UNIT-1-OVERVIEW

1.1 COMPUTER SECURITY CONCEPTS

A Definition Of Computer Security

These three concepts form what is often referred to as the CIA triad (Figure 1.1). The three concepts embody the fundamental security objectives for both data and for information and computing services. FIPS PUB 199 provides a useful characterization of these three objectives in terms of requirements and the definition of a loss of security in each category:

• Confidentiality (covers both data confidentiality and privacy): preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. A loss of confidentiality is the unauthorized disclosure of information.

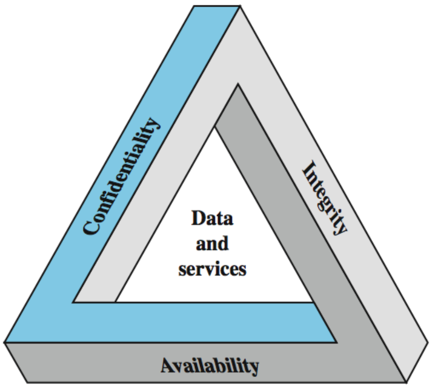
• Integrity (covers both data and system integrity): Guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity. A loss of integrity is the unauthorized modification or destruction of information.

• Availability: Ensuring timely and reliable access to and use of information. A loss of availability is the disruption of access to or use of information or an information system.

Although the use of the CIA triad to define security objectives is well established, some in the security field feel that additional concepts are needed to present a complete picture. Two of the most commonly mentioned are:

• Authenticity: The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator.

• Accountability: The security goal that generates the requirement for actions of an entity to be traced uniquely to that entity.



1.1.1 The challenges of computer security

1. The major requirements for security services can be given self-explanatory, one-word labels: confidentiality, authentication, nonrepudiation, or integrity.
2. In developing a particular security mechanism or algorithm, one must always consider potential attacks on those security features.
3. When the various aspects of the thread are considered that elaborate security mechanisms make sense.
4. It is necessary to decide where to use the security mechanisms, both in terms of physical placement(at what points in a network) and in a logical sense(at what layer or layers of a architecture such as TCP/IP)
5. Security mechanism typically involve more than a particular algorithm or protocol.
6. Computer and network security is essentially a battle of wits between a attackers and the designer/ administrator.
7. Security requires regular, even constant, monitoring.
8. Security is still too often an afterthought to be incorporated into a system after the design is complete rather than being an integral part of the design process.
9. Many users and even security administrator view strong security as an impediment to efficient and user-friendly operation of an information system.

1.2 THE OSI SECURITY ARCHITECTURE

The OSI security architecture focuses on security attack, security mechanism , security service

- **security attack:** Any action that compromises the security of information owned by an organization.
- **Security mechanism:** A process that is designed to detect, prevent, or recover from a security attack.
- **Security service:** A processing or communication service that enhances the security of the data processing systems and the information transfers of an organization.

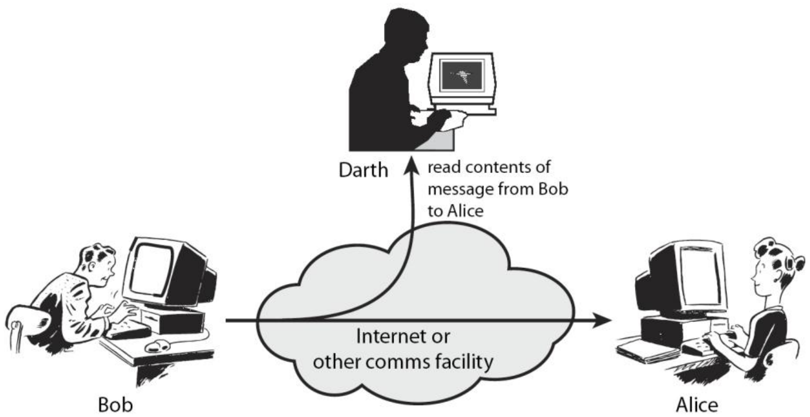
1.3 SECURITY ATTACKS

A useful means of classifying security attacks, used both in X.800 and RFC 2828, is in terms of *passive attacks* and *active attacks.* A passive attack attempts to learn or make use of information from the system but does not affect system resources.

Passive attacks are in the nature of eavesdropping on, or monitoring of, transmissions. The goal of the opponent is to obtain information that is being transmitted. Two types of passive attacks are:

- release of message contents - as shown above in Stallings Figure 1.2a here
- traffic analysis - monitor traffic flow to determine location and identity of communicating hosts and could observe the frequency and length of messages being exchanged

These attacks are difficult to detect because they do not involve any alteration of the data.



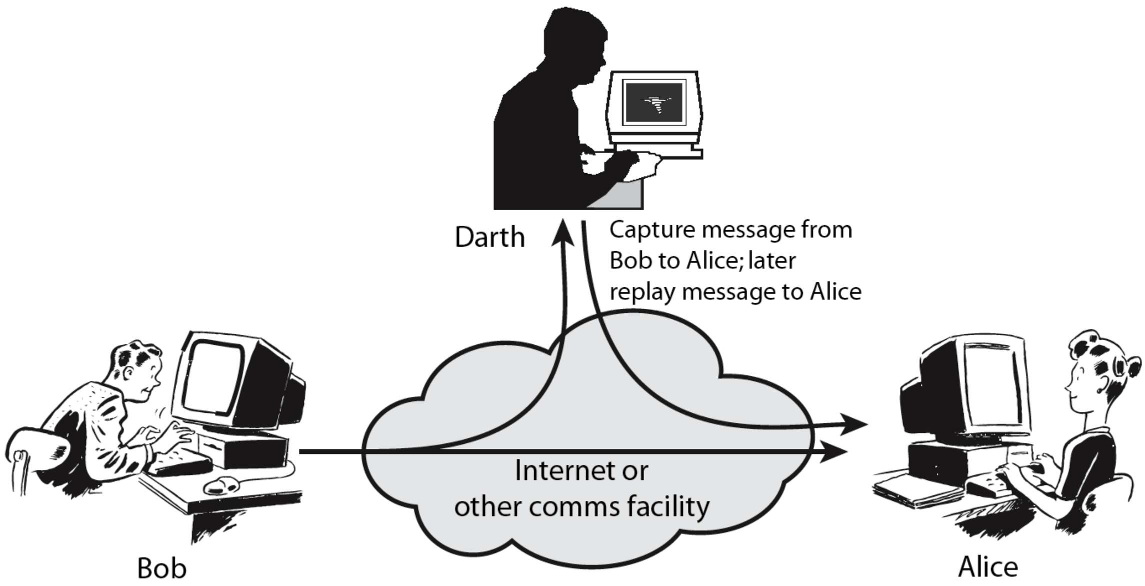
ACTIVE ATTACKS

Active attacks involve some modification of the data stream or the creation of a false stream and can be subdivided into four categories: masquerade, replay, modification of messages, and denial of service:

- masquerade of one entity as some other

- replay previous messages (as shown above in Stallings Figure 1.3b)
- modify/alter (part of) messages in transit to produce an unauthorized effect
- denial of service - prevents or inhibits the normal use or management of communications facilities

Active attacks present the opposite characteristics of passive attacks. Whereas passive attacks are difficult to detect, measures are available to prevent their success. On the other hand, it is quite difficult to prevent active attacks absolutely, because of the wide variety of potential physical, software, and network vulnerabilities. Instead, the goal is to detect active attacks and to recover from any disruption or delays caused by them.



1.4 SECURITY SERVICES

- enhance security of data processing systems and information transfers of an organization
- intended to counter security attacks
- using one or more security mechanisms
- often replicates functions normally associated with physical documents

which, for example, have signatures, dates; need protection from disclosure, tampering, or destruction; be notarized or witnessed; be recorded or licensed

Consider the role of a security service, and what may be required.

Note both similarities and differences with traditional paper documents, which for example:

- have signatures & dates;
- need protection from disclosure, tampering, or destruction;
- may be notarized or witnessed;
- may be recorded or licensed

Security Services (X.800)

- ✖ **Authentication** - assurance that communicating entity is the one claimed
    - ✚ have both peer-entity & data origin authentication
- ✖ **Access Control** - prevention of the unauthorized use of a resource
- ✖ **Data Confidentiality** –protection of data from unauthorized disclosure
- ✖ **Data Integrity** - assurance that data received is as sent by an authorized entity
- ✖ **Non-Repudiation** - protection against denial by one of the parties in a communication
- ✖ **Availability** – resource accessible/usable

1.5 SECURITY MECHANISM

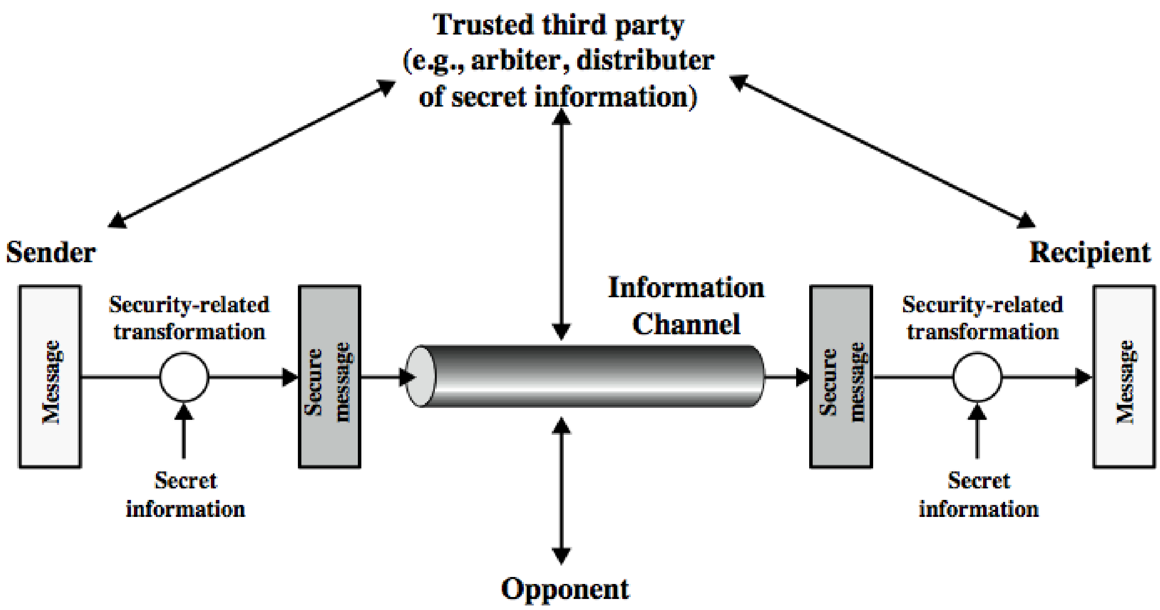
- ✖ feature designed to detect, prevent, or recover from a security attack
- ✖ no single mechanism that will support all services required
- ✖ however one particular element underlies many of the security mechanisms in use:
    - ✚ cryptographic techniques
- ✖ hence our focus on this topic
- ✖ specific security mechanisms:
    - ✚ encipherment, digital signatures, access controls, data integrity, authentication exchange, traffic padding, routing control, notarization
- ✖ pervasive security mechanisms:

+ trusted functionality, security labels, event detection, security audit trails, security recovery

1.6 Model for Network Security

In considering the place of encryption, its useful to use the following two models from Stallings section 1.6.

The first, illustrated in Figure models information being transferred from one party to another over an insecure communications channel, in the presence of possible opponents. The two parties, who are the principals in this transaction, must cooperate for the exchange to take place. They can use an appropriate security transform (encryption algorithm), with suitable keys, possibly negotiated using the presence of a trusted third party. Parts One through Four of this book concentrates on the types of security mechanisms and services that fit into the model shown here.



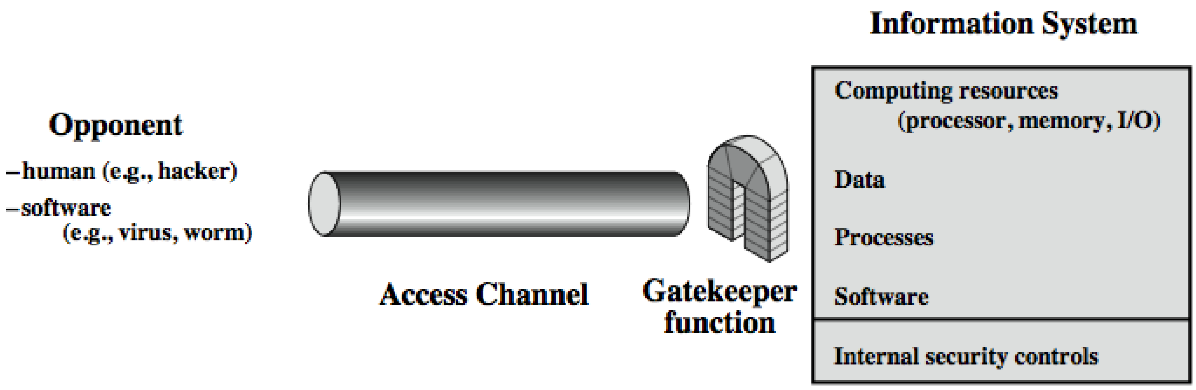
Model for Network Access Security

The second, illustrated in Figure 1.5, model is concerned with controlled access to information or resources on a computer system, in the presence of possible opponents. Here appropriate controls are needed on the access to and within the system, to provide suitable security.

The security mechanisms needed to cope with unwanted access fall into two broad categories (as shown in this figure).

The first category might be termed a *gatekeeper function*. It includes password-based login procedures that are designed to deny access to all but authorized users and screening logic that is designed to detect and reject worms, viruses, and other similar attacks. Once either an unwanted user or unwanted software gains access,

the second line of defense consists of a variety of internal controls that monitor activity and analyze stored information in an attempt to detect the presence of unwanted intruders.



1.7 SYMMETRIC CIPHER MODEL

A symmetric encryption scheme has five ingredients

- **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.

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 encryp-tion scheme, using Figure 2.2. A source produces a message in plaintext, $X = [X_1, X_2, \ldots.., X_M]$. 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 $K = [K_1, K_2, \ldots.., K_J]$ 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 chan-nel. Alternatively, a third party could generate the key and securely deliver it to both source and destination.

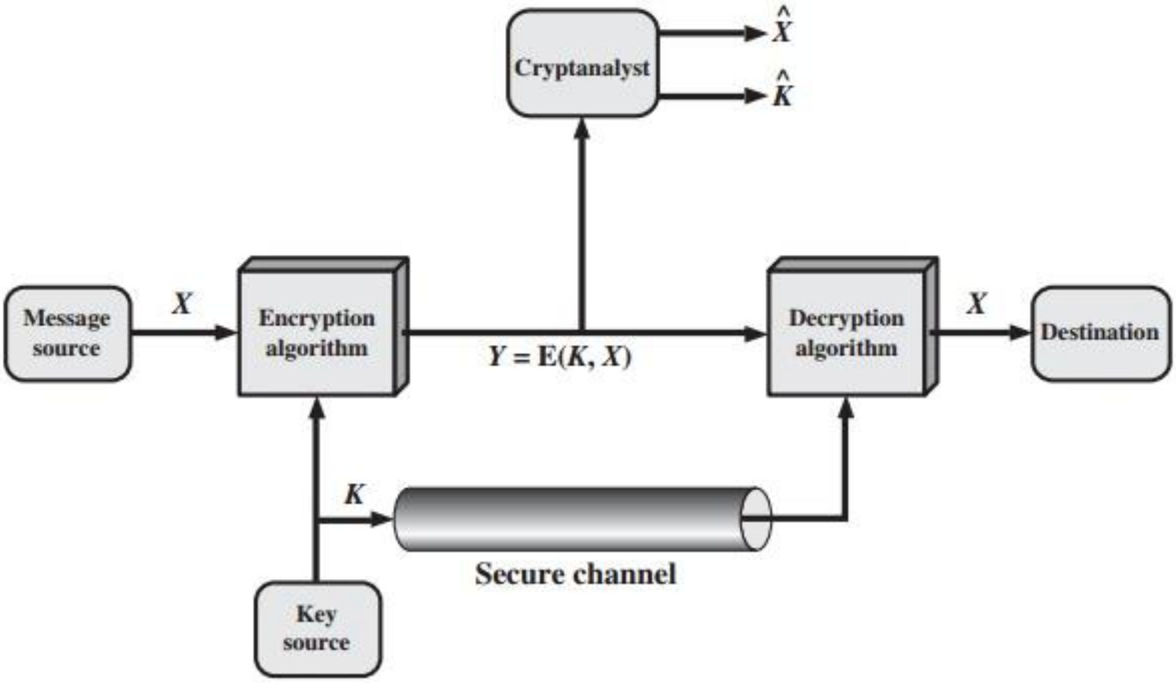


Figure 2.2 Model of Symmetric Cryptosystem

With the message X and the encryption key K as input, the encryption algo-rithm forms the ciphertext $Y = [Y_1, Y_2, \ldots.., Y_N]$. 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:
$X = D(K, Y)$

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 informa-tion 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 encryp-tion. 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 and Brute-Force Attack

Typically, the objective of attacking an encryption system is to recover the key in use rather than simply to recover the plaintext of a single ciphertext. There are two gen-eral 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 cipher-text 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 pre-sented 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 • Ciphertext chosen by cryptanalyst, together with its 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 • 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.

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 con-tain 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 encryp-tion 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. Unfortunately, it is very difficult to estimate the amount of effort required to cryptanalyze ciphertext successfully.

Table 2.2 Average Time Required for Exhaustive Key Search

Key Size (bits)	Number of Alternative Keys	Time Required at 1 Decryption/μs	Time Required at 10^6 Decryptions/μs
32	$2^{32} = 4.3 \times 10^9$	$2^{31} \mu s$ = 35.8 minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	$2^{55} \mu s$ = 1142 years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127} \mu s = 5.4 \times 10^{24}$ years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	$2^{167} \mu s = 5.9 \times 10^{36}$ years	5.9×10^{30} years
26 characters (permutation)	$26! = 4 \times 10^{26}$	$2 \times 10^{26} \mu s = 6.4 \times 10^{12}$ years	6.4×10^6 years

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.

1.8 SUBSTITUTION TECHNIQUES

In this section and the next, we examine a sampling of what might be called classical encryption techniques. A study of these techniques enables us to illustrate the basic approaches to symmetric encryption used today and the types of cryptanalytic attacks that must be anticipated.

The two basic building blocks of all encryption techniques are substitution and transposition. We examine these in the next two sections. Finally, we discuss a system that combines both substitution and transposition.

A substitution technique is one in which the letters of plaintext are replaced by other letters or by numbers or symbols. If the plaintext is viewed as a sequence of bits, then substitution involves replacing plaintext bit patterns with ciphertext bit patterns.

Caesar Cipher

The earliest known, and the simplest, use of a substitution cipher was by Julius Caesar. The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down the alphabet. For example,

Plain: meet me after the toga party
Cipher: PHHW PH DIWHU WKH WRJD SDUWB

Note that the alphabet is wrapped around, so that the letter following Z is A. We can define the transformation by listing all possibilities, as follows:

Plain: a b c d e f g h i j k l m n o p q r s t u v w x y z
cipher: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Let us assign a numerical equivalent to each letter:

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12

n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

Then the algorithm can be expressed as follows. For each plaintext letter p, substitute the ciphertext letter C:2

$$C = E(3, p) = (p + 3) \bmod 26$$

A shift may be of any amount, so that the general Caesar algorithm is

$$C = E(k, p) = (p + k) \bmod 26 \qquad \textbf{(2.1)}$$

where k takes on a value in the range 1 to 25. The decryption algorithm is simply

$$p = \mathrm{D}(k, C) = (C - k) \bmod 26 \qquad \textbf{(2.2)}$$

If it is known that a given ciphertext is a Caesar cipher, then a brute-force cryptanalysis is easily performed: simply try all the 25 possible keys. Figure 2.3 shows the results of applying this strategy to the example ciphertext. In this case, the plaintext leaps out as occupying the third line.

Three important characteristics of this problem enabled us to use a brute-force cryptanalysis:

- The encryption and decryption algorithms are known.
- There are only 25 keys to try.
- The language of the plaintext is known and easily recognizable.

In most networking situations, we can assume that the algorithms are known. What generally makes brute-force cryptanalysis impractical is the use of an algorithm that employs a large number of keys. For example, the triple DES algorithm, examined in Chapter 6, makes use of a 168-bit key, giving a key space of 2168 or greater than 3.7 * 1050 possible keys.

Monoalphabetic Ciphers

With only 25 possible keys, the Caesar cipher is far from secure. A dramatic increase in the key space can be achieved by allowing an arbitrary substitution. Before proceeding, we define the term *permutation.* A **permutation** of a finite set of elements S is an ordered sequence of all the elements of S, with each element appearing exactly once. For example, if S = {a, b, c}, there are six permutations of S:

abc, acb, bac, bca, cab, cba

In general, there are n! permutations of a set of n elements, because the first element can be chosen in one of n ways, the second in n - 1 ways, the third in n - 2 ways, and so on.

Recall the assignment for the Caesar cipher:

plain: a b c d e f g h i j k l m n o p q r s t u v w x y z
cipher: D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

If, instead, the "cipher" line can be any permutation of the 26 alphabetic characters, then there are 26! or greater than 4 * 1026 possible keys. This is 10 orders of magnitude greater than the key space for DES and would seem to eliminate brute-force techniques for cryptanalysis. Such an approach is referred to as a **monoalphabetic substitution cipher**, because a single cipher alphabet (mapping from plain alphabet to cipher alphabet) is used per message.

There is, however, another line of attack. If the cryptanalyst knows the nature of the plaintext (e.g., noncompressed English text), then the analyst can exploit the regularities of the language. The ciphertext to be solved is

UZQSOVUOHXMOPVGPOZPEVSGZWSZOPFPESXUDBMETSXAIZ
VUEPHZHMDZSHZOWSFPAPPDTSVPQUZWYMXUZUHSX
EPYEPOPDZSZUFPOMBZWPFUPZHMDJUDTMOHMQ

As a first step, the relative frequency of the letters can be determined and compared to a standard frequency distribution for English, such as is shown in Figure 2.5 (based on [LEWA00]). If the message were long enough, this technique alone might be sufficient, but because this is a relatively short message, we cannot expect an exact match. In any case, the relative frequencies of the letters in the ciphertext (in percentages) are as follows:

P 13.33	H 5.83	F 3.33	B 1.67	C 0.00
Z 11.67	D 5.00	W 3.33	G 1.67	K 0.00
S 8.33	E 5.00	Q 2.50	Y 1.67	L 0.00
U 8.33	V 4.17	T 2.50	I 0.83	N 0.00
O 7.50	X 4.17	A 1.67	J 0.83	R 0.00
M 6.67				

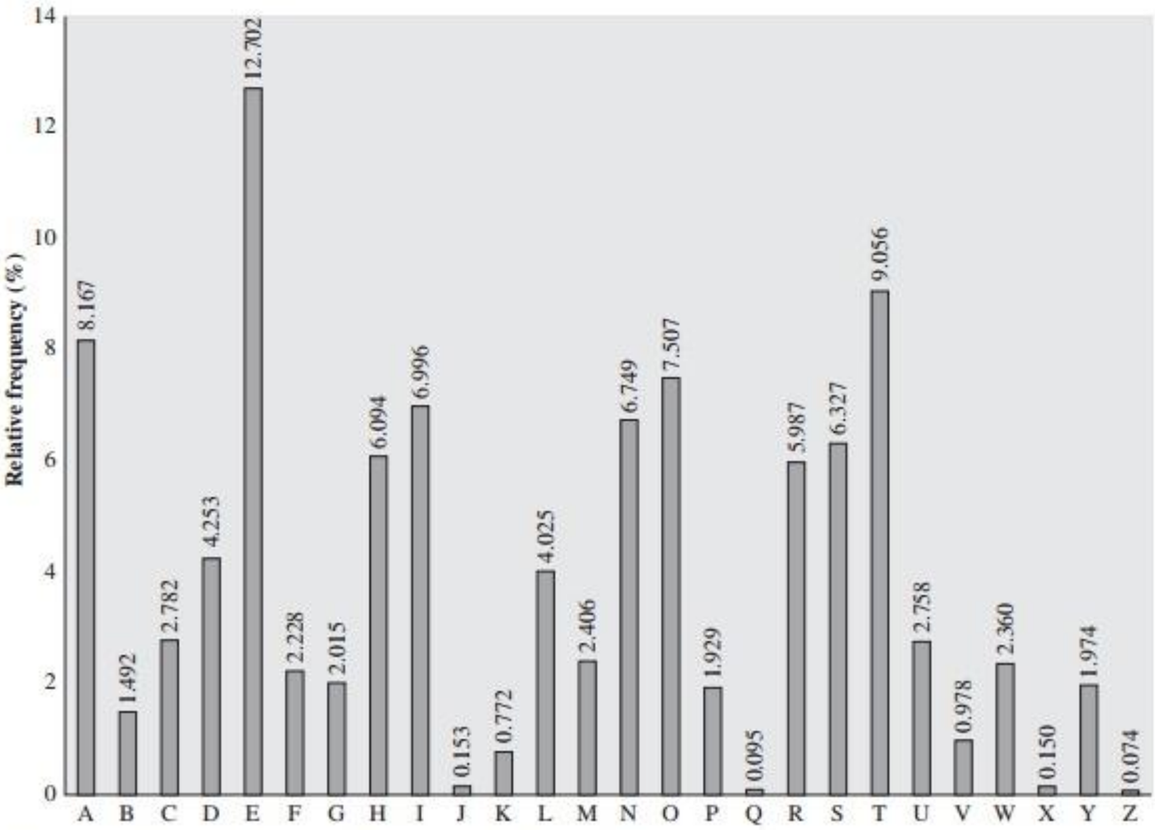


Figure 2.5 Relative Frequency of Letters in English Text

Comparing this breakdown with Figure 2.5, it seems likely that cipher letters P and Z are the equivalents of plain letters e and t, but it is not certain which is which. The letters S, U, O, M, and H are all of relatively high frequency and probably correspond to plain letters from the set {a, h, i, n, o, r, s}. The letters with the lowest frequencies (namely, A, B, G, Y, I, J) are likely included in the set {b, j, k, q, v, x, z}.

There are a number of ways to proceed at this point. We could make some tentative assignments and start to fill in the plaintext to see if it looks like a reasonable "skeleton" of a message. A more systematic approach is to look for other regularities. For example, certain words may be known to be in the text. Or we could look for repeating sequences of cipher letters and try to deduce their plaintext equivalents.

A powerful tool is to look at the frequency of two-letter combinations, known as **digrams**. A table similar to Figure 2.5 could be drawn up showing the relative frequency of digrams. The most common such digram is th. In our ciphertext, the most common digram is ZW, which appears three times. So we make the correspondence of Z with t and W with h. Then, by our earlier hypothesis, we can equate P with e. Now notice that the sequence ZWP appears in the ciphertext, and we can translate that sequence as "the." This is the most frequent trigram (three-letter combination) in English, which seems to indicate that we are on the right track.

Next, notice the sequence ZWSZ in the first line. We do not know that these four letters form a complete word, but if they do, it is of the form th_t. If so, S equates with a.

So far, then, we have

```
UZQSOVUOHXMOPVGPOZPEVSGZWSZOPFPESXUDBMETSXAIZ
 t a        e  e te  a that e e a        a
VUEPHZHMDZSHZOWSFPAPPDTSVPQUZWYMXUZUHSX
   e t   ta t ha e ee  a e  th    t  a
EPYEPOPDZSZUFPOMBZWPFUPZHMDJUDTMOHMQ
 e  e e tat  e   the   t
```

Only four letters have been identified, but already we have quite a bit of the message. Continued analysis of frequencies plus trial and error should easily yield a solution from this point. The complete plaintext, with spaces added between words, follows:

it was disclosed yesterday that several informal but direct contacts have been made with political representatives of the viet cong in Moscow

Monoalphabetic ciphers are easy to break because they reflect the frequency data of the original alphabet. A countermeasure is to provide multiple substitutes, known as homophones, for a single letter. For example, the letter e could be assigned a number of different cipher symbols, such as 16, 74, 35, and 21, with each homophone assigned to a letter in rotation or randomly. If the number of symbols assigned to each letter is proportional to the relative frequency of that letter, then single-letter frequency information is completely obliterated. The great mathematician Carl Friedrich Gauss believed that he had devised an unbreakable cipher using homo-phones. However, even with homophones, each element of plaintext affects only one element of ciphertext, and multiple-letter patterns (e.g., digram frequencies) still survive in the ciphertext, making cryptanalysis relatively straightforward.

