin and contents can be verified in case of a dispute. A more efficient way of achieving the same results is to encrypt a small block of bits that is a function of the document. Such a block, called an authenticator, must have the property that it is infeasible to change the document without changing the authenticator. If the authenticator is encrypted with the sender's private key, it serves as a signature that verifies origin, content, and sequencing. Chapter 13 examines this technique in detail.

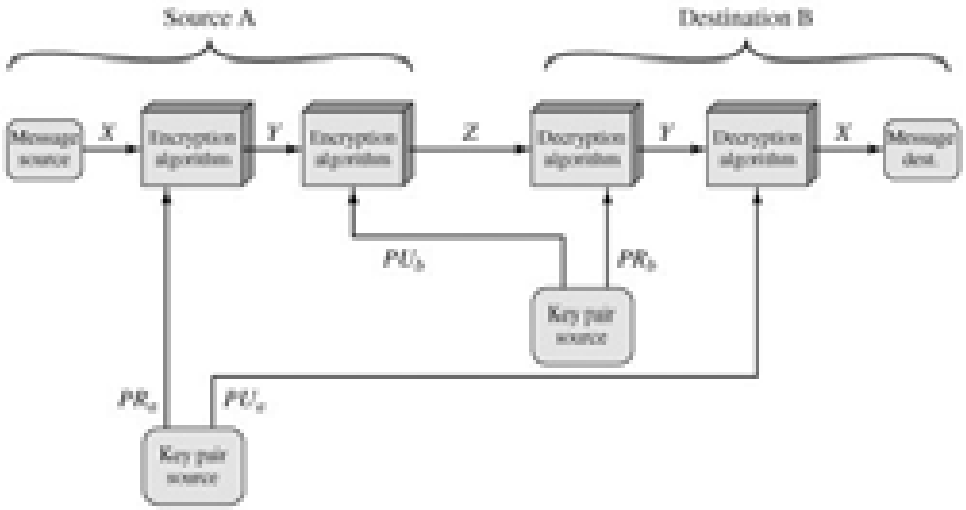
It is important to emphasize that the encryption process depicted in Figures 9.1b and 9.3 does not provide confidentiality. That is, the message being sent is safe from alteration but not from eavesdropping. This is obvious in the case of a signature based on a portion of the message, because the rest of the message is transmitted in the clear. Even in the case of complete encryption, as shown in Figure 9.3, there is no protection of confidentiality because any observer can decrypt the message by using the sender's public key.

It is, however, possible to provide both the authentication function and confidentiality by a double use of the public-key scheme (Figure 9.4):

$Z = E(PU_b, E(PR_a, X))$

$X = D(PU_a, E(PR_b, Z))$

Figure 9.4. Public-Key Cryptosystem: Authentication and Secrecy



In this case, we begin as before by encrypting a message, using the sender's private key. This provides the digital signature. Next, we encrypt again, using the receiver's public key. The final ciphertext can be decrypted only by the intended receiver, who alone has the matching private key. Thus, confidentiality is provided. The disadvantage of this approach is that the public-key algorithm, which is complex, must be exercised four times rather than two in each communication.

Applications for Public-Key Cryptosystems

Before proceeding, we need to clarify one aspect of public-key cryptosystems that is otherwise likely to lead to confusion. Public-key systems are characterized by the use of a cryptographic algorithm with two keys, one held private and one available publicly. Depending on the application, the sender uses either the sender's private key or the receiver's public key, or both, to perform some type of cryptographic function. In broad terms, we can classify the use of public-key cryptosystems into three categories:

- Encryption/decryption: The sender encrypts a message with the recipient's public key.
- Digital signature: The sender "signs" a message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.
- Key exchange: Two sides cooperate to exchange a session key. Several different approaches are possible, involving the private key(s) of one or both parties.

Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. Table 9.2 indicates the applications supported by the algorithms discussed in this book.

Table 9.2. Applications for Public-Key Cryptosystems

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes

Table 9.2. Applications for Public-Key Cryptosystems			
Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Requirements for Public-Key Cryptography

The cryptosystem illustrated in Figures 9.2 through 9.4 depends on a cryptographic algorithm based on two related keys. Diffie and Hellman postulated this system without demonstrating that such algorithms exist. However, they did lay out the conditions that such algorithms must fulfill :

1. It is computationally easy for a party B to generate a pair (public key PU_b, private key PR_b).
2. It is computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext:

   $C = E(PU_b, M)$

3. It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message:

   $M = D(PR_b, C) = D[PR_b, E(PU_b, M)]$

4. It is computationally infeasible for an adversary, knowing the public key, PU_b, to determine the private key, PR_b.
5. It is computationally infeasible for an adversary, knowing the public key, PU_b, and a ciphertext, C, to recover the original message, M.

   We can add a sixth requirement that, although useful, is not necessary for all public-key applications:

6. The two keys can be applied in either order:

$$M = D[PU_b, E(PR_b, M)] = D[PR_b, E(PU_b, M)]$$

These are formidable requirements, as evidenced by the fact that only a few algorithms (RSA, elliptic curve cryptography, Diffie-Hellman, DSS) have received widespread acceptance in the several decades since the concept of public-key cryptography was proposed.

Before elaborating on why the requirements are so formidable, let us first recast them. The requirements boil down to the need for a trap-door one-way function. A *one-way function*[3] is one that maps a domain into a range such that every function value has a unique inverse, with the condition that the calculation of the function is easy whereas the calculation of the inverse is infeasible:

[3] Not to be confused with a one-way hash function, which takes an arbitrarily large data field as its argument and maps it to a fixed output. Such functions are used for authentication .

$Y = f(X)$ easy

$X = f^1(X)$ infeasible

Generally, easy is defined to mean a problem that can be solved in polynomial time as a function of input length. Thus, if the length of the input is n bits, then the time to compute the function is proportional to n^a where a is a fixed constant. Such algorithms are said to belong to the class P. The term infeasible is a much fuzzier concept. In general, we can say a problem is infeasible if the effort to solve it grows faster than polynomial time as a function of input size. For example, if the length of the input is n bits and the time to compute the function is proportional to 2^n, the problem is considered infeasible. Unfortunately, it is difficult to determine if a particular algorithm exhibits this complexity. Furthermore, traditional notions of computational complexity focus on the worst-case or average-case complexity of an algorithm. These measures are inadequate for cryptography, which requires that it be infeasible to invert a function for virtually all inputs, not for the worst case or even average case.

We now turn to the definition of a trap-door one-way function, which is easy to calculate in one direction and infeasible to calculate in the other direction unless certain additional information is known. With the additional information the inverse can be calculated in polynomial time. We can

summarize as follows: A trap-door one-way function is a family of invertible functions f_k, such that

$Y = f_k(X)$	easy, if k and X are known
$X = f_k^{-1}(Y)$	easy, if k and Y are known
$X = f_k^{-1}(Y)$	infeasible, if Y is known but k is not known

Thus, the development of a practical public-key scheme depends on discovery of a suitable trap-door one-way function.

Public-Key Cryptanalysis

As with symmetric encryption, a public-key encryption scheme is vulnerable to a brute-force attack. The countermeasure is the same: Use large keys. However, there is a tradeoff to be considered. Public-key systems depend on the use of some sort of invertible mathematical function. The complexity of calculating these functions may not scale linearly with the number of bits in the key but grow more rapidly than that. Thus, the key size must be large enough to make brute-force attack impractical but small enough for practical encryption and decryption. In practice, the key sizes that have been proposed do make brute-force attack impractical but result in encryption/decryption speeds that are too slow for general-purpose use. Instead, as was mentioned earlier, public-key encryption is currently confined to key management and signature applications.

Another form of attack is to find some way to compute the private key given the public key. To date, it has not been mathematically proven that this form of attack is infeasible for a particular public-key algorithm. Thus, any given algorithm, including the widely used RSA algorithm, is suspect. The history of cryptanalysis shows that a problem that seems insoluble from one perspective can be found to have a solution if looked at in an entirely different way.

Finally, there is a form of attack that is peculiar to public-key systems. This is, in essence, a probable-message attack. Suppose, for example, that a message were to be sent that consisted solely of a 56-bit DES key. An adversary could encrypt all possible 56-bit DES keys using the public key and could discover the encrypted key by matching the transmitted ciphertext. Thus, no matter how large the key size of the public-key scheme, the attack

is reduced to a brute-force attack on a 56-bit key. This attack can be thwarted by appending some random bits to such simple messages.

Proof of the RSA Algorithm

The basic elements of the RSA algorithm can be summarized as follows. Given two prime numbers p and q, with n = pq and a message block M < n, two integers e and d are chosen such that

$M^{ed} \bmod n = M$

that the preceding relationship holds if e and d are multiplicative inverses modulo ϕ(n), where ϕ(n) is the Euler totient function. that for p, q prime, ϕ(n) = (p 1)(q 1). The relationship between e and d can be expressed as

ed mod ϕ(n) = 1

Another way to state this is that there is an integer k such that ed = k ϕ(n) + 1. Thus, we must show that

Equation 9-3

$$M^{k\phi(n)+1} \bmod n = M^{k(p-1)(q-1)+1} \bmod n = M$$

Basic Results

Before proving Equation (9.3), we summarize some basic results. a property of modular arithmetic is the following:

[(a mod n) x (b mod n)] mod n = (a x b) mod n

From this, it should be easy to see that if we have x mod n = 1 then x^2 mod n 1 and for any integer y, we have x^y mod n = 1. Similarly, if we have x mod n = 0, for any integer y, we have x^y mod n = 0.

Another property of modular arithmetic is

[(a mod n) (b mod n)] mod n = (a b) mod n

The other result we need is Euler's theorem. If integers a and n are relatively prime, than a ϕ (n) mod n = 1.

Proof

First we show that $M^{k(p1)(q1)+1} \bmod p = M \bmod p$. There are two cases to consider.

Case 1: M and p are not relatively prime ; that is, p divides M. In this case $M \bmod p = 0$ and therefore $M^{k(p1)(q1)+1} \bmod p = 0$. Thus, $M^{k(p1)(q1)+1} \bmod p = M \bmod p$.

Case 2: If M and p are relatively prime, by Euler's theorem, $M^{(p)} \bmod p = 1$. We proceed as follows:

$$
\begin{aligned}
M^{k(p1)(q1)+1} \bmod p &= [(M)M^{k(p1)(q1)}] \bmod p \\
&= [(M)(M^{p1})^{k(q1)} \bmod p \\
&= [(M)(M^{(p)})^{k(q1)} \bmod p \\
&= (M \bmod p) \times [(M^{(p)}) \bmod p]^{k(q1)} \\
&= (M \bmod p) \times (1)^{k(q1)} \qquad \text{(by Euler's theorem)} \\
&= M \bmod P
\end{aligned}
$$

We now observe that

$$[M^{k(p1)(q1)+1} M] \bmod p \; [M^{k(p1)(q1)+1} \bmod p] \; [M \bmod p] = 0$$

Thus, p divides $[M^{k(p1)(q1)+1} M]$. By the same reasoning, we can show that q divides $[M^{k(p1)(q1)+1} M]$. Because p and q are distinct primes, there must exist an integer r that satisfies

$$[M^{k(p1)(q1)+1} M] = (pq)r = nr$$

Therefore, p divides $[M^{k(p1)(q1)+1} M]$, and so $M^{k(n)+1} \bmod n = M^{k(p1)(q1)+1} \bmod n = M$.

Random Number Generation

Random numbers play an important role in the use of encryption for various network security applications. In this section, we provide a brief overview of the use of random numbers in network security and then look at some approaches to generating random numbers.

The Use of Random Numbers

A number of network security algorithms based on cryptography make use of random numbers. For example,

- Reciprocal authentication schemes, such as illustrated in Figures 7.9 and 7.11. In both of these key distribution scenarios, nonces are used for handshaking to prevent replay attacks. The use of random numbers for the nonces frustrates opponents' efforts to determine or guess the nonce.
- Session key generation, whether done by a key distribution center or by one of the principals.
- Generation of keys for the RSA public-key encryption algorithm .

These applications give rise to two distinct and not necessarily compatible requirements for a sequence of random numbers: randomness and unpredictability.

Randomness

Traditionally, the concern in the generation of a sequence of allegedly random numbers has been that the sequence of numbers be random in some well-defined statistical sense. The following two criteria are used to validate that a sequence of numbers is random:

- Uniform distribution: The distribution of numbers in the sequence should be uniform; that is, the frequency of occurrence of each of the numbers should be approximately the same.
- Independence: No one value in the sequence can be inferred from the others.

Although there are well-defined tests for determining that a sequence of numbers matches a particular distribution, such as the uniform distribution,

there is no such test to "prove" independence. Rather, a number of tests can be applied to demonstrate if a sequence does not exhibit independence. The general strategy is to apply a number of such tests until the confidence that independence exists is sufficiently strong.

In the context of our discussion, the use of a sequence of numbers that appear statistically random often occurs in the design of algorithms related to cryptography.

In general, it is difficult to determine if a given large number N is prime. A brute-force approach would be to divide N by every odd integer less than $\sqrt{N}$. If N is on the order, say, of 10^{150}, a not uncommon occurrence in public-key cryptography, such a brute-force approach is beyond the reach of human analysts and their computers. However, a number of effective algorithms exist that test the primality of a number by using a sequence of randomly chosen integers as input to relatively simple computations. If the sequence is sufficiently long (but far, far less than $\sqrt{10^{150}}$), the primality of a number can be determined with near certainty. This type of approach, known as randomization, crops up frequently in the design of algorithms. In essence, if a problem is too hard or time-consuming to solve exactly, a simpler, shorter approach based on randomization is used to provide an answer with any desired level of confidence.

Unpredictability

In applications such as reciprocal authentication and session key generation, the requirement is not so much that the sequence of numbers be statistically random but that the successive members of the sequence are unpredictable. With "true" random sequences, each number is statistically independent of other numbers in the sequence and therefore unpredictable. However, as is discussed shortly, true random numbers are seldom used; rather, sequences of numbers that appear to be random are generated by some algorithm. In this latter case, care must be taken that an opponent not be able to predict future elements of the sequence on the basis of earlier elements.

Pseudorandom Number Generators (PRNGs)

Cryptographic applications typically make use of algorithmic techniques for random number generation. These algorithms are deterministic and therefore produce sequences of numbers that are not statistically random. However, if the algorithm is good, the resulting sequences will pass many reasonable tests of randomness. Such numbers are referred to as pseudorandom numbers.

You may be somewhat uneasy about the concept of using numbers generated by a deterministic algorithm as if they were random numbers. Despite what might be called philosophical objections to such a practice, it generally works. As one expert on probability theory puts it:

For practical purposes we are forced to accept the awkward concept of "relatively random" meaning that with regard to the proposed use we can see no reason why they will not perform as if they were random (as the theory usually requires). This is highly subjective and is not very palatable to purists, but it is what statisticians regularly appeal to when they take "a random sample"they hope that any results they use will have approximately the same properties as a complete counting of the whole sample space that occurs in their theory.

Linear Congruential Generators

By far, the most widely used technique for pseudorandom number generation is an algorithm first proposed by Lehmer , which is known as the linear congruential method. The algorithm is parameterized with four numbers, as follows:

m	the modulus	$m > 0$
a	the multiplier	$0 < a < m$
c	the increment	$0 \leq c < m$
X_0	the starting value, or seed	$0 \leq X_0 < m$

The sequence of random numbers $\{X_n\}$ is obtained via the following iterative equation:

$$X_{n+1} = (aX_n + c) \bmod m$$

If m, a, c, and X_0 are integers, then this technique will produce a sequence of integers with each integer in the range $0 \leq X_n < m$.

The selection of values for a, c, and m is critical in developing a good random number generator. For example, consider a, = c = 1. The sequence produced is obviously not satisfactory. Now consider the values a = 7, c = 0, m = 32, and X_0 = 1. This generates the sequence {7, 17, 23, 1, 7, etc.}, which is also clearly unsatisfactory. Of the 32 possible values, only 4 are used; thus, the sequence is said to have a period of 4. If, instead, we change the value of a to 5, then the sequence is {5, 25, 29, 17, 21, 9, 13, 1, 5, etc.}, which increases the period to 8.

We would like m to be very large, so that there is the potential for producing a long series of distinct random numbers. A common criterion is that m be nearly equal to the maximum representable nonnegative integer for a given computer. Thus, a value of m near to or equal to 2^{31} is typically chosen.

proposes three tests to be used in evaluating a random number generator:

T_1: The function should be a full-period generating function. That is, the function should generate all the numbers between 0 and m before repeating.

T_2: The generated sequence should appear random. Because it is generated deterministically, the sequence is not random. There is a variety of statistical tests that can be used to assess the degree to which a sequence exhibits randomness.

T_3: The function should implement efficiently with 32-bit arithmetic.

With appropriate values of a, c, and m, these three tests can be passed. With respect to T_1 it can be shown that if m is prime and c = 0, then for certain values of a, the period of the generating function is m 1, with only the value 0 missing. For 32-bit arithmetic, a convenient prime value of m is 2^{31} 1. Thus, the generating function becomes

$X_{n+1} = (aX_n) \bmod (2^{31} 1)$

Of the more than 2 billion possible choices for a, only a handful of multipliers pass all three tests. One such value is $a = 7^5 = 16807$, which was originally designed for use in the IBM 360 family of computers.

This generator is widely used and has been subjected to a more thorough testing than any other PRNG. It is frequently recommended for statistical and simulation work.

The strength of the linear congruential algorithm is that if the multiplier and modulus are properly chosen, the resulting sequence of numbers will be statistically indistinguishable from a sequence drawn at random (but without replacement) from the set 1, 2,...,m 1. But there is nothing random at all about the algorithm, apart from the choice of the initial value X_0. Once that value is chosen, the remaining numbers in the sequence follow deterministically. This has implications for cryptanalysis.

If an opponent knows that the linear congruential algorithm is being used and if the parameters are known (e.g., a =7^5, c = 0, m = 2^{31} 1), then once a single number is discovered, all subsequent numbers are known. Even if the opponent knows only that a linear congruential algorithm is being used, knowledge of a small part of the sequence is sufficient to determine the parameters of the algorithm. Suppose that the opponent is able to determine values for X_0, X_1, X_2 and X_3 Then

$X_1 = (aX_0 + c) \bmod m$

$X_2 = (aX_1 + c) \bmod m$

$X_3 = (aX_2 + c) \bmod m$

These equations can be solved for a, c, and m.

Thus, although it is nice to be able to use a good PRNG, it is desirable to make the actual sequence used nonreproducible, so that knowledge of part of the sequence on the part of an opponent is insufficient to determine future elements of the sequence. This goal can be achieved in a number of ways. For example, suggests using an internal system clock to modify the random number stream. One way to use the clock would be to restart the sequence after every N numbers using the current clock value (mod m) as the new seed. Another way would be simply to add the current clock value to each random number (mod m).

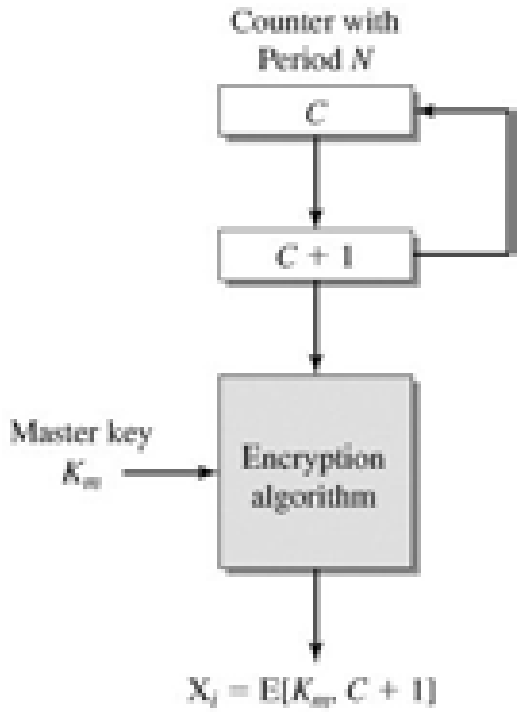
Cryptographically Generated Random Numbers

For cryptographic applications, it makes some sense to take advantage of the encryption logic available to produce random numbers. A number of means have been used, and in this subsection we look at three representative examples.

Cyclic Encryption

Figure 7.13 illustrates an approach. In this case, the procedure is used to generate session keys from a master key. A counter with period N provides input to the encryption logic. For example, if 56-bit DES keys are to be produced, then a counter with period 2^{56} can be used. After each key is produced, the counter is incremented by one. Thus, the pseudorandom numbers produced by this scheme cycle through a full period: Each of the outputs X_0, X_1, X_{N1} is based on a different counter value and therefore $X_0 \neq X_1 \neq ... \neq X_{N1}$. Because the master key is protected, it is not computationally feasible to deduce any of the session keys (random numbers) through knowledge of one or earlier session keys.

Figure 7.13. Pseudorandom Number Generation from a Counter



To strengthen the algorithm further, the input could be the output of a full-period PRNG rather than a simple counter.

DES Output Feedback Mode

The output feedback (OFB) mode of DES, illustrated in Figure 6.6, can be used for key generation as well as for stream encryption. Notice that the output of each stage of operation is a 64-bit value, of which the s leftmost bits are fed back for encryption. Successive 64-bit outputs constitute a sequence of pseudorandom numbers with good statistical properties. Again, as with the approach suggested in the preceding subsection, the use of a protected master key protects the generated session keys.

ANSI X9.17 PRNG

One of the strongest (cryptographically speaking) PRNGs is specified in ANSI X9.17. A number of applications employ this technique, including financial security applications and PGP .