Playfair Cipher

The best-known multiple-letter encryption cipher is the Playfair, which treats digrams in the plaintext as single units and translates these units into ciphertext digrams.

The Playfair algorithm is based on the use of a 5 x 5 matrix of letters con-structed using a keyword. Here is an example, solved by Lord Peter Wimsey in Dorothy Sayers's *Have His Carcase:*

M	O	N	A	R
C	H	Y	B	D
E	F	G	I/J	K
L	P	Q	S	T
U	V	W	X	Z

In this case, the keyword is *monarchy*. The matrix is constructed by filling in the letters of the keyword (minus duplicates) from left to right and from top to bot-tom, and then filling in the remainder of the matrix with the remaining letters in alphabetic order. The letters I and J count as one letter. Plaintext is encrypted two letters at a time, according to the following rules:

1. Repeating plaintext letters that are in the same pair are separated with a filler letter, such as x, so that balloon would be treated as ba lx lo on.

2. plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, ar is encrypted as RM.

3. Two plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the column circularly following the last. For example, mu is encrypted as CM.

4. Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter. Thus, hs becomes BP and ea becomes IM (or JM, as the encipherer wishes).

The Playfair cipher is a great advance over simple monoalphabetic ciphers. For one thing, whereas there are only 26 letters, there are 26 x 26 = 676 digrams, that identification of individual digrams is more difficult. Furthermore, the relative frequencies of individual letters exhibit a much greater range than that of digrams, making frequency analysis much more difficult. For these reasons, the Playfair cipher was for a long time considered unbreakable. It was used as the standard field system by the British Army in World War I and still enjoyed considerable use by the U.S. Army and other Allied forces during World War II.

Despite this level of confidence in its security, the Playfair cipher is relatively easy to break, because it still leaves much of the structure of the plaintext language intact. A few hundred letters of ciphertext are generally sufficient.

Hill Cipher

Another interesting multiletter cipher is the Hill cipher, developed by the mathe-matician Lester Hill in 1929.

CONCEPTS FROM LINEAR ALGEBRA Before describing the Hill cipher, let us briefly review some terminology from linear algebra. In this discussion, we are concerned with matrix arithmetic modulo 26. For the reader who needs a refresher on matrix multiplication and inversion, see Appendix E.

We define the inverse **M** - 1 of a square matrix **M** by the equation **M**(**M** - 1) = **M** - 1**M** = **I**, where **I** is the identity matrix. **I** is a square matrix that is all zeros except for ones along the main diagonal from upper left to lower right. The inverse of a matrix does not always exist, but when it does, it satisfies the preceding equation. For example,

$$\mathbf{A} = \begin{pmatrix} 5 & 8 \\ 17 & 3 \end{pmatrix} \qquad \mathbf{A}^{-1} \bmod 26 = \begin{pmatrix} 9 & 2 \\ 1 & 15 \end{pmatrix}$$

$$\mathbf{AA}^{-1} = \begin{pmatrix} (5 \times 9) + (8 \times 1) & (5 \times 2) + (8 \times 15) \\ (17 \times 9) + (3 \times 1) & (17 \times 2) + (3 \times 15) \end{pmatrix}$$
$$= \begin{pmatrix} 53 & 130 \\ 156 & 79 \end{pmatrix} \bmod 26 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

To explain how the inverse of a matrix is computed, we begin by with the con-cept of determinant. For any square matrix (*m* x *m*), the **determinant** equals the sum of all the products that can be formed by taking exactly one element from each row and exactly one element from each column, with certain of the product terms pre-ceded by a minus sign. For a 2 x 2 matrix,

$$\begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}$$

the determinant is $k_{11}k_{22} - k_{12}k_{21}$. For a 3 × 3 matrix, the value of the determinant is $k_{11}k_{22}k_{33} + k_{21}k_{32}k_{13} + k_{31}k_{12}k_{23} - k_{31}k_{22}k_{13} - k_{21}k_{12}k_{33} - k_{11}k_{32}k_{23}$. If a square matrix **A** has a nonzero determinant, then the inverse of the matrix is computed as $[\mathbf{A}^{-1}]_{ij} = (\det \mathbf{A})^{-1}(-1)^{i+j}(D_{ji})$, where (D_{ji}) is the subdeterminant formed by deleting the jth row and the ith column of $\mathbf{A}$, $\det(\mathbf{A})$ is the determinant of $\mathbf{A}$, and $(\det \mathbf{A})^{-1}$ is the multiplicative inverse of $(\det \mathbf{A}) \bmod 26$.

Continuing our example,

$$\det\begin{pmatrix} 5 & 8 \\ 17 & 3 \end{pmatrix} = (5 \times 3) - (8 \times 17) = -121 \bmod 26 = 9$$

We can show that $9^{-1} \bmod 26 = 3$, because $9 \times 3 = 27 \bmod 26 = 1$ (see Chapter 4 or Appendix E). Therefore, we compute the inverse of $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} 5 & 8 \\ 17 & 3 \end{pmatrix}$$

$$\mathbf{A}^{-1} \bmod 26 = 3\begin{pmatrix} 3 & -8 \\ -17 & 5 \end{pmatrix} = 3\begin{pmatrix} 3 & 18 \\ 9 & 5 \end{pmatrix} = \begin{pmatrix} 9 & 54 \\ 27 & 15 \end{pmatrix} = \begin{pmatrix} 9 & 2 \\ 1 & 15 \end{pmatrix}$$

THE HILL ALGORITHM This encryption algorithm takes m successive plaintext letters and substitutes for them m ciphertext letters. The substitution is determined by m linear equations in which each character is assigned a numerical value ($a = 0, b = 1, \ldots, z = 25$). For $m = 3$, the system can be described as

$$c_1 = (k_{11}p_1 + k_{12}p_2 + k_{13}p_3) \bmod 26$$
$$c_2 = (k_{21}p_1 + k_{22}p_2 + k_{23}p_3) \bmod 26$$
$$c_3 = (k_{31}p_1 + k_{32}p_2 + k_{33}p_3) \bmod 26$$

This can be expressed in terms of row vectors and matrices:[7]

$$(c_1\ c_2\ c_3) = (p\ \ p_2\ \ p_3)\begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} \bmod 26$$

or

$$\mathbf{C} = \mathbf{PK} \bmod 26$$

where **C** and **P** are row vectors of length 3 representing the plaintext and ciphertext, and **K** is a 3 * 3 matrix representing the encryption key. Operations are performed mod 26.
For example, consider the plaintext "paymoremoney" and use the encrypttion key

$$\mathbf{K} = \begin{pmatrix} 17 & 17 & 5 \\ 21 & 18 & 21 \\ 2 & 2 & 19 \end{pmatrix}$$

The first three letters of the plaintext are represented by the vector (15 0 24). Then (15 0 24)**K** = (303 303 531) mod 26 = (17 17 11) = RRL. Continuing in this fash-ion, the ciphertext for the entire plaintext is RRLMWBKASPDH.

Decryption requires using the inverse of the matrix **K**. We can compute det **K** = 23, and therefore, (det **K**)-1 mod 26 = 17. We can then compute the inverse as

$$\mathbf{K}^{-1} = \begin{pmatrix} 4 & 9 & 15 \\ 15 & 17 & 6 \\ 24 & 0 & 17 \end{pmatrix}$$

This is demonstrated as

$$\begin{pmatrix} 17 & 17 & 5 \\ 21 & 18 & 21 \\ 2 & 2 & 19 \end{pmatrix}\begin{pmatrix} 4 & 9 & 15 \\ 15 & 17 & 6 \\ 24 & 0 & 17 \end{pmatrix} = \begin{pmatrix} 443 & 442 & 442 \\ 858 & 495 & 780 \\ 494 & 52 & 365 \end{pmatrix} \bmod 26 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

It is easily seen that if the matrix **K** - 1 is applied to the ciphertext, then the plaintext is recovered.

In general terms, the Hill system can be expressed as

C = E(**K**, **P**) = **PK** mod 26

P = D(**K**, **C**) = **CK** - 1 mod 26 = **PKK** - 1 = **P**

As with Playfair, the strength of the Hill cipher is that it completely hides single-letter frequencies. Indeed, with Hill, the use of a larger matrix hides more fre-quency information. Thus, a 3 * 3 Hill cipher hides not only single-letter but also two-letter frequency information.

Although the Hill cipher is strong against a ciphertext-only attack, it is easily broken with a known plaintext attack. For an *m* * *m* Hill cipher, suppose we have *m* plaintext–ciphertext pairs, each of length *m*. We label the pairs **P**j = (*p*1*j p*1*j* ..... *pmj*) and **C**j = (*c*1*j c*1*j* ..... *cmj*) such that **C***j* = **P***j***K** for 1 … *j* … *m* and for some unknown key matrix **K**. Now define two *m* * *m* matrices **X** = (*pij*) and **Y** = (*cij*) . Then we can form the matrix equation **Y** = **XK**. If **X** has an inverse, then we can determine **K** = **X** - 1**Y**. If **X** is not invertible, then a new version of **X** can be formed with additional plaintext–ciphertext pairs until an invertible **X** is obtained.

Consider this example. Suppose that the plaintext “hillcipher” is encrypted using a 2 * 2 Hill cipher to yield the ciphertext HCRZSSXNSP. Thus, we know that
(7 8)**K** mod 26 = (7 2); (11 11)**K** mod 26 = (17 25); and so on. Using the first two plaintext–ciphertext pairs, we have

$$\begin{pmatrix} 7 & 2 \\ 17 & 25 \end{pmatrix} = \begin{pmatrix} 7 & 8 \\ 11 & 11 \end{pmatrix} \mathbf{K} \bmod 26$$

The inverse of **X** can be computed:

$$\begin{pmatrix} 7 & 8 \\ 11 & 11 \end{pmatrix}^{-1} = \begin{pmatrix} 25 & 22 \\ 1 & 23 \end{pmatrix}$$

$$\mathbf{K} = \begin{pmatrix} 25 & 22 \\ 1 & 23 \end{pmatrix} \begin{pmatrix} 7 & 2 \\ 17 & 25 \end{pmatrix} = \begin{pmatrix} 549 & 600 \\ 398 & 577 \end{pmatrix} \bmod 26 = \begin{pmatrix} 3 & 2 \\ 8 & 5 \end{pmatrix}$$

This result is verified by testing the remaining plaintext–ciphertext pairs.

Polyalphabetic Ciphers

Another way to improve on the simple monoalphabetic technique is to use different monoalphabetic substitutions as one proceeds through the plaintext message. The general name for this approach is **polyalphabetic substitution cipher**. All these tech-niques have the following features in common:

1. A set of related monoalphabetic substitution rules is used.
2. A key determines which particular rule is chosen for a given transformation.

VIGENERE` CIPHER The best known, and one of the simplest, polyalphabetic ciphers is the Vigenère cipher. In this scheme, the set of related monoalphabetic substitution rules consists of the 26 Caesar ciphers with shifts of 0 through 25. Each cipher is denoted by a key letter, which is the ciphertext letter that substitutes for the plaintext letter a. Thus, a Caesar cipher with a shift of 3 is denoted by the key value *d*.

We can express the Vigenère cipher in the following manner. Assume a sequence of plaintext letters $P = p0, p1, p2, ..., pn - 1$ and a key consisting of the sequence of letters $K = k0, k1, k2, ...$ $, km - 1$, where typically $m < n$. The sequence of ciphertext letters $C = C0, C1, C2, ..., Cn - 1$ is calculated as follows:

$C = C0, C1, C2, ..., Cn - 1 = E(K, P) = E[(k0, k1, k2, ..., km - 1), (p0, p1, p2, ..., pn - 1)]$

$= (p0 + k0) \bmod 26, (p1 + k1) \bmod 26, ..., (pm - 1 + km - 1) \bmod 26,$
$(pm + k0) \bmod 26, (pm + 1 + k1) \bmod 26, ..., (p2m - 1 + km - 1) \bmod 26, ...$

Thus, the first letter of the key is added to the first letter of the plaintext, mod 26, the second letters are added, and so on through the first *m* letters of the plaintext. For the next *m* letters of the plaintext, the key letters are repeated. This process continues until all of the plaintext sequence is encrypted. A general equation of the encryption process is

$$C_i = (p_i + k_{i \bmod m}) \bmod 26 \qquad \textbf{(2.3)}$$

Compare this with Equation (2.1) for the Caesar cipher. In essence, each plaintext character is encrypted with a different Caesar cipher, depending on the corresponding key character. Similarly, decryption is a generalization of Equation (2.2):

$$p_i = (C_i - k_{i \bmod m}) \bmod 26 \qquad \textbf{(2.4)}$$

To encrypt a message, a key is needed that is as long as the message. Usually, the key is a repeating keyword. For example, if the keyword is *deceptive*, the message "we are discovered save yourself" is encrypted as

```
key:           deceptivedeceptivedeceptive
plaintext:     wearediscoveredsaveyourself
ciphertext:    ZICVTWQNGRZGVTWAVZHCQYGLMGJ
```

Expressed numerically, we have the following result.

key	3	4	2	4	15	19	8	21	4	3	4	2	4	15
plaintext	22	4	0	17	4	3	8	18	2	14	21	4	17	4
ciphertext	25	8	2	21	19	22	16	13	6	17	25	6	21	19

key	19	8	21	4	3	4	2	4	15	19	8	21	4
plaintext	3	18	0	21	4	24	14	20	17	18	4	11	5
ciphertext	22	0	21	25	7	2	16	24	6	11	12	6	9

The strength of this cipher is that there are multiple ciphertext letters for each plaintext letter, one for each unique letter of the keyword. Thus, the letter frequency information is obscured. However, not all knowledge of the plaintext structure is lost. For example, Figure 2.6 shows the frequency distribution for a Vigenère cipher with a keyword of length 9. An improvement is achieved over the Playfair cipher, but considerable frequency information remains.

It is instructive to sketch a method of breaking this cipher, because the method reveals some of the mathematical principles that apply in cryptanalysis.

First, suppose that the opponent believes that the ciphertext was encrypted using either monoalphabetic substitution or a Vigenère cipher. A simple test can be made to make a determination. If a monoalphabetic substitution is used, then the statistical properties of the ciphertext should be the same as that of the language of the plaintext. Thus, referring to Figure 2.5, there should be one cipher letter with a relative frequency of occurrence of about 12.7%, one with about 9.06%, and so on. If only a single message is available for analysis, we would not expect an exact match of this small sample with the statistical profile of the plain-text language. Nevertheless, if the correspondence is close, we can assume a monoalphabetic substitution.

If, on the other hand, a Vigenère cipher is suspected, then progress depends on determining the length of the keyword, as will be seen in a moment. For now, let us con-centrate on how the keyword length can be determined.The important insight that leads to a solution is the following: If two identical sequences of plaintext letters occur at a dis-tance that is an integer multiple of the keyword length, they will generate identical ciphertext sequences. In the foregoing example, two instances of the sequence "red" are separated by nine character positions. Consequently, in both cases, r is encrypted using key letter *e*, e is encrypted using key letter *p*, and d is encrypted using key letter *t*. Thus, in both cases, the ciphertext sequence is VTW. We indicate this above by underlining the relevant ciphertext letters and shading the relevant ciphertext numbers.
An analyst looking at only the ciphertext would detect the repeated sequences VTW at a displacement of 9 and make the assumption that the keyword is either three or nine letters in length. The appearance of VTW twice could be by chance and not reflect identical plaintext letters encrypted with identical key letters. However, if the message is long enough, there will be a number of such repeated ciphertext sequences. By looking for common factors in the displacements of the various sequences, the analyst should be able to make a good guess of the keyword length.

Solution of the cipher now depends on an important insight. If the keyword length is *m*, then the cipher, in effect, consists of *m* monoalphabetic substitution ciphers. For example, with the keyword DECEPTIVE, the letters in positions 1, 10, 19, and so on are all encrypted with the same monoalphabetic cipher. Thus, we can use the known frequency characteristics of the plaintext language to attack each of the monoalphabetic ciphers separately.

The periodic nature of the keyword can be eliminated by using a nonrepeating keyword that is as long as the message itself. Vigenère proposed what is referred to as an **autokey system**, in which a keyword is concatenated with the plaintext itself to provide a running key. For our example,

key:	*deceptivewearediscoveredsav*
plaintext:	wearediscoveredsaveyourself
ciphertext:	ZICVTWQNGKZEIIGASXSTSLVVWLA

Even this scheme is vulnerable to cryptanalysis. Because the key and the plain-text share the same frequency distribution of letters, a statistical technique can be applied. For example, e enciphered by *e*, by Figure 2.5, can be expected to occur with a frequency of (0.127)2 L 0.016, whereas t enciphered by *t* would occur only about half as often. These regularities can be exploited to achieve successful cryptanalysis.

VERNAM CIPHER The ultimate defense against such a cryptanalysis is to choose a keyword that is as long as the plaintext and has no statistical relationship to it. Such a system was introduced by an AT&T engineer named Gilbert Vernam in 1918. His system works on binary data (bits) rather than letters. The system can be expressed succinctly as follows (Figure 2.7):

$$c_i = p_i \oplus k_i$$

where

p_i = ith binary digit of plaintext
k_i = ith binary digit of key
c_i = ith binary digit of ciphertext
$\oplus$ = exclusive-or (XOR) operation

where

pi = i th binary digit of plaintext ki = ith binary digit of key
ci = ith binary digit of ciphertext

Compare this with Equation (2.3) for the Vigenère cipher.

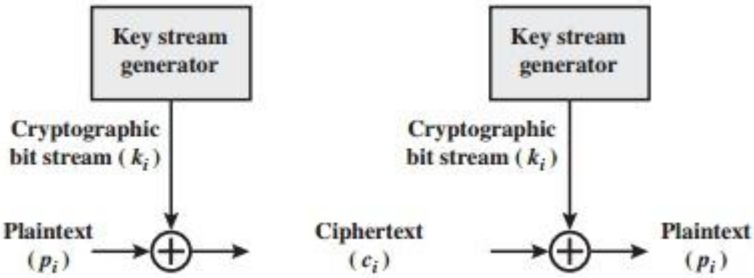


Figure 2.7 Vernam Cipher

Thus, the ciphertext is generated by performing the bitwise XOR of the plain-text and the key. Because of the properties of the XOR, decryption simply involves the same bitwise operation:

$$p_i = c_i \oplus k_i$$

which compares with Equation (2.4).

The essence of this technique is the means of construction of the key. Vernam proposed the use of a running loop of tape that eventually repeated the key, so that in fact the system worked with a very long but repeating keyword. Although such a scheme, with a long key, presents formidable cryptanalytic diffi-culties, it can be broken with sufficient ciphertext, the use of known or probable plaintext sequences, or both.

One-Time Pad

An Army Signal Corp officer, Joseph Mauborgne, proposed an improvement to the Vernam cipher that yields the ultimate in security. Mauborgne suggested using a random key that is as long as the message, so that the key need not be repeated. In addition, the key is to be used to encrypt and decrypt a single message, and then is discarded. Each new message requires a new key of the same length as the new mes-sage. Such a scheme, known as a **one-time pad**, is unbreakable. It produces random output that bears no statistical relationship to the plaintext. Because the ciphertext contains no information whatsoever about the plaintext, there is simply no way to break the code.

An example should illustrate our point. Suppose that we are using a Vigenère scheme with 27 characters in which the twenty-seventh character is the space character, but with a one-time key that is as long as the message. Consider the ciphertext

ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

We now show two different decryptions using two different keys:

ciphertext: ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

key: *pxlmvmsydofuyrvzwc tnlebnecvgdupahfzzlmnyih*
plaintext: mr mustard with the candlestick in the hall
ciphertext: ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

key: *mfugpmiydgaxgoufhklllmhsqdqogtewbqfgyovuhwt*
plaintext: miss scarlet with the knife in the library

Suppose that a cryptanalyst had managed to find these two keys. Two plausible plaintexts are produced. How is the cryptanalyst to decide which is the correct decryption (i.e., which is the correct key)? If the actual key were produced in a truly random fashion, then the cryptanalyst cannot say that one of these two keys is more likely than the other. Thus, there is no way to decide which key is correct and there-fore which plaintext is correct.

In fact, given any plaintext of equal length to the ciphertext, there is a key that produces that plaintext. Therefore, if you did an exhaustive search of all possible keys, you would end up with many legible plaintexts, with no way of knowing which was the intended plaintext. Therefore, the code is unbreakable.

The security of the one-time pad is entirely due to the randomness of the key. If the stream of characters that constitute the key is truly random, then the stream of characters that constitute the ciphertext will be truly random. Thus, there are no patterns or regularities that a cryptanalyst can use to attack the ciphertext.

In theory, we need look no further for a cipher. The one-time pad offers com-plete security but, in practice, has two fundamental difficulties

1. There is the practical problem of making large quantities of random keys. Any heavily used system might require millions of random characters on a regular basis. Supplying truly random characters in this volume is a significant task.
2. Even more daunting is the problem of key distribution and protection. For every message to be sent, a key of equal length is needed by both sender and receiver. Thus, a mammoth key distribution problem exists.

Because of these difficulties, the one-time pad is of limited utility and is useful primarily for low-bandwidth channels requiring very high security.

The one-time pad is the only cryptosystem that exhibits what is referred to as *perfect secrecy*.

1.9 TRANSPOSITION TECHNIQUES

All the techniques examined so far involve the substitution of a cipher text symbol for a plaintext symbol. A very different kind of mapping is achieved by performing some sort of permutation on the plaintext letters. This technique is referred to as a transposition cipher.

The simplest such cipher is the **rail fence** technique, in which the plaintext is written down as a sequence of diagonals and then read off as a sequence of rows. For example, to encipher the message “meet me after the toga party” with a rail fence of depth 2, we write the following:

m e m a t r h t g p r y
e t e f e t e o a a t
The encrypted message is

MEMATRHTGPRYETEFETEOAAT

This sort of thing would be trivial to cryptanalyze. A more complex scheme is to write the message in a rectangle, row by row, and read the message off, column by column, but permute the order of the columns. The order of the columns then becomes the key to the algorithm. For example,

```
Key:          4 3 1 2 5 6 7
Plaintext:    a t t a c k p
              o s t p o n e
              d u n t i l t
              w o a m x y z
Ciphertext:   TTNAAPTMTSUOAODWCOIXKNLYPETZ
```

Thus, in this example, the key is 4312567. To encrypt, start with the column that is labeled 1, in this case column 3. Write down all the letters in that column. Proceed to column 4, which is labeled 2, then column 2, then column 1, then columns 5, 6, and 7.

A pure transposition cipher is easily recognized because it has the same letter frequencies as the original plaintext. For the type of columnar transposition just shown, cryptanalysis is fairly straightforward and involves laying out the ciphertext in a matrix and playing around with column positions. Digram and trigram frequency tables can be useful.

The transposition cipher can be made significantly more secure by performing more than one stage of transposition. The result is a more complex permutation that is not easily reconstructed. Thus, if the foregoing message is reencrypted using the same algorithm,

```
Key:       4 3 1 2 5 6 7
Input:     t t n a a p t
           m t s u o a o
           d w c o i x k
           n l y p e t z
Output:    NSCYAUOPTTWLTMDNAOIEPAXTTOKZ
```

To visualize the result of this double transposition, designate the letters in the original plaintext message by the numbers designating their position. Thus, with 28 letters in the message, the original sequence of letters is

01 02 03 04 05 06 07 08 09 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28

After the first transposition, we have

03 10 17 24 04 11 18 25 02 09 16 23 01 08
15 22 05 12 19 26 06 13 20 27 07 14 21 28

which has a somewhat regular structure. But after the second transposition, we have

17 09 05 27 24 16 12 07 10 02 22 20 03 25
15 13 04 23 19 14 11 01 26 21 18 08 06 28

This is a much less structured permutation and is much more difficult to cryptanalyze.

1.10 ROTOR MACHINES

This is as true of substitution ciphers as it is of transposition ciphers. Before the introduction of DES, the most important application of the principle of multiple stages of encryption was a class of systems known as rotor machines.9

The basic principle of the rotor machine is illustrated in Figure 2.8. The machine consists of a set of independently rotating cylinders through which electrical pulses can flow. Each cylinder has 26 input pins and 26 output pins, with internal wiring that connects each input pin to a unique output pin. For simplicity, only three of the internal connections in each cylinder are shown.

If we associate each input and output pin with a letter of the alphabet, then a single cylinder defines a monoalphabetic substitution. For example, in Figure 2.8, if an operator depresses the key for the letter A, an electric signal is applied to the first pin of the first cylinder and flows through the internal connection to the twenty-fifth output pin.

Consider a machine with a single cylinder. After each input key is depressed, the cylinder rotates one position, so that the internal connections are shifted accordingly. Thus, a different monoalphabetic substitution cipher is defined. After 26 letters of plaintext, the cylinder would be back to the initial position. Thus, we have a poly-alphabetic substitution algorithm with a period of 26.

A single-cylinder system is trivial and does not present a formidable cryptana-lytic task. The power of the rotor machine is in the use of multiple cylinders, in which the output pins of one cylinder are connected to the input pins of the next. Figure 2.8 shows a three-cylinder system. The left half of the figure shows a position in which the input from the operator to the first pin (plaintext letter a) is routed through the three cylinders to appear at the output of the second pin (ciphertext letter B).

With multiple cylinders, the one closest to the operator input rotates one pin position with each keystroke. The right half of Figure 2.8 shows the system's config-uration after a single keystroke. For every complete rotation of the inner cylinder, the middle cylinder rotates one pin position. Finally, for every complete rotation of the middle cylinder, the outer cylinder rotates one pin position. This is the same type of operation seen with an odometer. The result is that there are 26 * 26 * 26 = 17,576 different substitution alphabets used before the system repeats. The addition of fourth and fifth rotors results in periods of 456,976 and 11,881,376 letters, respectively. As David Kahn eloquently put it, referring to a five-rotor machine [KAHN96,]:

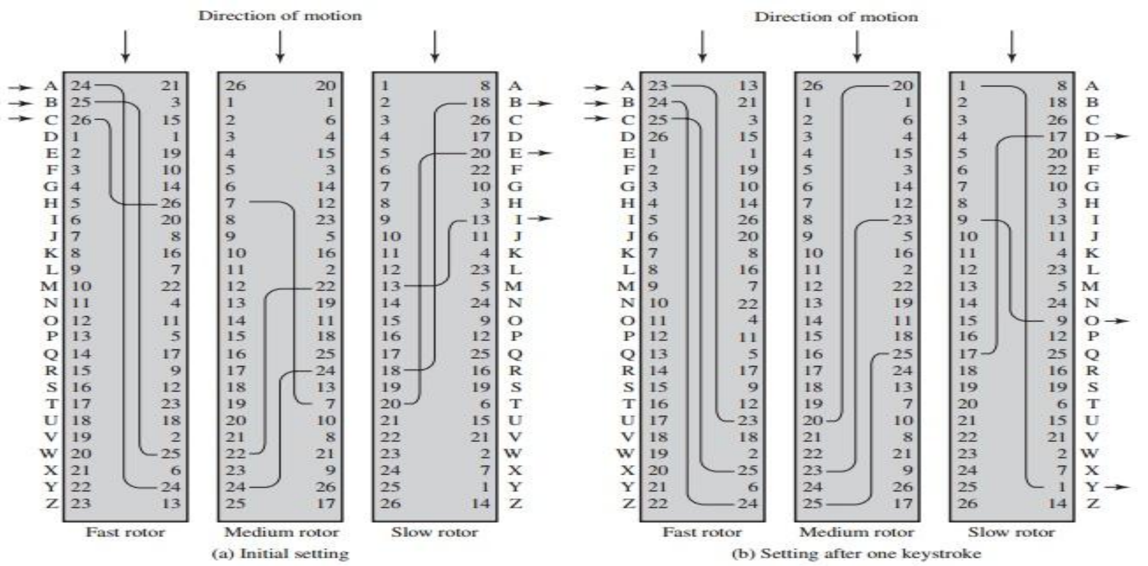


Figure 2.8 Three-Rotor Machine with Wiring Represented by Numbered Contacts

A period of that length thwarts any practical possibility of a straightforward solution on the basis of letter frequency. This general solution would need about 50 letters per cipher alphabet, meaning that all five rotors would have to go through their com-bined cycle 50 times. The ciphertext would have to be as long as all the speeches made on the floor of the Senate and the House of Representatives in three successive sessions of Congress. No cryptanalyst is likely to bag that kind of trophy in his lifetime; even diplomats, who can be as verbose as politicians, rarely scale those heights of loquacity.