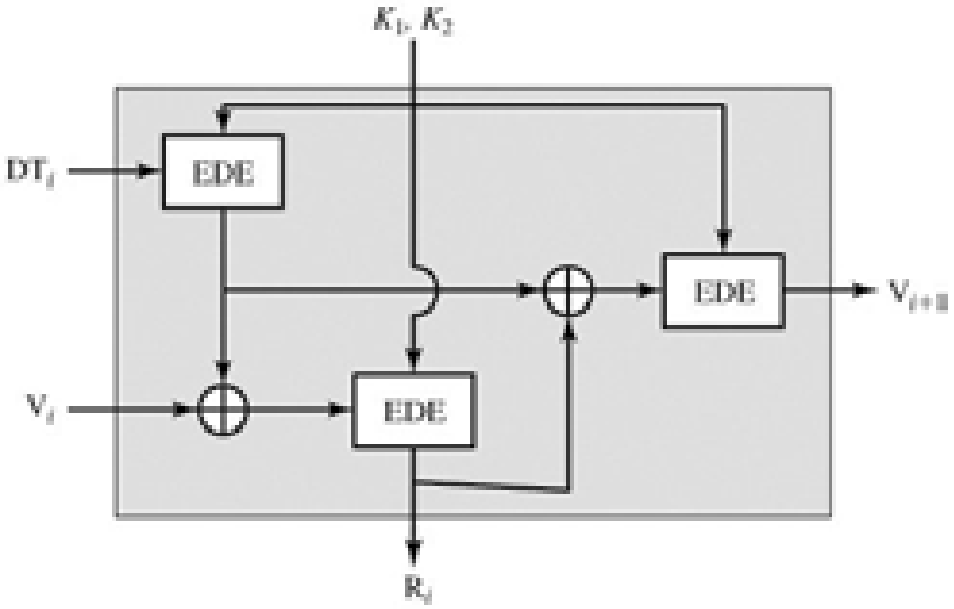
Figure 7.14 illustrates the algorithm, which makes use of triple DES for encryption. The ingredients are as follows:

- Input: Two pseudorandom inputs drive the generator. One is a 64-bit representation of the current date and time, which is updated on each number generation. The other is a 64-bit seed value; this is initialized to some arbitrary value and is updated during the generation process.
- Keys: The generator makes use of three triple DES encryption modules. All three make use of the same pair of 56-bit keys, which must be kept secret and are used only for pseudorandom number generation.
- Output: The output consists of a 64-bit pseudorandom number and a 64-bit seed value.

Define the following quantities:

DT_i	Date/time value at the beginning of ith generation stage
V_i	Seed value at the beginning of ith generation stage
R_i	Pseudorandom number produced by the ith generation stage
K_1, K_2	DES keys used for each stage

Figure 7.14. ANSI X9.17 Pseudorandom Number Generator



Then

$R_i = EDE([K_1, K_2], [V_i \oplus EDE([K_1, K_2], DT_i)])$

$V_{i+1} = EDE([K_1, K_2], [R_i \oplus EDE([K_1, K_2], DT_i)])$

Where EDE ([K_1, K2], X) refers to the sequence encrypt-decrypt-encrypt using two-key triple DES to encrypt X.

Several factors contribute to the cryptographic strength of this method. The technique involves a 112-bit key and three EDE encryptions for a total of

nine DES encryptions. The scheme is driven by two pseudorandom inputs, the date and time value, and a seed produced by the generator that is distinct from the pseudorandom number produced by the generator. Thus, the amount of material that must be compromised by an opponent is overwhelming. Even if a pseudorandom number R_i were compromised, it would be impossible to deduce the V_{i+1} from the R_i because an additional EDE operation is used to produce the V_{i+1}.

Blum Blum Shub Generator

A popular approach to generating secure pseudorandom number is known as the Blum, Blum, Shub (BBS) generator, named for its developers . It has perhaps the strongest public proof of its cryptographic strength. The procedure is as follows. First, choose two large prime numbers, p and q, that both have a remainder of 3 when divided by 4. That is,

$p \equiv q \equiv 3 \pmod 4$

This notation, simply means that (p mod 4) = (q mod 4) = 3. For example, the prime numbers 7 and 11 satisfy $7 \equiv 11 \equiv 3 \pmod 4$. Let n = p x q. Next, choose a random number s, such that s is relatively prime to n; this is equivalent to saying that neither p nor q is a factor of s. Then the BBS generator produces a sequence of bits Bi according to the following algorithm:

$X_0 = s^2 \bmod n$
for i = 1 to ∞
  $X_i = (X_{i1})^2 \bmod n$
  $B_i = X_i \bmod 2$

Thus, the least significant bit is taken at each iteration. Table 7.2, shows an example of BBS operation. Here, n = 192649 = 383 x 503 and the seed s = 101355.

Table 7.2. Example Operation of BBS Generator

(This item is displayed on page 226 in the print version)

i	X_i	B_i
0	20749	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0
11	137922	0
12	123175	1
13	8630	0
14	114386	0
15	14863	1
16	133015	1
17	106065	1
18	45870	0
19	137171	1
20	48060	0

The BBS is referred to as a cryptographically secure pseudorandom bit generator (CSPRBG). A CSPRBG is defined as one that passes the next-bit test, which, in turn, is defined as follows: A pseudorandom bit generator is said to pass the next-bit test if there is not a polynomial-time algorithm that,

on input of the first k bits of an output sequence, can predict the $(k + 1)^{st}$ bit with probability significantly greater than 1/2. In other words, given the first k bits of the sequence, there is not a practical algorithm that can even allow you to state that the next bit will be 1 (or 0) with probability greater than 1/2. For all practical purposes, the sequence is unpredictable. The security of BBS is based on the difficulty of factoring n. That is, given n, we need to determine its two prime factors p and q.

A polynomial-time algorithm of order k is one whose running time is bounded by a polynomial of order k.

True Random Number Generators

A true random number generator (TRNG) uses a nondeterministic source to produce randomness. Most operate by measuring unpredictable natural processes, such as pulse detectors of ionizing radiation events, gas discharge tubes, and leaky capacitors. Intel has developed a commercially available chip that samples thermal noise by amplifying the voltage measured across undriven resistors. A group at Bell Labs has developed a technique that uses the variations in the response time of raw read requests for one disk sector of a hard disk. LavaRnd is an open source project for creating truly random numbers using inexpensive cameras, open source code, and inexpensive hardware. The system uses a saturated CCD in a light-tight can as a chaotic source to produce the seed. Software processes the result into truly random numbers in a variety of formats.

There are problems both with the randomness and the precision of such numbers . to say nothing of the clumsy requirement of attaching one of these devices to every system in an internetwork. Another alternative is to dip into a published collection of good-quality random numbers. However, these collections provide a very limited source of numbers compared to the potential requirements of a sizable network security application. Furthermore, although the numbers in these books do indeed exhibit statistical randomness, they are predictable, because an opponent who knows that the book is in use can obtain a copy.

Skew

A true random number generator may produce an output that is biased in some way, such as having more ones than zeros or vice versa. Various methods of modifying a bit stream to reduce or eliminate the bias have been developed. These are referred to as deskewing algorithms. One approach to deskew is to pass the bit stream through a hash function such as MD5 or SHA-1 . The hash function produces an n-bit output from an input of arbitrary length. For deskewing, blocks of m input bits, with m ≥n can be passed through the hash function.

Simplified AES

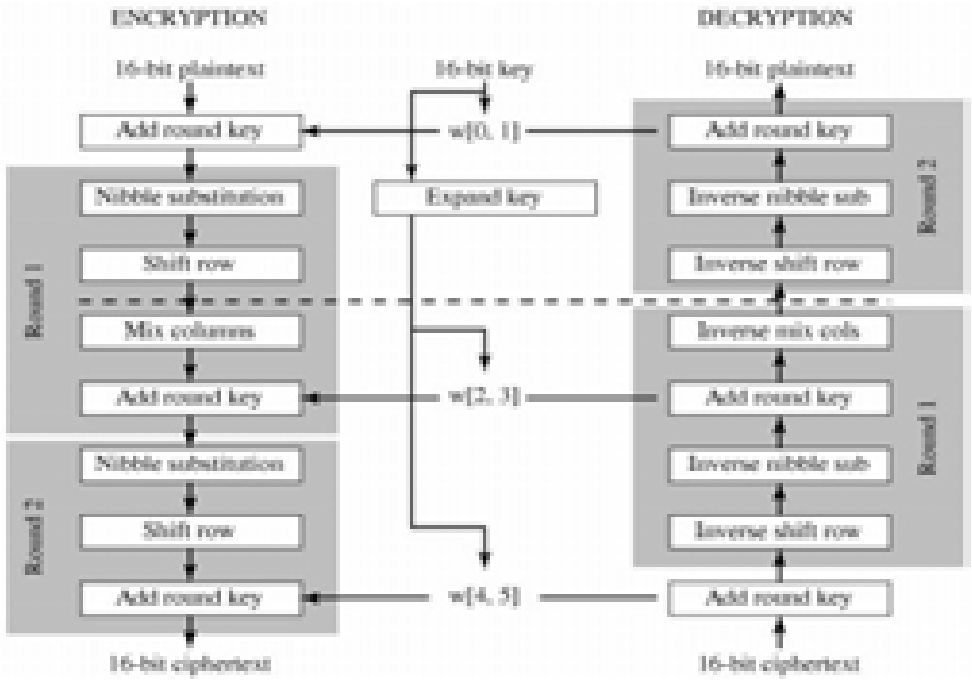
Simplified AES (S-AES) was developed by Professor Edward Schaefer of Santa Clara University and several of his students . It is an educational rather than a secure encryption algorithm. It has similar properties and structure to AES with much smaller parameters. The reader might find it useful to work through an example by hand while following the discussion in this appendix. A good grasp of S-AES will make it easier for the student to appreciate the structure and workings of AES.

Overview

Figure 5.8 illustrates the overall structure of S-AES. The encryption algorithm takes a 16-bit block of plaintext as input and a 16-bit key and produces a 16-bit block of ciphertext as output. The S-AES decryption algorithm takes an 16-bit block of ciphertext and the same 16-bit key used to produce that ciphertext as input and produces the original 16-bit block of plaintext as output.

Figure 5.8. S-AES Encryption and Decryption

(This item is displayed on page 165 in the print version)



The encryption algorithm involves the use of four different functions, or transformations: add key (A_K) nibble substitution (NS), shift row (SR), and mix column (MC), whose operation is explained subsequently.

We can concisely express the encryption algorithm as a composition of functions:

Definition: If f and g are two functions, then the function F with the equation y= F(x) = g[f(x)] is called the composition of f and g and is denoted as F = g º f.

A_{K2} º SR º NS º A_{K1} º MC º SR º NS º A_{K0}

so that A_{K0} is applied first.

The encryption algorithm is organized into three rounds. Round 0 is simply an add key round; round 1 is a full round of four functions; and round 2 contains only 3 functions. Each round includes the add key function, which makes use of 16 bits of key. The initial 16-bit key is expanded to 48 bits, so that each round uses a distinct 16-bit round key.

Each function operates on a 16-bit state, treated as a 2 x 2 matrix of nibbles, where one nibble equals 4 bits. The initial value of the state matrix is the 16-bit plaintext; the state matrix is modified by each subsequent function in the encryption process, producing after the last function the 16-bit ciphertext. As Figure 5.9a shows, the ordering of nibbles within the matrix is by column. So, for example, the first eight bits of a 16-bit plaintext input to the encryption cipher occupy the first column of the matrix, and the second eight bits occupy the second column. The 16-bit key is similarly organized, but it is somewhat more convenient to view the key as two bytes rather than four nibbles (Figure 5.9b). The expanded key of 48 bits is treated as three round keys, whose bits are labeled as follows: $K_0 = k_0...k_{15}$; $K_1 = k_{16}...k_{31}$; $K_2 = k_{32}...k_{47}$.

Figure 5.9. S-AES Data Structures

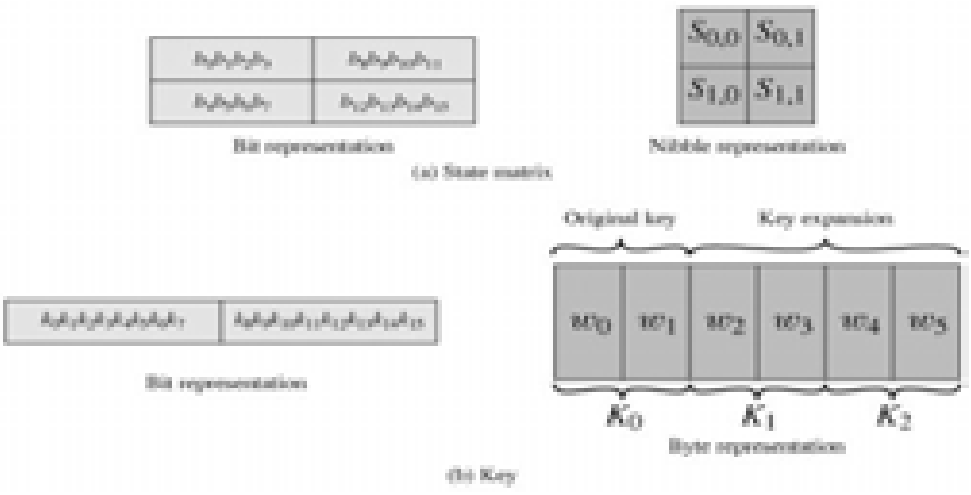
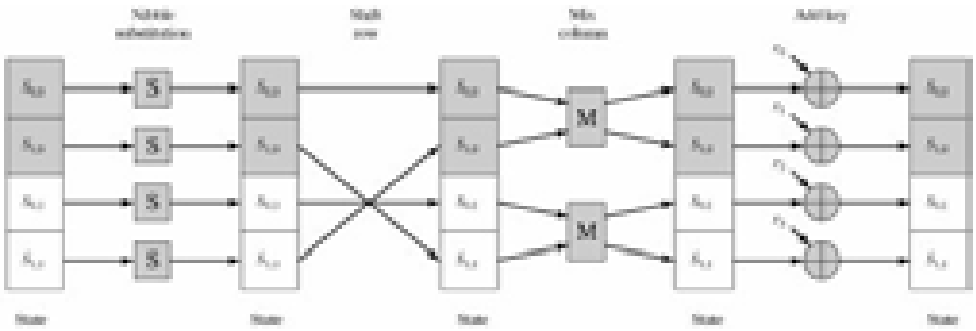


Figure 5.10 shows the essential elements of a full round of S-AES.

Figure 5.10. S-AES Encryption Round

(This item is displayed on page 167 in the print version)



Decryption is also shown in Figure 5.8 and is essentially the reverse of encryption:

$$A_{K0} \circ INS \circ ISR \circ IMC \circ A_{K1} \circ INS \circ ISR \circ A_{K2}$$

in which three of the functions have a corresponding inverse function: inverse nibble substitution (INS), inverse shift row (ISR), and inverse mix column (IMC).

S-AES Encryption and Decryption

We now look at the individual functions that are part of the encryption algorithm.

Add Key

The add key function consists of the bitwise XOR of the 16-bit state matrix and the 16-bit round key. Figure 5.11 depicts this as a columnwise operation, but it can also be viewed as a nibble-wise or bitwise operation. The following is an example:

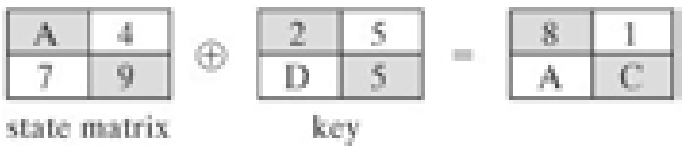
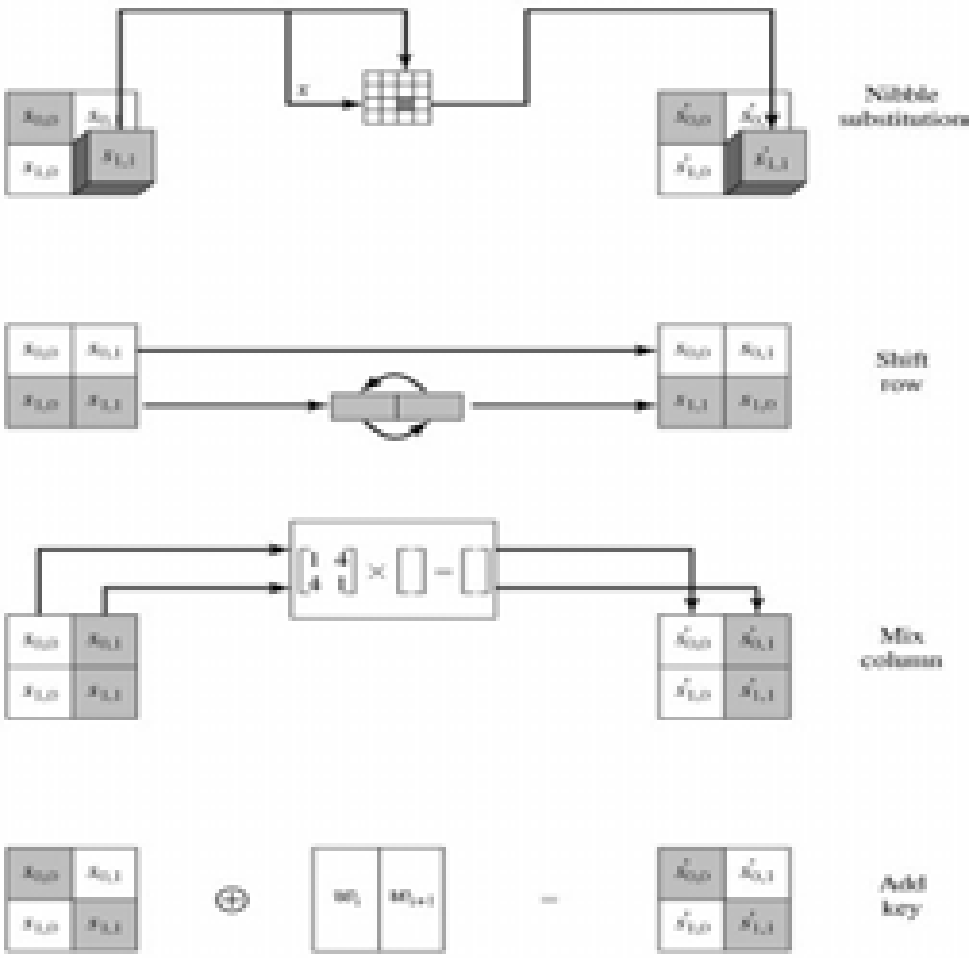


Figure 5.11. S-AES Transformations

[View full size image]



The inverse of the add key function is identical to the add key function, because the XOR operation is its own inverse.

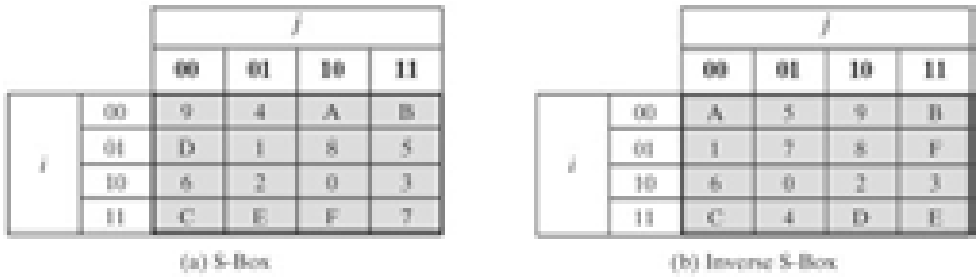
Nibble Substitution

The nibble substitution function is a simple table lookup (Figure 5.11). AES defines a 4 x 4 matrix of nibble values, called an S-box (Table 5.5a), that contains a permutation of all possible 4-bit values. Each individual nibble of the state matrix is mapped into a new nibble in the following way: The leftmost 2 bits of the nibble are used as a row value and the rightmost 2 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 4-bit output value. For example, the

hexadecimal value A references row 2, column 2 of the S-box, which contains the value 0. Accordingly, the value A is mapped into the value 0.

Table 5.5. S-AES S-Boxes

Note: Hexadecimal numbers in shaded boxes; binary numbers in unshaded boxes.



		j			
		00	01	10	11
i	00	9	4	A	B
	01	D	1	8	5
	10	6	2	0	3
	11	C	E	F	7

(a) S-Box

		j			
		00	01	10	11
i	00	A	5	9	B
	01	1	7	8	F
	10	6	0	2	3
	11	C	4	D	E

(b) Inverse S-Box

Here is an example of the nibble substitution transformation:

8	1	→	6	4
A	C		0	C

The inverse nibble substitution function makes use of the inverse S-box shown in Table 5.5b. Note, for example, that the input 0 produces the output A, and the input A to the S-box produces 0.

Shift Row

The shift row function performs a one-nibble circular shift of the second row of the state matrix; the first row is not altered (Figure 5.11). The following is an example:

6	4	→	6	4
0	C		C	0

The inverse shift row function is identical to the shift row function, because it shifts the second row back to its original position.

Mix Column

The mix column function operates on each column individually. Each nibble of a column is mapped into a new value that is a function of both nibbles in

that column. The transformation can be defined by the following matrix multiplication on the state matrix (Figure 5.11):

$$\begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} \\ S'_{1,0} & S'_{1,1} \end{bmatrix}$$

Performing the matrix multiplication, we get:

$S'_{0,0} = S_{0,0} \oplus (4 \cdot S_{1,0})$

$S'_{1,0} = (4 \cdot S_{0,0}) \oplus S_{1,0}$

$S'_{0,1} = S_{0,1} \oplus (4 \cdot S_{1,1})$

$S'_{1,1} = (4 \cdot S_{0,1}) \oplus S_{1,1}$

Where arithmetic is performed in GF(2^4), and the symbol $\cdot$ refers to multiplication in GF(2^4). Appendix E provides the addition and multiplication tables. The following is an example:

$$\begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 6 & 4 \\ C & 0 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 7 & 3 \end{bmatrix}$$

The inverse mix column function is defined as follows:

$$\begin{bmatrix} 9 & 2 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} \\ S'_{1,0} & S'_{1,1} \end{bmatrix}$$

We demonstrate that we have indeed defined the inverse in the following fashion:

$$\begin{bmatrix} 9 & 2 \\ 2 & 9 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} = \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix}$$

The preceding matrix multiplication makes use of the following results in GF(2^4): $9 + (2 \cdot 4) = 9 + 8 = 1$; $(9 \cdot 4) + 2 = 2 + 2 = 0$. These operations can be verified using the arithmetic tables in Appendix E or by polynomial arithmetic.