1.11 STEGANOGRAPHY

A plaintext message may be hidden in one of two ways. The methods of **steganography** conceal the existence of the message, whereas the methods of cryptography render the message unintelligible to outsiders by various transformations of the text.

A simple form of steganography, but one that is time-consuming to construct, is one in which an arrangement of words or letters within an apparently innocuous text spells out the real message. For example, the sequence of first letters of each word of the overall message spells out the hidden message. Figure 2.9 shows an example in which a subset of the words of the overall message is used to convey the hidden message. See if you can decipher this; it's not too hard.

- **Character marking:** Selected letters of printed or typewritten text are over-written in pencil. The marks are ordinarily not visible unless the paper is held at an angle to bright light.
- **Invisible ink:** A number of substances can be used for writing but leave no visible trace until heat or some chemical is applied to the paper.
- **Pin punctures:** Small pin punctures on selected letters are ordinarily not visi-ble unless the paper is held up in front of a light.
- **Typewriter correction ribbon:** Used between lines typed with a black ribbon, the results of typing with the correction tape are visible only under a strong light.

Advantage: It can be employed by parties who have something to lose should the fact of their secret communication be discovered.

Drawbacks: it requires lot of overhead to hide a relatively few bits of information.

UNIT II -BLOCK CIPHER AND DATA ENCRYPTION STANDARD

2.1 Block cipher principle

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. In the ideal case, a one-time pad version of the Vernam cipher would be used (Figure 2.7), in which the keystream (ki) is as long as the plaintext bit stream (pi). If the cryptographic keystream is random, then this cipher is unbreakable by any means other than acquiring the keystream. However, the keystream must be provided to both users in advance via some independent and secure channel. This introduces insurmountable logistical problems if the intended data traffic is very large.

Accordingly, for practical reasons, the bit-stream generator must be implemented as an algorithmic procedure, so that the cryptographic bit stream can be produced by both users. In this approach (Figure 3.1a), the bit-stream generator is a key-controlled algorithm and must produce a bit stream that is cryptographically strong. Now, the two users need only share the generating key, and each can produce the keystream.

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. As with a stream cipher, the two users share a symmetric encryption key (Figure 3.1b). Using some of the modes of operation explained in Chapter 6, 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 primarily 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 2n possible different plaintext blocks and, for the encryption to be reversible (i.e., for decryption to be possible), each must

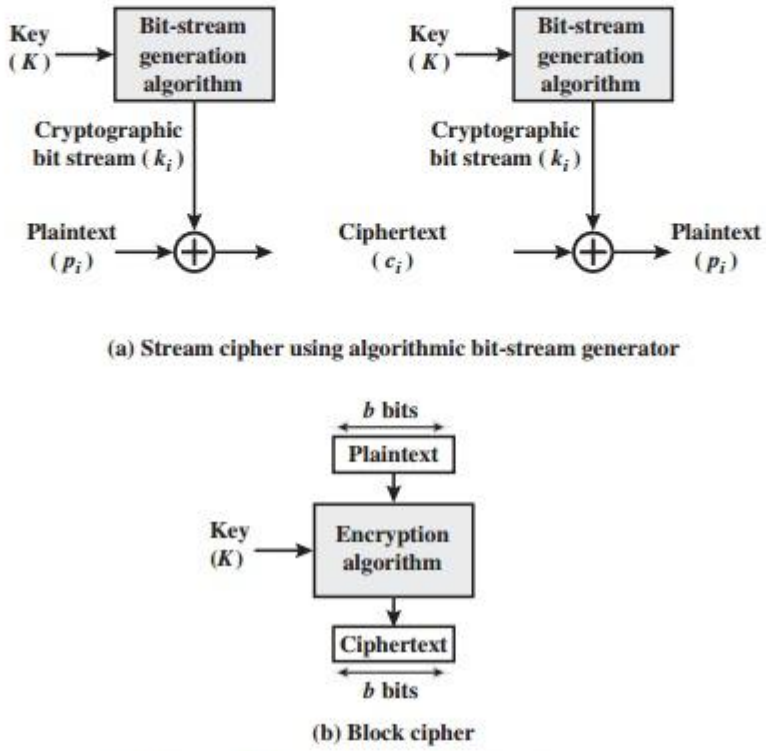


Figure 3.1 Stream Cipher and Block Cipher

produce a unique ciphertext block. Such a transformation is called reversible, or nonsingular. The following examples illustrate nonsingular and singular transfor- mations 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 plain-text blocks. So if we limit ourselves to reversible mappings, the number of different transformations is $2n!.2$

Figure 3.2 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

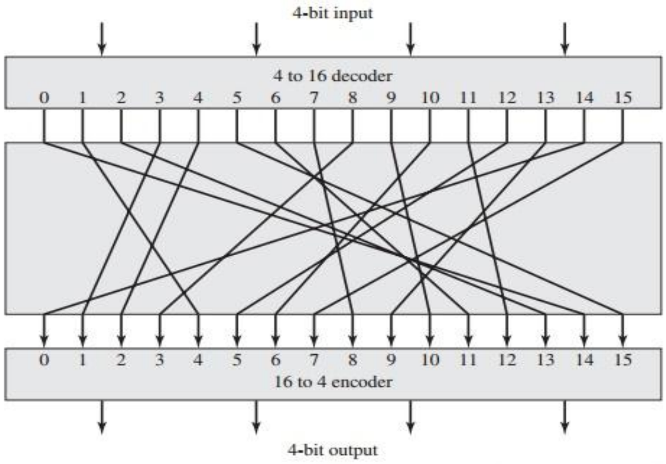


Figure 3.2 General n-bit-n-bit Block Substitution (shown with $n = 4$)

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 [FEIS75].

Table 3.1 Encryption and Decryption Tables for Substitution Cipher of Figure 3.2

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
1100	0101
1101	1001
1110	0000
1111	0111

Ciphertext	Plaintext
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 syste ms, as we have seen, are vulnerable to a statistical analysis of the plaintext. This weakness is not inh erent 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) * (16 rows) = 64 bits.

In general, foran n-bitideal block cipher, the length of the key defined in this fashion is $n * 2n$ bits. For a 64-bit block, which is a desirable length to thwart statistical attacks, the required key length is 64 * 264 = 270 ~~ 1021 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 compo- nents that are easily realizable [FEIS75]. 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 th e $2n!$ 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

$y1 = k11x1 + k12x2 + k13x3 + k14x4$ $y2 = k21x1 + k22x2 + k23x3 + k24x4$ $y3 = k31x1 + k32x2 + k33x3 + k34x4$ $y4 = k41x1 + k42x2 + k43x3 + k44x4$

where the xi are the four binary digits of the plaintext block, the yi are the four binary digits of the ciphertext block, the kij are the binary coefficients, and arith- metic is mod 2. The key size is just $n2$, 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. In this example, what we have is essentially the Hill cipher discussed in Chapter 2, applied to binary data rather than characters. As we saw in Chapter 2, a simple linear system such as this is quite vulnerable.

The Feistel Cipher

Feistel proposed [FEIS73] 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 $2k$ possible transfor mations, rather than the $2n!$ transformations available with the ideal block cipher.

In particular, Feistel proposed the use of a cipher that alternates substitutions and permutations, where these terms are defined as follows:

Substitution: Each plaintext element or group of elements is uniquely replaced by a

corresponding ciphertext element or group of elements.

Permutation: A sequence of plaintext elements is replaced by a permutation of that sequence. That is, no elements are added or deleted or replaced in the sequence, rather the order in which the elements appear in the sequence is changed.

In fact, Feistel's is a practical application of a proposal by Claude Shannon to develop a product cipher that alternates *confusion* and *diffusion* functions

DIFFUSION AND CONFUSION The terms *diffusion* and *confusion* were introduced by Claude Shannon to capture the two basic building blocks for any cryptographic system [SHAN49]. Shannon's concern was to thwart cryptanalysis based on statistical analysis. The reasoning is as follows. Assume the attacker has some knowledge of the statisticalcharacteristics of the plaint ext. 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. If these statistics are in any way reflected in the ciphertext, the cryptanalyst may be able to deduce the encryption key, 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.

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 plainte xt digits. An example of diffusion is to encrypt a message $M = m1, m2, m3, \ldots$ of characters with an averaging operation:

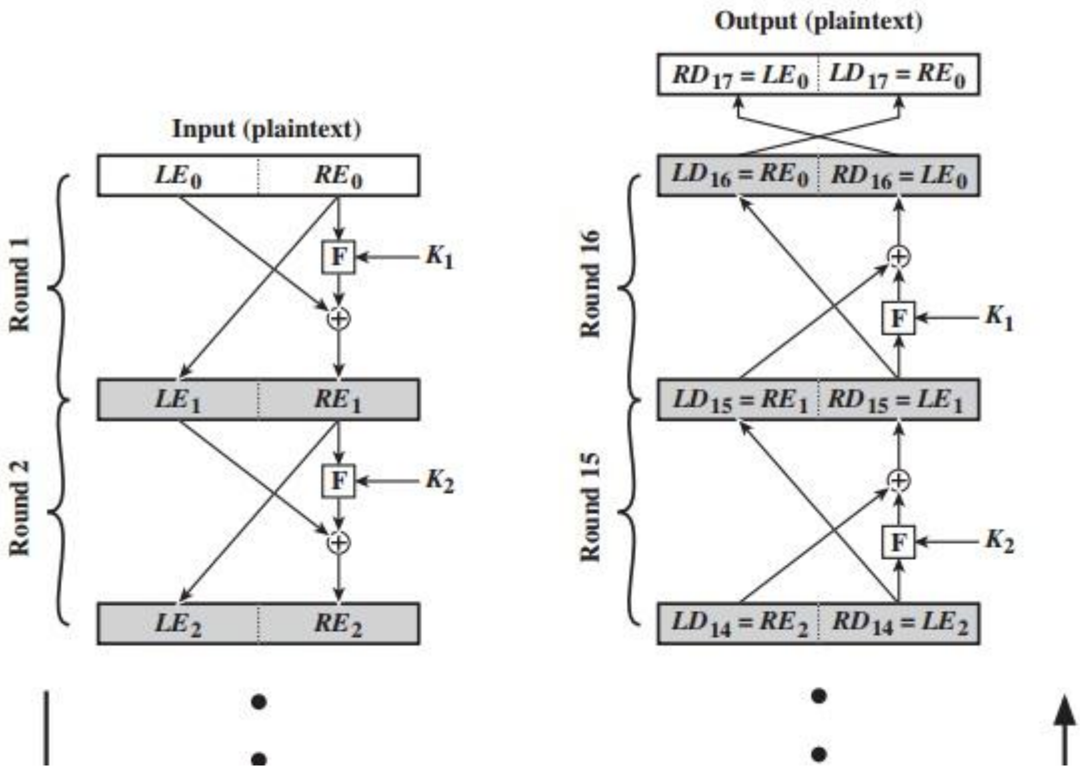
$$y_n = \left(\sum_{i=1}^{k} m_{n+i} \right) \bmod 26$$

adding k successive letters to get a ciphertext letter yn. One can show that the statis-
tical 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 repe atedly 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.5

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.

FEISTEL CIPHER STRUCTURE The left-hand side of Figure 3.3 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, $L0$ and $R0$. 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 $Li - 1$ and $Ri - 1$ derived from the previous round, as well as a subkey Ki derived from the overall K. In general, the subkeys Ki are different from K and from each other. In Figure 3.3, 16 rounds are used, although any number of rounds could be implemented.

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 Ki. Another way to express this is to say that F is a function of right-half block of w bits and a subkey of y bits, which produces an output value of length w bits: $F(REi, Ki + 1)$. Following this substitution, a



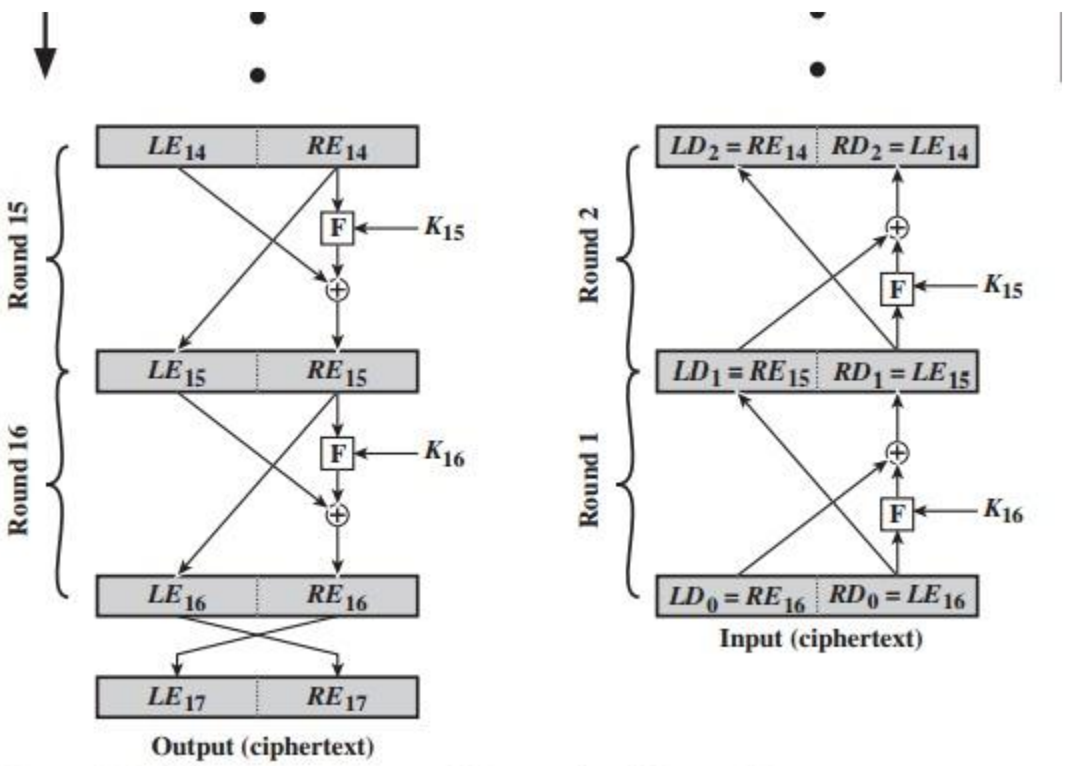


Figure 3.3 Feistel Encryption and Decryption (16 rounds)

permutation is performed that consists of the interchange of the two halves of the data.6 This structure is a particular form of the substitution-permutation network (SPN) proposed by Shannon.

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 F: 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 Ki in reverse order. That is, use Kn in the first round, $Kn - 1$ in the second round, and so on, until $K1$ 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, Figure 3.3 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. For clarity, we use the notation *LEi* and *REi* for data traveling through the encryption algorithm and *LDi* and *RDi* 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 *i*th encryption round be *LEi* 7 *REi* (*LEi* concatenated with *REi*). Then the corresponding output of the (16 – *i*)th decryption round is *REi* 7 *LEi* or, equivalently, *LD*16 - *i* 7 *RD*16 - *i*. 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 7 *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 7 *LE*16, which is equal to the 32-bit swap of the output of the sixteenth round of the encryption process.

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

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

On the decryption side,

$$LD_1 = RD_0 = LE_{16} = RE_{15}$$
$$RD_1 = LD_0 \oplus F(RD_0, K_{16})$$
$$= RE_{16} \oplus F(RE_{15}, K_{16})$$
$$= [LE_{15} \oplus F(RE_{15}, K_{16})] \oplus F(RE_{15}, K_{16})$$

The XOR has the following properties:

$$[A \oplus B] \oplus C = A \oplus [B \oplus C]$$
$$D \oplus D = 0$$
$$E \oplus 0 = E$$

Thus,we have *LD*1 = *RE*15 and *RD*1 = *LE*15. Therefore, the output of the first round of the decryption process is *RE*15 7 *LE*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 *i*th iteration of the encryption algorithm,

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

Rearranging terms:

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

Thus, we have described the inputs to the *i*th iteration as a function of the outputs, and these equations c onfirm 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.

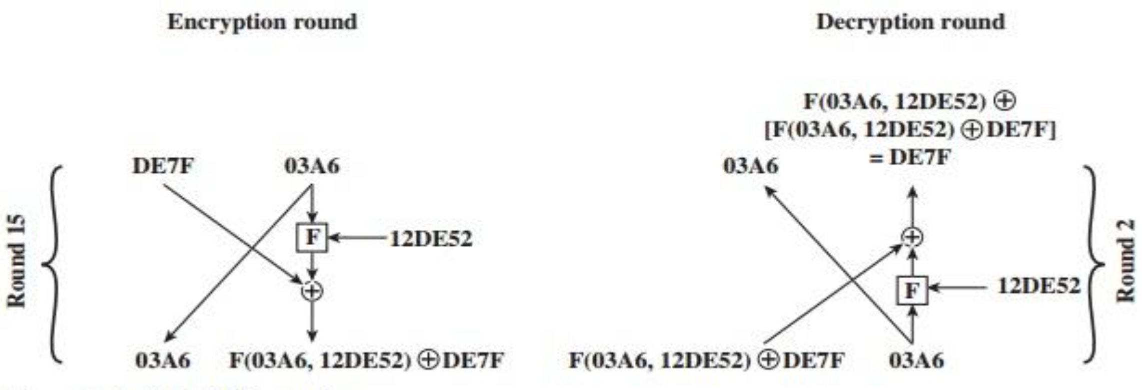


Figure 3.4 Feistel Example

To help clarify the preceding concepts, let us look at a specific example (Figure 3.4) and focus on the fifteenth round of encryption,corresponding to the second round of decryption. Suppose that the blocks at each stage are 32 bits (two 16bit halves) and that the key size is 24 bits. Suppose that at the end ofencryption round fourteen, the value of the intermediate block (in hexadecimal) is DE7F03A6. Then *LE*14 = DE7F and *RE*14 = 03A6. Also assume that the value of *K*15 is 12DE52. After round 15,

we have *LE*15 = 03A6 and *RE*15 = F(03A6, 12DE52) NOR DE7F.

Now let's look at the decryption. We assume that LD1 = RE15 and RD1 = LE15, as shown in Figure 3.3, and we want to demonstrate that LD2 = RE14 and RD2 = LE14. So, we start with LD1 = F(03A6, 12DE52) NOR DE7F and RD1 = 03A6.

Then, from Figure 3.3, LD2 = 03A6 = RE14 and RD2 = F(03A6, 12DE52) NOR [F(03A6, 12DE52) { DE7F]= DE7F = LE_{14}.

2.2 The Data Encryption Standard(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 [FEIS73], 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 the National Security Agency (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, in order 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 no t 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.

DES Encryption

The overall scheme for DES encryption is illustrated in Figure 3.5. 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.

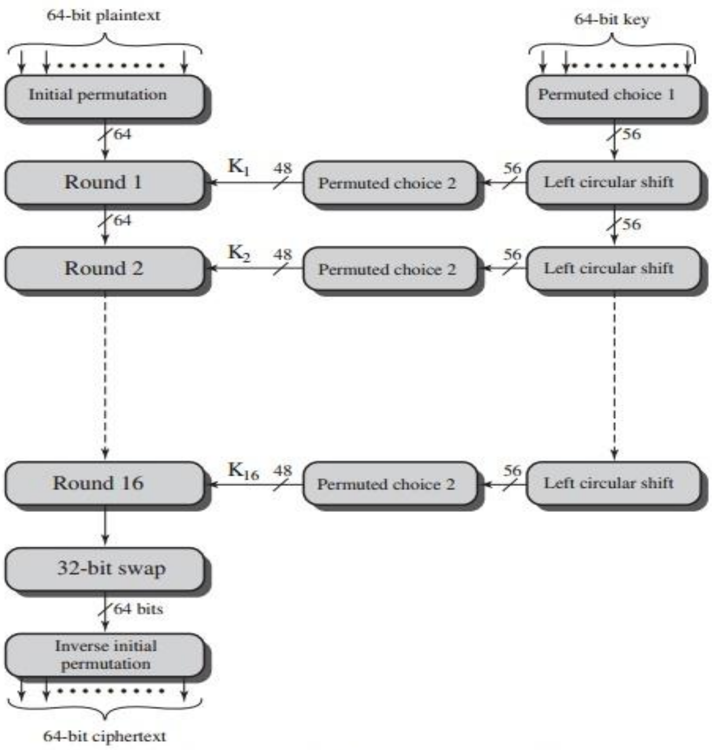


Figure 3.5 General Depiction of DES Encryption Algorithm

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 sixteen 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 3.3.

The right-hand portion of Figure 3.5 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 sixteen rounds, a *subkey* (*Ki*) 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

Table 3.2 Permutation Tables for DES

(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

(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

entry in the permutation table indicates the position of a numbered input bit in the output, which also consists of 64 bits.

To see that these two permutation functions are indeed the inverse of each other, consider the following 64-bit input *M*:

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}

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 3.6 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:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

The round key *Ki* 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 *Ki* . 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 3.7. 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 *Si* form a 2bit binary number to select o ne of four substitutions defined by the four rows in the table for *Si*. The middle four bits select one of th e sixteen columns. The decimal value in the cell selected by the row and column is then converted to i ts 4bit representation to produce the output. For example, in S1, 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.

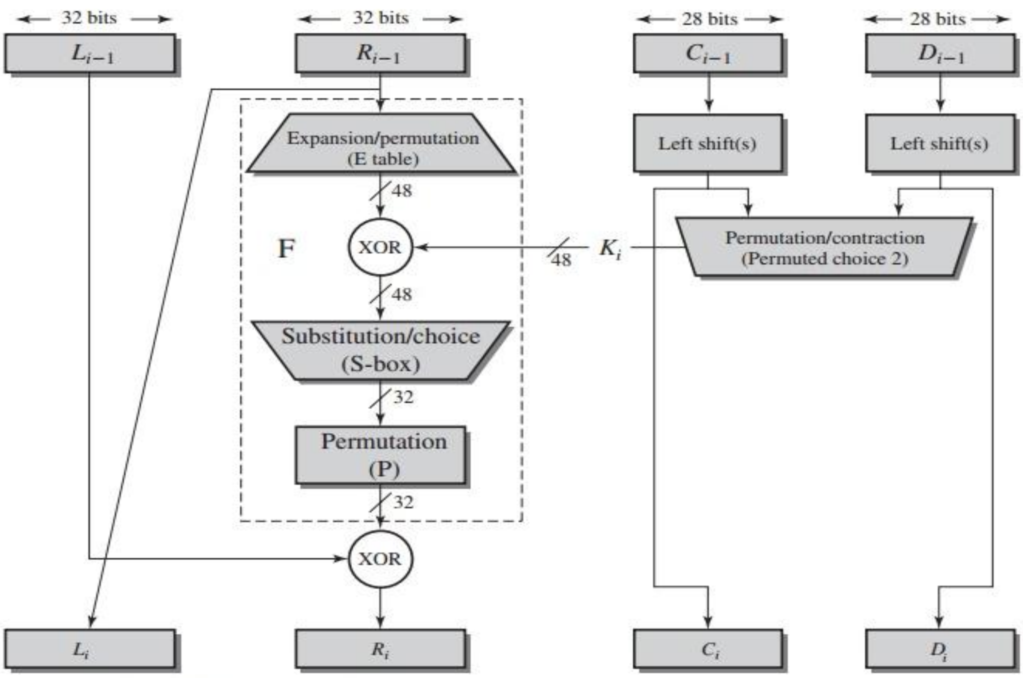


Figure 3.6 Single Round of DES Algorithm

Each row of an S-box defines a general reversible substitution. Figure 3.2 may be useful in understanding the mapping. The figure shows the substitution for row 0 of box S1.

The operation of the S-boxes is worth further comment. Ignore for the momentthe contribution of the key (*Ki*).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 ...

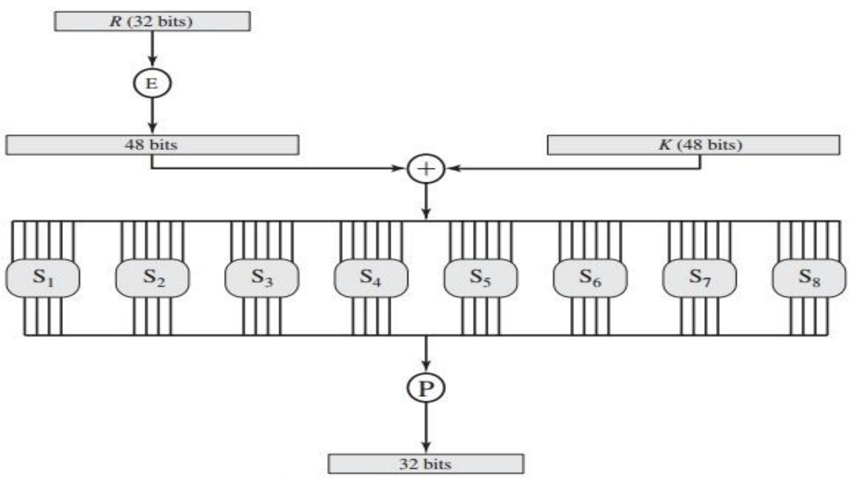


Figure 3.7 Calculation of F(R, K)

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 3.5 and 3.6, 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 C0 and D0. 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 the part labeled Permuted Choice Two (Table 3.4c), which produces a 48-bit output that serves as input to the function $F(R_{i-1}, K_i)$.

DES Decryption

As with any Feistel cipher, decryption uses the same algorithm as encryption, except that the application of the subkeys is reversed.

Table 3.3 Definition of DES S-Boxes

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

	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
S_5	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

	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
S_6	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

	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
S_7	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

	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
S_8	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

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
Bits Rotated	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

2.3 THE STRENGTH OF DES

Since its adoption as a federal standard, there have been lingering concerns about the level of security provided by DES. These concerns, by and large, fall into two areas: key size and the nature of the algorithm.

The Use of 56-Bit Keys

With a key length of 56 bits, there are 256 possible keys, which is approximately 7.2 * 1016 keys. Thus, on the face of it, a brute-force attack appears impractical. Assuming that, on average, half the key space has to be searched, a single machine performing one DES encryption per microsecond would take more than a thousand years (see Table 2.2) to break the cipher.However, the assumption of one encryption per microsecond is overly conservative. As far back as 1977, Diffie and Hellman postulated that the technology existed to build a parallel machine with 1 million encryption devices, each of which could perform one encryption per microsecond [DIFF77]. This would bring the average search time down to about 10 hours.

DES finally and definitively proved insecure in July 1998, when the Electronic Frontier Foundation (EFF) announced that it had broken a DES encryption using a special-purpose "DES cracker" machine that was built for less than $250,000. The attack took less than three days. The EFF has published a detailed description of the machine, enabling others tobuild their own cracker [EFF98]. And, of course, hardware prices will continue to drop as speeds increase, mak ing DES virtually worthless. It is important to note that there is more to a key-search attack than simply running through all possible keys. Unless known plaintext is provided, the analyst must be able to recognize plaintext as plaintext. If the message is just plain text in English, then the result pops out easily, although the task of recognizing English would have to be automated. If the text message has been compressed b efore encryption, then recognition is more difficult. And if the message is some more general type of data, such as a numerical file, and this has been compressed, the problem becomes even more difficult to automate. Thus, to supplement the brute-force approach, some degree of knowledge about the expected plaintext is needed, and some means of automatically distinguishing plaintext from garble is also needed.

The EFF approach addresses this issue as well and introduces some automated techniques that would be effective in many contexts.

The Nature of the DES Algorithm

Another concern is the possibility that cryptanalysis is possible by exploiting the characteristics of the DES algorithm. The focus of concern has been on the eight substitution tables, or S-boxes, that are used in each iteration. Because the design criteria for these boxes, and indeed for the entire algorithm, were not made public,there is a suspicion that the boxes were constructed in such a way th at cryptanalysis is possible for an opponent who knows the weaknesses in the S-boxes. This assertion is tantalizing, and over the years a number of regularities and unexpected behaviors of the S-boxes have been discovered. Despite this, no one has so far succeeded in discovering the supposed fatal weaknesses in the S-boxes.9

Timing Attacks

We discuss timing attacks in more detail in Part Two, as they relate to public-key algorithms. However, the issue may also berelevant for symmetric ciphers. In essence, a timi ng attack is onein which information about the key or the plaintextis obtained by observing how long it takes a given implementation to perform decryptions on various ciphertexts.