The mix column function is the most difficult to visualize. Accordingly, we provide an additional perspective on it in Appendix E.

Key Expansion

For key expansion, the 16 bits of the initial key are grouped into a row of two 8-bit words. Figure 5.12 shows the expansion into 6 words, by the calculation of 4 new words from the initial 2 words. The algorithm is as follows:

$w_2 = w_0 \oplus g(w_1) = w_0 \oplus RCON(1) \oplus SubNib(RotNib(w_1))$

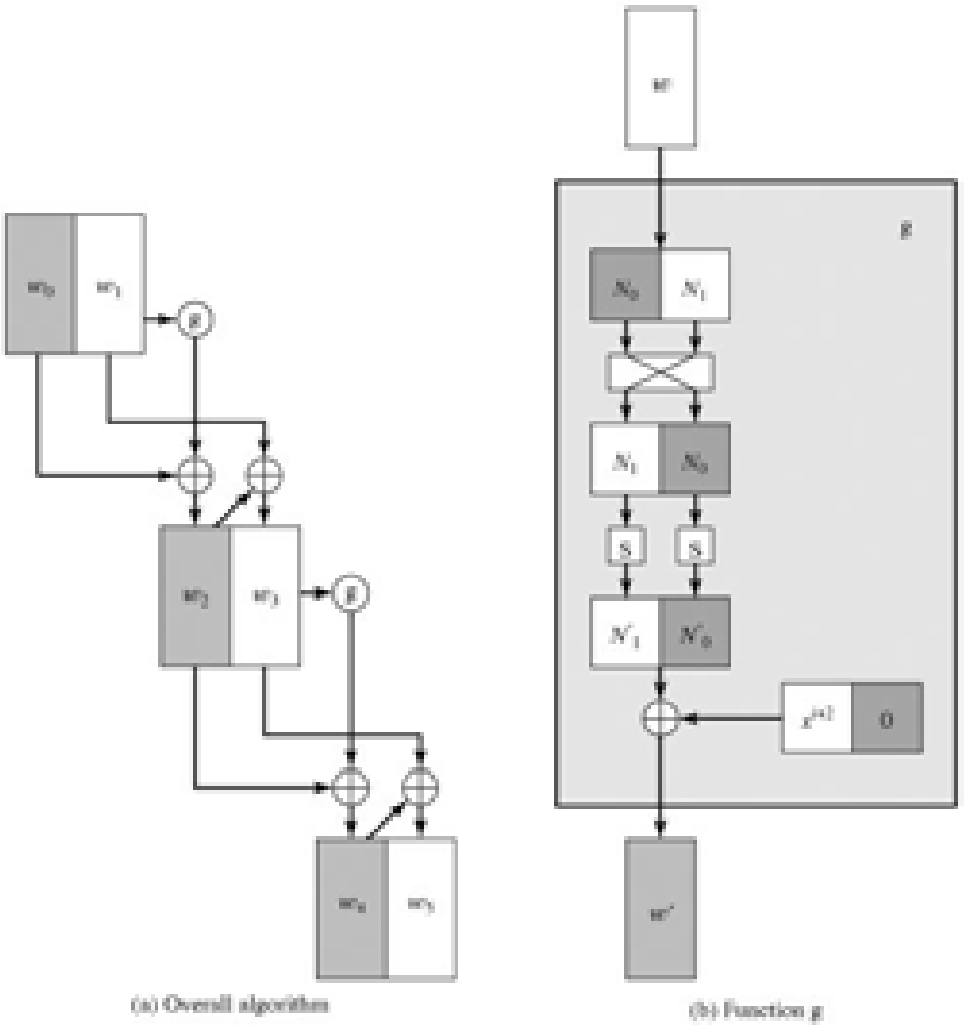
$w_3 = w_2 \oplus w_1$

$w_4 = w_2 \oplus g(w_3) = w_2 \oplus RCON(2) \oplus SubNib(RotNib(w_3))$

$w_5 = w_4 \oplus w_3$

Figure 5.12. S-AES Key Expansion

(This item is displayed on page 171 in the print version)



(a) Overall algorithm (b) Function g

RCON is a round constant, defined as follows: RC[i] = x^{i+2}, so that RC[1] = x^3 = 1000 and RC[2] = x^4 mod (x^4 + x + 1) = x + 1 = 0011. RC[i] forms the leftmost nibble of a byte, with the rightmost nibble being all zeros. Thus, RCON(1) = 10000000 and RCON(2) = 00110000.

For example, suppose the key is 2D55 = 0010 1101 0101 0101 = w_0w_1. Then

w_2 = 00101101 ⊕10000000 ⊕SubNib(01010101)

= 00101101 ⊕10000000 ⊕00010001 = 10111100

w_3 = 10111100 ⊕01010101 = 11101001

$w_4 = 10111110 \oplus 00110000 \oplus \text{SubNib}(10011110)$

$= 10111100 \oplus 00110000 \oplus 00101111 = 10100011$

$w_5 = 10100011 \oplus 11101001 = 01001010$

The S-Box

The S-box is constructed as follows:

> Initialize the S-box with the nibble values in ascending sequence row by row. The first row contains the hexadecimal values 0, 1, 2, 3; the second row contains 4, 5, 6, 7; and so on. Thus, the value of the nibble at row i, column j is 4i + j.

1. Treat each nibble as an element of the finite field GF(2^4) modulo $x^4 + x + 1$. Each nibble $a_0a_1a_2a_3$ represents a polynomial of degree 3.
2. Map each byte in the S-box to its multiplicative inverse in the finite field GF(2^4) modulo $x^4 + x + 1$; the value 0 is mapped to itself.
3. Consider that each byte in the S-box consists of 4 bits labeled (b_0, b_1, b_2, b_3). Apply the following transformation to each bit of each byte in the S-box: The AES standard depicts this transformation in matrix form as follows:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

> The prime (') indicates that the variable is to be updated by the value on the right. Remember that addition and multiplication are being calculated modulo 2.

Table 5.5a shows the resulting S-box. This is a nonlinear, invertible matrix. The inverse S-box is shown in Table 5.5b.

S-AES Structure

We can now examine several aspects of interest concerning the structure of AES. First, note that the encryption and decryption algorithms begin and end with the add key function. Any other function, at the beginning or end, is easily reversible without knowledge of the key and so would add no security but just a processing overhead. Thus, there is a round 0 consisting of only the add key function.

The second point to note is that round 2 does not include the mix column function. The explanation for this in fact relates to a third observation, which is that although the decryption algorithm is the reverse of the encryption algorithm, as clearly seen in Figure 5.8, it does not follow the same sequence of functions. Thus

Encryption: A_{K2} º SR º NS º A_{K1} º MC º SR º NS º A_{K0}

Decryption: A_{K0} º INS º ISR º IMC º A_{K1} º INS º ISR º A_{K2}

From an implementation point of view, it would be desirable to have the decryption function follow the same function sequence as encryption. This allows the decryption algorithm to be implemented in the same way as the encryption algorithm, creating opportunities for efficiency.

Note that if we were able to interchange the second and third functions, the fourth and fifth functions, and the sixth and seventh functions in the decryption sequence, we would have the same structure as the encryption algorithm. Let's see if this is possible. First, consider the interchange of INS and ISR. Given a state N consisting of the nibbles (N_0, N_1, N_2, N_3) the transformation INS(ISR(N)) proceeds as follows:

$$\begin{pmatrix} N_0 & N_2 \\ N_1 & N_3 \end{pmatrix} \rightarrow \begin{pmatrix} N_0 & N_2 \\ N_3 & N_1 \end{pmatrix} \rightarrow \begin{pmatrix} IS[N_0] & IS[N_2] \\ IS[N_3] & IS[N_1] \end{pmatrix}$$

Where IS refers to the inverse S-Box. Reversing the operations, the transformation ISR(INS(N) proceeds as follows:

$$\begin{pmatrix} N_0 & N_2 \\ N_1 & N_3 \end{pmatrix} \rightarrow \begin{pmatrix} IS[N_0] & IS[N_2] \\ IS[N_1] & IS[N_3] \end{pmatrix} \rightarrow \begin{pmatrix} IS[N_0] & IS[N_2] \\ IS[N_3] & IS[N_1] \end{pmatrix}$$

which is the same result. Thus, INS(ISR(N)) = ISR(INS(N)).

Now consider the operation of inverse mix column followed by add key: $IMC(A_{K1}(N))$ where the round key K_1 consists of the nibbles ($k_{0,0}$, $k_{1,0}$, $k_{0,1}$, $k_{1,1}$) Then:

$$\begin{aligned}
\begin{pmatrix} 9 & 2 \\ 2 & 9 \end{pmatrix}\left(\begin{pmatrix} k_{0,0} & k_{0,1} \\ k_{1,0} & k_{1,1} \end{pmatrix} \oplus \begin{pmatrix} N_0 & N_2 \\ N_1 & N_3 \end{pmatrix}\right) &= \begin{pmatrix} 9 & 2 \\ 2 & 9 \end{pmatrix}\begin{pmatrix} k_{0,0} \oplus N_0 & k_{0,1} \oplus N_2 \\ k_{1,0} \oplus N_1 & k_{1,1} \oplus N_3 \end{pmatrix} \\
&= \begin{pmatrix} 9(k_{0,0} \oplus N_0) \oplus 2(K_{1,0} \oplus N_1) & 9(k_{0,1} \oplus N_2) \oplus 2(K_{1,1} \oplus N_3) \\ 2(k_{0,0} \oplus N_0) \oplus 9(K_{1,0} \oplus N_1) & 2(k_{0,1} \oplus N_2) \oplus 9(K_{1,1} \oplus N_3) \end{pmatrix} \\
&= \begin{pmatrix} (9k_{0,0} \oplus 2k_{1,0}) \oplus (9N_0 \oplus 2N_1) & (9k_{0,1} \oplus 2k_{1,1}) \oplus (9N_2 \oplus 2N_3) \\ (2k_{0,0} \oplus 9k_{1,0}) \oplus (2N_0 \oplus 9N_1) & (2k_{0,1} \oplus 9k_{1,1}) \oplus (2N_2 \oplus 9N_3) \end{pmatrix} \\
&= \begin{pmatrix} (9k_{0,0} \oplus 2k_{1,0}) & (9k_{0,1} \oplus 2k_{1,1}) \\ (2k_{0,0} \oplus 9k_{1,0}) & (2k_{0,1} \oplus 9k_{1,1}) \end{pmatrix} \oplus \begin{pmatrix} (9N_0 \oplus 2N_1) & (9N_2 \oplus 2N_3) \\ (2N_0 \oplus 9N_1) & (2N_2 \oplus 9N_3) \end{pmatrix} \\
&= \begin{pmatrix} 9 & 2 \\ 2 & 9 \end{pmatrix}\begin{pmatrix} k_{0,0} & k_{0,1} \\ k_{1,0} & k_{1,1} \end{pmatrix} \oplus \begin{pmatrix} 9 & 2 \\ 2 & 9 \end{pmatrix}\begin{pmatrix} N_0 & N_2 \\ N_1 & N_3 \end{pmatrix}
\end{aligned}$$

All of the above steps make use of the properties of finite field arithmetic.

The result is that $IMC(A_{K1}(N)) = IMC(K_1 \oplus IMC(N)$. Now let us define the inverse round key for round 1 to be $IMC(K_1)$ and the inverse add key operation IA_{K1} to be the bitwise XOR of the inverse round key with the state vector. Then we have $IMC(A_{K1}(N)) = IA_{K1}(IMC(N))$. As a result, we can write the following:

Encryption: A_{K2} º SR º NS º A_{K1} º MC º SR º NS º A_{K0}

Decryption: A_{K0} º INS º ISR º IMC º A_{K1} º INS º ISR º A_{K2}

Decryption: A_{K0} º ISR º INS º $A_{IMC(K1)}$ º IMC º ISR º INS º A_{K2}

Both encryption and decryption now follow the same sequence. Note that this derivation would not work as effectively if round 2 of the encryption algorithm included the MC function. In that case, we would have

Encryption: A_{K2} º MC º SR º NS º A_{K1} º MC º SR º NS º A_{K0}

Decryption: A_{K0} º INS º ISR º IMC º A_{K1} º INS º ISR º IMC º A_{K2}

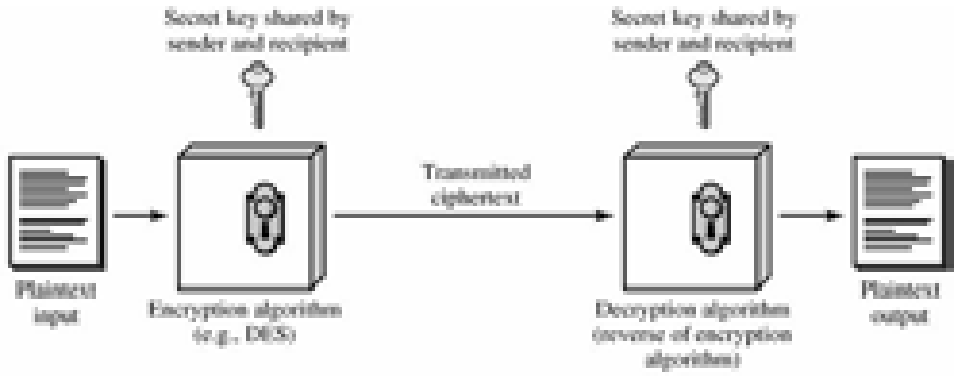
There is now no way to interchange pairs of operations in the decryption algorithm so as to achieve the same structure as the encryption algorithm.

Symmetric Cipher Model

A symmetric encryption scheme has five ingredients (Figure 2.1):

- Plaintext: This is the original intelligible message or data that is fed into the algorithm as input.
- Encryption algorithm: The encryption algorithm performs various substitutions and transformations on the plaintext.
- Secret key: The secret key is also input to the encryption algorithm. The key is a value independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key.
- Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and, as it stands, is unintelligible.
- Decryption algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

Figure 2.1. Simplified Model of Conventional Encryption



There are two requirements for secure use of conventional encryption:

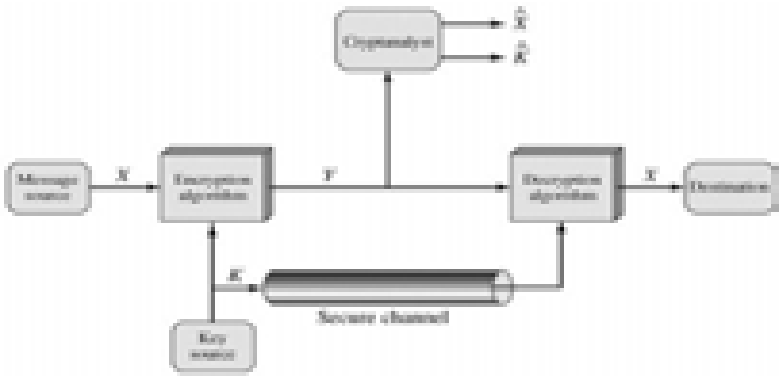
1. We need a strong encryption algorithm. At a minimum, we would like the algorithm to be such that an opponent who knows the algorithm and has access to one or more ciphertexts would be unable to decipher the ciphertext or figure out the key. This requirement is usually stated in a stronger form: The opponent should be unable to decrypt ciphertext or discover the key even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.
2. Sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key is readable.

We assume that it is impractical to decrypt a message on the basis of the ciphertext plus knowledge of the encryption/decryption algorithm. In other words, we do not need to keep the algorithm secret; we need to keep only the key secret. This feature of symmetric encryption is what makes it feasible for widespread use. The fact that the algorithm need not be kept secret means that manufacturers can and have developed low-cost chip implementations of data encryption algorithms. These chips are widely available and incorporated into a number of products. With the use of symmetric encryption, the principal security problem is maintaining the secrecy of the key.

Let us take a closer look at the essential elements of a symmetric encryption scheme, using Figure 2.2. A source produces a message in plaintext, $X = [X_1, X_2, ..., X_M]$. The M elements of X are letters in some finite alphabet. Traditionally, the alphabet usually consisted of the 26 capital letters.

Nowadays, the binary alphabet {0, 1} is typically used. For encryption, a key of the form $K = [K_1, K_2, ..., K_J]$ is generated. If the key is generated at the message source, then it must also be provided to the destination by means of some secure channel. Alternatively, a third party could generate the key and securely deliver it to both source and destination.

Figure 2.2. Model of Conventional Cryptosystem



With the message X and the encryption key K as input, the encryption algorithm forms the ciphertext $Y = [Y_1, Y_2, ..., Y_N]$. We can write this as

$Y = E(K, X)$

This notation indicates that Y is produced by using encryption algorithm E as a function of the plaintext X, with the specific function determined by the value of the key K.

The intended receiver, in possession of the key, is able to invert the transformation:

$X = D(K, Y)$

An opponent, observing Y but not having access to K or X, may attempt to recover X or K or both X and K. It is assumed that the opponent knows the encryption (E) and decryption (D) algorithms. If the opponent is interested in only this particular message, then the focus of the effort is to recover X by generating a plaintext estimate . Often, however, the opponent is interested in being able to read future messages as well, in which case an attempt is made to recover K by generating an estimate .

Cryptography

Cryptographic systems are characterized along three independent dimensions:

1. The type of operations used for transforming plaintext to ciphertext. All encryption algorithms are based on two general principles: substitution, in which each element in the plaintext (bit, letter, group of bits or letters) is mapped into another element, and transposition, in which elements in the plaintext are rearranged. The fundamental requirement is that no information be lost (that is, that all operations are reversible). Most systems, referred to as product systems, involve multiple stages of substitutions and transpositions.
2. The number of keys used. If both sender and receiver use the same key, the system is referred to as symmetric, single-key, secret-key, or conventional encryption. If the sender and receiver use different keys, the system is referred to as asymmetric, two-key, or public-key encryption.
3. The way in which the plaintext is processed. A *block cipher* processes the input one block of elements at a time, producing an output block for each input block. A *stream cipher* processes the input elements continuously, producing output one element at a time, as it goes along.

Cryptanalysis

Typically, the objective of attacking an encryption system is to recover the key in use rather then simply to recover the plaintext of a single ciphertext. There are two general approaches to attacking a conventional encryption scheme:

Cryptanalysis: Cryptanalytic attacks rely on the nature of the algorithm plus perhaps some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used.

- **Brute-force attack:** The attacker tries every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained. On average, half of all possible keys must be tried to achieve success.

If either type of attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised.

We first consider cryptanalysis and then discuss brute-force attacks.

Table 2.1 summarizes the various types of cryptanalytic attacks, based on the amount of information known to the cryptanalyst. The most difficult problem is presented when all that is available is the ciphertext only. In some cases, not even the encryption algorithm is known, but in general we can assume that the opponent does know the algorithm used for encryption. One possible attack under these circumstances is the brute-force approach of trying all possible keys. If the key space is very large, this becomes impractical. Thus, the opponent must rely on an analysis of the ciphertext itself, generally applying various statistical tests to it. To use this approach, the opponent must have some general idea of the type of plaintext that is concealed, such as English or French text, an EXE file, a Java source listing, an accounting file, and so on.

Table 2.1. Types of Attacks on Encrypted Messages

Type of Attack	**Known to Cryptanalyst**
Ciphertext only	• Encryption algorithm • Ciphertext
Known plaintext	• Encryption algorithm • Ciphertext • One or more plaintext-ciphertext pairs formed with the secret key
Chosen plaintext	• Encryption algorithm • Ciphertext • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key
Chosen ciphertext	• Encryption algorithm • Ciphertext • Purported ciphertext chosen by cryptanalyst, together with its

Type of Attack	Known to Cryptanalyst
	corresponding decrypted plaintext generated with the secret key
Chosen text	• Encryption algorithm • Ciphertext • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key • Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key

The ciphertext-only attack is the easiest to defend against because the opponent has the least amount of information to work with. In many cases, however, the analyst has more information. The analyst may be able to capture one or more plaintext messages as well as their encryptions. Or the analyst may know that certain plaintext patterns will appear in a message. For example, a file that is encoded in the Postscript format always begins with the same pattern, or there may be a standardized header or banner to an electronic funds transfer message, and so on. All these are examples of known plaintext. With this knowledge, the analyst may be able to deduce the key on the basis of the way in which the known plaintext is transformed.

Closely related to the known-plaintext attack is what might be referred to as a probable-word attack. If the opponent is working with the encryption of some general prose message, he or she may have little knowledge of what is in the message. However, if the opponent is after some very specific information, then parts of the message may be known. For example, if an entire accounting file is being transmitted, the opponent may know the placement of certain key words in the header of the file. As another example, the source code for a program developed by Corporation X might include a copyright statement in some standardized position.

If the analyst is able somehow to get the source system to insert into the system a message chosen by the analyst, then a chosen-plaintext attack is possible. if the analyst is able to choose the messages to encrypt, the analyst may deliberately pick patterns that can be expected to reveal the structure of the key.

Table 2.1 lists two other types of attack: chosen ciphertext and chosen text. These are less commonly employed as cryptanalytic techniques but are nevertheless possible avenues of attack.

Only relatively weak algorithms fail to withstand a ciphertext-only attack. Generally, an encryption algorithm is designed to withstand a known-plaintext attack.

Two more definitions are worthy of note. An encryption scheme is unconditionally secure if the ciphertext generated by the scheme does not contain enough information to determine uniquely the corresponding plaintext, no matter how much ciphertext is available. That is, no matter how much time an opponent has, it is impossible for him or her to decrypt the ciphertext, simply because the required information is not there. With the exception of a scheme known as the one-time pad (described later in this chapter), there is no encryption algorithm that is unconditionally secure. Therefore, all that the users of an encryption algorithm can strive for is an algorithm that meets one or both of the following criteria:

- The cost of breaking the cipher exceeds the value of the encrypted information.
- The time required to break the cipher exceeds the useful lifetime of the information.