A timing attack exploits the fact that an encryption or decryption algorithm often takes slightly different amounts of time on different inputs.reports on an approach that yields the Hamming weight(number of bits equal to one) of the secret key. This is a long way from knowing the actual key, but it is an intriguing first step. The authors conclude that DES appears to be fairly resistant to a successful timing attack but suggest some avenues to explore. Although this is an interesting line of attack, it so far appears unlikely that this technique will ever be successful against DES or more powerful symmetric ciphers such as triple DES and AES.

2.4 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 findingcryptanalytic 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 [MURP90]. This was followed by
a number of papers by Biham and Shamir, who demonstrated this form of attack on a variety of encr yption algorithms and hash functions; their results are summarized in [BIHA93].

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 255 encryptions. The scheme, as reported in [BIHA93], can successfully cryptanalyze DES with an effort on the order of 247 encryptions, requiring 247 chosen plaintexts. Although 247 is certainly significantly less than 255, the need for the adversary to find 247 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 [COPP94], 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 [BIHA93]. Differential cryptanalysis of an eight-round LUCIFER algorithm requires only 256 chosen plaintexts, whereas an attack on an eight-round version of DES requires 214 chosen plaintexts.

DIFFERENTIAL CRYPTANALYSIS ATTACK The differential cryptanalysis attack is complex; [BIHA93] provides a complete description. The rationale behind differential cryptanalysis is to observe the behavior of pairs of text blocks evolving along eac h 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 *mi* (2 … *i* … 17), then the intermediate message halves are related as follows:

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

In differential cryptanalysis, we start with two messages, *m* and *m*¿, with a known XOR differe nce Δ*m* = *m* NOR *m*¿, and consider the difference between the intermediate message halves: Δ*mi* = *mi* NOR *m*¿*i*. Then we have

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: $\Delta m_i = m_i \oplus m'_i$. Then we have

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

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 ¢*mi* - 1 and ¢*mi* with high probability, then we know ¢*mi* + 1 with high probability.

Furthermore, if a number of such differences aredetermined, it is feasible to determine the subkey use d in the function f.

Theoverallstrategyofdifferentialcryptanalysisisbasedontheseconsiderations for a single round. The procedure is to begin with two plaintext messages *m* and *m¿* with a given difference and trace throu gh a probable pattern of differences after each
round to yield a probable difference for the ciphertext. Actually, there are two proba-
blepatternsofdifferencesforthetwo32-
bithalves:(¢*m*17 || ¢*m*16).Next,wesubmit*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,

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

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 3.8, based on a figure in [BIHA93], illustrates the propagation of differ- ences 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

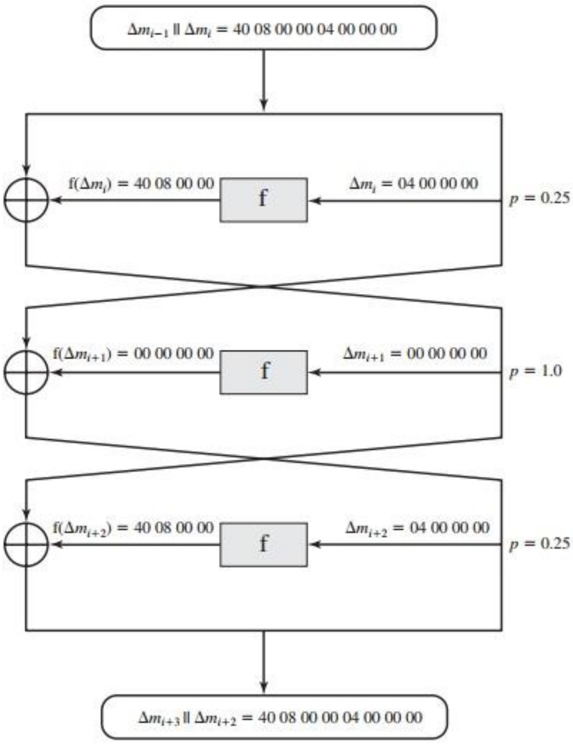


Figure 3.8 Differential Propagation through Three Rounds of DES (numbers in hexadecimal)

of the input differences. Overall, after three rounds, the probability that the output difference is as shown is equal to 0.25 * 1 * 0.25 = 0.0625.

Linear Cryptanalysis

A more recent development is linear cryptanalysis, described in [MATS93]. This attack is based on findin g linear approximations to describe the transformations performed in DES. This method can find a DES k ey given 243 known plaintexts, as compared to 247 chosen plaintexts for differential cryptanalysis. Alth ough 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

$$A[i, j, \ldots, k] = A[i] \oplus A[j] \oplus \ldots \oplus A[k]$$

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

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

(where *x* = 0 or 1; 1 <= *a*; *b* <= *n*; *c* <= *m*; and where the *a*, *b*, and *g* terms represent fixed, unique bit locations) that holds with probability *p* != 0.5. The further *p* is from 0.5, the more effective the equation. Once a proposed relation is determined, the pro- cedure 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[*g*1, *g*2, ….. , *gc*] = 0. If it is 1 most of the time, assume K[*g*1, *g*2, ….. , *gc*] = 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.

2.5 Block Cipher Design Principles

DES Design Criteria

The criteria used in the design of DES, as reported in [COPP94], focused on the design of the S-boxes and on the P function that takes the output of the S-boxes (Figure 3.7). The criteria for the S-boxes are as follows.

1. No output bit of any Sbox should be too close a linear function of the input bits. Sp ecifically, if we select any output bit and any subset of the six input bits, the fracton of inputs for which this output bit equals the XOR of these input bits should not be close to 0 or 1, but rather should be near 1/2.
2. Each row of an S-box (determined by a fixed value of the leftmost and right-most input bits) should include all 16 possible output bit combinations.
3. If two inputs to an S-box differ in exactly one bit, the outputs must differ in at least two bits.

4. If two inputs to an S-box differ in the two middle bits exactly, the outputs must differ in at least two bits.
5. If two inputs to an S-box differ in their first two bits and are identical in their last two bits, the two outputs must not be the same.
6. For any nonzero 6-bit difference between inputs, no more than eight of the 32 pairs of inputs exhibiting that difference may result in the same output difference.
7. This is a criterion similar to the previous one, but for the case of three S-boxes.

Coppersmith pointed out that the first criterion in the preceding list was needed because the S-boxes are the only nonlinear part of DES. If the S-boxes were linear (i.e., each output bit is a linear combination of the input bits), the entire algorithm would be linear and easily broken. We have seen this phenomenon with the Hill cipher, which is linear. The remaining criteria were primarily aimed at thwarting differential cryptanalysis and at providing good confusion properties.

The criteria for the permutation P are as follows.

1. The four output bits from each S-box at round *i* are distributed so that two of them affect (provide input for) "middle bits" of round (*i* + 1) and the other two affect end bits. The two middle bits of input to an S-box are not shared with adjacent S-boxes. The end bits are the two left-hand bits and the two right-hand bits, which are shared with adjacent S-boxes.
2. The four output bits from each S-box affect six different S-boxes on the next round, and no two affect the same S-box.
3. For two S-boxes *j*, *k*, if an output bit from S*j* affects a middle bit of S*k* on the next round, then an output bit from S*k* cannot affect a middle bit of S*j*. This implies that, for *j* = *k*, an output bit from S*j* must not affect a middle bit of S*j*.

These criteria are intended to increase the diffusion of the algorithm.

Number of Rounds

The cryptographic strength of a Feistel cipher derives from three aspects of the design: the number of rounds, the function F, and the key schedule algorithm. Let us look first at the choice of the number of rounds.

The greater the number of rounds, the more difficult it is to perform crypt- analysis, even for a relatively weak F. In general, the criterion should be that the number of rounds is chosen so that known cryptanalytic efforts require greater effort than a simple brute-force key search attack. This criterion was certainly used in the design of DES. Schneier [SCHN96] observes that for 16-round DES, a differential cryptanalysis attack is slightly less efficient than brute force: The differential cryptanalysis attack requires 255.1 operations,10 whereas brute force requires 255. If DES had 15 or fewer rounds, differential cryptanalysis would require less effort than a brute-force key search.

This criterion is attractive, because it makes it easy to judge the strength of an algorithm and to compare different algorithms. In the absence of a cryptanalytic breakthrough, the strength of any algorithm that satisfies the criterion can be judged solely on key length.

Design of Function F

The heart of a Feistel block cipher is the function F. As we have seen, in DES, this function relies on the use of S-boxes. This is also the case for many other symmetric block ciphers. However, we can make some general comments about the criteria for designing F. After that, we look specifically at S-box design.

DESIGN CRITERIA FOR F The function F provides the element of confusion in a Feistel cipher. Thus, it must be difficult to "unscramble" the substitution performed by F. One obvious criterion is that F be nonlinear, as we discussed previously. The more nonlinear F, the more difficult any type of cryptanalysis will be. There are several measures of nonlinearity, which are beyond the scope of this book. In rough terms, the more difficult it is to approximate F by a set of linear equations, the more nonlinear F is.

Several other criteria should be considered in designing F.

We would like the algorithm to have good avalanche properties. Recall that, in general, this means that a change in one bit of the input should produce a change in many bits of the output. A more stringent version of this is the **strict avalanche criterion (SAC)** [WEBS86], which states that any output bit j of an S-box should change with probability 1/2 when any single input bit i is inverted for all i, j. Although SAC is expressed in terms of S-boxes, a similar criterion could be applied to F as a whole. This is important when considering designs that do not include S-boxes. Another criterion proposed in [WEBS86] is the **bit independence criterion (BIC)**, which states that output bits j and k should change independently when any single input bit i is inverted for all i, j, and k. The SAC and BIC criteria appear to strengthen the effectiveness of the confusion function.

S-BOX DESIGN One of the most intense areas of research in the field of symmetric block ciphers is that of S-box design. The papers are almost too numerous to count. Here we mention some general principles. In essence, we would like any change to the input vector to an S-box to result in random-looking changes to the output. The relationship should be nonlinear and difficult to approximate with linear functions.

One obvious characteristic of the S-box is its size. An $n * m$ S-box has n input bits and m output bits. DES has 6×4 S-boxes. The encryption algorithm Blowfish, has 8×32 S-boxes. Larger S-boxes, by and large, are more resistant to differential and linear cryptanalysis [SCHN96]. On the other hand, the larger the dimension n, the (exponentially) larger the lookup table. Thus, for practical reasons, a limit of n equal to about 8 to 10 is usually imposed. Another practical consideration is that the larger the S-box, the more difficult it is to design it properly.

S-boxes are typically organized in a different manner than used in DES. An *n* * *m* S-box typically consists of 2*n* rows of *m* bits each. The *n* bits of input select one of the rows of the S-box, and the *m* bits in that row are the output. For example, in an 8 * 32 S-box, if the input is **00001001**, the output consists of the 32 bits in row 9 (the first row is labeled row 0).

A related criterion for S-boxes is proposed and analyzed in [HEYS95]. The authors define the **guaranteed avalanche (GA)** criterion as follows: An S-box satisfies GA of order *g* if, for a 1-bit input change, at least *g* output bits change. The authors conclude that a GA in the range of order 2 to order 5 provides strong diffusion characteristics for the overall encryption algorithm.

- **Random:** Use some pseudorandom number generation or some table of random digits to generate the entries in the S-boxes. This may lead to boxes with undesirable characteristics for small sizes (e.g., 6 * 4) but should be acceptable for large S-boxes (e.g., 8 * 32).
- **Random with testing:** Choose S-box entries randomly, then test the results against various criteria, and throw away those that do not pass.
- **Human-made:** This is a more or less manual approach with only simple mathematics to support it. It is apparently the technique used in the DES design. This app roach is difficult to carry through for large S-boxes.
- **Math-made:** Generate Sboxes according to mathematical principles. By using mathematical construction, Sboxes can be constructed that offer proven sec urity against linear and differential cryptanalysis, together with good diffusion.

A variation on the first technique is to use S-boxes that are both random and key dependent. An example of this approach is Blowfish, which starts with S-boxes filled with pseudorandom digits and then alters the contents using the key. A tremendous advantage of key-dependent S-boxes is that, because they are not fixed, it is impossible to analyze the S-boxes ahead of time to look for weaknesses.

Key Schedule Algorithm

A final area of block cipher design, and one that has received less attention than S-box design, is the key schedule algorithm. With any Feistel block cipher, the key is used to generate one subkey for each round. In general, we would like to select subkeys to maximize the difficulty of deducing individual subkeys and the difficulty of working back to the main key. No general principles for this have yet been promulgated.

ADVANCED STANDARD ENCRYPTION

2.6 FINITE FIELD ARITHMETIC

In AES, all operations are performed on 8-bit bytes. In particular, the arithmetic operations of addition, multiplication, and division are performed over the finite field GF(28). Section 4.7 discusses such operations in some detail. For the reader who has not studied Chapter 4, and as a quick review for those who have, this section summarizes the important concepts.

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: $a/b = a(b - 1)$.

An example of a finite field (one with a finite number of elements) is the set Zp consisting of all the integers $\{0, 1, \ldots, p - 1\}$, where p is a prime number and in which arithmetic is carried out modulo p.

Virtually all encryption algorithms, both conventional and public-key, involve arithmetic operations on integers. If one of the operations used in the algorithm is division, then we need to work in arithmetic defined over a field; this is because division requires that each nonzero element have a multiplicative inverse. 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 $2n - 1$, which fit into an n-bit word. Unfortunately, the set of such integers, $Z2n$, using modular arithmetic, is not a field. For example, the integer 2 has no multiplicative inverse in $Z2n$, that is, there is no integer b, such that $2b \bmod 2n = 1$.

There is a way of defining a finite field containing $2n$ elements; such a field is referred to as GF($2n$). Consider the set, S, of all polynomials of degree $n - 1$ or less with binary coefficients. Thus, each polynomial has the form

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

where each ai takes on the value 0 or 1. There are a total of $2n$ different polynomials in S. For $n = 3$, the $23 = 8$ polynomials in the set are

0	x	x^2	$x^2 + x$
1	$x + 1$	$x^2 + 1$	$x^2 + x + 1$

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.
2. Arithmetic on the coefficients is performed modulo 2. This is the same as the XOR operation.
3. 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 $r(x) = f(x) \bmod m(x)$. A polynomial $m(x)$ is called **irreduci ble** if and only if $m(x)$ cannot be expressed as a product of two polynomials, both of degree lower than that of $m(x)$.

For example, to construct the finite field GF(23), we need to choose an irre- ducible polynomial of degree 3. There are only two such polynomials: ($x3 + x2 + 1$) and ($x3 + x + 1$). Addition is equivalent to taking the XOR of like terms. Thus, $(x + 1) + x = 1$.

A polynomial in GF(2n) can be uniquely represented by its n binary coeffi- cients (an - 1an - 2 ... a0). Therefore, every polynomial in GF(2n) can be represented by an n-bit number. Addition is performed by taking the bitwise XOR of the two n-bit elements. There is no simple XOR operation that will accomplish multiplication in GF(2n). However, a reasonably straightforward, easily implemented, technique is available. In essence, it can be shown that multiplication of a number in GF(2n) by 2 consists of a left shift followed by a conditional XOR with a constant. Multiplication by larger numbers can be achieved by repeated application of this rule.

To summarize, AES operates on 8-bit bytes. Addition of two bytes is defined as the bitwise XOR operation. Multiplication of two bytes is defined as multiplication in the finite field GF(28), with the irreducible polynomial2 $m(x) = x8 + x4 + x3 + x + 1$. 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 given in [LIDL94].

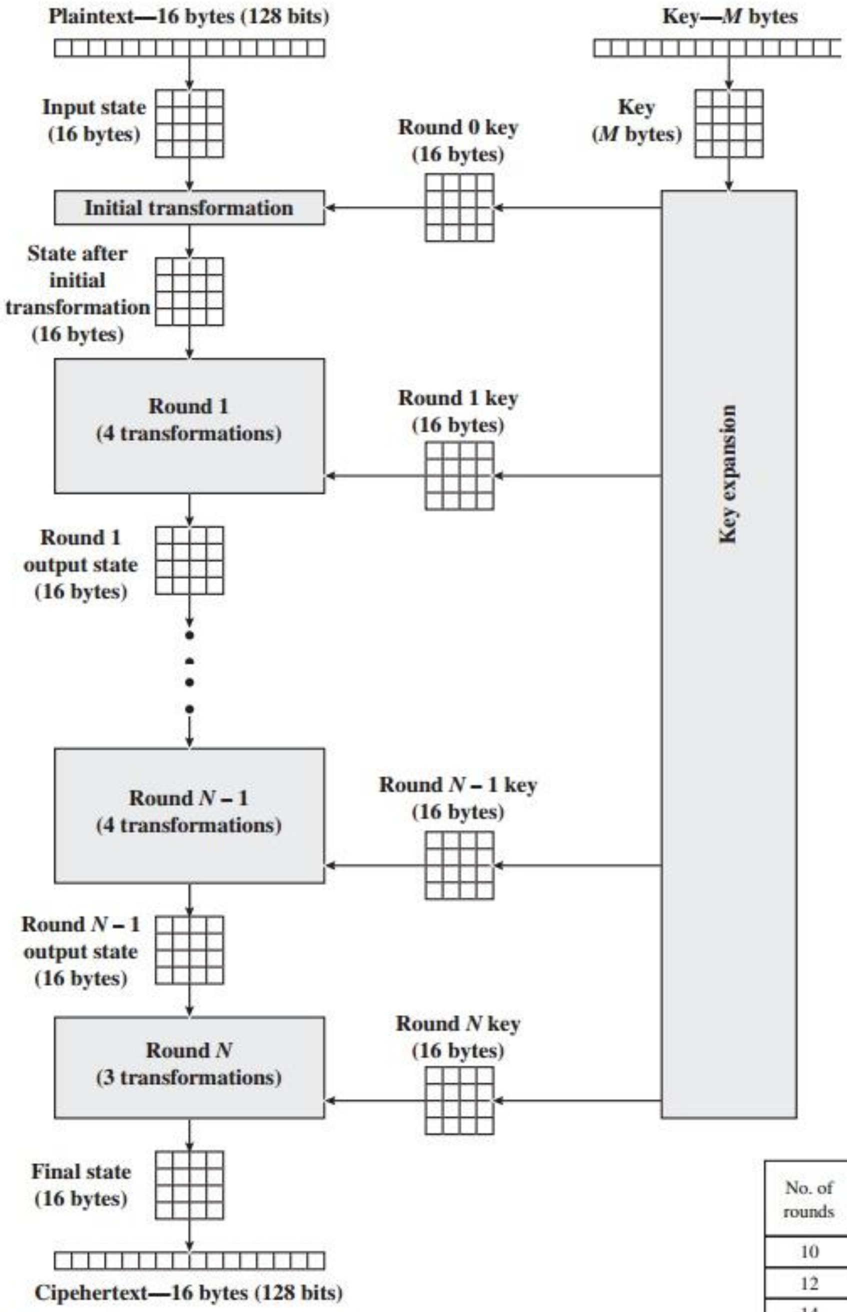
2.7 AES STRUCTURE

General Structure

Figure 5.1 shows the overall structure of the AES encryption process. The cipher takes a plaintext block size of 128 bits, or 16 bytes. The key length can be 16, 24, or 32 bytes (128, 192, or 256 bits). The algorithm is referred to as AES-128, AES-192, or AES-256, depending on the key length.

The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a 4 * 4 square matrix of bytes. This blockis copied into the **S tate** array, which is modified at each stage of encryption or decryption. After the final stage, **State** is co pied to an output matrix. These operations aredepicted in Figure 5.2a. Similarly, the key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words. Figure 5.2b shows the expansion for the 128-bit key. Each word is four bytes, and the total key schedule is 44 words for the 128-bit key. 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.

The cipher consists of N rounds, where the number of rounds depends on the key length: 10 rounds for a 16-byte key, 12 rounds for a 24-byte key, and 14 rounds for a 32-byte key (Table 5.1). The first N - 1 rounds consist of four distinct trans- formation functions: SubBytes, ShiftRows, MixColumns, and AddRoundKey, which are described subsequently. The final round contains only three transformations, and there is a initial single transformation (AddRoundKey) before the first round,



No. of rounds	Key Length (bytes)
10	16
12	24
14	32

Figure 5.1 AES Encryption Process

which can be considered Round 0. Each transformation takes one or more 4 * 4 matrices as input and produces a 4 * 4 matrix as output. Figure 5.1 shows that the output of each round is a 4 * 4 matrix, with the output of the final round being the ciphertext. Also, the key expansion function generates $N + 1$ round keys, each of which is a distinct 4 * 4 matrix. Each round key serve as one of the inputs to the AddRoundKey transformation in each round.

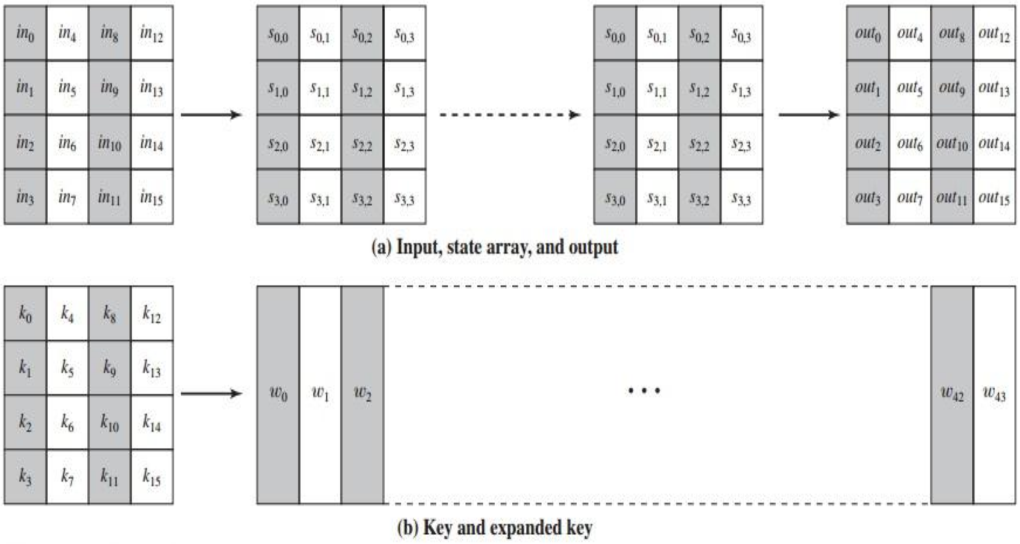


Figure 5.2 AES Data Structures

Table 5.1 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

Detailed Structure

Figure 5.3 shows the AES cipher in more detail, indicating the sequence of transformations in each round and showing the corresponding decryption function. As was done in Chapter 3, we show encryption proceeding down the page and decryption proceeding up the page.

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. AES instead processes the entire data block as a single matrix 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 5.3.
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(28)

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, f ollowed by a tenth round of three stages. Figure 5.4 depicts the structure of a full encryption round.

5. Only the AddRoundKey stage makes use of the key. For this reason, the cipher begins and ends wit h an AddRoundKey stage. Any other stage, applied at the beginning or end, is reversible without kno wledge 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 AddRo undKey stage, the inverse is achieved by XORing the same round key to the block, using the result that A Ⓧ B Ⓧ B = A.

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.

9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 5.3 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 an d decryption.

10. 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.

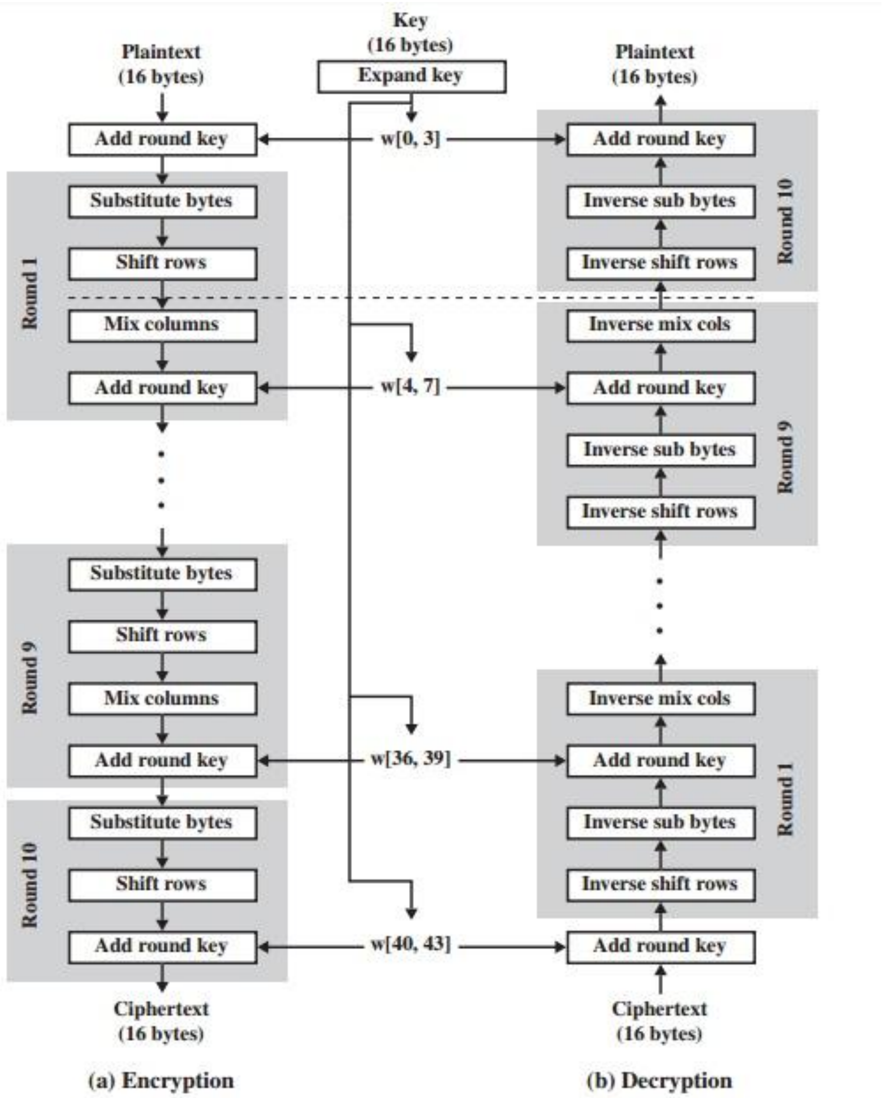


Figure 5.3 AES Encryption and Decryption

2.8 AES TRANSFORMATION FUNCTIONS

We now turn to a discussion of each of the four transformations used in AES. For each stage, we describe the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rationale for the stage.

Substitute Bytes Transformation

FORWARD AND INVERSE TRANSFORMATIONS The **forward substitute byte transfor mation**, called SubBytes, is a simple table lookup (Figure 5.5a). AES defines a 16 * 16 matrix of byte values, called an S-box (Table 5.2a), 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 value3 {95} references row 9, column 5

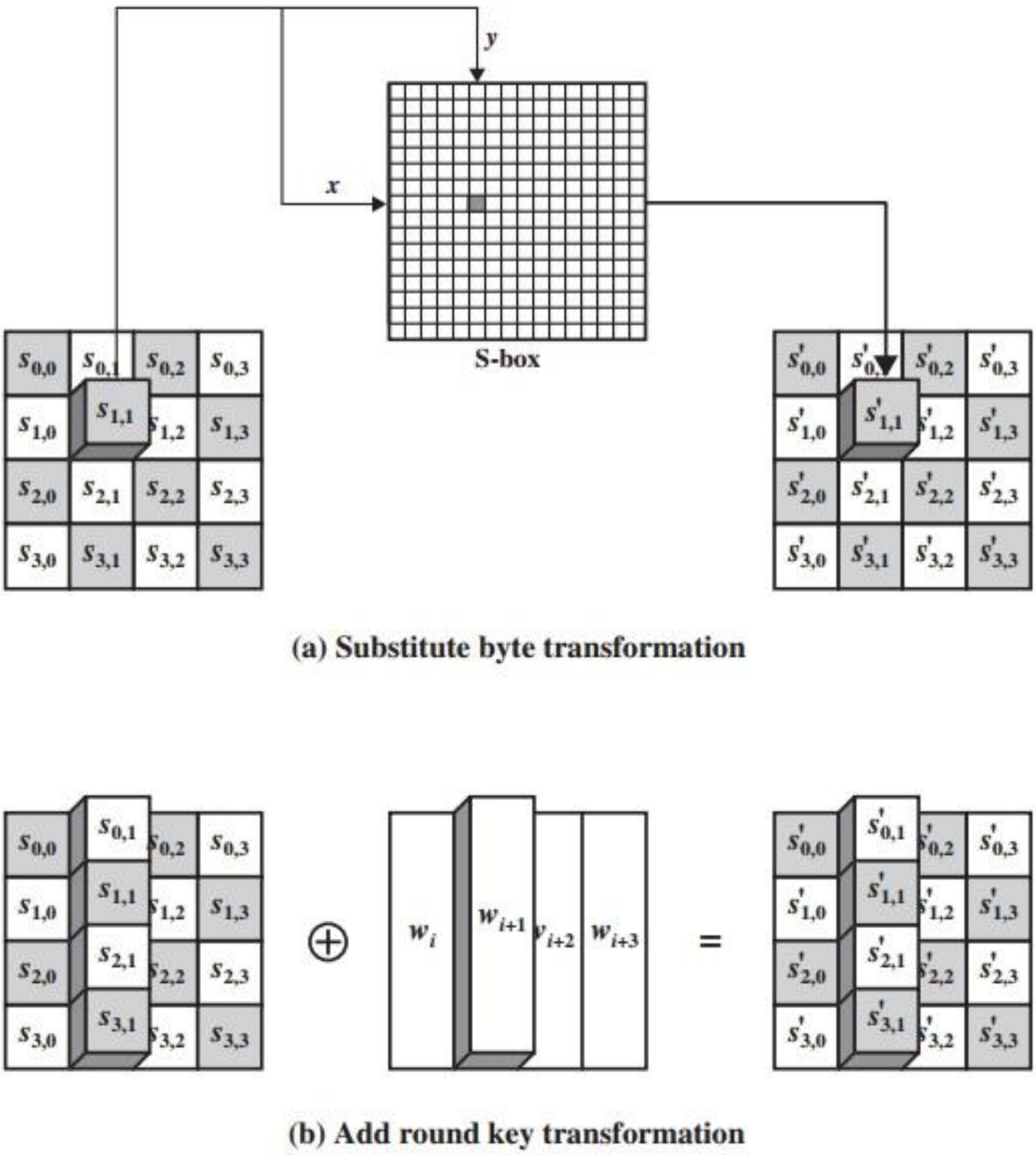


Figure 5.5 AES Byte-Level Operations

Table 5.2 AES S-Boxes

		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

(a) 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

(b) Inverse S-box

of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

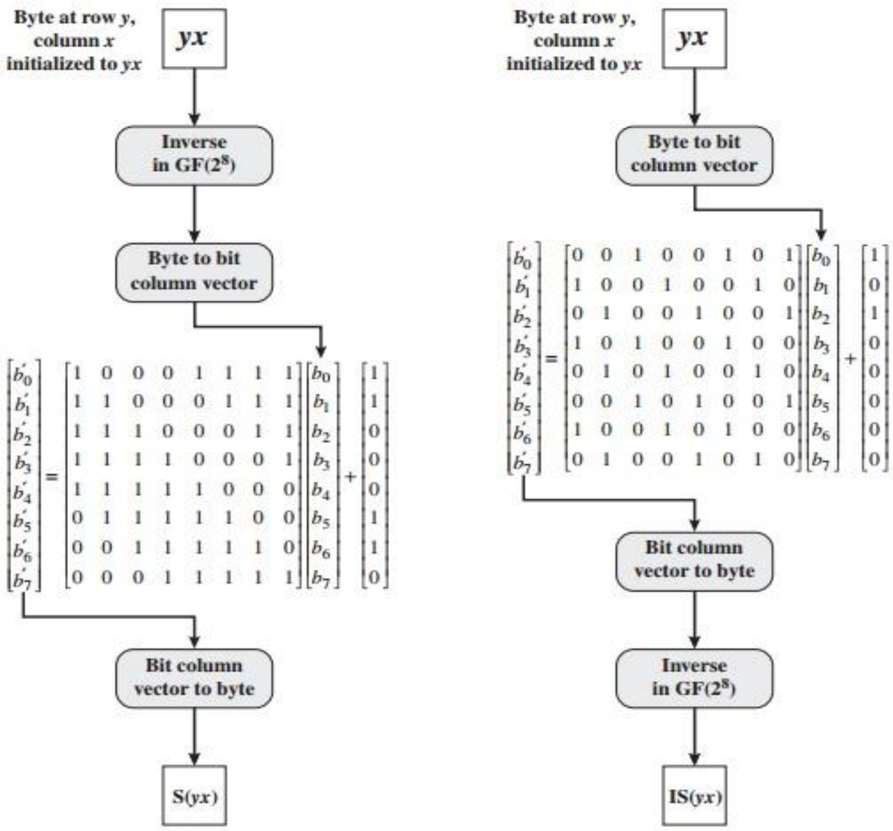
Here is an example of the SubBytes transformation:

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 (Figure 5.6a).



(a) Calculation of byte at row y, column x of S-box

(a) Calculation of byte at row y, column x of IS-box

Figure 5.6 Constuction of S-Box and IS-Box

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 y, column x is {yx}.

2. Map each byte in the S-box to its multiplicative inverse in the finite field GF(28); the value {00} is mapped to itself.

3. Consider that each byte in the S-box consists of 8 bits labeled ($b7$, $b6$, $b5$, $b4$, $b3$, $b2$, $b1$, $b0$). Apply the following transformation to each bit of each byte in the S-box:

$$b'_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i \qquad \textbf{(5.1)}$$

where ci is the ith bit of byte c with the value {63}; that is, ($c7c6c5c4c3c2c1c0$) = (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.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \textbf{(5.2)}$$

Equation (5.2) has to be interpreted carefully. In ordinary matrix multiplica- tion,4 each element in the product matrix is the sum of products of the elements of 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. Furthermore, the final addition shown in Equation (5.2) is a bitwise XOR. Recall from Section 4.7 that the bitwise XOR is addition in GF(28).

As an example, consider the input value {95}. The multiplicative inverse in GF(28) is {95} - 1 = {8A}, which is 10001010 in binary. Using Equation (5.2),

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

The result is {2A}, which should appear in row {09} column {05} of the S-box.

This is verified by checking Table 5.2a.

The **inverse substitute byte transformation**, called InvSubBytes, makes use of the inverse S-box shown in Table 5.2b. 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 (Figure 5.6b) by applying the inverse of the transformation in Equation (5.1) followed by taking the multiplicative inverse in GF(28).

The inverse transformation is

$$b_i' = b_{(i+2) \bmod 8} \otimes b_{(i+5) \bmod 8} \otimes b_{(i+7) \bmod 8} \otimes d_i$$

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 B, respectively, and the vector versions of con- stants c and d as C and D, respectively. For some 8-bit vector B, Equation (5.2) becomes B¿ = XB $\otimes$ C. We need to show that Y(XB $\otimes$ C) $\otimes$ D = B. To multiply out, we must show YXB $\otimes$ YC $\otimes$ D = B. This becomes

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus$$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}$$

We have demonstrated that YX equals the identity matrix, and the YC = D, so that YC $\otimes$ 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 is not a linear mathematical function of the input [DAEM01]. The nonlinearity is due to the use of the multiplicative inverse.

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*) = Bar *a*], where Bar *a* 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 does 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}.

FORWARD AND INVERSE TRANSFORMATIONS The **forward shift row transformation**, called ShiftRows, is depicted in Figure 5.7a. 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.

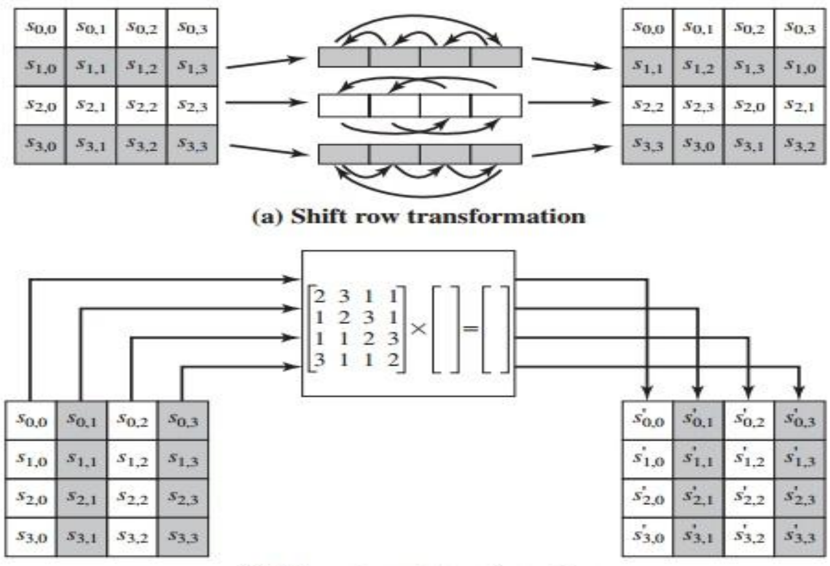
87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

→

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

The **inverse shift row transformation**, called InvShiftRows, performs the circu- lar shifts in the opposite direction for each of the last three rows, with a 1-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. Furthermore, 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



(a) Shift row transformation

(b) Mix column transformation

Figure 5.7 AES Row and Column Operations

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 5.4 illustrates the effect.

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 5.7b):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad \textbf{(5.3)}$$

Each element in the product matrix is the sum of products of elements of one row and one column. In thi s case, the individual additions and multiplications5 are performed in GF(28). The MixColumns transformation on a single column of **State** can be expressed as

$$\begin{aligned} s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\ s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j}) \end{aligned} \quad \textbf{(5.4)}$$

The following is an example of MixColumns:

87	F2	4D	97		47	40	A3	4C
6E	4C	90	EC		37	D4	70	9F
46	E7	4A	C3	→	94	E4	3A	42
A6	8C	D8	95		ED	A5	A6	BC

Let us verify the first column of this example. Recall from Section 4.7 that, in GF(28), addition is the bitwise XOR operation and that multiplication can be performed according to the rule established in Equation (4.14). 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

$$\begin{aligned} (\{02\} \cdot \{87\}) \oplus (\{03\} \cdot \{6E\}) \oplus \{46\} \oplus \{A6\} &= \{47\} \\ \{87\} \oplus (\{02\} \cdot \{6E\}) \oplus (\{03\} \cdot \{46\}) \oplus \{A6\} &= \{37\} \\ \{87\} \oplus \{6E\} \oplus (\{02\} \cdot \{46\}) \oplus (\{03\} \cdot \{A6\}) &= \{94\} \\ (\{03\} \cdot \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \cdot \{A6\}) &= \{ED\} \end{aligned}$$

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,

$$\begin{aligned} \{02\} \cdot \{87\} &= 0001\ 0101 \\ \{03\} \cdot \{6E\} &= 1011\ 0010 \\ \{46\} &= 0100\ 0110 \\ \{A6\} &= \underline{1010\ 0110} \\ & \ 0100\ 0111 = \{47\} \end{aligned}$$

The other equations can be similarly verified.

The **inverse mix column transformation**, called InvMixColumns, is defined by the following matrix multiplication:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad \textbf{(5.5)}$$

It is not immediately clear that Equation (5.5) is the **inverse** of Equation (5.3).

We need to show

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

which is equivalent to showing

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.6)}$$

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

$$\begin{aligned} (\{0E\} \cdot \{02\}) \oplus \{0B\} \oplus \{0D\} \oplus (\{09\} \cdot \{03\}) &= \{01\} \\ (\{09\} \cdot \{02\}) \oplus \{0E\} \oplus \{0B\} \oplus (\{0D\} \cdot \{03\}) &= \{00\} \\ (\{0D\} \cdot \{02\}) \oplus \{09\} \oplus \{0E\} \oplus (\{0B\} \cdot \{03\}) &= \{00\} \\ (\{0B\} \cdot \{02\}) \oplus \{0D\} \oplus \{09\} \oplus (\{0E\} \cdot \{03\}) &= \{00\} \end{aligned}$$

For the first equation, we have {0E} . {02} = 00011100 and {09} . {03} = {09} Ⓧ ({09} # {02}) = 00001001 Ⓧ 00010010 = 00011011. Then

$$\begin{aligned} \{0E\} \cdot \{02\} &= 00011100 \\ \{0B\} &= 00001011 \\ \{0D\} &= 00001101 \\ \{09\} \cdot \{03\} &= \underline{00011011} \\ & 00000001 \end{aligned}$$

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 poly- nomial with coefficients in GF(28). Each column is multiplied modulo (x4 + 1) by the fixed polynomial $a(x)$, given by

$a(x)$ = {03}x3 + {01}x2 + {01}x + {02} **(5.7)**

Appendix 5A 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

$b(x) = \{0B\}x3 + \{0D\}x2 + \{09\}x + \{0E\}$ **(5.8)**

It readily can be shown that $b(x) = a\text{-}1(x) \bmod (x4 + 1)$.

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 amongthe 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. See [DAEM99] for a discussion.

In addition, the choice of coefficients in MixColumns, which are all {01}, { 02}, or { 03}, was influenced by implementation considerations. As was discussed, multi- plication 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 (Figures 6.5 and 6.6; described in Chapter 6), only encrypti on is used.

2. As with any block cipher, AES can be used to construct a message authentica- tion code (Chapter 12), and for this, only encryption is used.

AddRoundKey Transformation

FORWARD AND INVERSE TRANSFORMATIONS In the **forward add round key transfor- mation**, called AddRoundKey, the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. As shown in Figure 5.5b, 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:

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

⊕

AC	19	28	57
77	FA	D1	5C
66	DC	29	00
F3	21	41	6A

=

EB	59	8B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D6

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.

Figure 5.8 is another view of a single round of AES, emphasizing the mecha- nisms and inputs of each transformation.

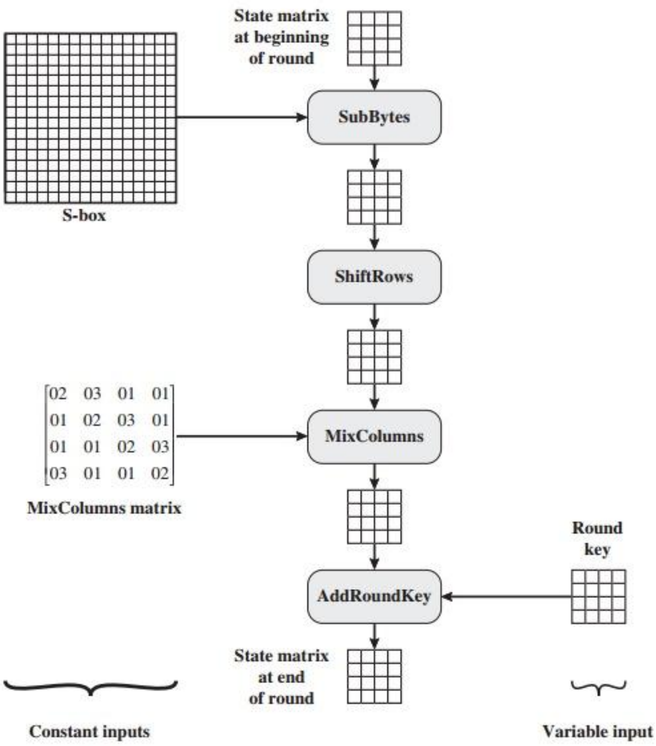


Figure 5.8 Inputs for Single AES Round

2.9 AES KEY EXPANSION

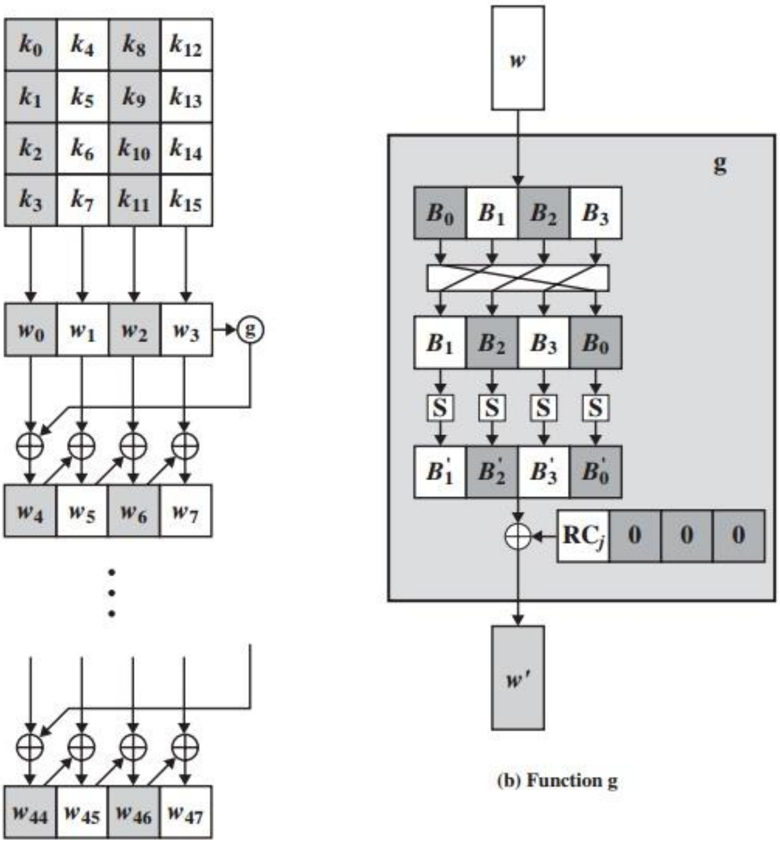
Key Expansion Algorithm

The AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The pseudocode on the next page describes the expansion.

The key is copied into the first four words of the expanded key. The remain- der 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 posi- tions 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 5.9 illustrates the generation of the expanded key, using the symbol g to represent that complex function. The function g consists of the following subfunctions.

```
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[i-4] ⊕ temp
    }
}
```



(b) Function g

(a) Overall algorithm

Figure 5.9 AES Key Expansion

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 5.2a).

3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

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 left-most 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(28). 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.

The specific criteria that were used are [DAEM99]

- 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.

2.10 AES IMPLEMENTATION

As was mentioned, the AES decryption cipher is not identical to the encryption cipher (Figure 5.3). 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. As illustrated in Figure 5.3, 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** *Si*,

InvShiftRows [InvSubBytes (*Si*)] = InvSubBytes [InvShiftRows (*Si*)]

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** *Si* and a given round key *wj*,

InvMixColumns (*Si* ⊗ *wj*) = [InvMixColumns (*Si*)] ⊗ [InvMixColumns (*wj*)]

To see this, suppose that the first column of **State** *Si* is the sequence is the sequence ($y0, y1, y2, y3$) and the first column of the round key *wj* is ($k0, k1, k2, k3$). Then we need to show

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} y_0 \oplus k_0 \\ y_1 \oplus k_1 \\ y_2 \oplus k_2 \\ y_3 \oplus k_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \oplus \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix}$$

Let us demonstrate that for the first column entry. We need to show

$$\begin{aligned} &[\{0E\} \cdot (y_0 \oplus k_0)] \oplus [\{0B\} \cdot (y_1 \oplus k_1)] \oplus [\{0D\} \cdot (y_2 \oplus k_2)] \oplus [\{09\} \cdot (y_3 \oplus k_3)] \\ &= [\{0E\} \cdot y_0] \oplus [\{0B\} \cdot y_1] \oplus [\{0D\} \cdot y_2] \oplus [\{09\} \cdot y_3] \oplus \\ &\quad [\{0E\} \cdot k_0] \oplus [\{0B\} \cdot k_1] \oplus [\{0D\} \cdot k_2] \oplus [\{09\} \cdot k_3] \end{aligned}$$

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 transformatio ns are not interchanged with InvMixColumns to produce the equivalent decryption algorithm.

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(28), which means that all operations are carried out on bytes.

The transformation MixColumns requires matrix multiplication in the field GF(28), which means that all operations are carried out on bytes. MixColumns only requi res 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 $\{03\}\ x = (\{02\}\ x) \otimes x$, we can rewrite Equation set (5.4) as follows.

$$
\begin{aligned}
Tmp &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\
s'_{0,j} &= s_{0,j} \oplus Tmp \oplus [2 \cdot (s_{0,j} \oplus s_{1,j})] \\
s'_{1,j} &= s_{1,j} \oplus Tmp \oplus [2 \cdot (s_{1,j} \oplus s_{2,j})] \\
s'_{2,j} &= s_{2,j} \oplus Tmp \oplus [2 \cdot (s_{2,j} \oplus s_{3,j})] \\
s'_{3,j} &= s_{3,j} \oplus Tmp \oplus [2 \cdot (s_{3,j} \oplus s_{0,j})]
\end{aligned}
\qquad \textbf{(5.9)}
$$

Equation set (5.9) is verified by expanding and eliminating terms.

The multiplication by {02} involves a shift and a conditional XOR. Such an imple-mentation 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 t able lookup. Define the 256-byte table X2, such that $X2[i] = \{02\}\ i$. Then Equation set (5.9) can be rewritten as

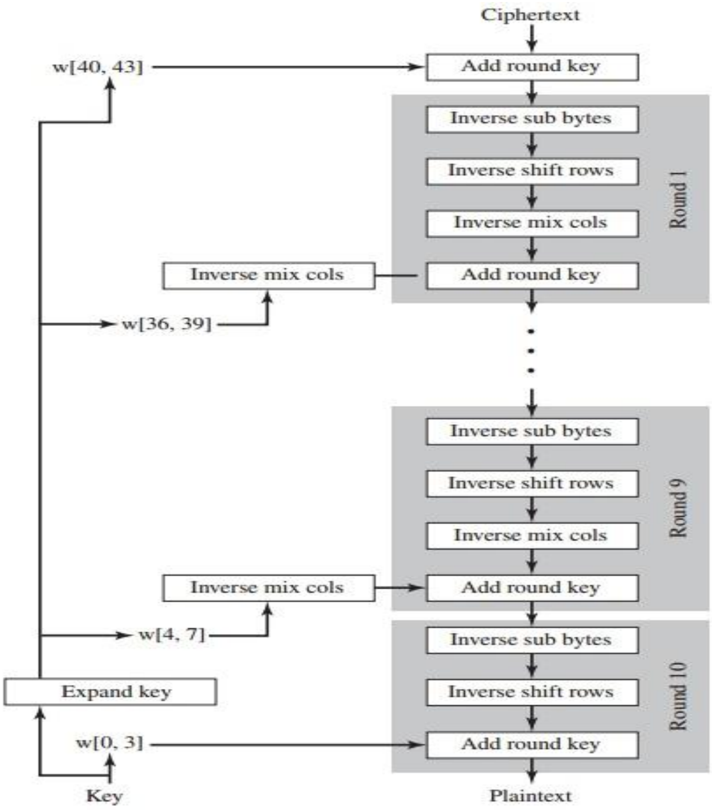


Figure 5.10 Equivalent Inverse Cipher

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 $X2[i] = \{02\} \cdot i$. Then Equation set (5.9) can be rewritten as

$$Tmp = s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j}$$
$$s'_{0,j} = s_{0,j} \oplus Tmp \oplus X2[s_{0,j} \oplus s_{1,j}]$$
$$s'_{1,c} = s_{1,j} \oplus Tmp \oplus X2[s_{1,j} \oplus s_{2,j}]$$
$$s'_{2,c} = s_{2,j} \oplus Tmp \oplus X2[s_{2,j} \oplus s_{3,j}]$$
$$s'_{3,j} = s_{3,j} \oplus Tmp \oplus X2[s_{3,j} \oplus s_{0,j}]$$

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 roundkey 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 com- bine 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 develop ers of Rijndael believe that this compact, efficient implementation was probably one of the most important factors in the selection of Rijndael for AES.

UNIT III- PUBLIC-KEY CRYPTOGRAPHY AND RSA

3.1 Principles Of Public-Key Cryptosystems

Public-Key Cryptosystems

Applications for Public-Key Cryptosystems Requirements for Public-Key Cryptography Public-Key Cryptanalysis

3.2 The RSA Algorithm

The Security of RSA

KEY POINTS

- Asymmetric encryption is a form of cryptosystem in which encryption and decryption are performed using the different keys—one a public key and one a private key. It is also known as public-key encryption.

- Asymmetric encryption transforms plaintext into ciphertext using a one of two keys and an encryption algorithm. Using the paired key and a decryption algorithm, the plaintext is recovered from the ciphertext.

- Asymmetric encryption can be used for confidentiality, authentication, or both.

- The most widely used public-key cryptosystem is RSA. The difficulty of attacking RSA is based on the difficulty of finding the prime factors of a composite number.

The development of public-key cryptography is the greatest and perhaps the only true revolution in the entire history of cryptography. From its earliest beginnings to modern times, virtually all cryptographic systems have been based on the elementary tools of substitution and permutation. After millennia of working with algorithms that could be calculated by hand, a major advance in symmetric cryptography occurred with the development of the rotor encryption/decryption machine. The electromechanical rotor enabled the development of fiendishly complex cipher systems. With th e availability of computers, even more complex systems were devised, the most prominent of which was the Lucifer effort at IBM that culminated in the Data Encryption Standard (DES). But both rotor machines and DES, although representing significant advances, still relied on the bread-and-butter tools of substitution and permutation. Public-key cryptography provides a radical departure from all that has gone before. For one thing, public-key algorithms are based on mathematical functions rather than on substitution and permutation. More important, public-key cryptography is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication, as we shall see.

Before proceeding, we should mention several common misconceptions con- cerning public-key encryption. One such misconception is that public-key encryption is more secure from cryptanalysis than is symmetricencryption. In fact, the securityof any encryption scheme depends on the length of the key and the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis.

A second misconception is that public-key encryption is a general-purpose technique that has made symmetricencryption obsolete. Onthe contrary, because of the computational overhead of current publ ic-key encryption schemes, there seems no foreseeable likelihood that symmetric encryption will be abandoned. As one of the inventors of public -key encryption has put it [DIFF88], "the restriction of public-key cryptography to key management and signature applications is almost universally accepted."

Finally, there is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centers for symmetric encryption. In fact, some form of protocol is needed, generally involving a central agent, and the procedures involved are not simpler nor any more efficient than those required for symmetric encryption (e.g., see analysis in [NEED78]).

This chapter and the next provide an overview of publickey cryptography. First, we look at its conceptual framework. Interestingly, the concept for this technique was developed and published before it was shown to be practical to adopt it. Next, we examine the RSA algorithm, which is the most important encryption/decryption algorithm that has been shown to be feasible for public-key encryption.

Much of the theory of public-key cryptosystems is based on number theory. If one is prepared to accept the results given in this chapter, an understanding of number theory is not strictly necessary. However, to gain a full appreciation of public-key algorithms, some understanding of number theory is required.

Table 9.1 defines some key terms.

Table 9.1 Terminology Related to Asymmetric Encryption

Asymmetric Keys

Two related keys, a public key and a private key, that are used to perform complementary operations, such as encryption and decryption or signature generation and signature verification.

Public Key Certificate

A digital document issued and digitally signed by the private key of a Certification Authority that bindsthename of a subscriber to a public key. The certificate indicates that the subscriber identified in the certificate has sole control and access to the corresponding private key.

Public Key (Asymmetric) Cryptographic Algorithm

A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys ha ve the property that deriving the private key from the public key is computationally infeasible.

Public Key Infrastructure (PKI)

A set of policies, processes, server platforms, software and workstations used for the purpose of administering certificates and public-private key pairs, including the ability to issue, maintain, and revoke public key certificates.

3.2 THE RSA ALGORITHM

The pioneering paper by Diffie and Hellman [DIFF76b] introduced a new approach to cryptography and, in effect, challenged cryptologists to come up with a crypto- graphic algorithm that met the requirements for public-key systems. A number of algorithms have been proposed for public-key cryptography. Some of these, though initially promising, turned out to be breakable.

One of the first successful responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78].5 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.

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 dec- imal digits. That is, n is less than 21024. 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

RSA 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 log2(n) + 1; in practice, the block size is i bits, where 2i 6 n £ 2i+1. Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C.

$C = Me \bmod n$

$M = Cd \bmod n = 1Me\ d \bmod n = Med \bmod n$

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 $PR = \{d, n\}$. For this algorithm to be satisfactory for public-key encryption, the following require- ments must be met.

1. It is possible to find values of e, d, n such that Med mod $n = M$ for all $M < n$.

2. It is relatively easy to calculate Me mod n and Cd mod n 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

Med mod $n = M$

The preceding relationship holds if e and d are multiplicative inverses modulo $\phi(n)$, where $\phi(n)$ is the Euler totient function. It is shown in Chapter 8 that for p, q prime, $\phi(pq) = (p - 1)(q - 1)$. The relationship between e and d can be expressed as

ed mod $\phi(n) = 1$ **(9.1)**

This is equivalent to saying

$$ed \equiv 1 \bmod \phi(n)$$
$$d \equiv e^{-1} \bmod \phi(n)$$

That is, e and d are multiplicative inverses mod $\phi(n)$. Note that, according to the rules of modular arithmetic, this is true only if d (and therefore e) is relatively prime to $\phi(n)$. Equivalently, $\gcd(\phi(n), d) = 1$. See Appendix 9A for a proof that Equation (9.1) satisfies the requirement for RSA.