An encryption scheme is said to be computationally secure if either of the foregoing two criteria are met. The rub is that it is very difficult to estimate the amount of effort required to cryptanalyze ciphertext successfully.

All forms of cryptanalysis for symmetric encryption schemes are designed to exploit the fact that traces of structure or pattern in the plaintext may survive encryption and be discernible in the ciphertext. This will become clear as we examine various symmetric encryption schemes in this chapter. that cryptanalysis for public-key schemes proceeds from a fundamentally different premise, namely, that the mathematical properties of the pair of keys may make it possible for one of the two keys to be deduced from the other.

A brute-force attack involves trying every possible key until an intelligible translation of the ciphertext into plaintext is obtained. On average, half of all possible keys must be tried to achieve success. Table 2.2 shows how much time is involved for various key spaces. Results are shown for four binary

key sizes. The 56-bit key size is used with the DES (Data Encryption Standard) algorithm, and the 168-bit key size is used for triple DES. The minimum key size specified for AES (Advanced Encryption Standard) is 128 bits. Results are also shown for what are called substitution codes that use a 26-character key (discussed later), in which all possible permutations of the 26 characters serve as keys. For each key size, the results are shown assuming that it takes 1 μs to perform a single decryption, which is a reasonable order of magnitude for today's machines. With the use of massively parallel organizations of microprocessors, it may be possible to achieve processing rates many orders of magnitude greater. The final column of Table 2.2 considers the results for a system that can process 1 million keys per microsecond. As you can see, at this performance level, DES can no longer be considered computationally secure.

Table 2.2. Average Time Required for Exhaustive Key Search

Key size (bits)	Number of alternative keys		Time required at 1 decryption/ s		Time required at 10^6 decryption/ s
32	2^{32}	= 4.3 x 10^9	2^{31} μs	= 35.8 minutes	2.15 milliseconds
56	2^{56}	= 7.2 x 10^{16}	2^{55} μs	= 1142 years	10.01 hours
128	2^{128}	= 3.4 x 10^{38}	2^{127} μs	= 5.4 x 10^{24} years	5.4 x 10^{18} years
168	2^{168}	= 3.7 x 10^{50}	2^{167} μs	= 5.9 x 10^{36} years	5.9 x 10^{30} years
26 characters (permutation)	26!	= 4 x 10^{26}	2 x 10^{26} μs	= 6.4 x 10^{12} years	6.4 x 10^6 years

Testing for Primality

For many cryptographic algorithms, it is necessary to select one or more very large prime numbers at random. Thus we are faced with the task of determining whether a given large number is prime. There is no simple yet efficient means of accomplishing this task.

In this section, we present one attractive and popular algorithm. You may be surprised to learn that this algorithm yields a number that is not necessarily a prime. However, the algorithm can yield a number that is almost certainly a prime. This will be explained presently. We also make reference to a deterministic algorithm for finding primes. The section closes with a discussion concerning the distribution of primes.

Miller-Rabin Algorithm

The algorithm due to Miller and Rabin is typically used to test a large number for primality. Before explaining the algorithm, we need some background. First, any positive odd integer n $\geq$3 can be expressed as follows:

n 1 = 2^kq with k > 0, q odd

To see this, note that (n 1) is an even integer. Then, divide (n 1) by 2 until the result is an odd number q, for a total of k divisions. If n is expressed as a binary number, then the result is achieved by shifting the number to the right until the rightmost digit is a 1, for a total of k shifts. We now develop two properties of prime numbers that we will need.

Two Properties of Prime Numbers

The first property is stated as follows: If p is prime and a is a positive integer less than p, then a^2 mod p = 1 if and only if either a mod p = 1 or a mod p= 1 mode p = p 1. By the rules of modular arithmetic (a mode p) (a mode p) = a^2 mod p. Thus if either a mode p = 1 or a mod p = 1, then a^2 mod p = 1. Conversely, if a^2 mod p = 1, then $(a \bmod p)^2$ = 1, which is true only for a mod p = 1 or a mod p = 1.

The second property is stated as follows: Let p be a prime number greater than 2. We can then write p 1 = 2^kq, with k > 0 q odd. Let a be any integer in the range 1 < a < p 1. Then one of the two following conditions is true:

1. a^q is congruent to 1 modulo p. That is, a^q mod p = 1, or equivalently, $a^q \equiv 1 \pmod{p}$.
2. One of the numbers a^q, a^{2q}, a^{4q},..., a^{2k-1q} is congruent to 1 modulo p. That is, there is some number j in the range ($1 \leq j \leq k$) such that a^{2j-1q} mod p = 1 mod p = p 1, or equivalently, $a^{2j-1q} \equiv 1 \pmod{p}$.

Proof: Fermat's theorem [Equation (8.2)] states that $a^{n1} \equiv 1 \pmod{n}$ if n is prime. We have p 1 = 2^kq. Thus, we know that a^{p1} mod p = a^{2kq} mod p = 1. Thus, if we look at the sequence of numbers

Equation 8-6

$$a^q \bmod p, a^{2q} \bmod p, a^{4q} \bmod p, \ldots, a^{2^{k-1}q} \bmod p, a^{2^k q} \bmod p$$

we know that the last number in the list has value 1. Further, each number in the list is the square of the previous number. Therefore, one of the following possibilities must be true:

1. The first number on the list, and therefore all subsequent numbers on the list, equals 1.
2. Some number on the list does not equal 1, but its square mod p does equal 1. By virtue of the first property of prime numbers defined above, we know that the only number that satisfies this condition p 1 is So, in this case, the list contains an element equal to p 1.

   This completes the proof.

Details of the Algorithm

These considerations lead to the conclusion that if n is prime, then either the first element in the list of residues, or remainders, (a^q, a^{2q},..., a^{2k-1q}, a^{2kq}) modulo n equals 1, or some element in the list equals (n 1); otherwise n is composite (i.e., not a prime). On the other hand, if the condition is met, that does not necessarily mean that n is prime. For example, if n = 2047 = 23 x

89, then n 1 = 2 x 1023. Computing, 2^{1023} mod 2047 = 1, so that 2047 meets the condition but is not prime.

We can use the preceding property to devise a test for primality. The procedure TEST takes a candidate integer n as input and returns the result `composite` if n is definitely not a prime, and the result `inconclusive` if n may or may not be a prime.

TEST (n)
1. Find integers k, q, with k > 0, q odd, so that (n 1 = 2^kq);
2. Select a random integer a, 1 < a < n 1;
3. If a^q mod n = 1 then return ("inconclusive");
4. For j = 0 to k 1 do
5. if a^{2jq} mod n ≡ n 1 then return("inconclusive");
6. Return ("composite");

[

Let us apply the test to the prime number n = 29. We have (n 1) = 28 = 2^2(7) = 2^kq. First, let us try a = 10. We compute 10^7 mod 29 = 17, which is neither 1 nor 28, so we continue the test. The next calculation finds that $(10^7)^2$ mod 29 = 28, and the test returns `inconclusive` (i.e., 29 may be prime). Let's try again with a = 2. We have the following calculations: 2^7 mod 29 = 12;2^{14} mod 29 = 28; and the test again returns `inconclusive`. If we perform the test for all integers a in the range 1 through 28, we get the same inconclusive result, which is compatible with n being a prime number.

Now let us apply the test to the composite number n = 13 x 17 = 221. Then (n 1) = 220 = 2^2(55) = 2^kq. Let us try a = 5. Then we have 5^{55} mod 221 = 112, which is neither 1 nor 220; $(5^{55})^2$ mod 221 = 168. Because we have used all values of j (i.e., j = 0 andj = 1) in line 4 of the TEST algorithm, the test returns `composite`, indicating that 221 is definitely a composite number. But suppose we had selected a = 21. Then we have 21^{55} mod 221 = 200; $(21^{55})^2$ mod 221 = 220; and the test returns `inconclusive`, indicating that 221 may be prime. In fact, of the 218 integers from 2 through 219, four of these will return an inconclusive result, namely 21, 47, 174, and 200.

How can we use the Miller-Rabin algorithm to determine with a high degree of confidence whether or not an integer is prime? It can be shown that given an odd number n that is not prime and a randomly chosen integer, a with 1 < a < n 1, the probability that TEST will return `inconclusive` (i.e., fail to detect that n is not prime) is less than 1/4. Thus, if t different values of a are chosen, the probability that all of them will pass TEST (return inconclusive) for n is less than $(1/4)^t$ For example, for t = 10, the probability that a nonprime number will pass all ten tests is less than 10^6. Thus, for a sufficiently large value of t, we can be confident that n is prime if Miller's test always returns `inconclusive`.

This gives us a basis for determining whether an odd integer n is prime with a reasonable degree of confidence. The procedure is as follows: Repeatedly invoke TEST (n) using randomly chosen values for a. If, at any point, TEST returns `composite`, then n is determined to be nonprime. If TEST continues to return `inconclusive` for t tests, for a sufficiently large value of t, assume that n is prime.

A Deterministic Primality Algorithm

Prior to 2002, there was no known method of efficiently proving the primality of very large numbers. All of the algorithms in use, including the most popular (Miller-Rabin), produced a probabilistic result. In 2002, Agrawal, Kayal, and Saxena developed a relatively simple deterministic algorithm that efficiently determines whether a given large number is a prime. The algorithm, known as the AKS algorithm, does not appear to be as efficient as the Miller-Rabin algorithm. Thus far, it has not supplanted this older, probabilistic technique .

Distribution of Primes

It is worth noting how many numbers are likely to be rejected before a prime number is found using the Miller-Rabin test, or any other test for primality. A result from number theory, known as the prime number theorem, states that the primes near n are spaced on the average one every (ln n) integers. Thus, on average, one would have to test on the order of ln(n) integers before a prime is found. Because all even integers can be immediately rejected, the correct figure is 0.5 ln(n). For example, if a prime on the order

of magnitude of 2^{200} were sought, then about $0.5 \ln(2^{200}) = 69$ trials would be needed to find a prime. However, this figure is just an average. In some places along the number line, primes are closely packed, and in other places there are large gaps.

> The two consecutive odd integers 1,000,000,000,061 and 1,000,000,000,063 are both prime. On the other hand, 1001! + 2,1001! + 3,..., 1001! + 1000, 1001! + 1001 is a sequence of 1000 consecutive composite integers.

The AES Ciphe

The Rijndael proposal for AES defined a cipher in which the block length and the key length can be independently specified to be 128, 192, or 256 bits. The AES specification uses the same three key size alternatives but limits the block length to 128 bits. A number of AES parameters depend on the key length (Table 5.3). In the description of this section, we assume a key length of 128 bits, which is likely to be the one most commonly implemented.

Table 5.3. AES Parameters

Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

Rijndael was designed to have the following characteristics:

- Resistance against all known attacks
- Speed and code compactness on a wide range of platforms
- Design simplicity

Figure 5.1 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a square matrix of bytes. This block is copied into the

State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix. These operations are depicted in Figure 5.2a. Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is four bytes and the total key schedule is 44 words for the 128-bit key (Figure 5.2b). Note that the ordering of bytes within a matrix is by column. So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the in matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the w matrix.

Figure 5.1. AES Encryption and Decryption

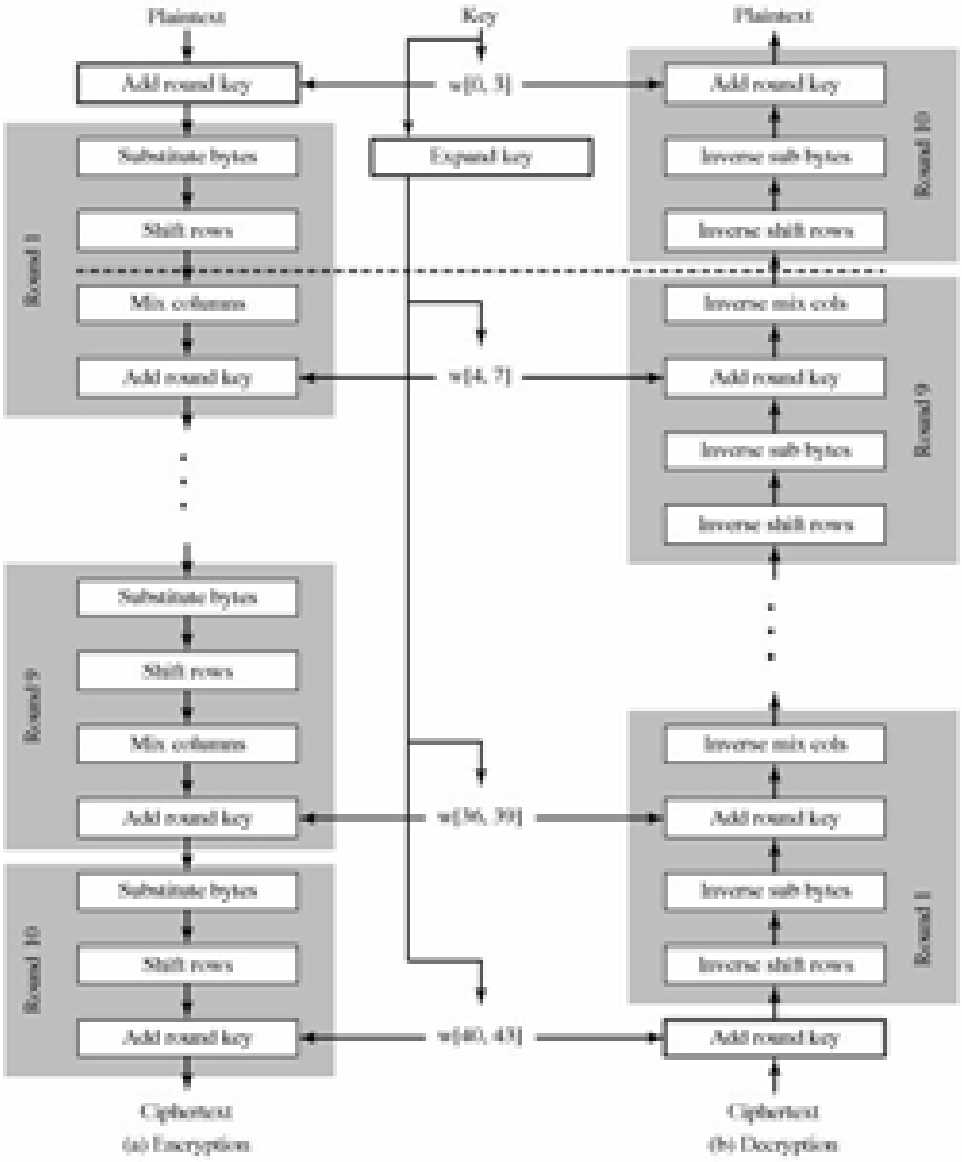
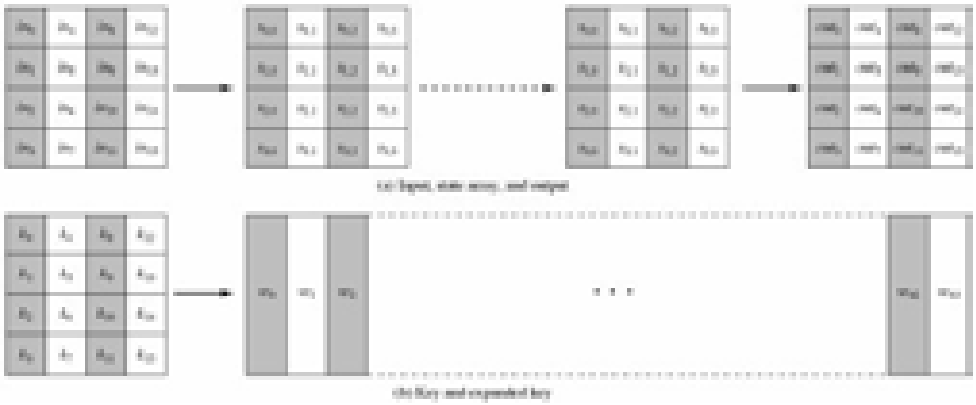


Figure 5.2. AES Data Structures

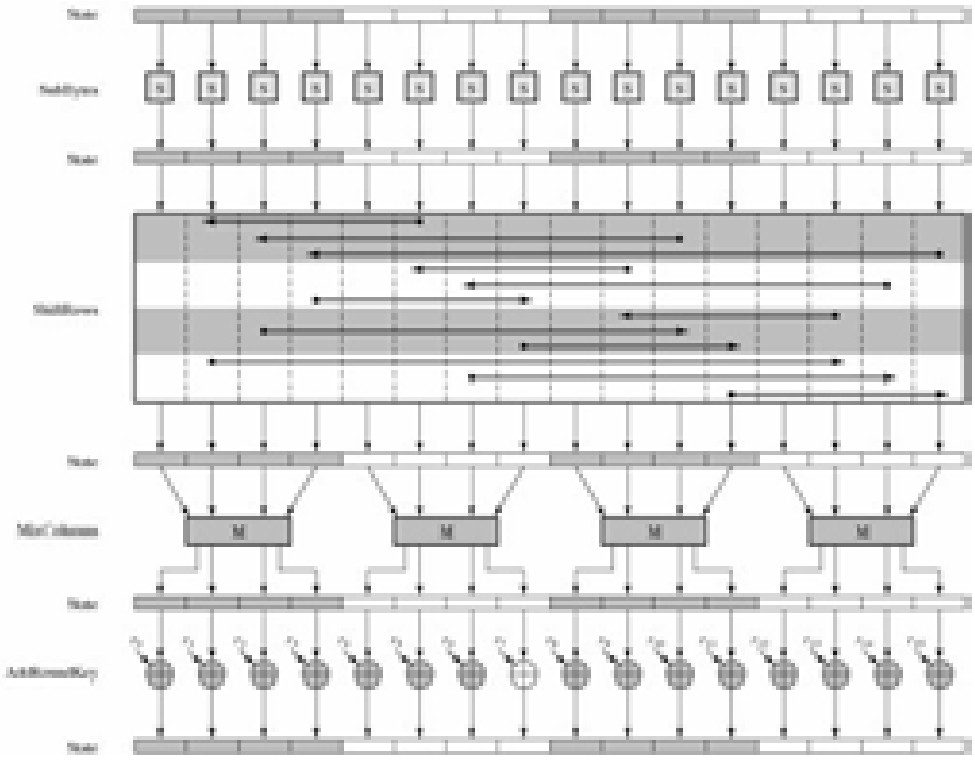


Before delving into details, we can make several comments about the overall AES structure:

1. One noteworthy feature of this structure is that it is not a Feistel structure. Recall that in the classic Feistel structure, half of the data block is used to modify the other half of the data block, and then the halves are swapped. Two of the AES finalists, including Rijndael, do not use a Feistel structure but process the entire data block in parallel during each round using substitutions and permutation.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, w[i]. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 5.1.
3. Four different stages are used, one of permutation and three of substitution:
    - Substitute bytes: Uses an S-box to perform a byte-by-byte substitution of the block
    - ShiftRows: A simple permutation
    - MixColumns: A substitution that makes use of arithmetic over GF(2^8)
    - AddRoundKey: A simple bitwise XOR of the current block with a portion of the expanded key
4. The structure is quite simple. For both encryption and decryption, the cipher begins with an AddRoundKey stage, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages. Figure 5.3 depicts the structure of a full encryption round.

Figure 5.3. AES Encryption Round

(This item is displayed on page 144 in the print version)



5. Only the AddRoundKey stage makes use of the key. For this reason, the cipher begins and ends with an AddRoundKey stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.
6. The AddRoundKey stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. We can view the cipher as alternating operations of XOR encryption (AddRoundKey) of a block, followed by scrambling of the block (the other three stages), followed by XOR encryption, and so on. This scheme is both efficient and highly secure.
7. Each stage is easily reversible. For the Substitute Byte, ShiftRows, and MixColumns stages, an inverse function is used in the decryption algorithm. For the AddRoundKey stage, the inverse is achieved by XORing the same round key to the block, using the result that A $\oplus$A $\oplus$B = B.

8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.

   Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 5.1 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), State is the same for both encryption and decryption.

9. The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

We now turn to a discussion of each of the four stages used in AES. For each stage, we describe the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rationale for the stage. This is followed by a discussion of key expansion.

AES uses arithmetic in the finite field $GF(2^8)$, with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. The developers of Rijndael give as their motivation for selecting this one of the 30 possible irreducible polynomials of degree 8 that it is the first one on the list .

Substitute Bytes Transformation

Forward and Inverse Transformations

The forward substitute byte transformation, called SubBytes, is a simple table lookup (Figure 5.4a). AES defines a 16 x 16 matrix of byte values, called an S-box (Table 5.4a), that contains a permutation of all possible 256 8-bit values. Each individual byte of State is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value. For example, the hexadecimal value {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

In FIPS PUB 197, a hexadecimal number is indicated by enclosing it in curly brackets. We use that convention in this chapter.

Figure 5.4. AES Byte-Level Operations

(This item is displayed on page 145 in the print version)

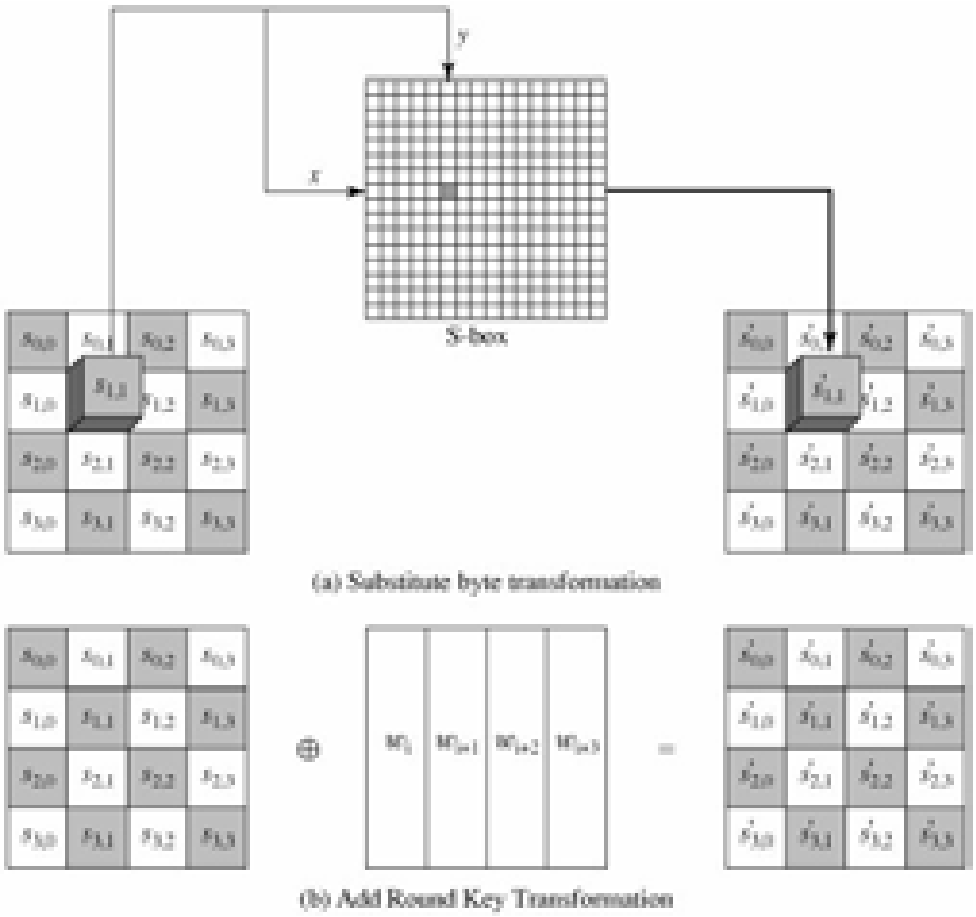


Table 5.4. AES S-Boxes

(This item is displayed on page 146 in the print version)

(a) S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(b) Inverse S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Here is an example of the SubBytes transformation:

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

→

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

The S-box is constructed in the following fashion:

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains {00}, {01}, {02},.... {0F}; the second row contains {10}, {11}, etc.; and so on. Thus, the value of the byte at row x, column y is {xy}.
2. Map each byte in the S-box to its multiplicative inverse in the finite field GF(2^8); the value {00} is mapped to itself.
3. Consider that each byte in the S-box consists of 8 bits labeled (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0). Apply the following transformation to each bit of each byte in the S-box:

   Equation 5-1

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

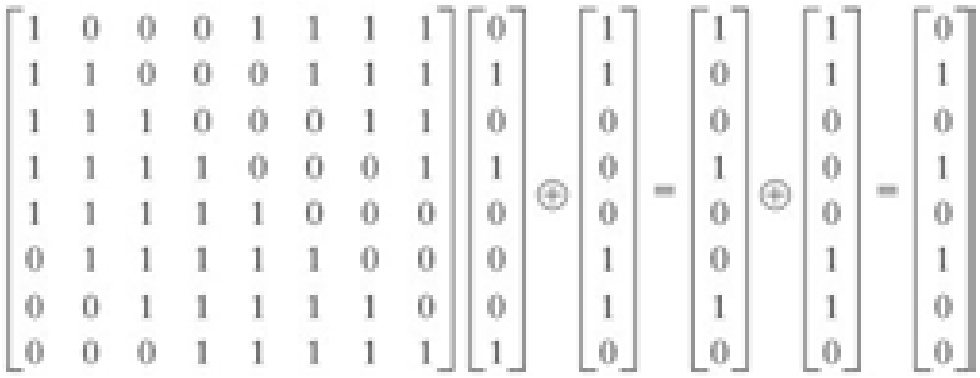
   where c_i is the ith bit of byte c with the value {63}; that is, ($c_7c_6c_5c_4c_3c_2c_1c_0$) = (01100011). The prime (') indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows:

   Equation 5-2

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Equation (5.2) has to be interpreted carefully. In ordinary matrix multiplication,[5] each element in the product matrix is the sum of products of the elements or one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Further, the final addition shown in Equation (5.2) is a bitwise XOR.

As an example, consider the input value {95}. The multiplicative inverse in GF(2^8) is $\{95\}^{-1} = \{8A\}$, which is 10001010 in binary. Using Equation (5.2),



The result is {2A}, which should appear in row {09} column {05} of the S-box. This is verified by checking Table 5.4a.

The inverse substitute byte transformation, called InvSubBytes, makes use of the inverse S-box shown in Table 5.4b. Note, for example, that the input {2A} produces the output {95} and the input {95} to the S-box produces {2A}. The inverse S-box is constructed by applying the inverse of the transformation in Equation (5.1) followed by taking the multiplicative inverse in GF(2^8). The inverse transformation is:

$$b_i' = b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i$$

where byte d = {05}, or 00000101. We can depict this transformation as follows:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To see that InvSubBytes is the inverse of SubBytes, label the matrices in SubBytes and InvSubBytes as X and Y, respectively, and the vector versions of constants c and d as C and D, respectively. For some 8-bit vector B, Equation (5.2) becomes B' = XB ⊕C. We need to show that Y(XB ⊕C) ⊕D = B. Multiply out, we must show YXB ⊕YC ⊕D = B. This becomes

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus$$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}$$

We have demonstrated that YX equals the identity matrix, and the YC = D, so that YC ⊕D equals the null vector.

Rationale

The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input . In addition, the constant in Equation (5.1) was chosen so that the S-box has no fixed points [S-box(a) = a] and no "opposite fixed points" [S-box(a) = ā], where ā is the bitwise complement of a.

Of course, the S-box must be invertible, that is, IS-box[S-box(a)] = a. However, the S-box is not self-inverse in the sense that it is not true that S-box(a) = IS-box(a). For example, [S-box({95}) = {2A}, but IS-box({95}) = {AD}.

ShiftRows Transformation

Forward and Inverse Transformations

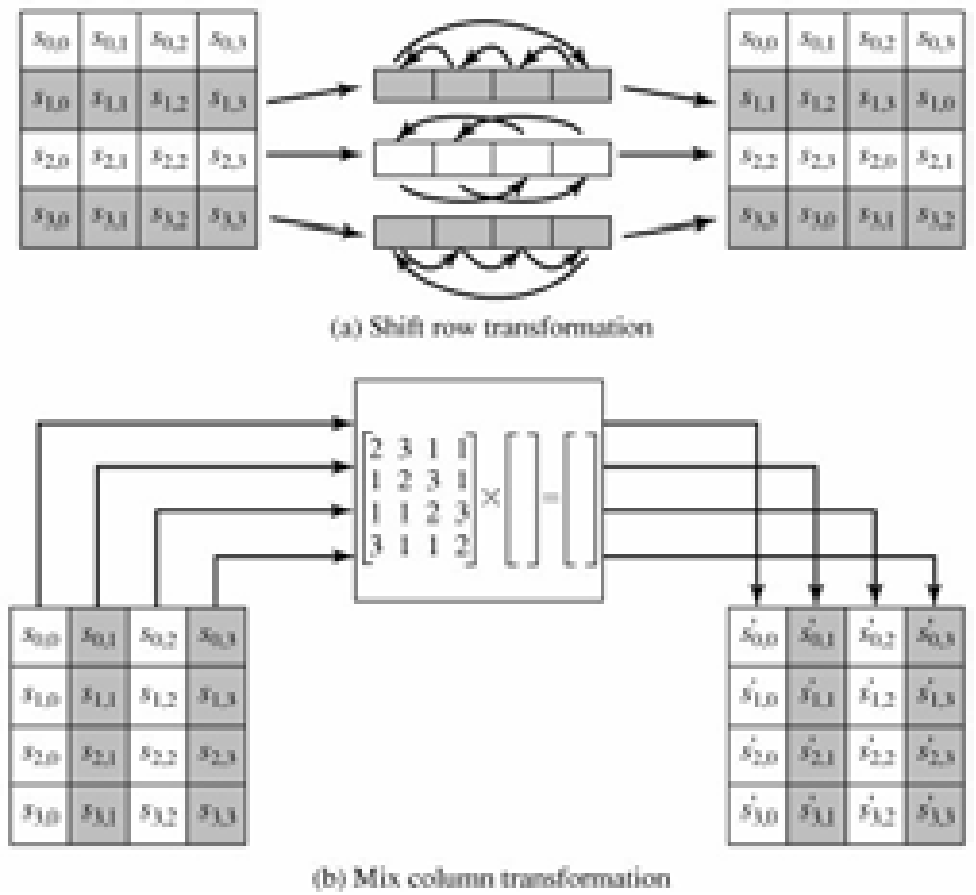
The forward shift row transformation, called ShiftRows, is depicted in Figure 5.5a. The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. The following is an example of ShiftRows:

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

→

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

Figure 5.5. AES Row and Column Operations



The inverse shift row transformation, called InvShiftRows, performs the circular shifts in the opposite direction for each of the last three rows, with a one-byte circular right shift for the second row, and so on.

Rationale

The shift row transformation is more substantial than it may first appear. This is because the State, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of State, and so on. Further, as will be seen, the round key is applied to State column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Also note that the transformation ensures

that the 4 bytes of one column are spread out to four different columns. Figure 5.3 illustrates the effect.

MixColumns Transformation

Forward and Inverse Transformations

The forward mix column transformation, called MixColumns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The transformation can be defined by the following matrix multiplication on State (Figure 5.5b):

Equation 5-3

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

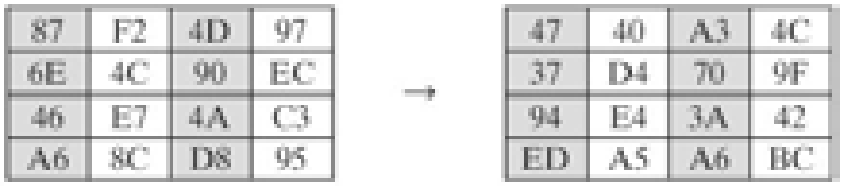
Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications are performed in GF(2^8). The MixColumns transformation on a single column j($0 \leq j \leq 3$) of State can be expressed as

We follow the convention of FIPS PUB 197 and use the symbol · to indicate multiplication over the finite field GF(2^8) and ⊕to indicate bitwise XOR, which corresponds to addition in GF(2^8).

Equation 5-4

$$\begin{aligned} s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\ s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j}) \end{aligned}$$

The following is an example of MixColumns:



87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

Let us verify the first column of this example, in GF(2^8), addition is the bitwise XOR operation and that multiplication can be performed according to the rule established in Equation (4.10). In particular, multiplication of a value by x (i.e., by {02}) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (0001 1011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, we need to show that

$$(\{02\} \cdot \{87\}) \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\}$$

$$\{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} = \{37\}$$

$$\{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) = \{94\}$$

$$(\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) = \{ED\}$$

For the first equation, we have {02} · {87} = (0000 1110) ⊕ (0001 1011) = (0001 0101); and {03} · {6E} = {6E} ⊕ ({02} · {6E}) = (0110 1110) ⊕ (1101 1100) = (1011 0010). Then

$\{02\} \cdot \{87\} = 0001\ 0101$

$\{03\} \cdot \{6E\} = 1011\ 0010$

$\{46\} = 0100\ 0110$

$\{A6\} = 1010\ 0110$

$0100\ 0111 = \{47\}$

The other equations can be similarly verified.

The inverse mix column transformation, called InvMixColumns, is defined by the following matrix multiplication:

Equation 5-5

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

It is not immediately clear that Equation (5.5) is the inverse of Equation (5.3). We need to show that:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

which is equivalent to showing that:

Equation 5-6

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of Equation (5.6), we need to show that:

$(\{0E\} \cdot \{02\}) \oplus \{0B\} \oplus \{0D\} \oplus (\{09\} \cdot \{03\}) = \{01\}$

$(\{09\} \cdot \{02\}) \oplus \{0E\} \oplus \{0B\} \oplus (\{0D\} \cdot \{03\}) = \{00\}$

$(\{0D\} \cdot \{02\}) \oplus \{09\} \oplus \{0E\} \oplus (\{0B\} \cdot \{03\}) = \{00\}$

$(\{0B\} \cdot \{02\}) \oplus \{0D\} \oplus \{09\} \oplus (\{0E\} \cdot \{03\}) = \{00\}$

For the first equation, we have {0E} · {02}) ⊕00011100; and {09} · {03} = {09} ⊕ ({09} · {02}) = 00001001 ⊕00010010 = 00011011. Then

{0E} · {02}	=	00011100
{0B}	=	00001011
{0D}	=	00001101
{09} · {03}	=	00011011
		00000001

The other equations can be similarly verified.

The AES document describes another way of characterizing the MixColumns transformation, which is in terms of polynomial arithmetic. In the standard, MixColumns is defined by considering each column of State to be a four-term polynomial with coefficients in GF(2^8). Each column is multiplied modulo ($x^4 + 1$) by the fixed polynomial a(x), given by

Equation 5-7

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

demonstrates that multiplication of each column of State by a(x) can be written as the matrix multiplication of Equation (5.3). Similarly, it can be seen that the transformation in Equation (5.5) corresponds to treating each column as a four-term polynomial and multiplying each column by b(x), given by

Equation 5-8

$$b(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$$

It can readily be shown that $b(x) = a^1 (x) \bmod (x^4 + 1)$.

Rationale

The coefficients of the matrix in Equation (5.3) are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits.

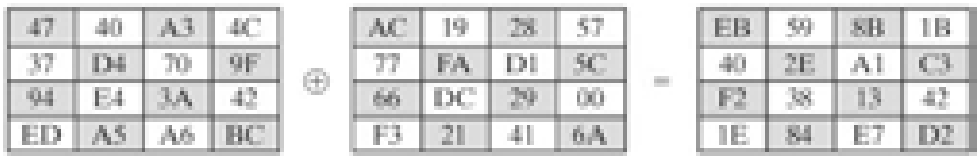
In addition, the choice of coefficients in MixColumns, which are all {01}, {02}, or {03}, was influenced by implementation considerations. As was discussed, multiplication by these coefficients involves at most a shift and an XOR. The coefficients in InvMixColumns are more formidable to implement. However, encryption was deemed more important than decryption for two reasons:

1. For the CFB and OFB cipher modes only encryption is used.
2. As with any block cipher, AES can be used to construct a message authentication code (Part Two), and for this only encryption is used.

AddRoundKey Transformation

Forward and Inverse Transformations

In the forward add round key transformation, called AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. As shown in Figure 5.4b, the operation is viewed as a columnwise operation between the 4 bytes of a State column and one word of the round key; it can also be viewed as a byte-level operation. The following is an example of AddRoundKey:



The first matrix is State, and the second matrix is the round key.

The inverse add round key transformation is identical to the forward add round key transformation, because the XOR operation is its own inverse.

Rationale

The add round key transformation is as simple as possible and affects every bit of State. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensure security.

AES Key Expansion

Key Expansion Algorithm

The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a 4-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The following pseudocode describes the expansion:

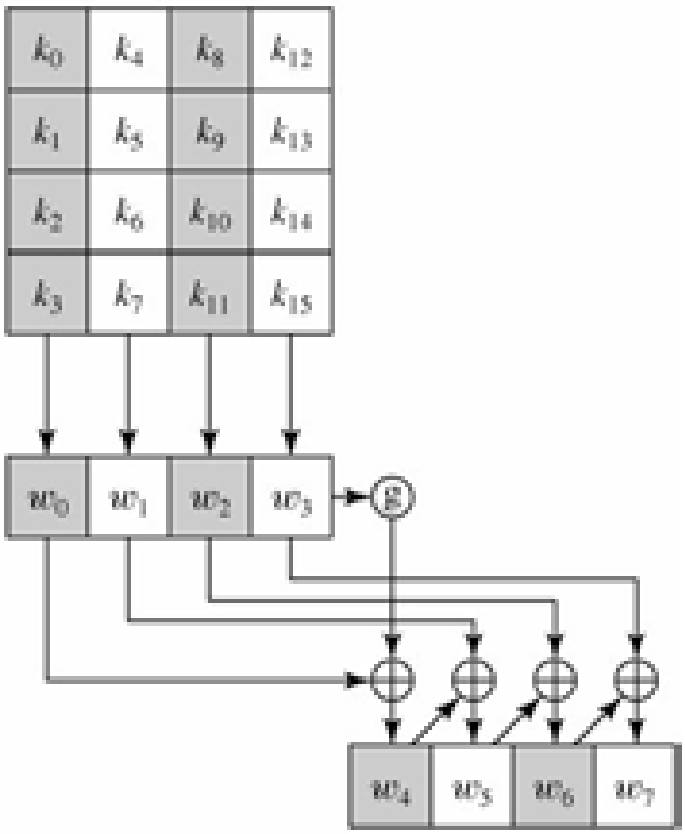
```
KeyExpansion (byte key[16], word w[44])
{
   word temp
   for (i = 0; i < 4; i++) w[i] = (key[4*i],
    key[4*i+1],
                              key[4*i+2],
                              key[4*i+3]);
   for (i = 4; i < 44; i++)
   {
    temp = w[i  1];
    if (i mod 4 = 0) temp = SubWord (RotWord (temp))
                        ⊕ Rcon[i/4];
    w[i] = w[i4] ≈temp
   }
}
```

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word w[i] depends on the immediately preceding word, w[i 1], and the word four positions back,w[i 4]. In three out of four cases, a simple XOR is used. For a word whose position in the w array is a

multiple of 4, a more complex function is used. Figure 5.6 illustrates the generation of the first eight words of the expanded key, using the symbol g to represent that complex function. The function g consists of the following subfunctions:

1. RotWord performs a one-byte circular left shift on a word. This means that an input word [b0, b1, b2, b3] is transformed into [b1, b2, b3, b0].
2. SubWord performs a byte substitution on each byte of its input word, using the S-box (Table 5.4a).
3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

Figure 5.6. AES Key Expansion



The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as Rcon[j] = (RC[j], 0, 0, 0), with RC[1] = 1,

RC[j] = 2 · RC[j - 1] and with multiplication defined over the field GF(2^8). The values of RC[j] in hexadecimal are

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

For example, suppose that the round key for round 8 is

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as follows:

i (decimal)	temp	After RotWord	After SubWord	Rcon (9)	After XOR with Rcon	w[i 4]	w[i] = temp ⊕w[i 4]
36	7F8D292F	8D292F7F	5DA515D2	1B000000	46A515D2	EAD27321	AC7766F3

Rationale

The Rijndael developers designed the expansion key algorithm to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the ways in which round keys are generated in different rounds:

- Knowledge of a part of the cipher key or round key does not enable calculation of many other round key bits
- An invertible transformation [i.e., knowledge of any Nk consecutive words of the Expanded Key enables regeneration the entire expanded key (Nk = key size in words)]
- Speed on a wide range of processors
- Usage of round constants to eliminate symmetries
- Diffusion of cipher key differences into the round keys; that is, each key bit affects many round key bits
- Enough nonlinearity to prohibit the full determination of round key differences from cipher key differences only
- Simplicity of description

The authors do not quantify the first point on the preceding list, but the idea is that if you know less than Nk consecutive words of either the cipher key or one of the round keys, then it is difficult to reconstruct the remaining unknown bits. The fewer bits one knows, the more difficult it is to do the reconstruction or to determine other bits in the key expansion.

Equivalent Inverse Cipher

As was mentioned, the AES decryption cipher is not identical to the encryption cipher (Figure 5.1). That is, the sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm. The equivalent version has the same sequence of transformations as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

Two separate changes are needed to bring the decryption structure in line with the encryption structure. An encryption round has the structure SubBytes, ShiftRows, MixColumns, AddRoundKey. The standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Thus, the first two stages of the decryption round need to be interchanged, and the second two stages of the decryption round need to be interchanged.