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 $\gcd(\phi(n), e) = 1$; $1 < e < \phi(n)$	(public, chosen)
d K e-1 (mod $\phi(n)$)	(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 $C = Me$ mod n and transmits C. On receipt of this cipher- text, user A decrypts by calculating $M = Cd$ mod n.

Figure 9.5 summarizes the RSA algorithm. It corresponds to Figure 9.1a: Alice generates a public/private key pair; Bob encrypts using Alice's public key; and Alice decrypts using her private key. An example from [SING99] 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 ´ 11 = 187.

3. Calculate ϕ(n) = (p - 1)(q - 1) = 16 ´ 10 = 160.

4. Select e such that e is relatively prime to ϕ(n) = 160 and less than $f(n)$; we choose e = 7.

5. Determine d such that de K 1 (mod 160) and d < 160. The correct value is d = 23, because 23 ´ 7 = 161 = (1 ´ 160) + 1; d can be calculated using the extended Euclid's algorithm (Chapter 4).

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 encryp- tion, we need to calculate C = 887 mod 187. Exploiting the properties of modular arithmetic, we can do this as follows.

887 mod 187 = [(884 mod 187) ´ (882 mod 187)´ (881 mod 187)] mod 187

881 mod 187 = 88

882 mod 187 = 7744 mod 187 = 77

884 mod 187 = 59,969,536 mod 187 = 132

887 mod 187 = (88 ´ 77 ´ 132) mod 187 = 894,432 mod 187 = 11

Key Generation Alice

Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calcuate $\phi(n) = (p-1)(q-1)$	
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 by Bob with Alice's Public Key

Plaintext:	$M < n$
Ciphertext:	$C = M^e \bmod n$

Decryption by Alice with Alice's Public Key

Ciphertext:	C
Plaintext:	$M = C^d \bmod n$

Figure 9.5 The RSA Algorithm

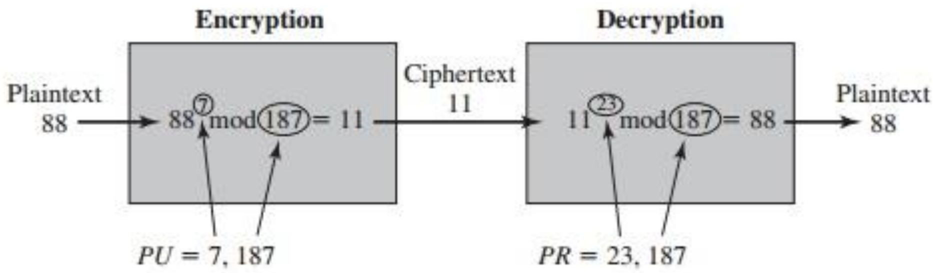


Figure 9.6 Example of RSA Algorithm

For decryption, we calculate *M* = 1123 mod 187:

1123 mod 187 = [(111 mod 187) ´ (112 mod 187) ´ (114 mod 187)
´ (118 mod 187) ´ (118 mod 187)] mod 187

111 mod 187 = 11

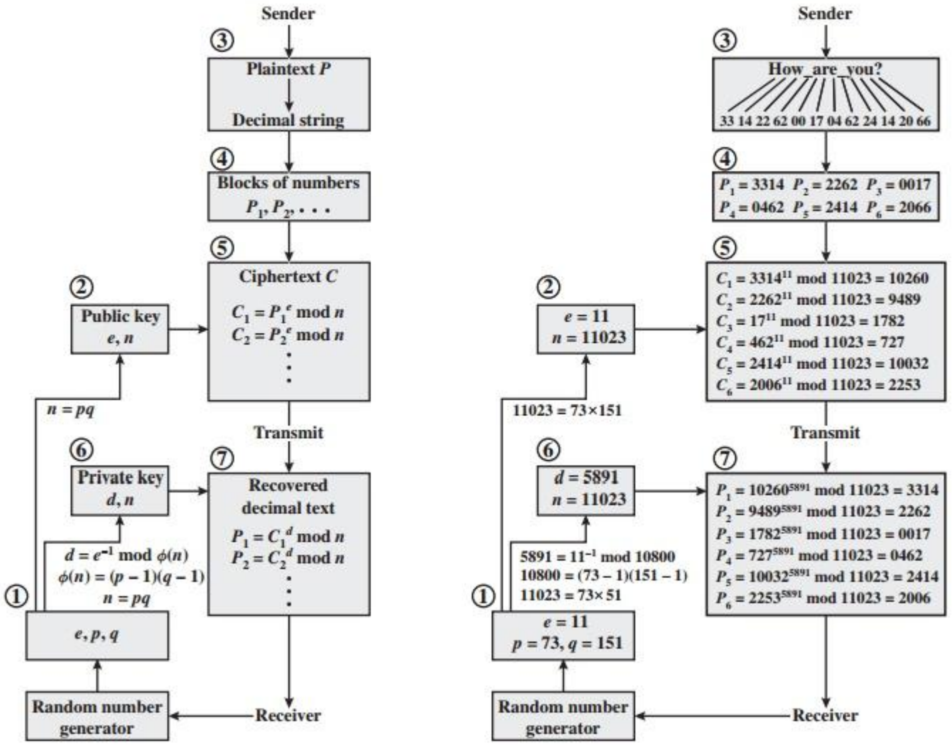
112 mod 187 = 121

114 mod 187 = 14,641 mod 187 = 55

118 mod 187 = 214,358,881 mod 187 = 33

1123 mod 187 = (11 ´ 121 ´ 55 ´ 33 ´ 33) mod 187 = 79,720,245 mod 187 = 88

We now look at an example from [HELL79], which shows the use of RSA to process multiple blocks of data. In this simple example, the plaintext is an alphanumeric string. Each plaintext symbol is assigned a unique code of two decimal digits (e.g., a = 00, A = 26).6 A plaintext block consists of four decimal digits, or two alphanumeric characters. Figure 9.7a illustrates the sequence of events for the encryption of multiple blocks, and Figure 9.7b gives a specific example. The circled numbers indicate the order in which operations are performed.



(a) General approach (b) Example

Figure 9.7 RSA Processeing of Multiple Blocks

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 Bothencryption and decryption in RSAinvolve 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:

$$[(a \bmod n) * (b \bmod n)] \bmod n = (a * b) \bmod n$$

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 x16. A straightforward approach requires 15 multiplications:

*x*16 = *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *x* * *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 mod *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* mod *n*, *x*2 mod *n*, *x*4 mod *n*, an d *x*8 mod *n* and then calculate [(*x* mod *n*) ´ (*x*2 mod *n*) ´ (*x*8 mod *n*)] mod *n*.

More generally, suppose we wish to find the value *ab* with *a* and *m* positive integers. If we express *b* as a binary number *bkbk*-1. . . *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 algorithm7 for computing *ab* mod *n*, shown in Figure 9.8. Table 9.4 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.

```
c ← 0; f ← 1
for i ← k downto 0
    do  c ← 2 × c
        f ← (f × f) mod n
    if  b_i = 1
        then c ← c + 1
             f ← (f × a) mod n
return f
```

Note: The integer b is expressed as a binary number $b_k b_{k-1} \ldots b_0$.

Figure 9.8 Algorithm for Computing a^b mod n

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 (216 + 1); two other popular choices are 3 and 17. Each of these choices has only two 1 bits, 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 $(n1, n2, n3)$. If user A sends the same encrypted message M to all three users, then the three ciphertexts are $C1 = M3 \bmod n1$, $C2 = M3 \bmod n2$, and $C3 = M3 \bmod n3$. It is likely that $n1$, $n2$, and $n3$ are pairwise relatively prime. Therefore, one can use the Chinese remainder theorem (CRT) to compute $M3 \bmod (n1n2n3)$. By the rules of the RSA algorithm, M is less than each of the ni; therefore $M3 < n1n2n3$. Accordingly, the attacker need only compute the cube root of $M3$. This attack canbe countered by adding a uniquepseudorandom bit string as padding to each insta nce 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 $f(n)$. Thus, if a value of e is selected first and the primes p and q are generated, it may turn out that gcd($f(n)$, e) Z 1. In that case, the user must reject the p, q values and generate a new p, q pair.

Table 9.4 Result of the Fast Modular Exponentiation Algorithm for ab mod n, where $a = 7$, $b = 560 = 1000110000$, and $n = 561$

Table 9.4 Result of the Fast Modular Exponentiation Algorithm for a^b mod n, where $a = 7$, $b = 560 = 1000110000$, and $n = 561$

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
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 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 [WIEN90]. However, there is a way to speed up computation using the CRT. We wish to compute the value $M = Cd \bmod n$. Let us define the following intermediate results:

$Vp = Cd \bmod p \quad Vq = Cd \bmod q$

Following the CRT using Equation (8.8), define the quantities

$Xp = q * (q - 1 \bmod p) \quad Xq = p * (p - 1 \bmod q)$

The CRT then shows, using Equation (8.9), that

$M = (Vp\,Xp + Vq\,Xq) \bmod n$

Furthermore, we can simplify the calculation of *Vp* and *Vq* using Fermat's theorem, which states that *ap*-1 K 1 (mod *p*) if *p* and *a* are relatively prime. Some thought should convince you that the following are valid.

$Vp = Cd \bmod p = Cd \bmod (p - 1) \bmod p \qquad Vq = Cd \bmod q = Cd \bmod (q - 1) \bmod q$

The quantities *d* mod (*p* - 1) and *d* mod (*q* - 1) can be precalculated. The end result is that the calculation is approximately four times as fast as evaluating *M* = *Cd* mod *n* directly [BONE02].

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, in order 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 l arge 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 magni-tude 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 prob-abilistic. 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, the Miller-Rabin algorithm, is described in Chapter 8. With this algorithm and most such algorithms, the procedure for test- ing 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 2200 were sought, then about ln(2200)/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(*f*(*n*), *e*) = 1 and then calculate *d* K *e*-1 (mod *f*(*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. The algorithm, referred to as the extended Euclid's algorithm, is explained in Chapter 4. Thus, the procedure is to generate a series of random numbers, testing each against *f*(*n*) until a number relatively prime to *f*(*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 *f*(*n*)? It can be shown easily that the probabi lity that two random numbers are relatively prime is about 0.6; thus, very few tests would be needed to find a suitable integer (see Problem 8.2).

The Security of RSA

Four possible approaches to attacking the RSA algorithm are

- **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, to 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.

1. Factor *n* into its two prime factors. This enables calculation of ϕ(*n*) = (*p* - 1) ´(*q* - 1), which in turn enables determination of *d* K *e*-1 (mod ϕ(*n*)).

2. Determine ϕ(*n*) directly, without first determining *p* and *q*. Again, this enables determination of *d* K *e*-1 (mod ϕ(*n*)).

3. Determine *d* directly, without first determining ϕ(*n*).

Table 9.5 Progress in Factorization

Table 9.5 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

million-instructions-per-second processor running for one year, which is about 3 ´x 1013 instructions executed. A 1 GHz Pentium is about a 250-MIPS machine.

A striking fact about Table 9.5 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 conside rably faster than the generalized number field sieve. Figure 9.9 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 [ODLY95]. 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.

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 1075 to 10100.

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 $d < n1/4$, then d can be easily determined [WIEN90].

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 [KOCH96, KALI96b]. 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 9.8, 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 theleftmost bit, bk.

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 operationof the subsequent step depends on the unknown exponent b it. If the bit is set, d ;(d ′ a)mod n will be executed. For a few values of a and d, the modular multiplicatio n will beextremely slow, and the attacker knows which these are. Therefore, if the observed time to exec ute the decryption algorithm is always slow when this particular iteration is slowwith 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 bi t is assumed to be 0.

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 cipher- text 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 $M = Cd \bmod n$ is implemented as follows.

1.Generate a secret random number r between 0 and n -1.

2.Compute $C' = C(re) \bmod n$, where e is the public exponent.

3.Compute $M' = (C')d \bmod n$ with the ordinary RSA implementation.

4.Compute $M = M'r\text{-}1 \bmod n$. In this equation, r-1 is the multiplicative inverse of $r \bmod n$; see Chapter 4 for a discussion of this concept. It can be demonstrated that this is the correct result by observing that $red \bmod n = r \bmod n$.

RSA Data Security reports a 2 to 10% performance penalty for blinding.

CHOSEN CIPHERTEXT ATTACK AND OPTIMAL ASYMMETRIC ENCRYPTION PAD DING

Thebasic RSA algorithm is vulnerable to a **chosen ciphertext attack** (CCA). CCA is defined as an attack in which the adversary chooses a number of ciphertexts and is then given the corresponding plaintexts, decrypted with the target's private key. Thus, the adversary c ould select a plaintext, encrypt it with the target's public key, and thenbe able to get the plaintext back by having it decrypted with the private key. Clearly, this provides the adversary with no new inform ation. Instead, the adversary exploitsproperties of RSA and selects blocks of data that, when process ed 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:

$E(PU, M1) \,'\, E(PU, M2) = E(PU, [M1 \,'\, M2])$ **(9.2)**

We can decrypt $C = Me \bmod n$ using a CCA as follows.

1. Compute $X = (C \,'\, 2e) \bmod n$.

2. Submit X as a chosen ciphertext and receive back $Y = Xd \bmod n$. But now note that

$X = (C \bmod n) * (2e \bmod n)$

$= (Me \bmod n) * (2e \bmod n)$

$= (2M)e \bmod n$

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; see [POIN02] for an introduction and [BELL94a] for a thorough analysis. Here, we simply summarize the OAEP procedure.

Figure 9.10 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.8 The output is then padded with zeros to get the desired length in the overall data

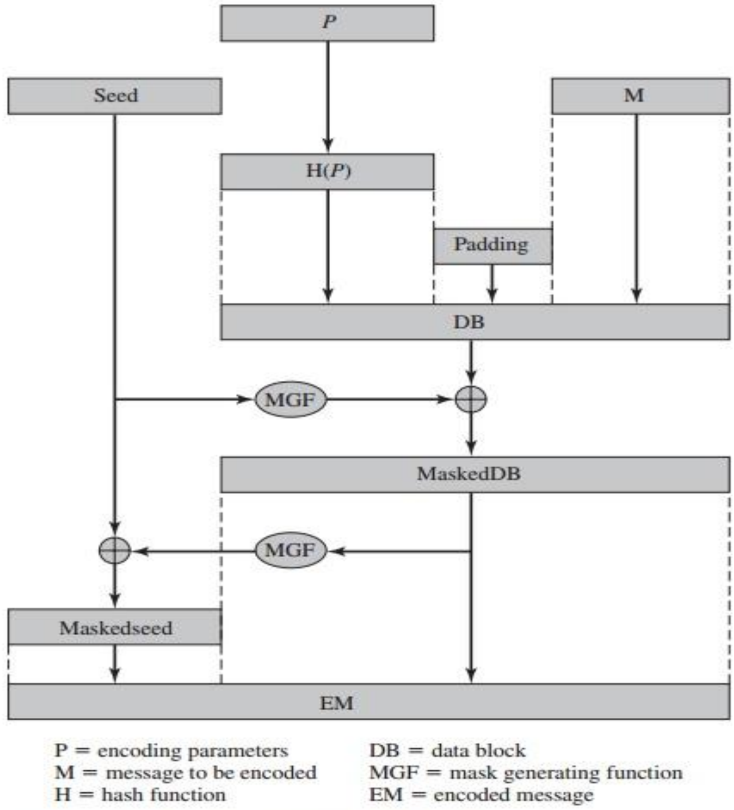


Figure 9.10 Encryption Using Optimal Assymetric Encryption Padding (OAEP)

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.

3.3 OTHER PUBLIC-KEY CRYPTOSYSTEMS

3.4 ELGAMAL CRYPTOGRAPHIC SYSTEM

In 1984, T. Elgamal announced a public-key scheme based on discrete logarithms, closely related to the Diffie-Hellman technique [ELGA84, ELGA85]. The ElGamal2 cryptosystem is used in some form in a number of standards including the digital signature standard (DSS), which is covered in Chapter 13, and the S/MIME e-mail standard (Chapter 18).

As with DiffieHellman, the global elements of ElGamal are a prime number q and a, which is a primi tive root of q. User A generates a private/public key pair as follows:

1. Generate a random integer XA, such that 1 6 XA 6 q - 1.
2. Compute $YA = aXA \bmod q$.
3. A's private key is XA; A's pubic key is $\{q, a, YA\}$.

Any user B that has access to A's public key can encrypt a message as follows:

1. Represent the message as an integer M in the range 0 … M … q - 1. Longer messages are sent as a sequence of blocks, with each block being an integer less than q.
2. Choose a random integer k such that 1 … k … q - 1.
3. Compute a one-time key $K = (YA)k \bmod q$.
4. Encrypt M as the pair of integers ($C1$, $C2$) where $C1 = ak \bmod q$; $C2 = KM \bmod q$

User A recovers the plaintext as follows:

1. Recover the key by computing $K = (C1)XA \bmod q$.
2. Compute $M = (C2K\text{-}1) \bmod q$.

These steps are summarized in Figure 10.3. It corresponds to Figure 9.1a: Alice generates a public/private key pair; Bob encrypts using Alice's public key; and Alice decrypts using her private key.

Let us demonstrate why the ElGamal scheme works. First, we show how K is recovered by the decryption process:

$K = (YA)k \bmod q$ K is defined during the encryption process

$K = (aXA \bmod q)k \bmod q$ substitute using $YA = aXA \bmod q$ $K = akXA \bmod q$ by the rules of modular arithmetic

$K = (C1)XA \bmod q$ substitute using $C1 = ak \bmod q$

Next, using K, we recover the plaintext as

$C2 = KM \bmod q$

$(C2K\text{-}1) \bmod q = KMK\text{-}1 \bmod q = M \bmod q = M$

We can restate the ElGamal process as follows, using Figure 10.3.

1. Bob generates a random integer k.

2. Bob generates a one-time key K using Alice's public-key components YA, q, and k.

3. Bob encrypts k using the publickey component a, yielding $C1$. $C1$ provides sufficient information for Alice to recover K.

4. Bob encrypts the plaintext message M using K.

5. Alice recovers K from $C1$ using her private key.

6. Alice uses K-1 to recover the plaintext message from $C2$.

Global Public Elements	
q	prime number
α	$\alpha < q$ and α a primitive root of q

Key Generation by Alice	
Select private X_A	$X_A < q - 1$
Calculate Y_A	$Y_A = \alpha^{XA} \bmod q$
Public key	$PU = \{q, \alpha, Y_A\}$
Private key	X_A

Encryption by Bob with Alice's Public Key	
Plaintext:	$M < q$
Select random integer k	$k < q$
Calculate K	$K = (Y_A)^k \bmod q$
Calculate C_1	$C_1 = \alpha^k \bmod q$
Calculate C_2	$C_2 = KM \bmod q$
Ciphertext:	(C_1, C_2)

Decryption by Alice with Alice's Private Key	
Ciphertext:	(C_1, C_2)
Calculate K	$K = (C_1)^{XA} \bmod q$
Plaintext:	$M = (C_2 K^{-1}) \bmod q$

Figure 10.3 The ElGamal Cryptosystem

Thus, K functions as a one-time key, used to encrypt and decrypt the message.

For example, let us start with the prime field GF(19); that is, $q = 19$. It has primitive roots {2, 3, 10, 13, 14, 15}, as shown in Table 8.3. We choose $a = 10$.

Alice generates a key pair as follows:

1. Alice chooses $XA = 5$.
2. Then $YA = aXA \bmod q = a5 \bmod 19 = 3$ (see Table 8.3).

3. Alice's private key is 5; Alice's pubic key is {q, a, YA} = {19, 10, 3}.

Suppose Bob wants to send the message with the value M = 17. Then,

1. Bob chooses k = 6.

2. Then $K = (YA)k \bmod q$ = 36 mod 19 = 729 mod 19 = 7.

3. So $C1 = ak \bmod q = a6 \bmod 19 = 11C2 = KM \bmod q$ = 7 * 17 mod 19 = 119 mod 19 = 5

4. Bob sends the ciphertext (11, 5).

For decryption:

1. Alice calculates $K = (C1)XA \bmod q$ = 115 mod 19 = 161051 mod 19 = 7.

2. Then K -1 in GF(19) is 7-1 mod 19 = 11.

3. Finally, $M = (C2K- 1) \bmod q$ = 5 * 11 mod 19 = 55 mod 19 = 17.

If a message must be broken up into blocks and sent as a sequence of encrypted blocks,

a unique value of k should be used for each block. If k is used formore than one block, knowledge of one block $m1$ of the message enables the user to compute other blocks as follows. Let

$$C_{1,1} = \alpha^k \bmod q;\ C_{2,1} = KM_1 \bmod q$$

$$C_{1,2} = \alpha^k \bmod q;\ C_{2,2} = KM_2 \bmod q$$

Then,

$$\frac{C_{2,1}}{C_{2,2}} = \frac{KM_1 \bmod q}{KM_2 \bmod q} = \frac{M_1 \bmod q}{M_2 \bmod q}$$

If M_1 is known, then M_2 is easily computed as

$$M_2 = (C_{2,1})^{-1} C_{2,2} M_1 \bmod q$$

The security of ElGamal is based on the difficulty of computing discrete logarithms. To recover A's private key, an adversary would have to compute $XA = \text{dlog}a,q(YA)$. Alternatively, to recover the one-time key K, an adversary would have to determine the random number k, and this would require comput- ing the discrete logarithm $k = \text{dlog}a,q(C1)$. [STIN06] points out that these calcu- lations are regarded as infeasible if p is at least 300 decimal digits and q - 1 has at least one "large" prime factor.

3.5 ELLIPTIC CURVE ARITHMETIC

Most of the products and standards that use public-key cryptography for encryp- tion 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. A competing system challenges RSA: elliptic curve cryptography (ECC). 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.

The reader may wish to review the material on finite fields in Chapter 4 before proceeding.

Abelian Groups

Recall from Chapter 4 that an **abelian group** G, sometimes denoted by $\{G, \cdot\}$, is a set of elements with a binary operation, denoted by $\cdot$, that associates to each ordered pair (a, b) of elements in G an element $(a \cdot b)$ in G, such that the following axioms are obeyed:[3]

(A1) Closure: If a and b belong to G, then $a \cdot b$ is also in G.

(A2) Associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all a, b, c in G.

(A3) Identity element: There is an element e in G such that $a \cdot e = e \cdot a = a$ for all a in G.

(A4) Inverse element: For each a in G there is an element a' in G such that $a \cdot a' = a' \cdot a = e$.

(A5) Commutative: $a \cdot b = b \cdot a$ 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 inte- gers modulo a prime number q. Keys are generated by exponentiation over the group, with exponentiation defined as repeated multiplication. For example,

$a^k \bmod q = \underbrace{(a \times a \times \dots \times a)}_{k \text{ times}} \bmod q$. To attack Diffie-Hellman, the attacker must determine k given a and a^k; this is the discrete logarithm problem.

determine k given a and *ak*; this is the discrete logarithm problem.

For elliptic curve cryptography, an operation over elliptic curves, called addi- tion, is used. Multiplication is defined by repeated addition. For example,

$$a \times k = \underbrace{(a + a + \dots + a)}_{k \text{ times}}$$

where the addition is performed over an elliptic curve. Cryptanalysis involves determining k given a and $(a * k)$.

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 following form, known as a **Weierstrass equation**:

$$y2 + axy + by = x3 + cx2 + dx + e$$

where a, b, c, d, e are real numbers and x and y take on values in the real numbers.4 For our purpose, i t is sufficient to limit ourselves to equations of the form

$$y^2 = x^3 + ax + b \qquad \textbf{(10.1)}$$

Such equations are said to be cubic, or of degree 3, because the highest expo-
nent 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 10.4 shows two examples of elliptic curves. As you can see, the formula sometimes produces weird- looking curves.

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 10.4 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:

$$4a^3 + 27b^2 \neq 0 \qquad \textbf{(10.2)}$$

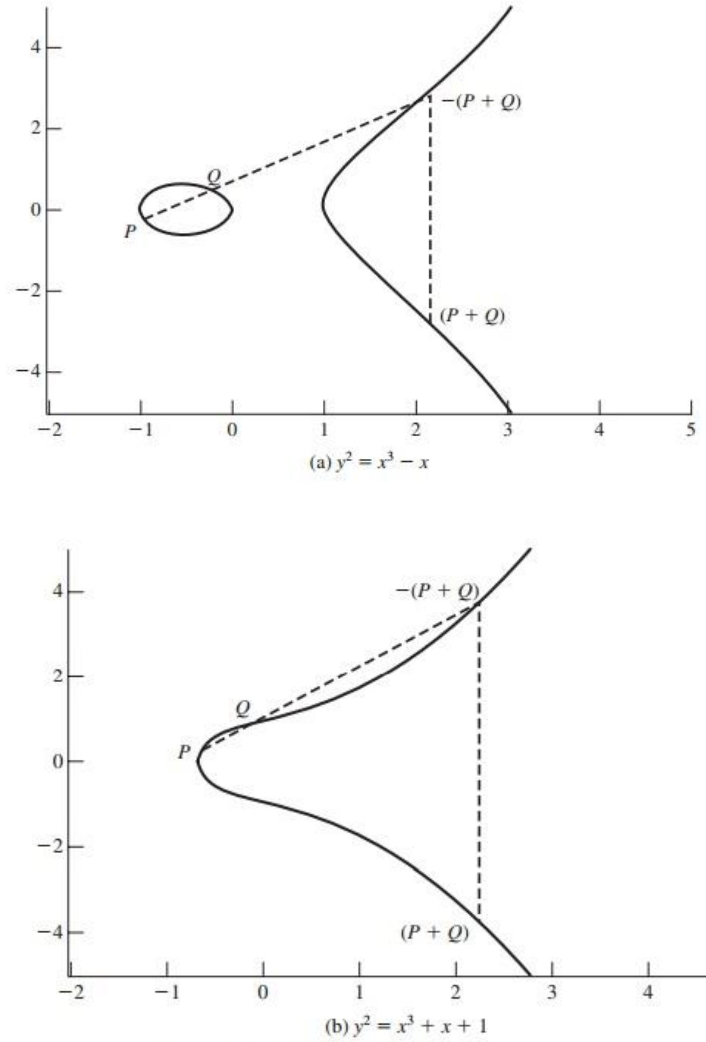


Figure 10.4 Example of Elliptic Curves

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 $P \mathrel{!=} O$ and $Q \mathrel{!=} O$.

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 $P + (-P) = P - P = O$.

1. 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 tan- gent 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: $P + Q = -R$. That is, we define $P + Q$ to be the mirror image (with respect to the x axis) of the third point of intersection. Figure 10.4 illustrates this construction.

2. 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 there- fore have $P + (-P) = O$, which is consistent with item (2).

3. To double a point Q, draw the tangent line and find the other point of inter- section 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.5 For two distinct points, $P = (xP , yP)$ and $Q = (xQ, yQ)$, that are not negatives of each other, the slope of the line l that joins them is $\Delta = (yQ - yP)>(xQ - xP)$. 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

$$\begin{aligned} x_R &= \Delta^2 - x_P - x_Q \\ y_R &= -y_P + \Delta(x_P - x_R) \end{aligned} \qquad \textbf{(10.3)}$$

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

$$\begin{aligned} x_R &= \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P \\ y_R &= \left(\frac{3x_P^2 + a}{2y_P}\right)(x_P - x_R) - y_P \end{aligned} \qquad \textbf{(10.4)}$$

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 binarycurves over GF(2*m*).

For a **prime curve** over Z*p*, we use a cubic equation in which thevariables 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** definedover GF(2*m*), the vari ables and coefficients all take on values in GF(2*m*) and in cal- culations are performed over GF(2*m*). [FERN99] 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 *Zp*, 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 *Zp*:

$$y^2 \bmod p = (x^3 + ax + b) \bmod p \qquad \textbf{(10.5)}$$

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

$$7^2 \bmod 23 = (9^3 + 9 + 1) \bmod 23$$
$$49 \bmod 23 = 739 \bmod 23$$
$$3 = 3$$

Now consider the set E*p*(*a*, *b*) consisting of all pairs of integers (*x*, *y*) that sat- isfy 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 *Zp*.