Interchanging InvShiftRows and InvSubBytes

InvShiftRows affects the sequence of bytes in State but does not alter byte contents and does not depend on byte contents to perform its transformation. InvSubBytes affects the contents of bytes in State but does not alter byte sequence and does not depend on byte sequence to perform its transformation. Thus, these two operations commute and can be interchanged. For a given State S_i,

InvShiftRows [InvSubBytes (S_i)] = InvSubBytes [InvShiftRows (S_i)]

Interchanging AddRoundKey and InvMixColumns

The transformations AddRoundKey and InvMixColumns do not alter the sequence of bytes in State. If we view the key as a sequence of words, then both AddRoundKey and InvMixColumns operate on State one column at a time. These two operations are linear with respect to the column input. That is, for a given State S_i and a given round key w_j:

$$\text{InvMixColumns}\ (S_i \oplus w_j) = [\text{InvMixColumns}\ (S_i)] \oplus [\text{InvMixColumns}\ (w_j)]$$

To see this, suppose that the first column of State S_i is the sequence (y_0, y_1, y_2, y_3) and the first column of the round key w_j is (k_0, k_1, k_2, k_3). Then we need to show that

Let us demonstrate that for the first column entry. We need to show that:

$$[\{0E\} \cdot (y_0 \oplus k_0)] \oplus [\{0B\} \cdot (y_1 \oplus k_1)] \oplus [\{0D\} \cdot (y_2 \oplus k_2)] \oplus [\{09\} \cdot (y_3 \oplus k_3)]$$

$$= [\{0E\} \cdot y_0] \oplus [\{0B\} \cdot y_1] \oplus [\{0D\} \cdot y_2] \oplus [\{09\} \cdot y_3]$$

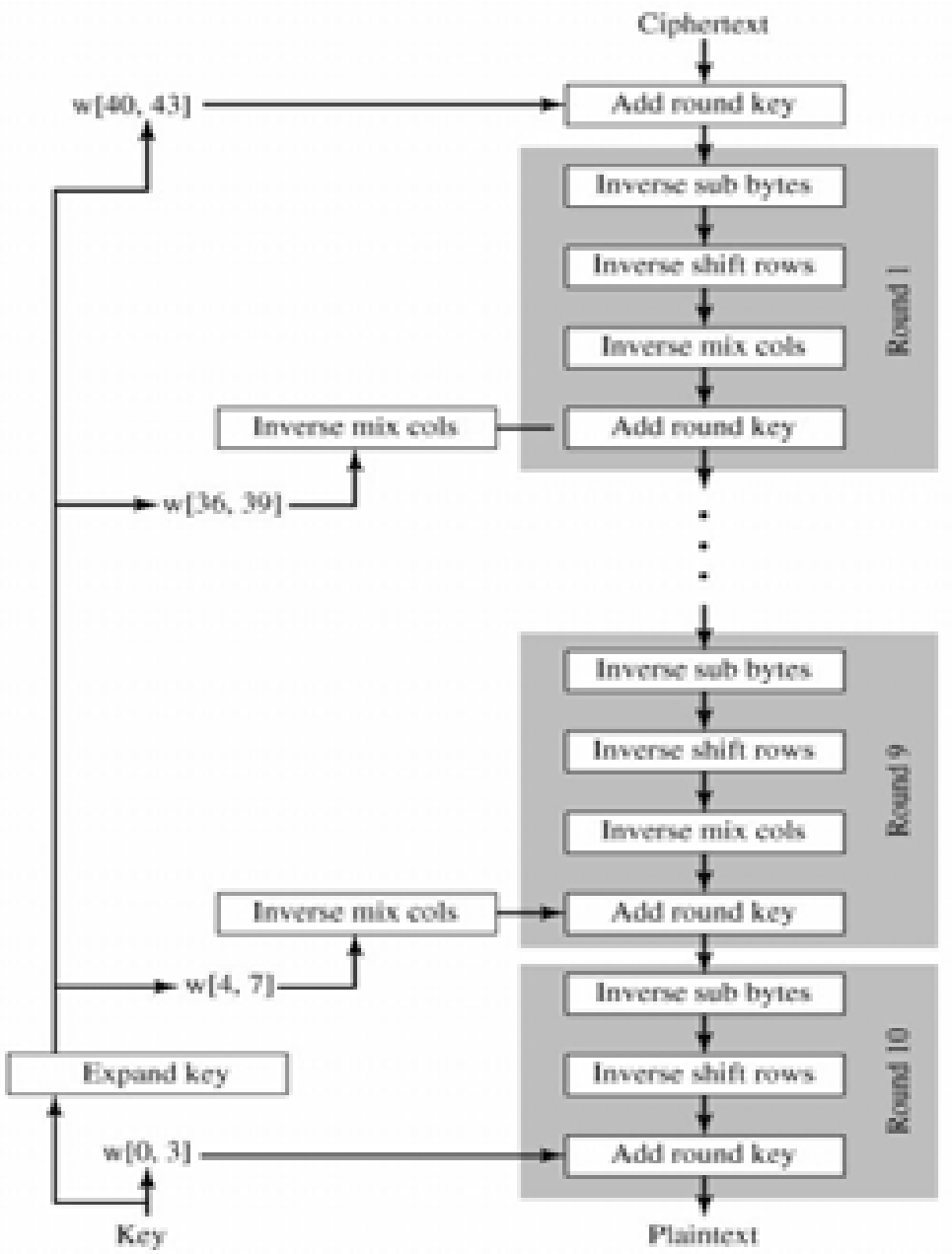
$$\oplus [[\{0E\} \cdot k_0] \oplus] [\{0B\} \cdot k_1] \oplus [\{0D\} \cdot k_2] \oplus [\{09\} \cdot k_3]$$

This equation is valid by inspection. Thus, we can interchange AddRoundKey and InvMixColumns, provided that we first apply InvMixColumns to the round key. Note that we do not need to apply InvMixColumns to the round key for the input to the first AddRoundKey transformation (preceding the first round) nor to the last AddRoundKey transformation (in round 10). This is because these two AddRoundKey transformations are not interchanged with InvMixColumns to produce the equivalent decryption algorithm.

Figure 5.7 illustrates the equivalent decryption algorithm.

Figure 5.7. Equivalent Inverse Cipher

(This item is displayed on page 158 in the print version)



Implementation Aspects

The Rijndael proposal provides some suggestions for efficient implementation on 8-bit processors, typical for current smart cards, and on 32-bit processors, typical for PCs.

8-Bit Processor

AES can be implemented very efficiently on an 8-bit processor. AddRoundKey is a bytewise XOR operation. ShiftRows is a simple byte shifting operation. SubBytes operates at the byte level and only requires a table of 256 bytes.

The transformation MixColumns requires matrix multiplication in the field GF(2^8), which means that all operations are carried out on bytes. MixColumns only requires multiplication by {02} and {03}, which, as we have seen, involved simple shifts, conditional XORs, and XORs. This can be implemented in a more efficient way that eliminates the shifts and conditional XORs. Equation Set (5.4) shows the equations for the MixColumns transformation on a single column. Using the identity {03} · x = ({02} · x) ⊕x, we can rewrite Equation Set (5.4) as follows:

Equation 5-9

$$
\begin{aligned}
Tmp &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\
s'_{0,j} &= s_{0,j} \oplus Tmp \oplus [2 \cdot (s_{0,j} \oplus s_{1,j})] \\
s'_{1,j} &= s_{1,j} \oplus Tmp \oplus [2 \cdot (s_{1,j} \oplus s_{2,j})] \\
s'_{2,j} &= s_{2,j} \oplus Tmp \oplus [2 \cdot (s_{2,j} \oplus s_{3,j})] \\
s'_{3,j} &= s_{3,j} \oplus Tmp \oplus [2 \cdot (s_{3,j} \oplus s_{0,j})]
\end{aligned}
$$

Equation Set (5.9) is verified by expanding and eliminating terms.

The multiplication by {02} involves a shift and a conditional XOR. Such an implementation may be vulnerable to a timing attack of the sort described in Section 3.4. To counter this attack and to increase processing efficiency at the cost of some storage, the multiplication can be replaced by a table lookup. Define the 256-byte table X2, such that X2[i] = {02} · i. Then Equation Set (5.9) can be rewritten as

Tmp = s_o, j $\oplus s_1$, j $\oplus$s2, j $\oplus$s3, j

s'_0, j = s_0, j $\oplus$Tmp $\oplus$X2[s_o, j $\oplus s_1$, j]

s'_1, c = s_1, j $\oplus$Tmp $\oplus$X2[s_1, j $\oplus s_2$, j]

s'_2, c = s_2, j $\oplus$Tmp $\oplus$X2[s_2, j $\oplus s_3$, j]

s'_3, j = s_3, j $\oplus$Tmp $\oplus$X2[s_3, j $\oplus s_0$, j]

32-Bit Processor

The implementation described in the preceding subsection uses only 8-bit operations. For a 32-bit processor, a more efficient implementation can be achieved if operations are defined on 32-bit words. To show this, we first define the four transformations of a round in algebraic form. Suppose we begin with a State matrix consisting of elements $a_{i,j}$ and a round key matrix consisting of elements $k_{i,j}$. Then the transformations can be expressed as follows:

SubBytes	$b_{i,j} = S[a_{i,j}]$

ShiftRows

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$$

MixColumns

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$$

AddRoundKey

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

In the ShiftRows equation, the column indices are taken mod 4. We can combine all of these expressions into a single equation:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

$$= \left(\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \cdot S[a_{0,j}] \right) \oplus \left(\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \cdot S[a_{1,j-1}] \right) \oplus \left(\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \cdot S[a_{2,j-2}] \right)$$

$$\oplus \left(\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \cdot S[a_{3,j-3}] \right) \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

In the second equation, we are expressing the matrix multiplication as a linear combination of vectors. We define four 256-word (1024-byte) tables as follows:

$T_0[x] = \left(\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \cdot S[x] \right)$	$T_1[x] = \left(\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \cdot S[x] \right)$	$T_2[x] = \left(\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \cdot S[x] \right)$	$T_3[x] = \left(\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \cdot S[x] \right)$

Thus, each table takes as input a byte value and produces a column vector (a 32-bit word) that is a function of the S-box entry for that byte value. These tables can be calculated in advance.

We can define a round function operating on a column in the following fashion:

$$\begin{bmatrix} s'_{0,j} \\ s'_{1,j} \\ s'_{2,j} \\ s'_{3,j} \end{bmatrix} = T_0[s_{0,j}] \oplus T_1[s_{1,j-1}] \oplus T_2[s_{2,j-2}] \oplus T_3[s_{3,j-3}] \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$$

As a result, an implementation based on the preceding equation requires only four table lookups and four XORs per column per round, plus 4 Kbytes to store the table. The developers of Rijndael believe that this compact, efficient implementation was probably one of the most important factors in the selection of Rijndael for AES.

The Chinese Remainder Theorem

One of the most useful results of number theory is the Chinese remainder theorem (CRT). In essence, the CRT says it is possible to reconstruct integers in a certain range from their residues modulo a set of pairwise relatively prime moduli.

The CRT is so called because it is believed to have been discovered by the Chinese mathematician Sun-Tsu in around 100 A.D.

The 10 integers in Z_{10}, that is the integers 0 through 9, can be reconstructed from their two residues modulo 2 and 5 (the relatively prime factors of 10). Say the known residues of a decimal digit x are $r_2 = 0$ and $r_5 = 3$; that is, x mod 2 =0 and x mod 5 = 3. Therefore, x is an even integer in Z_{10} whose remainder, on division by 5, is 3. The unique solution is x = 8.

The CRT can be stated in several ways. We present here a formulation that is most useful from the point of view of this text. An alternative formulation is explored in Problem 8.17. Let

$$M = \prod_{i=1}^{k} m_i$$

where the m_i are pairwise relatively prime; that is, $\gcd(m_i, m_j) = 1$ for $1 \leq i, j \leq k$, and $i \neq j$. We can represent any integer A in Z^M by a k-tuple whose elements are in Z_{mi} using the following correspondence:

Equation 8-7

$$A \leftrightarrow (a_1, a_2, \ldots, a_k)$$

where A $\in Z_M$, $a_i \in Z_{mi}$ and $a_i = A \bmod m_i$ for $1 \leq i \leq k$. The CRT makes two assertions.

1. The mapping of Equation (8.7) is a one-to-one correspondence (called a bijection) between Z_M and the Cartesian product $Z_{m1} \times Z_{m2} \times \ldots \times Z_{mk}$. That is, for every integer A such that $0 \leq A < M$ there is a unique k-tuple $(a_1, a_2, \ldots, a_k)$ with $0 \leq a_i < m_i$ that represents it, and for every such k-tuple $(a_1, a_2, \ldots, a_k)$ there is a unique integer A in Z_M.
2. Operations performed on the elements of Z_M can be equivalently performed on the corresponding k-tuples by performing the operation independently in each coordinate position in the appropriate system.

Let us demonstrate the first assertion. The transformation from A to $(a_1, a_1, \ldots, a_k)$ is obviously unique; that is, each a_i is uniquely calculated as $a_i = A \bmod m_i$. Computing A from $(a_1, a_1, \ldots, a_k)$ can be done as follows. Let $M_i = M/m_i$ for $1 \leq i \leq k$. Note that $M_i = m_1 \times m_2 \times \ldots \times m_{i-1} \times m_{i+1} \times \ldots \times m_k$ so that $M_i \equiv 0 \pmod{m_j}$ for all $j \neq i$. Then let

Equation 8-8

$$c_i = M_i \times (M_i^{-1} \bmod m_i) \quad \text{for } 1 \leq i \leq k$$

By the definition of M_i it is relatively prime to m_i and therefore has a unique multiplicative inverse mod m_i So Equation (8.8) is well defined and produces a unique value c_i. We can now compute:

Equation 8-9

$$A = \left(\sum_{i=1}^{k} a_i c_i \right) \pmod{M}$$

To show that the value of A produced by Equation (8.9) is correct, we must show that $a_i = A \bmod m_i$ for $1 \leq i \leq k$. Note that $c_j \equiv M_j \equiv 0 (\bmod m_i)$ if $j \neq i$ and that $c_i \equiv 1 (\bmod m_i)$. It follows that $a_i = A \bmod m_i$.

The second assertion of the CRT, concerning arithmetic operations, follows from the rules for modular arithmetic. That is, the second assertion can be stated as follows: If

$$A \longleftrightarrow (a_1, a_2, \ldots, a_k)$$
$$B \longleftrightarrow (b_1, b_2, \ldots, b_k)$$

then

$$(A + B) \bmod M \longleftrightarrow ((a_1 + b_1) \bmod m_1, \ldots, (a_k + b_k) \bmod m_k)$$
$$(A - B) \bmod M \longleftrightarrow ((a_1 - b_1) \bmod m_1, \ldots, (a_k - b_k) \bmod m_k)$$
$$(A \times B) \bmod M \longleftrightarrow ((a_1 \times b_1) \bmod m_1, \ldots, (a_k \times b_k) \bmod m_k)$$

One of the useful features of the Chinese remainder theorem is that it provides a way to manipulate (potentially very large) numbers mod M in terms of tuples of smaller numbers. This can be useful when M is 150 digits or more. However, note that it is necessary to know beforehand the factorization of M.

To represent 973 mod 1813 as a pair of numbers mod 37 and 49, define[8]

$m_1 = 37$

$m_2 = 49$

$M = 1813$

$A = 973$

We also have $M_1 = 49$ and $M_2 = 37$. Using the extended Euclidean algorithm, we compute $M^{-1}{}_1 = 34 \bmod m_1$ and $M^{-1}{}_2 = 4 \bmod m_2$. (Note that we only need to compute each M_i and each $M^{-1}{}_i$ once.) Taking residues modulo 37 and 49, our representation of 973 is (11, 42), because 973 mod 37 = 11 and 973 mod 49 = 42.

Now suppose we want to add 678 to 973. What do we do to (11, 42)? First we compute (678) (678 mod 37, 678 mod 49) = (12, 41). Then we add the tuples element-wise and reduce (11 + 12 mod 37, 42 + 41 mod 49) = (23, 34). To verify that this has the correct effect, we compute

$(23,34) \longleftrightarrow a_1M_1M^{-1}{}_1 + a_2M_2M^{-1}{}_2 \bmod M$

$= [(23)(49)(34) + (34)(37)(4)] \bmod 1813$

$= 43350 \bmod 1813$

$= 1651$

and check that it is equal to (973 + 678) mod 1813 = 1651. Remember that in the above derivation, $M^{-1}{}_1$ is the multiplicative inverse of M_1 modulo m_1, and $M^{-1}{}_2$ is the multiplicative inverse of M_2 modulo m_2.

Suppose we want to multiply 1651 (mod 1813) by 73. We multiply (23, 34) by 73 and reduce to get (23 x 73 mod 37, 34 x 73 mod 49) = (14,32). It is easily verified that

$(32,14) \longleftrightarrow [(14)(49)(34) + (32)(37)(4)] \bmod 1813$

$= 865$

$= 1651 \times 73 \bmod 1813$

The Data Encryption Standard

The most widely used encryption scheme is based on the Data Encryption Standard (DES) adopted in 1977 by the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as Federal Information Processing Standard 46 (FIPS PUB 46). The algorithm itself is referred to as the Data Encryption Algorithm (DEA). For DES, data are encrypted in 64-bit blocks using a 56-bit key. The algorithm transforms 64-bit input in a series of steps into a 64-bit output. The same steps, with the same key, are used to reverse the encryption.

The terminology is a bit confusing. Until recently, the terms DES and DEA could be used interchangeably. However, the most recent edition of the DES document includes a specification of the DEA described here plus the triple DEA (TDEA) described in Chapter 6. Both DEA and TDEA are part of the Data Encryption Standard. Further, until the recent adoption of the official term TDEA, the triple DEA algorithm was typically referred to as triple DES and written as 3DES. For the sake of convenience, we use the term 3DES.

The DES enjoys widespread use. It has also been the subject of much controversy concerning how secure the DES is. To appreciate the nature of the controversy, let us quickly review the history of the DES.

In the late 1960s, IBM set up a research project in computer cryptography led by Horst Feistel. The project concluded in 1971 with the development of an algorithm with the designation LUCIFER . which was sold to Lloyd's of London for use in a cash-dispensing system, also developed by IBM. LUCIFER is a Feistel block cipher that operates on blocks of 64 bits, using a key size of 128 bits. Because of the promising results produced by the LUCIFER project, IBM embarked on an effort to develop a marketable commercial encryption product that ideally could be implemented on a single chip. The effort was headed by Walter Tuchman and Carl Meyer, and it involved not only IBM researchers but also outside consultants and technical advice from NSA. The outcome of this effort was a refined version of LUCIFER that was more resistant to cryptanalysis but that had a reduced key size of 56 bits, to fit on a single chip.

In 1973, the National Bureau of Standards (NBS) issued a request for proposals for a national cipher standard. IBM submitted the results of its Tuchman-Meyer project. This was by far the best algorithm proposed and was adopted in 1977 as the Data Encryption Standard.

Before its adoption as a standard, the proposed DES was subjected to intense criticism, which has not subsided to this day. Two areas drew the critics' fire. First, the key length in IBM's original LUCIFER algorithm was 128 bits, but that of the proposed system was only 56 bits, an enormous reduction in key size of 72 bits. Critics feared that this key length was too short to withstand brute-force attacks. The second area of concern was that the design criteria for the internal structure of DES, the S-boxes, were classified. Thus, users could not be sure that the internal structure of DES was free of any hidden weak points that would enable NSA to decipher messages without benefit of the key. Subsequent events, particularly the recent work on differential cryptanalysis, seem to indicate that DES has a very strong internal structure. Furthermore, according to IBM participants, the only changes that were made to the proposal were changes to the S-boxes, suggested by NSA, that removed vulnerabilities identified in the course of the evaluation process.

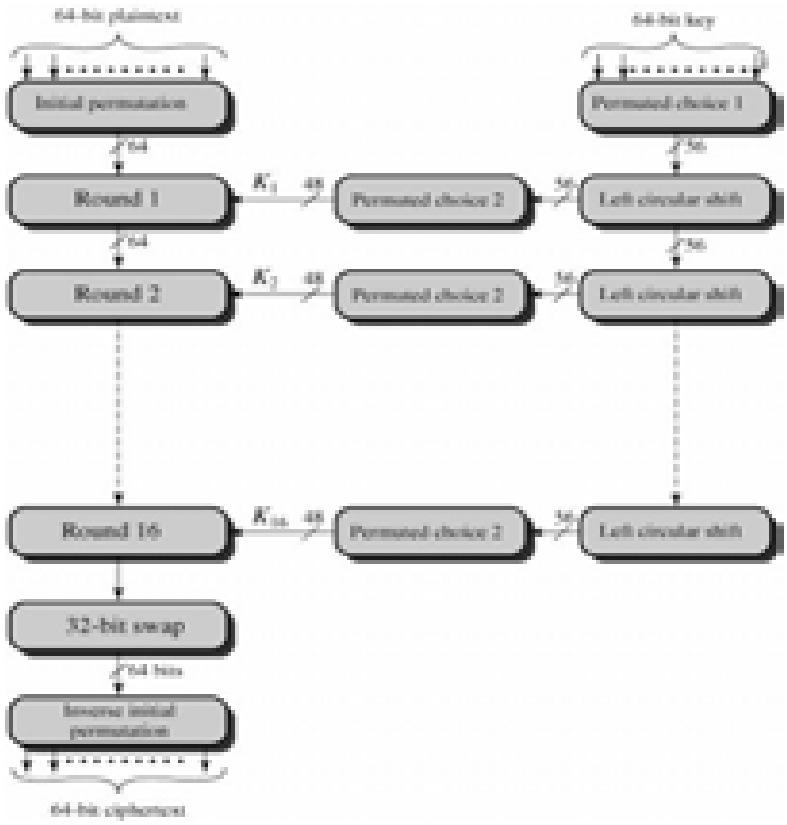
Whatever the merits of the case, DES has flourished and is widely used, especially in financial applications. In 1994, NIST reaffirmed DES for federal use for another five years; NIST recommended the use of DES for applications other than the protection of classified information. In 1999, NIST issued a new version of its standard (FIPS PUB 46-3) that indicated that DES should only be used for legacy systems and that triple DES (which in essence involves repeating the DES algorithm three times on the plaintext using two or three different keys to produce the ciphertext) be used.

DES Encryption

The overall scheme for DES encryption is illustrated in Figure 3.4. As with any encryption scheme, there are two inputs to the encryption function: the plaintext to be encrypted and the key. In this case, the plaintext must be 64 bits in length and the key is 56 bits in length.

Actually, the function expects a 64-bit key as input. However, only 56 of these bits are ever used; the other 8 bits can be used as parity bits or simply set arbitrarily.

Figure 3.4. General Depiction of DES Encryption Algorithm



Looking at the left-hand side of the figure, we can see that the processing of the plaintext proceeds in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that rearranges the bits to produce the permuted input. This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last (sixteenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation (IP^{-1}) that is the inverse of the initial permutation function, to produce the 64-bit ciphertext. With the exception of the initial and final permutations, DES has the exact structure of a Feistel cipher, as shown in Figure 3.2.

The right-hand portion of Figure 3.4 shows the way in which the 56-bit key is used. Initially, the key is passed through a permutation function. Then, for each of the 16 rounds, a *subkey* (K_i) is produced by the combination of a left circular shift and a permutation. The permutation function is the same for

each round, but a different subkey is produced because of the repeated shifts of the key bits.

Initial Permutation

The initial permutation and its inverse are defined by tables, as shown in Tables 3.2a and 3.2b, respectively. The tables are to be interpreted as follows. The input to a table consists of 64 bits numbered from 1 to 64. The 64 entries in the permutation table contain a permutation of the numbers from 1 to 64. Each entry in the permutation table indicates the position of a numbered input bit in the output, which also consists of 64 bits.

Table 3.2. Permutation Tables for DES

(This item is displayed on page 76 in the print version)

(a) Initial Permutation (IP)

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

(b) Inverse Initial Permutation (IP^{1})

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

Table 3.2. Permutation Tables for DES

(This item is displayed on page 76 in the print version)

(a) Initial Permutation (IP)

(c) Expansion Permutation (E)

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

(d) Permutation Function (P)

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

To see that these two permutation functions are indeed the inverse of each other, consider the following 64-bit input M:

$$
\begin{array}{cccccccc}
M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 \\
M_9 & M_{10} & M_{11} & M_{12} & M_{13} & M_{14} & M_{15} & M_{16} \\
M_{17} & M_{18} & M_{19} & M_{20} & M_{21} & M_{22} & M_{23} & M_{24} \\
M_{25} & M_{26} & M_{27} & M_{28} & M_{29} & M_{30} & M_{31} & M_{32} \\
M_{33} & M_{34} & M_{35} & M_{36} & M_{37} & M_{38} & M_{39} & M_{40} \\
M_{41} & M_{42} & M_{43} & M_{44} & M_{45} & M_{46} & M_{47} & M_{48} \\
M_{49} & M_{50} & M_{51} & M_{52} & M_{53} & M_{54} & M_{55} & M_{56} \\
M_{57} & M_{58} & M_{59} & M_{60} & M_{61} & M_{62} & M_{63} & M_{64}
\end{array}
$$

where M_i is a binary digit. Then the permutation X = IP(M) is as follows:

M_{58}	M_{50}	M_{42}	M_{34}	M_{26}	M_{18}	M_{10}	M_2
M_{60}	M_{52}	M_{44}	M_{36}	M_{28}	M_{20}	M_{12}	M_4
M_{62}	M_{54}	M_{46}	M_{38}	M_{30}	M_{22}	M_{14}	M_6
M_{64}	M_{56}	M_{48}	M_{40}	M_{32}	M_{24}	M_{16}	M_8
M_{57}	M_{49}	M_{41}	M_{33}	M_{25}	M_{17}	M_9	M_1
M_{59}	M_{51}	M_{43}	M_{35}	M_{27}	M_{19}	M_{11}	M_3
M_{61}	M_{53}	M_{45}	M_{37}	M_{29}	M_{21}	M_{13}	M_5
M_{63}	M_{55}	M_{47}	M_{39}	M_{31}	M_{23}	M_{15}	M_7

If we then take the inverse permutation $Y = IP^{-1}(X) = IP^{-1}(IP(M))$, it can be seen that the original ordering of the bits is restored.

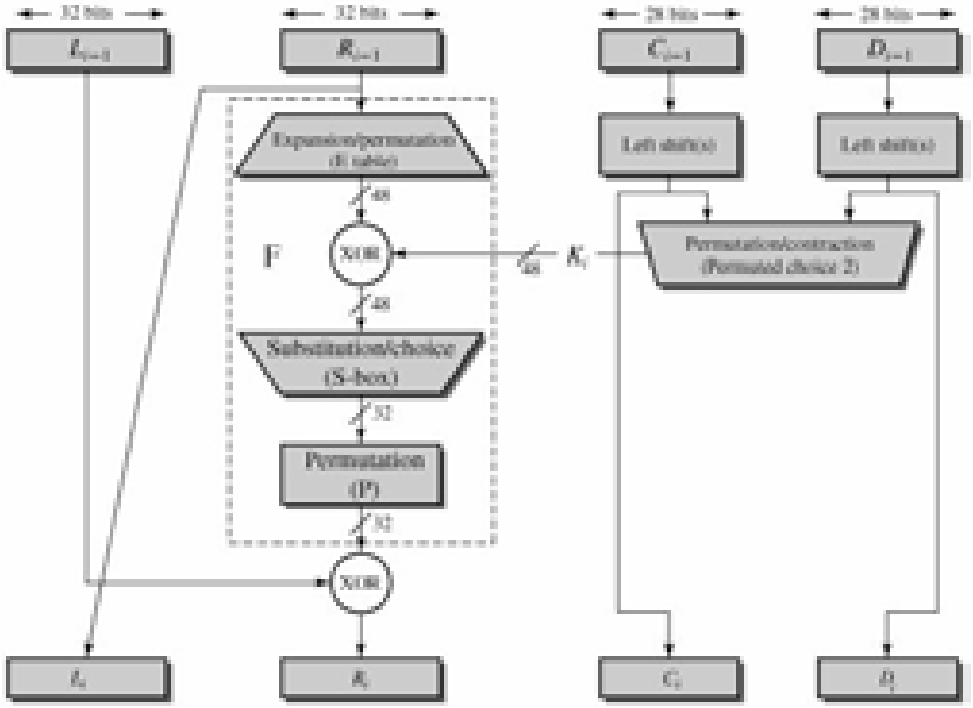
Details of Single Round

Figure 3.5 shows the internal structure of a single round. Again, begin by focusing on the left-hand side of the diagram. The left and right halves of each 64-bit intermediate value are treated as separate 32-bit quantities, labeled L (left) and R (right). As in any classic Feistel cipher, the overall processing at each round can be summarized in the following formulas:

$L_i = R_{i-1}$

$R_i = L_{i-1} \times F(R_{i-1}, K_i)$

Figure 3.5. Single Round of DES Algorithm



The round key K_i is 48 bits. The R input is 32 bits. This R input is first expanded to 48 bits by using a table that defines a permutation plus an expansion that involves duplication of 16 of the R bits (Table 3.2c). The resulting 48 bits are XORed with K_i. This 48-bit result passes through a substitution function that produces a 32-bit output, which is permuted as defined by Table 3.2d.

The role of the S-boxes in the function F is illustrated in Figure 3.6. The substitution consists of a set of eight S-boxes, each of which accepts 6 bits as input and produces 4 bits as output. These transformations are defined in Table 3.3, which is interpreted as follows: The first and last bits of the input to box S_i form a 2-bit binary number to select one of four substitutions defined by the four rows in the table for S_i. The middle four bits select one of the sixteen columns. The decimal value in the cell selected by the row and column is then converted to its 4-bit representation to produce the output. For example, in S_1 for input 011001, the row is 01 (row 1) and the column is 1100 (column 12). The value in row 1, column 12 is 9, so the output is 1001.

Figure 3.6. Calculation of F(R, K)

(This item is displayed on page 78 in the print version)

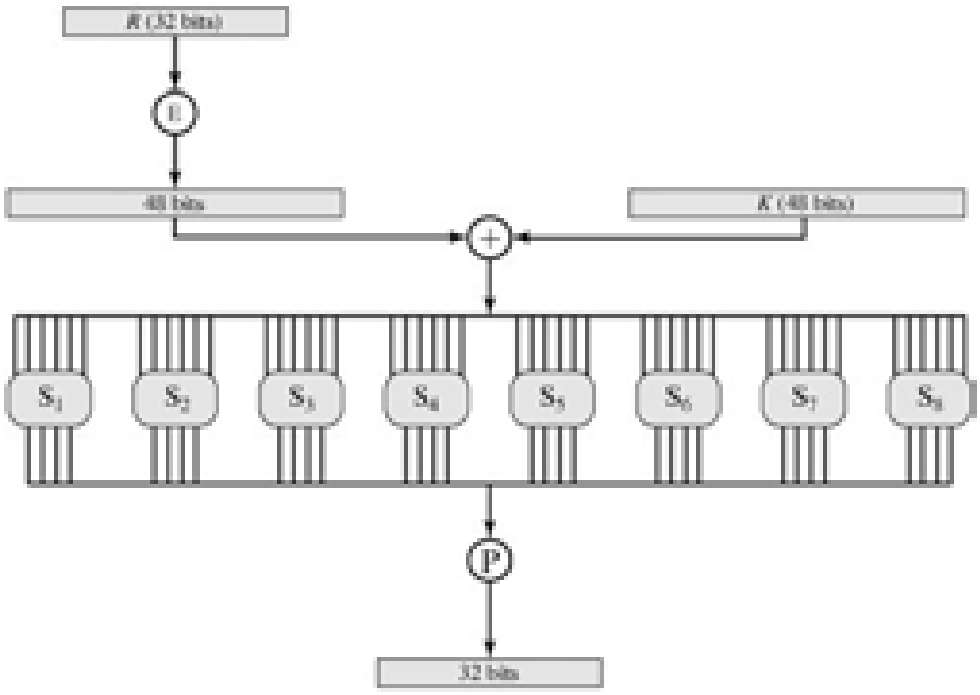


Table 3.3. Definition of DES S-Boxes

(This item is displayed on page 79 in the print version)

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S_3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S_5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S_7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S_8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Each row of an S-box defines a general reversible substitution. Figure 3.1 may be useful in understanding the mapping. The figure shows the substitution for row 0 of box S_1.

The operation of the S-boxes is worth further comment. Ignore for the moment the contribution of the key (K_i). If you examine the expansion table, you see that the 32 bits of input are split into groups of 4 bits, and then become groups of 6 bits by taking the outer bits from the two adjacent groups. For example, if part of the input word is

... efgh ijkl mnop ...

this becomes

... defghi hijklm lmnopq ...

The outer two bits of each group select one of four possible substitutions (one row of an S-box). Then a 4-bit output value is substituted for the particular 4-bit input (the middle four input bits). The 32-bit output from the eight S-boxes is then permuted, so that on the next round the output from each S-box immediately affects as many others as possible.

Key Generation

Returning to Figures 3.4 and 3.5, we see that a 64-bit key is used as input to the algorithm. The bits of the key are numbered from 1 through 64; every eighth bit is ignored, as indicated by the lack of shading in Table 3.4a. The key is first subjected to a permutation governed by a table labeled Permuted Choice One (Table 3.4b). The resulting 56-bit key is then treated as two 28-bit quantities, labeled C_0 and D_0. At each round, C_{i-1} and D_{i-1} are separately subjected to a circular left shift, or rotation, of 1 or 2 bits, as governed by Table 3.4d. These shifted values serve as input to the next round. They also serve as input to Permuted Choice Two (Table 3.4c), which produces a 48-bit output that serves as input to the function $F(R_{i-1}, K_i)$.

Table 3.4. DES Key Schedule Calculation

(a) Input Key

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

(b) Permuted Choice One (PC-1)

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

(c) Permuted Choice Two (PC-2)

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

(d) Schedule of Left Shifts

Round number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

DES Decryption

As with any Feistel cipher, decryption uses the same algorithm as encryption, except that the application of the subkeys is reversed.

The Avalanche Effect

A desirable property of any encryption algorithm is that a small change in either the plaintext or the key should produce a significant change in the ciphertext. In particular, a change in one bit of the plaintext or one bit of the key should produce a change in many bits of the ciphertext. If the change were small, this might provide a way to reduce the size of the plaintext or key space to be searched.

DES exhibits a strong avalanche effect. Table 3.5 shows some results taken from. In Table 3.5a, two plaintexts that differ by one bit were used:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

with the key

0000001 1001011 0100100 1100010 0011100 0011000 0011100 0110010

Table 3.5. Avalanche Effect in DES

(a) Change in Plaintext		(b) Change in Key	
Round	Number of bits that differ	Round	Number of bits that differ
0	1	0	0
1	6	1	2
2	21	2	14
3	35	3	28
4	39	4	32
5	34	5	30
6	32	6	32

Table 3.5. Avalanche Effect in DES			
(a) Change in Plaintext		(b) Change in Key	
Round	Number of bits that differ	Round	Number of bits that differ
7	31	7	35
8	29	8	34
9	42	9	40
10	44	10	38
11	32	11	31
12	30	12	33
13	30	13	28
14	26	14	26
15	29	15	34
16	34	16	35

The Table 3.5a shows that after just three rounds, 21 bits differ between the two blocks. On completion, the two ciphertexts differ in 34 bit positions.

Table 3.5b shows a similar test in which a single plaintext is input:

01101000 10000101 00101111 01111010 00010011 01110110 11101011 10100100

with two keys that differ in only one bit position:

1110010 1111011 1101111 0011000 0011101 0000100 0110001 11011100

0110010 1111011 1101111 0011000 0011101 0000100 0110001 11011100

Again, the results show that about half of the bits in the ciphertext differ and that the avalanche effect is pronounced after just a few rounds.

The Euclidean Algorithm

One of the basic techniques of number theory is the Euclidean algorithm, which is a simple procedure for determining the greatest common divisor of two positive integers.

Greatest Common Divisor

Recall that nonzero b is defined to be a divisor of a if a = mb for some m, where a, b, and m are integers. We will use the notation gcd(a, b) to mean the greatest common divisor of a and b. The positive integer c is said to be the greatest common divisor of a and b if

1. c is a divisor of a and of b;
2. any divisor of a and b is a divisor of c.

An equivalent definition is the following:

gcd(a, b) = max[k, such that k|a and k|b]

Because we require that the greatest common divisor be positive, gcd(a, b) = gcd(a, b) = gcd(a, b) = gcd(a, b). In general, gcd(a, b) = gcd(|a|, |b|).

gcd(60, 24) = gcd(60, 24) = 12

Also, because all nonzero integers divide 0, we have gcd(a, 0) = |a|.

We stated that two integers a and b are relatively prime if their only common positive integer factor is 1. This is equivalent to saying that a and b are relatively prime if gcd(a, b) = 1.

8 and 15 are relatively prime because the positive divisors of 8 are 1, 2, 4, and 8, and the positive divisors of 15 are 1, 3, 5, and 15, so 1 is the only integer on both lists.

Finding the Greatest Common Divisor

The Euclidean algorithm is based on the following theorem: For any nonnegative integer a and any positive integer b,

Equation 4-4

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

gcd(55, 22) = gcd(22, 55 mod 22) = gcd(22, 11) = 11

To see that Equation (4.4) works, let d = gcd(a, b). Then, by the definition of gcd, d|a and d|b. For any positive integer b, a can be expressed in the form

a = kb + r≡≡ r (mod b)

a mod b = r

with k, r integers. Therefore, (a mod b) = a kb for some integer k. But because d|b, it also divides kb. We also have d|a. Therefore, d|(a mod b). This shows that d is a common divisor of b and (a mod b). Conversely, if d is a common divisor of b and (a mod b), then d|kb and thus d|[kb + (a mod b)], which is equivalent to d|a. Thus, the set of common divisors of a and b is equal to the set of common divisors of b and (a mod b). Therefore, the gcd of one pair is the same as the gcd of the other pair, proving the theorem.

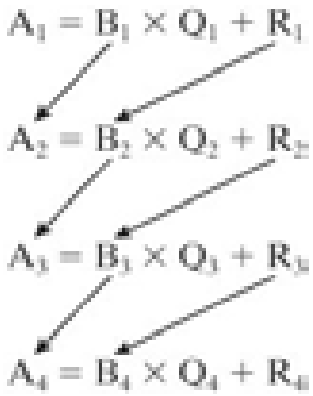
Equation (4.4) can be used repetitively to determine the greatest common divisor.

gcd(18, 12) = gcd(12, 6) = gcd(6, 0) = 6
gcd(11, 10) = gcd(10, 1) = gcd(1, 0) = 1

The Euclidean algorithm makes repeated use of Equation (4.4) to determine the greatest common divisor, as follows. The algorithm assumes a > b > 0. It is acceptable to restrict the algorithm to positive integers because gcd(a, b) = gcd(|a|, |b|).

EUCLID(a, b)
1. A ← a; B ← b
2. if B = 0 return A = gcd(a, b)
3. R = A mod B
4. A ← B
5. B ← R
6. goto 2

The algorithm has the following progression:



To find gcd(1970, 1066)		
1970	= 1 x 1066 + 904	gcd(1066, 904)
1066	= 1 x 904 + 162	gcd(904, 162)
904	= 5 x 162 + 94	gcd(162, 94)
162	= 1 x 94 + 68	gcd(94, 68)
94	= 1 x 68 + 26	gcd(68, 26)
68	= 2 x 26 + 16	gcd(26, 16)
26	= 1 x 16 + 10	gcd(16, 10)
16	= 1 x 10 + 6	gcd(10, 6)
10	= 1 x 6 + 4	gcd(6, 4)
6	= 1 x 4 + 2	gcd(4, 2)
4	= 2 x 2 + 0	gcd(2, 0)
Therefore, gcd(1970, 1066) = 2		

The alert reader may ask how we can be sure that this process terminates. That is, how can we be sure that at some point B divides A? If not, we would get an endless sequence of positive integers, each one strictly smaller than the one before, and this is clearly impossible.

The RSA Algorithm

The pioneering paper by Diffie and Hellman introduced a new approach to cryptography and, in effect, challenged cryptologists to come up with a cryptographic algorithm that met the requirements for public-key systems. One of the first of the responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 . The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose approach to public-key encryption.

Apparently, the first workable public-key system for encryption/decryption was put forward by Clifford Cocks of Britain's CESG in 1973 ; Cocks's method is virtually identical to RSA.

The RSA scheme is a block cipher in which the plaintext and ciphertext are integers between 0 and n 1 for some n. A typical size for n is 1024 bits, or 309 decimal digits. That is, n is less than 2^{1024}. We examine RSA in this section in some detail, beginning with an explanation of the algorithm. Then we examine some of the computational and cryptanalytical implications of RSA.

Description of the Algorithm

The scheme developed by Rivest, Shamir, and Adleman makes use of an expression with exponentials. Plaintext is encrypted in blocks, with each block having a binary value less than some number n. That is, the block size must be less than or equal to $\log_2(n)$; in practice, the block size is i bits, where $2^i < n \leq 2^{i+1}$. Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$C = M^e \bmod n$

$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$

Both sender and receiver must know the value of n. The sender knows the value of e, and only the receiver knows the value of d. Thus, this is a public-key encryption algorithm with a public key of PU = {e, n} and a private key

of PU = {d, n}. For this algorithm to be satisfactory for public-key encryption, the following requirements must be met:

1. It is possible to find values of e, d, n such that M^{ed} mod n = M for all M < n.
2. It is relatively easy to calculate mod M^e mod n and C^d for all values of M < n.
3. It is infeasible to determine d given e and n.

For now, we focus on the first requirement and consider the other questions later. We need to find a relationship of the form

M^{ed} mod n = M

The preceding relationship holds if e and d are multiplicative inverses modulo ϕ(n), where ϕ(n) is the Euler totient function. That for p, q prime, ϕ(pq) = (p 1)(q 1) The relationship between e and d can be expressed as

Equation 9-1

$$ed \bmod \phi(n) = 1$$

This is equivalent to saying

ed ≡ 1 mod ϕ (n)

d ≡ e^1 mod ϕ (n)

That is, e and d are multiplicative inverses mod ϕ (n). Note that, according to the rules of modular arithmetic, this is true only if d (and therefore e) is relatively prime to ϕ (n). Equivalently, gcd(ϕ (n),d) = 1.

We are now ready to state the RSA scheme. The ingredients are the following:

p,q, two prime numbers	(private, chosen)
n = pq	(public, calculated)
e, with gcd(ϕ(n),e) = 1;1 < e < ϕ (n)	(public, chosen)
d≡ e^{1}(mod ϕ (n))	(private, calculated)

The private key consists of {d, n} and the public key consists of {e, n}. Suppose that user A has published its public key and that user B wishes to send the message M to A. Then B calculates C = M^{e} mod n and transmits C. On receipt of this ciphertext, user A decrypts by calculating M = C^{d} mod n.

Figure 9.5 summarizes the RSA algorithm. An example, is shown in Figure 9.6. For this example, the keys were generated as follows:

1. Select two prime numbers, p = 17 and q = 11.
2. Calculate n = pq = 17 x 11 = 187.
3. Calculate ϕ (n) = (p 1)(q 1) = 16 x 10 = 160.
4. Select e such that e is relatively prime to ϕ (n) = 160 and less than ϕ(n) we choose e = 7.
5. Determine d such that de 1 (mod 160) and d < 160. The correct value is d = 23, because 23 x 7 = 161 = 10 x 160 + 1; d can be calculated using the extended Euclid's algorithm.

Figure 9.5. The RSA Algorithm

Key Generation

Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p - 1)(q - 1)$	
Select integer e	$\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$d \equiv e^{-1} \pmod{\phi(n)}$
Public key	$PU = \{e, n\}$
Private key	$PR = \{d, n\}$

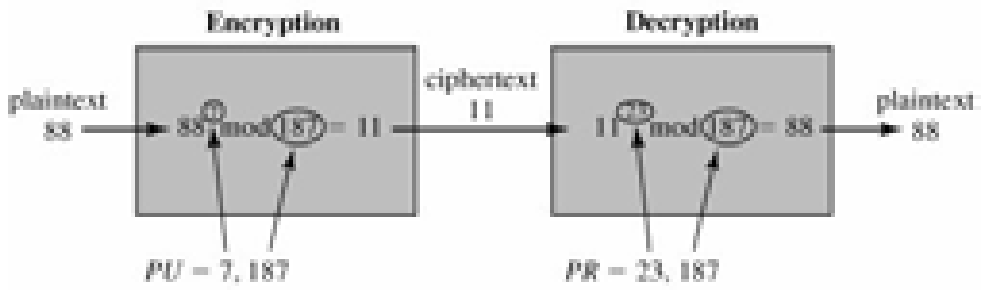
Encryption

Plaintext:	$M < n$
Ciphertext:	$C = M^e \bmod n$

Decryption

Ciphertext:	C
Plaintext:	$M = C^d \bmod n$

Figure 9.6. Example of RSA Algorithm



The resulting keys are public key PU = {7,187} and private key PR = {23,187}. The example shows the use of these keys for a plaintext input of M = 88. For encryption, we need to calculate $C = 88^7 \bmod 187$. Exploiting the properties of modular arithmetic, we can do this as follows:

$88^7 \bmod 187 = [(88^4 \bmod 187) \times (88^2 \bmod 187) \times (88^1 \bmod 187)] \bmod 187$

$88^1 \bmod 187 = 88$

$88^2 \bmod 187 = 7744 \bmod 187 = 77$

$88^4 \bmod 187 = 59{,}969{,}536 \bmod 187 = 132$

$88^7 \bmod 187 = (88 \times 77 \times 132) \bmod 187 = 894{,}432 \bmod 187 = 11$

For decryption, we calculate $M = 11^{23} \bmod 187$:

$11^{23} \bmod 187 = [(11^1 \bmod 187) \times (11^2 \bmod 187) \times (11^4 \bmod 187) \times (11^8 \bmod 187) \times (11^8 \bmod 187)] \bmod 187$

$11^1 \bmod 187 = 11$

$11^2 \bmod 187 = 121$

$11^4 \bmod 187 = 14{,}641 \bmod 187 = 55$

$11^8 \bmod 187 = 214{,}358{,}881 \bmod 187 = 33$

$11^{23} \bmod 187 = (11 \times 121 \times 55 \times 33 \times 33) \bmod 187 = 79{,}720{,}245 \bmod 187 = 88$

Computational Aspects

We now turn to the issue of the complexity of the computation required to use RSA. There are actually two issues to consider: encryption/decryption and key generation. Let us look first at the process of encryption and decryption and then consider key generation.