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 10.4b. The figure shows a continuous curve with all of the real points that satisfy the equation. For the set E23(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 E23(1, 1). Figure 10.5 plots the points of E23(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)

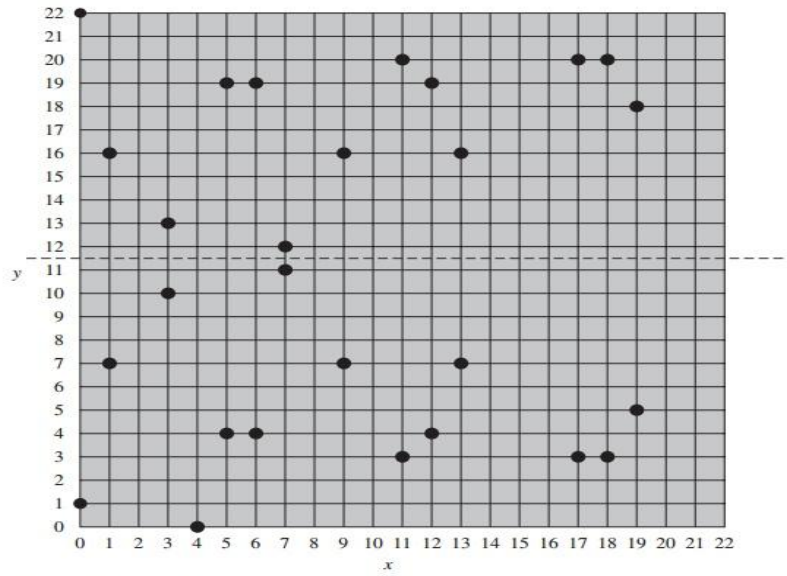


Figure 10.5 The Elliptic Curve $E_{23}(1, 1)$

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*) mod *p* has no repeated factors. This is equiva- lent to the condition

$$(4a^3 + 27b^2) \bmod p \neq 0 \bmod p \qquad \textbf{(10.6)}$$

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 numbers. For all points *P*, *Q* Î E*p*(*a*, *b*):

1. $P + O = P$.
2. 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 $-7 \bmod 23 = 16$. Therefore, $-P = (13, 16)$, which is also in $E_{23}(1, 1)$.
3. If $P = (x_p, y_p)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$, 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

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

4. 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

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

$$x_R = (11^2 - 3 - 9) \bmod 23 = 109 \bmod 23 = 17$$

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

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 Z23. This can be done using the extended Euclidean algorithm defined in Section 4.4. To confirm, note that (6 * 4) mod 23 = 24 mod 23 = 1.

xR = (62 - 3 - 3) mod 23 = 30 mod 23 = 7

yR = (6(3 - 7) - 10) mod 23 = (-34) mod 23 = 12

and 2*P* = (7, 12).

For determining the security of various elliptic curve ciphers, it is of some inter- est to know the number of points in a finite abelian group defined over an elliptic curve. In the case of the finite group EP(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 *Ep*(*a*, *b*) is approximately equal to the number of elements in *Zp*, namely *p* elements.

Elliptic Curves over GF(2*m*)

Recall from Chapter 4 that 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 *Zp*. The form is

$$y^2 + xy = x^3 + ax^2 + b \quad \textbf{(10.7)}$$

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})	$(g^{12}, 0)$
(g^3, g^{13})	(g^9, g^{10})	(g^{12}, g^{12})

where it is understood that the variables *x* and *y* and the coefficients *a* and *b* are elements of GF(2*m*) 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(24) with the irreducible polynomial *f*(*x*) = *x*4 + *x* + 1. This yields a generator *g* that satisfies *f*(*g*) = 0 with a value of *g*4 = *g* + 1 , or in binary, *g* = 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 E241*g*4, 12. Figure 10.6 plots the points of E241*g*4, 12.

It can be shown that a finite abelian group can be defined based on the set E2*m*(*a*, *b*), provided that *b* != 0. The rules for addition can be stated as follows. For all points *P*, *Q* Î *E*2*m*(*a*, *b*):

1. $P + O = P$.
2. 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, which is denoted as $-P$.
3. 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:

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

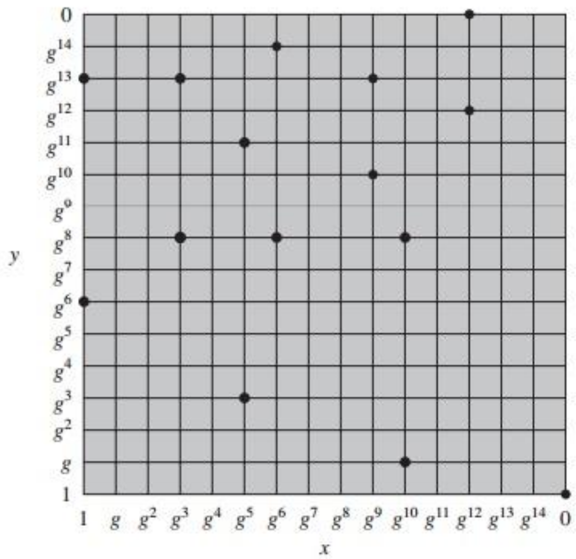


Figure 10.6 The Elliptic Curve $E_{2^4}(g^4, 1)$

3.6

where

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

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

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

where

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

3.6 ELLIPTIC CURVE CRYPTOGRAPHY

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple addition is 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 Î $EP(a, b)$ 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 c alled the discrete logarithm problem for elliptic curves.

We give an example taken from the Certicom Web site(www.certicom.com).

Consider the group E23(9,17). This is the group defined by the equation $y2$ mod 23 = ($x3 + 9x$ + 17) mod 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 = 17, 32; 7P = 18, 72; 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 $2m$, and elliptic curve parameters a and b for Equation (10.5) or Equation (10.7). This defines the elliptic group of points $Eq(a, b)$. Next, pick a *base point* $G = (x1, y1)$ in $Ep(a, b)$ 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 $nG = 0$ 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 10.7).

1. A selects an integer nA less than n. This is A's private key. A then generates a public key $PA = nA * G$; the public key is a point in $Eq(a, b)$.
2. B similarly selects a private key nB and computes a public key PB.
3. A generates the secret key $k = nA * PB$. B generates the secret key $k = nB * PA$.

The two calculations in step 3 produce the same result because

$n\mathrm{A} * PB = nA * (nB * G) = nB * (nA * G) = nB * PA$

To break this scheme, an attacker would need to be able to compute k given G

and kG, which is assumed to be hard.

As an example,6 take $p = 211$; Ep(0, - 4), which is equivalent to the curve

$y2 = x3 - 4$; and $G = (2, 2)$. One can calculate that $240G = O$. A's private key is

$nA = 121$, so A's public key is $PA = 121(2, 2) = (115, 48)$. B's private key

is $nB = 203$, so B's public key is 203(2, 3) = (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 gener- ated. We could simply use the x coordinates or some simple function of the x coordinate.

Global Public Elements	
$E_q(a, b)$	elliptic curve with parameters a, b, and q, where q is a prime or an integer of the form 2^m
G	point on elliptic curve whose order is large value n

User A Key Generation	
Select private n_A	$n_A < n$
Calculate public P_A	$P_A = n_A \times G$

User B Key Generation	
Select private n_B	$n_B < n$
Calculate public P_B	$P_B = n_B \times G$

Calculation of Secret Key by User A

$K = n_A \times P_B$

Calculation of Secret Key by User B

$K = n_B \times P_A$

Figure 10.7 ECC Diffie-Hellman Key Exchange

Elliptic Curve Encryption/Decryption

Several approaches to encryption/decryption using elliptic curves have been ana- lyzed 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 Pm. It is the point Pm that will be encrypted as a ciphertext and subsequently decrypted. Note that we cannot simply encode t he message as the x or y coordinate of a point, because not all such coordinates are in $Eq(a, b)$; for example, see Table

Again, there are several approaches to this encoding, which we will not

address here, but suffice it to say that there are relatively straightforward tech- niques 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 nA and generates a public key $PA = nA * G$.

To encrypt and send a message Pm to B, A chooses a random positive integer k

and produces the ciphertext Cm consisting of the pair of points:

$$Cm = \{kG, Pm + kPB\}$$

Note that A has used B's public key PB. 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:

$$Pm + kPB - nB(kG) = Pm + k(nBG) - nB(kG) = Pm$$

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

As an example of the encryption process (taken from [KOBL94]), take $p = 751$; $Ep(-1, 188)$, which is equivalent to the curve $y2 = x3 - x + 188$; and $G = (0, 376)$. Suppose that A wishes to send a message to B that is encoded in the elliptic point $Pm = (562, 201)$ and that A selects the random number $k = 386$. B's public key is $PB = (201, 5)$. 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 [JURI97]. 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

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
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

Source: Certicom

3.7 PSEUDORANDOM NUMBER GENERATION BASED ON AN ASYMMETRIC CIPHER

We noted in Chapter 7 that because a symmetric block cipher produces an apparently random output, it can serve as the basis of a pseudorandom number generator (PRNG). Si milarly, an asymmetric encryption algorithm produces apparently random output and can be used to build a PRNG. Because asymmetric algorithms are typically much slower than symmetric algorithms, asymmetric algorithms are not used to generate open-ended PRNG bit streams. Rather, the asymmetric approach is useful for creating a pseudorandom function (PRF) for generating a short pseudorandom bit sequence.

In this section, we examine two PRNG designs based on pseudorandom functions.

PRNG Based on RSA

For a sufficient key length, the RSA algorithm is considered secure and is a good candidate to form the basis of a PRNG. Such a PRNG, known as the Micali-Schnorr PRNG [MICA91], is recommended in the ANSI standard X9.82 (*Random Number Generation*) and in the ISO standard 18031 (*Random Bit Generation*).

The PRNG is illustrated in Figure 10.8. As can be seen, this PRNG has much the same structure as the o utput feedback (OFB) mode used as a PRNG (see Figure 7.3b and the portion of Figure 6.6a enclosed with a dashed box). In this case, the encryption algorithm is RSA rather than a symmetric block cipher. Also, a portion of the output is fed back to the next iteration of the encryption algorithm and the remain- der of the output is used as pseudorandom bits. The motivation for this separation of the output into two distinct parts is so that the pseudorandom bits from one stage do not provide input to the next stage. This separation should contribute to forward unpredictability.

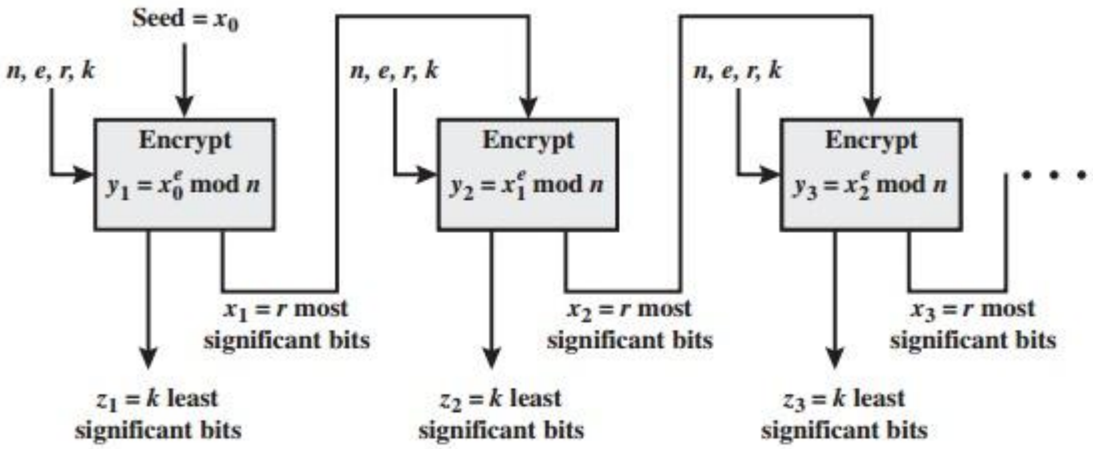


Figure 10.8 Micali-Schnorr Pseudorandom Bit Generator

We can define the PRNG as follows.

Setup Select p, q primes; $n = pq$; $\phi(n) = (p-1)(q-1)$. Select e such that $\gcd(e, \phi(n)) = 1$. These are the standard RSA setup selections (see Figure 9.5). In addition, let $N = \lfloor \log_2 n \rfloor + 1$ (the bitlength of n). Select r, k such that $r + k = N$.

Seed Select a random seed x_0 of bitlength r.

Generate Generate a pseudorandom sequence of length $k \times m$ using the loop
for i **from** 1 to m **do**

$$y_i = x_{i-1}^e \bmod n$$
$$x_i = r \text{ most significant bits of } y_i$$
$$z_i = k \text{ least significant bits of } y_i$$

Output The output sequence is $z_1 \| z_2 \| \ldots \| z_m$.

The parameters n, r, e, and k are selected to satisfy the following six requirements.

1. $n = pq$	n is chosen as the product of two primes to have the cryptographic strength required of RSA.
2. $1 < e < \phi(n)$; $\gcd(e, \phi(n)) = 1$	Ensures that the mapping $s \rightarrow s^e \bmod n$ is 1 to 1.
3. $re \geq 2N$	Ensures that the exponentiation requires a full modular reduction.
4. $r \geq 2\,strength$	Protects against a cryptographic attacks.
5. k, r are multiples of 8	An implementation convenience.
6. $k \geq 8$; $r + k = N$	All bits are used.

The variable *strength* in requirement 4 is defined in NIST SP 800-90 as follows: A number associated with the amount of work (that is, the number of operations) required to break a cryptographic algorithm or system; a security strength is speci- fied in bits and is a specific value from the set (112, 128, 192, 256) for this Recommendation. The amount of work needed is 2*strength*.

There is clearly a tradeoff between *r* and *k*. Because RSA is computation- ally intensive compared to a block cipher, we would like to generate as many pseudorandom bits per iteration as possible and therefore would
like a large value of *k*. However, for cryptographic strength, we would like *r* to be as large as poss ible.

For example, if e = 3 and N = 1024 , then we have the inequality 3r 7 1024, yielding a minimum required size for r of 683 bits. For r set to that size, k = 341 bits are generated for each exponentiation (each RSA encryption). In this case, each exponentiation requires only one modular squaring of a 683-bit number and one modular multiplication. That is, we need only calculate $1x_i$ * $1x_i$ mod n22mod n .

PRNG Based on Elliptic Curve Cryptography

In this subsection, we briefly summarize a technique developed by the U.S. National Security Agency (NSA) known as dual elliptic curve PRNG (DEC PRNG). This technique is recommended in NIST SP 800-90, the ANSI standard X9.82, and the ISO standard 18031. There has been some controversy regarding both the security and efficiency of this algorithm compared to other alternatives (e.g., see [SCHO06], [BROW07]).

[SCHO06] summarizes the algorithm as follows: Let P and Q be two known points on a given elliptic curve. The seed of the DEC PRNG is a random integer *s*0 Î {0, 1, ... , #E(GF(p)) - 1}, where # E(GF(p)) denotes the number of points on the curve. Let x denote a function that gives the *x*-coordinate of a point of the curve. Let *lsbi*1*s*2 denote the *i* least significant bits of an integer *s*. The DEC PRNG transforms the seed into the pseudorandom sequence of length 240*k*, *k* > 0, as follows.

```
for i = 1 to k do
   Set s_i ← x(s_{i-1} P)
   Set r_i ← lsb_240 (x(s_i Q))
end for
   Return r_1, . . . , r_k
```

Given the security concerns expressed for this PRNG, the only motivation for its use would be that it is used in a system that already implements ECC but does not implement any other symmetric, asymmetric, or hash cryptographic algorithm that could be used to build a PRNG.

UNIT-IV

4.1 MESSAGE AUTHENTICATION CODES

- **Message Authentication Requirements**
- **Message Authentication Functions**
  - Message Encryption
  - Message Authentication Code
- **Requirements for Message Authentication Codes**
- **Security of MACs**
  - Brute-Force Attacks Cryptanalysis
- **MACs Based on Hash Functions: HMAC**
  - HMAC Design Objectives HMAC Algorithm Security of HMAC
- **MACs Based on Block Ciphers: DAA and CMAC**
  - Data Authentication Algorithm
  - Cipher-Based Message Authentication Code (CMAC)
- **Authenticated Encryption: CCM and GCM**
  - Counter with Cipher Block Chaining-
    Message Authentication Code Galois/Counter Mode
- **Pseudorandom Number Generation Using Hash Functions and Macs**
  - PRNG Based on Hash function PRNG Based on MAC function

4.2 MESSAGE AUTHENTICATION REQUIREMENTS

In the context of communications across a network, the following attacks can be identified.

Disclosure: Release of message contents to any person or process not possessing the appropriate cryp tographic key.

Traffic analysis: Discovery of the pattern of traffic between parties. In a connection-oriented application, the frequency and duration of connections could be determined. In either a connection-oriented or connectionless environment, the number and length of messages between parties could be d etermined.

Masquerade: Insertion of messages into the network from a fraudulent source. This includes the creation of messages by an opponent that are purported to come from an authorized entity. Also included are fraudulent acknowledgments of message receipt or non-receipt by someone other than the message recipient.

Content modification: Changes to the contents of a message, including insertion, deletion, transposition, a nd modification.

Sequence modification: Any modification to a sequence of messages between parties, including inserti on, deletion, and reordering.

Timing modification: Delay or replay of messages. In a connectionoriented application, an entire sess ion or sequence of messages could be a replay of someprevious valid session, or individual messages in the sequence could be delayed or replayed. In a connectionless application, an individual message (e.g., data- gram) could be delayed or replayed.

Source repudiation: Denial of transmission of message by source.

Destination repudiation: Denial of receipt of message by destination.

Measures to deal with the first two attacks are in the realm of message confidentiality and are dealt with in Part One. Measures to deal with items (3) through(2) in the foregoing list are generally regarded as message authentication. Mechanisms for dealing specifically with item (7) come under t he heading of digitalsignatures. Generally, a digital signature technique will also counter some or all of the attacks listed under items (3) through (6). Dealing with item (8) may require a combination of the use of digital signatures and a protocol designed to counter this attack.

In summary, message authentication is a procedure to verify that received messages come from the alleged source and have not been altered. Message authentication may also verify Sequencing and timeliness. A digital signature is an authentication technique that also includes measures to counter repudiation by the source.

4.3 MESSAGE AUTHENTICATION FUNCTIONS

Any message authentication or digital signature mechanism has two levels of functionality. At the lower level, there must be some sort of function that produces an authenticator: a value to be used to authenticate a message. This lower-level function is then used as a primitive in a higher-level authentication protocol that enables a receiver to verify the authenticity of a message.

This section is concerned with the types of functions that may be used to produce an authenticator. These may be grouped into three classes.

Hash function: A function that maps a message of any length into a fixed-length hash value, which serves as the authenticator

Message encryption: The ciphertext of the entire message serves as its authenticator

Message authentication code (MAC): A function of the message and a secret key that produces a fixed-length value that serves as the authenticator Hash functions, and how they may serve for messageauthentication,are discussed in Chapter 11. The remainder of this section briefly examines th e remaining two topics. The remainder of the chapter elaborates on the topic of MACs.

Message encryption by itself can provide a measure of authentication. The analysis differs for symmetric and public-key encryption schemes.

SYMMETRIC ENCRYPTION Consider the straightforward use of symmetric encryption (Figure 1 2.1a). A message *M* transmitted from source A to destination B is encrypted using a secret key *K* shared by A and B. If no other party knows the key, then confidentiality is provided: No other party can recover the plaintext of the message. In addition, B is assured that the message was generated by A. Why? The message must have come from A, because A is the only other party that pos sesses *K* and therefore the only other party with the information necessary to construct cipher- text that can be decrypted with *K*. Furthermore, if *M* is recovered, B knows that none of the bits of *M* have been altered, because an opponent that does not know *K* would not know h ow to alter bits in the ciphertext to produce the desired changes in the plaintext.

So we may say that symmetric encryption provides authentication as well as confidentiality. However, this flat statement needs to be qualified. Consider exactly what is happening at B. Given a decryption function D and a secret key *K*, the destination will accept *any* Input*X* and produce output $Y = D(K, X)$. If *X* is the ciphertext of a legitimate message *M* produced by the corresponding encryption function, then *Y* is some plaintext message *M*. Otherwise, *Y* will likely be a meaningless sequence of bits. There may need to be some automated means of determining at B whether *Y* is legitimate plaintext and therefore must have come from A.

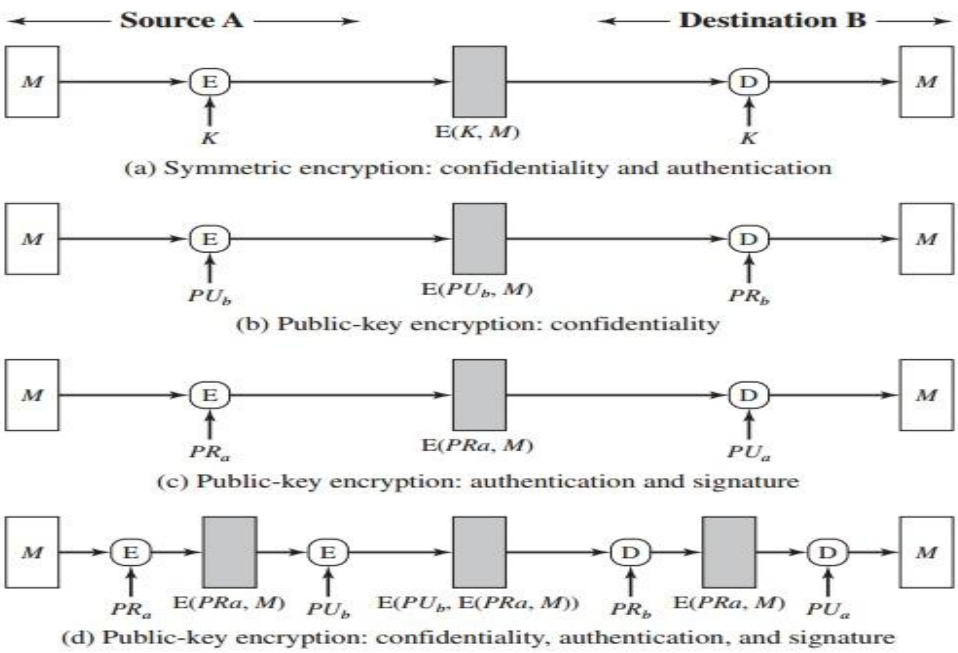


Figure 12.1 Basic Uses of Message Encryption

The implications of the line of reasoning in the preceding paragraph are profound from the point of view of authentication. Suppose the message *M* can be any arbitrary bit pattern. In that case, there is no way to determine automatically, at thedestination, whether an incoming message is the ciphertext of a legitimate message. This conclusion is incontrovertible: If *M* can be any bit pattern, then regardless of the value of *X*, the value $Y = D(K, X)$ is *some* bit pattern and therefore must be accepted as authentic plaintext.

Thus, in general, we require that only a small subset of all possible bit pat- terns be considered legitimate plaintext. In that case, any spurious ciphertext is unlikely to produce legitimate plaintext. For example, suppose that only one bit pattern in 106 is legitimate plaintext. Then the probability that any randomly chosen bit pattern, treated as ciphertext, will produce a legitimate plaintext message is only 10-6.

For a number of applications and encryption schemes, the desired conditions prevail as a matter of course. For example, suppose that we are transmitting English- language messages using a Caesar cipher with a shift of one ($K = 1$). A sends the following legitimate ciphertext:

Nbsftfbupbutboeepftfbupbutboemjuumfmbnctfbujwz

B decrypts to produce the following plaintext:

mareseatoatsanddoeseatoatsandlittlelambseativy

A simple frequency analysis confirms that this message has the profile of ordinary English. On the other hand, if an opponent generates the following random sequence of letters:

zuvrsoevgqxlzwigamdvnmhpmccxiuureosfbcebtqxsxq

this decrypts to

ytuqrndufpwkyvhfzlcumlgolbbwhttqdnreabdaspwrwp

which does not fit the profile of ordinary English.

It may be difficult to determine *automatically* if incoming ciphertext decrypts to intelligible plaintext. If the plaintext is, say, a binary object file or digitized X-rays, determination of properly formed and therefore authentic plaintext may be difficult. Thus, an opponent could achieve a certain level of disruption simply by issuing messages with random content purporting to come from a legitimate user.

One solution to this problem is to force the plaintext to have some structure that is easily recognized but that cannot be replicated without recourse to the encryption function. We could, for example, append an error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption, as illustrated in Figure 12.2a. A prepares a plaintext message *M* and then provides this as input to a function F that produces an FCS. The FCS is appended to *M* and the entire block is then encrypted. At the destination, B

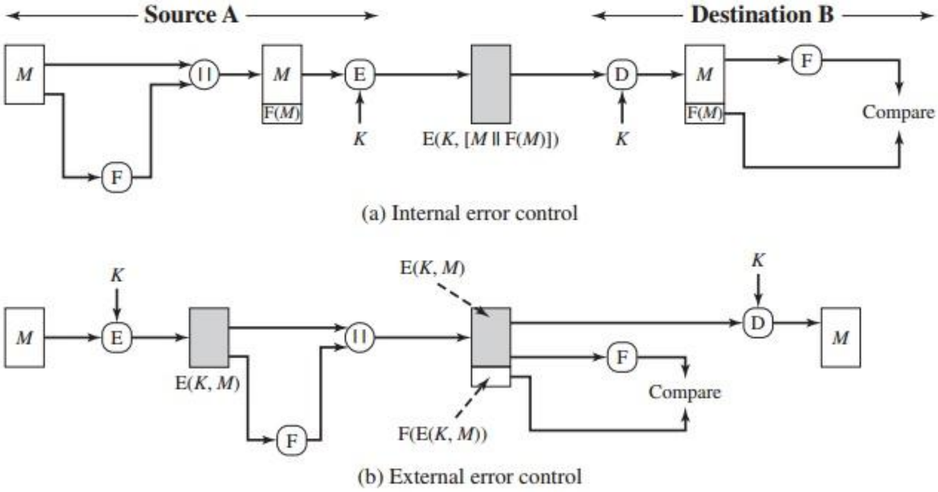


Figure 12.2 Internal and External Error Control

decrypts the incoming block and treats the results as a message with an appended FCS. B applies the same function F to attempt to reproduce the FCS. If the calcu- lated FCS is equal to the incoming FCS, then the message is considered authentic. It is unlikely that any random sequence of bi ts would exhibit the desired relationship.

Note that the order in which the FCS and encryption functions are performed is critical.

The sequenceillustrated in Figure 12.2a as **internal error control**, which the authors contrast wi th **external error control** (Figure 12.2b). With internal error control, authentication is provided because an opponent would have difficulty generating ciphertext that, when decrypted, would have valid error control bits. If instead the FCS is the outer code, an opponent can construct messages with valid error-control codes. Although the opponent cannot know what the decrypted plaintext will be, he or she can still hope to create confusion and disrupt operations.

An errorcontrol code is just one example; in fact, any sort of structuring added to the transmitted message serves to strengthen the authentication capability.Such structure is provided by the use of a co mmunications architecture consisting of layered protocols. As an example, consider the structure of messages transmitted using the TCP/IP protocol architecture.

Figure 12.3 shows the format of a TCP segment, illustrating the TCP header. Now suppose that each pair of hosts shared a unique secret key, so that all exchanges between a pair of hosts used the same key, regardless of application. Then we could simply encrypt all of the datagram except the IP header. Again, if an opponent substituted some arbitrary bit pattern for the encrypted TCP segment, the resulting plaintext would not include a meaningful header. In this case, the header includes not only a checksum (which covers the header) but also other useful information, such as the sequence number. Because successive TCP segments on a given connection are numbered sequentially, encryp- tion assures that an opponent does not delay, misorder, or delete any segments.

PUBLIC-KEY ENCRYPTION

The straightforward use of publickey encryption (Figure 12.1b) provides confidentiality but not a uthentication. The source (A) usesthe public key *PUb* of the destination (B) to encrypt *M*. Because only B has the

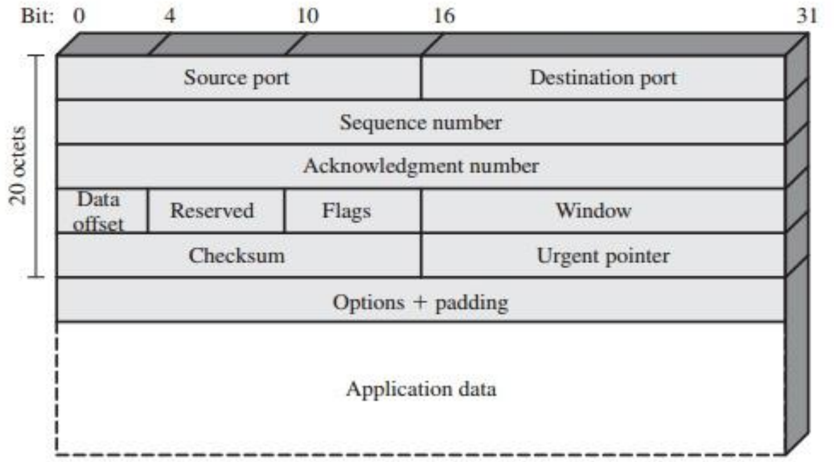


Figure 12.3 TCP Segment

Corresponding private key *PRb*, only B can decrypt the message.