Exponentiation in Modular Arithmetic

Both encryption and decryption in RSA involve raising an integer to an integer power, mod n. If the exponentiation is done over the integers and then reduced modulo n, the intermediate values would be gargantuan. Fortunately, as the preceding example shows, we can make use of a property of modular arithmetic:

$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$

Thus, we can reduce intermediate results modulo n. This makes the calculation practical.

Another consideration is the efficiency of exponentiation, because with RSA we are dealing with potentially large exponents. To see how efficiency might be increased, consider that we wish to compute x^{16}. A straightforward approach requires 15 multiplications:

$x^{16} = x \times x$

However, we can achieve the same final result with only four multiplications if we repeatedly take the square of each partial result, successively forming x^2, x^4, x^8, x^{16}. As another example, suppose we wish to calculate $x^{11} \bmod n$ for some integers x and n. Observe that $x^{11} = x^{1+2+8} = (x)(x^2)(x^8)$. In this case we compute $x \bmod n$, $x^2 \bmod n$, $x^4 \bmod n$, and $x^8 \bmod n$ and then calculate $[(x \bmod n) \times (x^2 \bmod n) \times (x^8 \bmod n)] \bmod n$.

More generally, suppose we wish to find the value a^b with a and b positive integers. If we express b as a binary number $b_k b_{k1} \ldots b_0$ then we have

$$b = \sum_{b_i \neq 0} 2^i$$

Therefore,

$$a^b = a^{\left(\sum_{b_i \neq 0} 2^i\right)} = \prod_{b_i \neq 0} a^{(2^i)}$$

$$a^b \bmod n = \left[\prod_{b_i \neq 0} a^{(2^i)}\right] \bmod n = \left(\prod_{b_i \neq 0} \left[a^{(2^i)} \bmod n\right]\right) \bmod n$$

We can therefore develop the algorithm for computing a^b mod n, shown in Figure 9.7. Table 9.3 shows an example of the execution of this algorithm. Note that the variable c is not needed; it is included for explanatory purposes. The final value of c is the value of the exponent.

The algorithm has a long history; this particular pseudocode expression is from.

Figure 9.7. Algorithm for computing a^b mod n

Note: The integer b is expressed as a binary number $b_k b_k 1 \ldots b_0$

```
c ← 0; f ← 1
for i ← k downto 0
    do  c ← 2 × c
        f ← (f × f) mod n
    if  bi = 1
        then c ← c + 1
             f ← (f × a) mod n
return f
```

Table 9.3. Result of the Fast Modular Exponentiation Algorithm for a^b mod n, where a = 7, b = 560 = 1000110000, n = 561

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0

Table 9.3. Result of the Fast Modular Exponentiation Algorithm for a^b mod n, where a = 7, b = 560 = 1000110000, n = 561

i	9	8	7	6	5	4	3	2	1	0
c	1	2	4	8	17	35	70	140	280	560
f	7	49	157	526	160	241	298	166	67	1

Efficient Operation Using the Public Key

To speed up the operation of the RSA algorithm using the public key, a specific choice of e is usually made. The most common choice is 65537 (2^{16} 1); two other popular choices are 3 and 17. Each of these choices has only two 1 bits and so the number of multiplications required to perform exponentiation is minimized.

However, with a very small public key, such as e = 3, RSA becomes vulnerable to a simple attack. Suppose we have three different RSA users who all use the value e = 3 but have unique values of n, namely n_1, n_2, n_3. If user A sends the same encrypted message M to all three users, then the three ciphertexts are $C_1 = M^3 \bmod n_1$; $C_2 = M^3 \bmod n_2$; $C_3 = M^3 \bmod n_3$. It is likely that n_1, n_2, and n_3 are pairwise relatively prime. Therefore, one can use the Chinese remainder theorem (CRT) to compute $M^3 \bmod (n_1 n_2 n_3)$. By the rules of the RSA algorithm, M is less than each of the n_i therefore $M^3 < n_1 n_2 n_3$. Accordingly, the attacker need only compute the cube root of M^3. This attack can be countered by adding a unique pseudorandom bit string as padding to each instance of M to be encrypted. This approach is discussed subsequently.

The reader may have noted that the definition of the RSA algorithm (Figure 9.5) requires that during key generation the user selects a value of e that is relatively prime to $\phi(n)$. Thus, for example, if a user has preselected e = 65537 and then generated primes p and q, it may turn out that $\gcd(\phi(n),e) \neq 1$, Thus, the user must reject any value of p or q that is not congruent to 1 (mod 65537).

Efficient Operation Using the Private Key

We cannot similarly choose a small constant value of d for efficient operation. A small value of d is vulnerable to a brute-force attack and to other forms of cryptanalysis. However, there is a way to speed up computation using the CRT. We wish to compute the value $M = C^d \bmod n$. Let us define the following intermediate results:

$V_p = C^d \bmod p \quad V_q = C^d \bmod q$

Following the CRT, Equation (8.8), define the quantities:

$X_p = q \times (q^1 \bmod p) \quad X_q = p \times (p^1 \bmod q)$

The CRT then shows, using Equation (8.9), that

$M = (V_p X_p + V_q X_q) \bmod n$

Further, we can simplify the calculation of V_p and V_q using Fermat's theorem, which states that $a^{p1} \equiv 1 \pmod p$ if p and a are relatively prime. Some thought should convince you that the following are valid:

$V_p = C^d \bmod p = C^{d \bmod (p1)} \bmod p \quad V_q = C^d \bmod q = C^{d \bmod (q1)} \bmod q$

The quantities d mod (P1) and d mod (q1) can be precalculated. The end result is that the calculation is approximately four times as fast as evaluating $M = C^d \bmod n$ directly .

Key Generation

Before the application of the public-key cryptosystem, each participant must generate a pair of keys. This involves the following tasks:

- Determining two prime numbers, p and q
- Selecting either e or d and calculating the other

First, consider the selection of p and q. Because the value of n = pq will be known to any potential adversary, to prevent the discovery of p and q by exhaustive methods, these primes must be chosen from a sufficiently large set (i.e., p and q must be large numbers). On the other hand, the method used for finding large primes must be reasonably efficient.

At present, there are no useful techniques that yield arbitrarily large primes, so some other means of tackling the problem is needed. The procedure that is generally used is to pick at random an odd number of the desired order of magnitude and test whether that number is prime. If not, pick successive random numbers until one is found that tests prime.

A variety of tests for primality have been developed Almost invariably, the tests are probabilistic. That is, the test will merely determine that a given integer is probably prime. Despite this lack of certainty, these tests can be run in such a way as to make the probability as close to 1.0 as desired. As an example, one of the more efficient and popular algorithms, With this algorithm and most such algorithms, the procedure for testing whether a given integer n is prime is to perform some calculation that involves n and a randomly chosen integer a. If n "fails" the test, then n is not prime. If n "passes" the test, then n may be prime or nonprime. If n passes many such tests with many different randomly chosen values for a, then we can have high confidence that n is, in fact, prime.

In summary, the procedure for picking a prime number is as follows.

1. Pick an odd integer n at random (e.g., using a pseudorandom number generator).

2. Pick an integer a < n at random.

3. Perform the probabilistic primality test, such as Miller-Rabin, with a as a parameter. If n fails the test, reject the value n and go to step 1.

4. If n has passed a sufficient number of tests, accept n; otherwise, go to step 2.

This is a somewhat tedious procedure. However, remember that this process is performed relatively infrequently: only when a new pair (PU, PR) is needed.

It is worth noting how many numbers are likely to be rejected before a prime number is found. A result from number theory, known as the prime number theorem, states that the primes near N are spaced on the average one every (ln N) integers. Thus, on average, one would have to test on the order of ln(N) integers before a prime is found. Actually, because all even integers can be immediately rejected, the correct figure is ln(N)/2. For example, if a prime on the order of magnitude of 2^{200} were sought, then about $\ln(2^{200})/2 = 70$ trials would be needed to find a prime.

Having determined prime numbers p and q, the process of key generation is completed by selecting a value of e and calculating d or, alternatively, selecting a value of d and calculating e. Assuming the former, then we need to select an e such that gcd(ϕ(n), e) = 1 and then calculate $d \equiv e^{1} (\text{mod} \quad (n))$. Fortunately, there is a single algorithm that will, at the same time, calculate the greatest common divisor of two integers and, if the gcd is 1, determine the inverse of one of the integers modulo the other.. Thus, the procedure is to generate a series of random numbers, testing each against (n) until a number relatively prime to ϕ(n) is found. Again, we can ask the question: How many random numbers must we test to find a usable number, that is, a number relatively prime to ϕ(n)? It can be shown easily that the probability that two random numbers are relatively prime is about 0.6; thus, very few tests would be needed to find a suitable integer

The Security of RSA

Four possible approaches to attacking the RSA algorithm are as follows:

- Brute force: This involves trying all possible private keys.
- Mathematical attacks: There are several approaches, all equivalent in effort to factoring the product of two primes.
- Timing attacks: These depend on the running time of the decryption algorithm.
- Chosen ciphertext attacks: This type of attack exploits properties of the RSA algorithm.

The defense against the brute-force approach is the same for RSA as for other cryptosystems, namely, use a large key space. Thus, the larger the number of bits in d, the better. However, because the calculations involved,

both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run.

In this subsection, we provide an overview of mathematical and timing attacks.

The Factoring Problem

We can identify three approaches to attacking RSA mathematically:

- Factor n into its two prime factors. This enables calculation of $\phi(n) = (p\ 1) \times (q\ 1)$, which, in turn, enables determination of $d \equiv e^{1} \pmod{\phi(n)}$.
- Determine $\phi(n)$ directly, without first determining p and q. Again, this enables determination of $d \equiv e^{1} \pmod{\phi(n)}$.
- Determine d directly, without first determining $\phi(n)$.

Most discussions of the cryptanalysis of RSA have focused on the task of factoring n into its two prime factors. Determining $\phi(n)$ given n is equivalent to factoring n. With presently known algorithms, determining d given e and n appears to be at least as time-consuming as the factoring problem. Hence, we can use factoring performance as a benchmark against which to evaluate the security of RSA.

For a large n with large prime factors, factoring is a hard problem, but not as hard as it used to be. A striking illustration of this is the following. In 1977, the three inventors of RSA dared Scientific American readers to decode a cipher they printed in Martin Gardner's "Mathematical Games" column . They offered a $100 reward for the return of a plaintext sentence, an event they predicted might not occur for some 40 quadrillion years. In April of 1994, a group working over the Internet claimed the prize after only eight months of work .This challenge used a public key size (length of n) of 129 decimal digits, or around 428 bits. In the meantime, just as they had done for DES, RSA Laboratories had issued challenges for the RSA cipher with key sizes of 100, 110, 120, and so on, digits. The latest challenge to be met is the RSA-200 challenge with a key length of 200 decimal digits, or about 663 bits. Table 9.4 shows the results to date. The level of effort is measured in MIPS-years: a million-instructions-per-second processor running for one

year, which is about 3 x 10^{13} instructions executed. A 1 GHz Pentium is about a 250-MIPS machine.

Table 9.4. Progress in Factorization

Number of Decimal Digits	Approximate Number of Bits	Date Achieved	MIPS-years	Algorithm
100	332	April 1991	7	Quadratic sieve
110	365	April 1992	75	Quadratic sieve
120	398	June 1993	830	Quadratic sieve
129	428	April 1994	5000	Quadratic sieve
130	431	April 1996	1000	Generalized number field sieve
140	465	February 1999	2000	Generalized number field sieve
155	512	August 1999	8000	Generalized number field sieve
160	530	April 2003		Lattice sieve
174	576	December 2003		Lattice sieve
200	663	May 2005		Lattice sieve

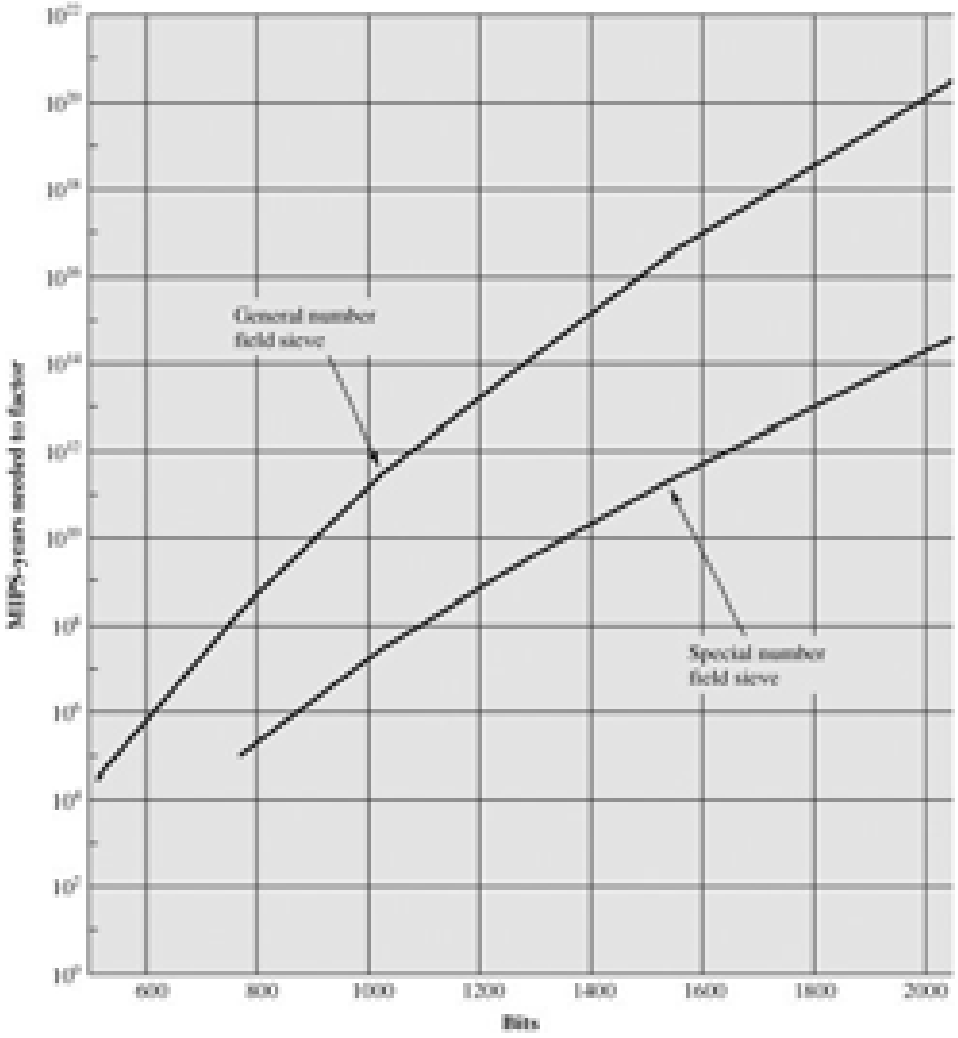
A striking fact about Table 9.4 concerns the method used. Until the mid-1990s, factoring attacks were made using an approach known as the quadratic sieve. The attack on RSA-130 used a newer algorithm, the generalized number field sieve (GNFS), and was able to factor a larger number than RSA-129 at only 20% of the computing effort.

The threat to larger key sizes is twofold: the continuing increase in computing power, and the continuing refinement of factoring algorithms. We have seen that the move to a different algorithm resulted in a tremendous speedup. We can expect further refinements in the GNFS, and the use of an even better algorithm is also a possibility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve. Figure 9.8

compares the performance of the two algorithms. It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better. Thus, we need to be careful in choosing a key size for RSA. For the near future, a key size in the range of 1024 to 2048 bits seems reasonable.

Figure 9.8. MIPS-years needed to Factor

(This item is displayed on page 277 in the print version)



In addition to specifying the size of n, a number of other constraints have been suggested by researchers. To avoid values of n that may be factored more easily, the algorithm's inventors suggest the following constraints on p and q:

1. p and q should differ in length by only a few digits. Thus, for a 1024-bit key (309 decimal digits), both p and q should be on the order of magnitude of 10^{75} to 10^{100}.
2. Both (p 1) and (q 1) should contain a large prime factor.
3. gcd(p 1, q 1) should be small.

In addition, it has been demonstrated that if e < n and d < $n^{1/4}$, then d can be easily determined .

Timing Attacks

If one needed yet another lesson about how difficult it is to assess the security of a cryptographic algorithm, the appearance of timing attacks provides a stunning one. Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer takes to decipher messages. Timing attacks are applicable not just to RSA, but to other public-key cryptography systems. This attack is alarming for two reasons: It comes from a completely unexpected direction and it is a ciphertext-only attack.

A timing attack is somewhat analogous to a burglar guessing the combination of a safe by observing how long it takes for someone to turn the dial from number to number. We can explain the attack using the modular exponentiation algorithm of Figure 9.7, but the attack can be adapted to work with any implementation that does not run in fixed time. In this algorithm, modular exponentiation is accomplished bit by bit, with one modular multiplication performed at each iteration and an additional modular multiplication performed for each 1 bit.

As Kocher points out in his paper, the attack is simplest to understand in an extreme case. Suppose the target system uses a modular multiplication function that is very fast in almost all cases but in a few cases takes much more time than an entire average modular exponentiation. The attack proceeds bit-by-bit starting with the leftmost bit, b_k. Suppose that the first j bits are known (to obtain the entire exponent, start with j = 0 and repeat the attack until the entire exponent is known). For a given ciphertext, the attacker can complete the first j iterations of the for loop. The operation of the subsequent step depends on the unknown exponent bit. If the bit is set, d
(d x a) mod n will be executed. For a few values of a and d, the modular multiplication will be extremely slow, and the attacker knows which these are. Therefore, if the observed time to execute the decryption algorithm is always slow when this particular iteration is slow with a 1 bit, then this bit is assumed to be 1. If a number of observed execution times for the entire algorithm are fast, then this bit is assumed to be 0.

In practice, modular exponentiation implementations do not have such extreme timing variations, in which the execution time of a single iteration can exceed the mean execution time of the entire algorithm. Nevertheless, there is enough variation to make this attack practical. For details.

Although the timing attack is a serious threat, there are simple countermeasures that can be used, including the following:

- Constant exponentiation time: Ensure that all exponentiations take the same amount of time before returning a result. This is a simple fix but does degrade performance.
- Random delay: Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack. Kocher points out that if defenders don't add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.
- Blinding: Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.

RSA Data Security incorporates a blinding feature into some of its products. The private-key operation $M = C^d \bmod n$ is implemented as follows:

1. Generate a secret random number r between 0 and n 1.
2. Compute $C' = C(r^e) \bmod n$, where e is the public exponent.
3. Compute $M' = (C')^d \bmod n$ with the ordinary RSA implementation.
4. Compute $M = M'r^1 \bmod n$. In this equation, r^1 is the multiplicative inverse of r mod n. It can be demonstrated that this is the correct result by observing that $r^{ed} \bmod n = r \bmod n$.

RSA Data Security reports a 2 to 10% performance penalty for blinding.

Chosen Ciphertext Attack and Optimal Asymmetric Encryption Padding

The basic RSA algorithm is vulnerable to a chosen ciphertext attack (CCA). CCA is defined as an attack in which adversary chooses a number of ciphertexts and is then given the corresponding plaintexts, decrypted with the target's private key. Thus, the adversary could select a plaintext, encrypt it with the target's public key and then be able to get the plaintext back by having it decrypted with the private key. Clearly, this provides the adversary with no new information. Instead, the adversary exploits properties of RSA and selects blocks of data that, when processed using the target's private key, yield information needed for cryptanalysis.

A simple example of a CCA against RSA takes advantage of the following property of RSA:

Equation 9-2

$$E(PU, M_1) \times E(PU, M_2) = E(PU, [M_1 \times M_2])$$

We can decrypt $C = M^e$ using a CCA as follows.

1. Compute $X = (C \times 2^e) \bmod n$.
2. Submit X as a chosen ciphertext and receive back $Y = X^d \bmod n$.

But now note the following:

$X = (C \bmod n) \times (2^e \text{ mode } n)$

$= (M^e \bmod n) \times (2^e \text{ mode } n)$

$= (2M)^e \bmod n$

Therefore, $Y = (2M) \bmod n$ from this, we can deduce M. To overcome this simple attack, practical RSA-based cryptosystems randomly pad the plaintext prior to encryption. This randomizes the ciphertext so that Equation (9.2) no longer holds. However, more sophisticated CCAs are possible and a simple padding with a random value has been shown to be insufficient to provide the desired security. To counter such attacks RSA Security Inc., a leading RSA vendor and former holder of the RSA patent, recommends modifying the plaintext using a procedure known as optimal asymmetric encryption padding (OAEP). A full discussion of the threats and OAEP are beyond our scope.

Figure 9.9 depicts OAEP encryption. As a first step the message M to be encrypted is padded. A set of optional parameters P is passed through a hash function H.The output is then padded with zeros to get the desired length in the overall data block (DB). Next, a random seed is generated and passed through another hash function, called the mask generating function (MGF). The resulting hash value is bit-by-bit XORed with DB to produce a maskedDB. The maskedDB is in turn passed through the MGF to form a hash that is XORed with the seed to produce the masked seed. The

concatenation of the maskedseed and the maskedDB forms the encoded message EM. Note that the EM includes the padded message, masked by the seed, and the seed, masked by the maskedDB. The EM is then encrypted using RSA.

A hash function maps a variable-length data block or message into a fixed-length value called a hash code.

Figure 9.9. Encryption Using Optimal Asymmetric Encryption Padding (OAEP)