This scheme provides no authentication, because any opponent could also use B's public key to encrypt a message and claim to be A.

To provide authentication, A uses its private key to encrypt the message, and Buses A's public key to decrypt (Figure 12.1c). This provides authentication using the same type of reasoning as in the symmetric encryption case: The message must havecome from A because A is the only party that possesses *PRa* and therefore the only party with the information necessary to construct ciphertext that can be decrypted with *PUa*. Again, the same reasoning as before applies: There must be some internal structure to the plaintext so that the receiver can distinguish between well-formed plaintext and random bits.

Assuming there is such structure, then the scheme of Figure 12.1c does provide authentication. It also provides what is known as digtal signature.1 Only A could have constructed the ciphertext because only A possesses *PRa*.Not even B, the recipient, could have constructed the ciphertext. Therefore, if B is in possession ofthe ciphertext, B has the means to prove that the message must have come from A.In effect, A has "signed" the message by using its private key to encrypt. Note thatthis scheme doe s not provide confidentiality. Anyone in possession of A's public key can decrypt the ciphertext.
To provide both confidentiality and authentication, A can encrypt *M* first using its private key, which provides the digital signature, and then using B's public key, which provides confidentiality (Figure 12.1d). The disadvantage of this approach is that the publickey algorithm, which is comple x, must be exercised four times rather than two in each communication.

Message Authentication Code

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as a **cryptographic checksum** or MAC, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key *K*. When A has a message to send to B, it calculates the MAC as a function of the message and the key:

MAC = MAC(*K*, *M*)

where

M = input message

C = MAC function

K = shared secret key

MAC = message authentication code

The message plus MAC are transmitted to the intended recipient. The recipient per- forms the same calculation on the received message, using the same secret key, to

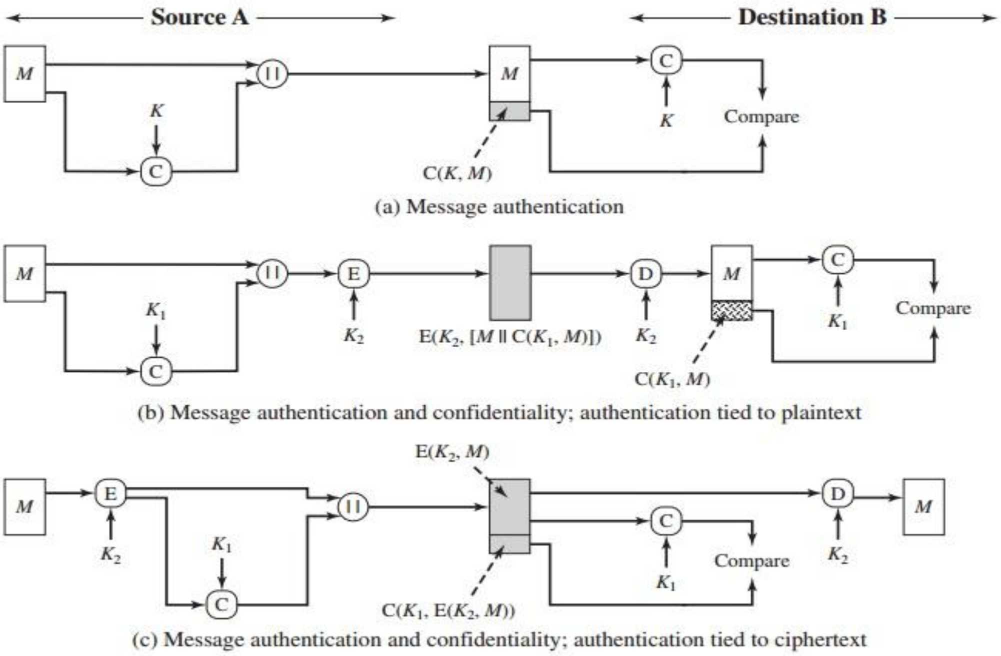


Figure 12.4 Basic Uses of Message Authentication code (MAC)

generate a new MAC. The received MAC is compared to the calculated MAC (Figure 12.4a). If we assume that only the receiver and the sender know the identity of the secret key, and if the received MAC matches the calculated MAC, then

1. The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the MAC, then the receiver's calculation of the MAC will differ from the received MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to cor- respond to the alterations in the message.

2. The receiver is assured that the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper MAC.

3. If the message includes a sequence number (such as is used with HDLC, X.25, and TCP), then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.AMAC function is similar to encryption. On e difference is that the MAC algorithm need not be reversible, as it must be for decryption.

In general, the MAC function is a many-one function. The domain of the function consists of messages of some arbitrary length, whereas the range consists of all possible MACs and all possible keys. If an *n*-bit MAC is used, then there are 2*n* possible MACs, whereas there are *N* possible messages with *N* 77 2*n*. Furthermore, with a *k*-bit key, there are 2*k* possible keys.
For example, suppose that we are using 100-bit messages and a 10-bit MAC. Then, there are a total of 2100 different messages but only 210 different MACs. So, on average, each MAC value is generated by a total of 2100/210 = 290 different messages. If a 5-bit key is used, then there are 25 = 32 different mappings from the set of messages to the set of MAC values.

It turns out that, because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption.

The process depicted in Figure 12.4a provides authentication but not confidentiality, because the messa ge as a whole is transmitted in the clear. Confidentiality canbe provided by performing message encryp tion either after (Figure 12.4b) or before(Figure 12.4c) the MAC algorithm. In both these cases, two sep arate keys are needed,each of which is shared by the sender and the receiver. In the first case, the MA C is calculated with the message as input and is then concatenated to the message. The entire block is then encrypted. In the second case, the message is encrypted first. Then the MAC is calculated using the resulting ciphertext and is concatenated to theciphertext to form the transmitted block. Typically, it is preferable to tie the authentication directly to the plaintext, so the method of Figure 12.4b is used.

Because symmetric encryption will provide authentication and because it is widely used with readily available products, why not simply use this instead of a separate message authentication code? suggests three situations in which a message authentication code is used.

1. There are a number of applications in which the same message is broadcast toa number of destina tions. Examples are notification to users that the networkis now unavailable or an alarm signal in a m ilitary control center. It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication code. The responsible system has the secret key and performs authentication. If a violation occurs, the other destination systems are alerted by a general alarm.

2. Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages. Authentication is carried out on a selective basis, messages being chosen at random for checking.

3. Authentication of a computer program in plaintext is an attractive service. The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources.However, if a message authentiation code were attached to the program, it could be checked whenever assurance was required of the integrity of the program.

Three other rationales may be added.

For some applications, it may not be of concern to keep messages secret, but it is important to authenticate messages. An example is the Simple Network Management Protocol Version 3 (SNMPv3), which separates the functions of confidentiality and auth entication. For this application, it is usually important for a managed system to authenticate incoming SN MP messages, particularly if themessage contains a command to change parameters at the managed syst em. On the other hand, it may not be necessary to conceal the SNMP traffic.

4. Separation of authentication and confidentiality functions affords architectural flexibility.

For example, it may be desired to perform authentication at the application level but to provide confiden tiality at a lower level, such as the transport layer.

5.A user may wish to prolong the period of protection beyond the time of reception and yet allow proc essing of message contents. With message encryption, the protection is lost when the message is decrypted, so the message is pro- tected against fraudulent modifications only in transit but Not within the target system.

6. Finally, note that the MAC does not provide a digital signature, because both sender and receiver share the same key.

4.4 REQUIREMENTS FOR MESSAGE AUTHENTICATION CODES

A MAC, also known as a cryptographic checksum, is generated by a function C of the form

$T = \text{MAC}(K, M)$

where M is a variable-length message, K is a secret key shared only by sender and receiver, and MAC(K, M) is the fixed-length authenticator, sometimes called a **tag**. The tag is appended to the message at the source at a time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the tag.

When an entire message is encrypted for confidentiality, using either sym- metric or asymmetric encryption, the security of the scheme generally depends on the bit length of the key. Barring some weakness in the algorithm, the opponent must resort to a brute-force attack using all possible keys. On average, such an attack will require $2(k - 1)$ attempts for a k-bit key. In particular, for a ciphertext- only attack, the opponent, given ciphertext C, performs $Pi = D(Ki, C)$ for all pos- sible key values Ki until a Pi is produced that matches the form of acceptable plaintext.In the case of a MAC, the considerations are entirely different. In gener al, the MAC function is a many-to-one function, due to the many-to-one nature of the function. Using brute-force methods, how would an opponent attempt to discover a key? If confidentiality is not employed, the opponent has access to plaintext mes- sages and their associated MACs. Suppose k 7 n; that is, suppose that the key size is greater than th e MAC size. Then, given a known $M1$ and $T1$, with $T1 = \text{MAC}(K, M1)$, the cryptanalyst can perform $Ti = \text{MAC}(Ki, M1)$ for all possi- ble key values ki. At least one key is guaranteed to produce a match of $Ti = T1$. Note that a total of $2k$ tags will be produced, but there are only

2*n* 6 2*k* different tagvalues. Thus, a number of keys will produce the correct tag and the opponent has no way of knowing which is the correct key. On average, a total of 2*k*/2*n* = 2(*k* - *n*) keys will produce a match. Thus, the opponent must iterate the attack.

- **Round 1**

$$\begin{aligned}&\text{Given: } M_1, T_1 = \text{MAC}(K, M_1)\\ &\text{Compute } T_i = \text{MAC}(K_i, M_1) \quad \text{for all } 2^k \text{ keys}\\ &\text{Number of matches} \approx 2^{(k-n)}\end{aligned}$$

- **Round 2**

$$\begin{aligned}&\text{Given: } M_2, T_2 = \text{MAC}(K, M_2)\\ &\text{Compute } T_i = \text{MAC}(K_i, M_2) \quad \text{for the } 2^{(k-n)} \text{ keys resulting from Round 1}\\ &\text{Number of matches} \approx 2^{(k-2\times n)}\end{aligned}$$

And so on. On average, a rounds will be needed if *k* = *a* * *n*. For example, if an 80-bit key is used and the tab is 32 bits, then the first round will produce about 248 possible keys. The second round will narrow the possible keys to about 216 possibili- ties. The third round should produce only a single key, which must be the one used by the sender.

If the key length is less than or equal to the tag length, then it is likely that a first round will produce a single match. It is possible that more than one key will produce such a match, in which case the opponent would need to perform the same test on a new (message, tag) pair.

Thus, a brute-force attempt to discover the authentication key is no less effort and may be more effort than that required to discover a decryption key of the same length. However, other attacks that do not require the discovery of the key are possible.

Consider the following MAC algorithm. Let *M* = (*X*1 || *X*2 || Á || *Xm*) be a message that is treated as a concatenation of 64-bit blocks *Xi*. Then define

$$\begin{aligned}\Delta(M) &= X_1 \oplus X_2 \oplus \cdots \oplus X_m\\ \text{MAC}(K, M) &= \text{E}(K, \Delta(M))\end{aligned}$$

where Ⓧ is the exclusive-OR (XOR) operation and the encryption algorithm is

DES in electronic codebook mode. Thus, the key length is 56 bits, and the tag length is 64 bits. If an opponent observes {*M* || MAC(*K*, *M*)}, a brute-force attempt to determine *K* will require at least 256 encryptions. But the opponent can attack the system by replacing *X*1 through *Xm* - 1 with any desired values *Y*1 through *Ym* - 1 and replacing *Xm* with *Ym*, where *Ym* is calculated as

$$Y_m = Y_1 \oplus Y_2 \oplus \cdots \oplus Y_{m-1} \oplus \Delta(M)$$

The opponent can now concatenate the new message, which consists of *Y*1 through *Ym*, using the original tag to form a message that will be accepted as authentic by the receiver. With this tactic, any message of length 64 * (m - 1) bits can be fraudulently inserted.

Thus, in assessing the security of a MAC function, we need to consider the types of attacks that may be mounted against it. With that in mind, let us state the requirements for the function. Assume that an opponent knows the MAC function but does not know K. Then the MAC function should satisfy the following requirements.

1. If an opponent observes M and MAC(K, M), it should be computationally infeasible for the opponent to construct a message M¿ such that MAC(K,'') = MAC(K, M)

2. MAC(K, M) should be uniformly distributed in the sense that for randomly chosen messages, M and M¿, the probability that MAC(K, M) = MAC(K, M') is 2 -n, where n is the number of bits in the tag.

3. Let M'''' be equal to some known transformation on M. That is, M'' = f(M). For example, f may involve inverting one or more specific bits. In that case,Pr [MAC(K, M) = MAC(K, M')] = 2-n

The first requirement speaks to the earlier example, in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key. The second requirement deals with the need to thwart a brute-force attack based on chosen plaintext. That is, if we assume that the opponent does not know K but does have access to the MAC function and can present messages for MAC generation, then the opponent could try various messages until finding one that matches a given tag. If the MAC function exhibits uniform distribution, then a brute-force method would require, on average, 2(n - 1) attempts before finding a message that fits a given tag.

The final requirement dictates that the authentication algorithm should not beweaker with respect to certain parts or bits of the message than others. If this were not the case, then an opponent who had M and MAC(K, M) could attempt varia tions on M at the known "weak spots"with a likelihood of early success at producing a new message that matched the old tags.

4.5 SECURITY OF MACS

Just as with encryption algorithms and hash functions, we can group attacks on MACs into two categories: brute-force attacks and cryptanalysis.

Brute-Force Attacks

A brute-force attack on a MAC is a more difficult undertaking than a brute-force attack on a hash function because it requires known message-tag pairs. Let us see why this is so. To attack a hash code, we can proceed in the following way. Given a fixed message x with n-bit hash code h = H(x), a brute-force method of finding a collision is to pick a random bit string y and check if H(y) = H(x). The attacker can do this repeatedly off line. Whether an off-line attack can be used on a MAC Algorithm depends on the relative size of the key and the tag.

To proceed, we need to state the desired security property of a MAC algo- rithm, which can be expressed as follows.

Computation resistance: Given one or more text-MAC pairs [*xi*, MAC(*K*, *x*i)], it is

Computationally infeasible to compute any text-MAC pair [*x*, MAC(*K*, *x*)] for any new input *x* |=*xi*.

In other words, the attacker would like to come up with the valid MAC code for a given message *x*. There are two lines of attack possible: attack the key space and attack the MAC value. We examine each of these in turn.

If an attacker can determine the MAC key, then it is possible to generate a valid MAC value for any input *x*. Suppose the key size is *k* bits and that the attacker has one known text–tag pair. Then the attacker can compute the *n*-bit tag on the known text for all possible keys. At least one key is guaranteed to produce the correct tag, namely, th e valid key that was initially used to produce the known text–tag pair. This phase of the attack takes a level of effort proportional to 2*k* (that is, one operation for each of the 2*k* possible key values). However, as was described earlier, because the MAC is a many-to-one mapping, there may be other keys that produce the correct value. Thus, if more than one key is found to produce the correct value, additional text–tag pairs must be tested. It can be shown that the level of effort drops off rapidly with each additional text–MAC pair and that the overall level of effort is roughly 2*k* [MENE97].

An attacker can also work on the tag without attempting to recover the key. Here, the objective is to generate a valid tag for a given message or to find a message that matches a given tag. In either case, the level of effort is comparable to that for attacking the one-way or weak collision-resistant property of a hash code, or 2*n*. In the case of the MAC, the attack cannot be conducted off line without further input; the attacker will require chosen text–tag pairs or knowledge of the key.

To summarize, the level of effort for brute-force attack on a MAC algorithm can be expressed as min(2*k*, 2*n*). The assessment of strength is similar to that for symmetric encryption algorithms. It would appear reasonable to require that the key length and tag length satisfy a relationship such as min(*k*, *n*) Ú *N*, where *N* is perhaps in the range of 128 bits.

Cryptanalysis

As with encryption algorithms and hash functions, cryptanalytic attacks on MAC algorithms seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a MAC algorithm to cryptanalysis is to compare its strength to the effort required for a brute-force attack. That is, an ideal MAC algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.

There is much more variety in the structure of MACs than in hash functions, so it is difficult to generalize about the cryptanalysis of MACs. Furthermore, far less work has been done on developing such attacks. A useful survey of some methods for specific MACs is [PREN96].

4.6 MACS BASED ON HASH FUNCTIONS: HMAC

Later in this chapter, we look at examples of a MAC based on the use of a symmetric block cipher. This has traditionally been the most common approach to constructing a MAC. In recent years, there has been increased interest in developing a MAC derived from a cryptographic hash function. The motivations for this interest are

1. Cryptographic hash functions such as MD5 and SHA generally execute faster in software than symmetric block ciphers such as DES.

2. Library code for cryptographic hash functions is widely available.

With the development of AES and the more widespread availability of code for encryption algorithms, these considerations are less significant, but hash-based MACs continue to be widely used.

A hash function such as SHA was not designed for use as a MAC and cannot be used directly for that purpose, because it does not rely on a secret key. There have been a number of proposals for the incorporation of a secret key into an existing hash algorithm. The approach that has received the most support is HMAC [BELL96a, BELL96b]. HMAC has been issued as RFC 2104, has been chosen as the mandatory-to-implement MAC for IP security, and is used in other Internet protocols, such as SSL. HMAC has also been issued as a NIST standard (FIPS 198).

HMAC Design Objectives

RFC 2104 lists the following design objectives for HMAC.

- To use, without modifications, available hash functions. In particular, to use hash functions that perform well in software and for which code is freely and widely available.
- To allow for easy replace ability of the embedded hash function in case faster or more secure hash functions are found or required.
- To preserve the original performance of the hash function without incurring a significant degradation.
- To use and handle keys in a simple way.
- To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions about the embedded hash function.

The first two objectives are important to the acceptability of HMAC. HMAC treats the hash function as a "black box." This has two benefits. First, an existing implementation of a hash function can be used as a module in implementing HMAC. In this way, the bulk of the HMAC code is prepackaged and ready to use without modification. Second, if it is ever desired to replace a given hash function inan HMAC implementation, all that is required is to remove the existing hash function module and drop in the new module. This could be done if a faster hash function were desired. More important, if the security of the embedded hash function were compromised, the security of HMAC could be retained simply by replacing the embedded hash function with a more secure one (e.g., replacing SHA-2 with SHA-3).

The last design objective in the preceding list is, in fact, the main advantage of HMAC over other proposed hash-based schemes. HMAC can be proven secure provided that the embedded hash function has some reasonable cryptographic strength.

We return to this point later in this section, but first we examine the structure of HMAC.

HMAC Algorithm

Figure 12.5 illustrates the overall operation of HMAC. Define the following terms. H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)

IV = initial value input to hash function

M = message input to HMAC (including the padding specified in the embed- ded hash function)

Y_i = *i* th block of *M*,0 Š *i* Š (*L* – 1)

L = number of blocks in *M*

b = number of bits in a block

n = length of hash code produced by embedded hash function

K = secret key; recommended length is Š *n*; if key length is greater than *b*, the key is input to the hash function to produce an *n*-bit key

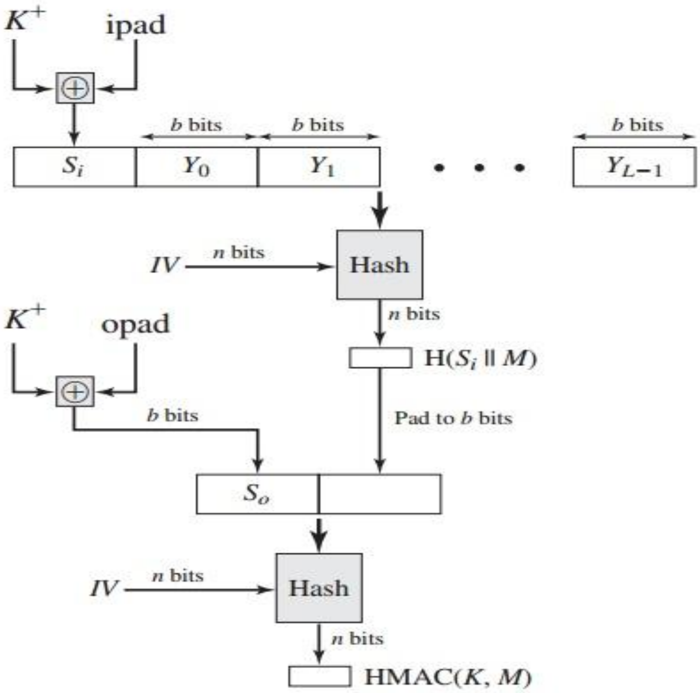


Figure 12.5 HMAC Structure

K^+ = *K* padded with zeros on the left so that the result is *b* bits in length ipad = 00110110 (36 in hexadecimal) repeated *b*/8 times

opad = 01011100 (5C in hexadecimal) repeated *b*/8 times Then HMAC can be expressed as

HMAC(*K*, *M*) = H[(*K*+ ⊗ opad) || H[(*K*+ ⊗ ipad) || *M*]]

We can describe the algorithm as follows.

1. Append zeros to the left end of *K* to create a *b*-bit string *K*+ (e.g., if *K* is of length 160 bits and *b* = 512, then *K* will be appended with 44 zeroes).
2. XOR (bitwise exclusive-OR) *K*+ with ipad to produce the *b*-bit block *Si*.
3. Append *M* to *Si*.
4. Apply H to the stream generated in step 3.
5. XOR *K*+ with opad to produce the b-bit block *So*.
6. Append the hash result from step 4 to *So*.
7. Apply H to the stream generated in step 6 and output the result.

Note that the XOR with ipad results in flipping one-half of the bits of *K*.

Similarly, the XOR with opad results in flipping onehalf of the bits of *K*, using a different set of bits. In effect, by passing *Si* and *So* through the compression function ofthe hash algorithm, we have pseud orandomly generated two keys from *K*.

HMAC should execute in approximately the same time as the embedded hashfunction for long messa ges. HMAC adds three executions of the hash compressionfunction (for *Si*, *So*, and the block produce d from the inner hash).

A more efficient implementation is possible, as shown in Figure 12.6. Two quantities are precomputed:

f(*IV*, (*K* + ⊗ ipad))

f(*IV*, (*K* + ⊗ opad))

where f(cv, block) is the compression function for the hash function, which takes as arguments a chaining variable of *n* bits and a block of *b* bits and produces a chaining variable of *n* bits. These quantities only need to be computed initially and every time the key changes. In effect, the precomputed quantities substitute for the initial value (*IV*) in the hash function. With this implementation, only one additional instance of the compression function is added to the processing normally produced by the hash function. This more efficient implementation is especially worthwhile if most of the messages for which a MAC is computed are short.

Security of HMAC

The security of any MAC function based on an embedded hash function depends in some way on the cryptographic strength of the underlying hash function. The appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC.

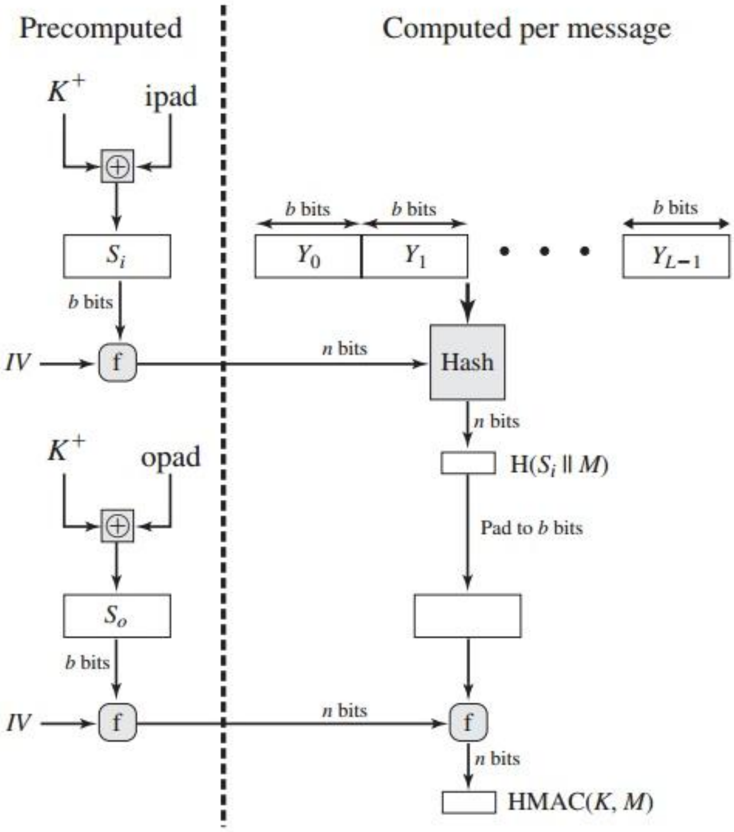


Figure 12.6 Efficient Implementation of HMAC

The security of a MAC function is generally expressed in terms of the probability of successful forgery with a given amount of time spent by the forger and a given number of message–tag pairs created with the same key. In essence, it is proved in [BELL96a] that for a given level of effort (time, message–tag pairs) on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function.

1. The attacker is able to compute an output of the compression function even with an *IV* that is random, secret, and unknown to the attacker.
2. The attacker finds collisions in the hash function even when the *IV* is random and secret.

In the first attack, we can view the compression function as equivalent to the hash function applied to a message consisting of a single *b*bit block. For this attack, the *IV* of the hash function is replaced by a secret, random value of *n* bits. An attack on this hash function requires either a brute-force attack on the key, which is a level of effort on the order of 2*n*, or a birthday attack, which is a special case of the second attack, discussed next.

In the second attack, the attacker is looking for two messages *M* and *M*¿ that produce the same ha sh: H(*M*) = H(*M*¿). This is the birthday attack discussed in Chapter 11. We have shown that this requires a level of effort of 2*n*/2 for a hash length of *n*. On this basis, the security of MD5 is called into question, because a level of effort of 264 looks feasible with today's technology. Does this mean that a 128-bit hash function such as MD5 is unsuitable for HMAC? The answer is no, because of the following argument. To attack MD5, the attacker can choose any set of messages and work on these off line on a dedicated computing facility to find a collision. Because the attacker knows the hash algorithm and the default *IV*, the attacker can generate the hash code for each of the messages that the attacker generates. However, when attacking HMAC, the attacker cannot generate message/code pairs off line because the attacker does not know *K*. Therefore, the attacker must observe a sequence of messages generated by HMAC under the same key and perform the attack on these known messages. For a hash code length of 128 bits, this requires 264 observed blocks (272 bits) generated using the same key. On a 1-Gbps link, one would need to observe a continuous stream of messages with no change in key for about 150,000 years in order to succeed. Thus, if speed is a concern, it is fully acceptable to use MD5 rather than SHA-1 as the embedded hash function for HMAC.

4.7 MACS BASED ON BLOCK CIPHERS: DAA AND CMAC

In this section, we look at two MACs that are based on the use of a block cipher mode of operation. We begin with an older algorithm, the Data Authentication Algorithm (DAA), which is now obsolete. Then we examine CMAC, which is designed to overcome the deficiencies of DAA.

Data Authentication Algorithm

The **Data Authentication Algorithm** (DAA), based on DES, has been one of the most widely used MACs for a number of years. The algorithm is both a FIPS publication (FIPS PUB 113) and an ANSI standard (X9.17). However, as we discuss subsequently, security weaknesses in this algorithm have been discovered, and it is being replaced by newer and stronger algorithms. The algorithm can be defined as using the cipher block chaining (CBC) mode of operation of DES (Figure 6.4) with an initialization vector of zero. The data (e.g., message, record, file, or program) tobe authenticated are grouped into contiguous 64-bit blocks: *D*1, *D*2, ... , *DN*. If necessary, the final block ispadded on the right with zeroes to form a full 64bit block. Using the DES encryption algori thm E and a secret key *K*, a data authentication code (DAC) is calculated as follows (Figure 12.7).

$$
\begin{aligned}
O_1 &= \mathrm{E}(K, D) \\
O_2 &= \mathrm{E}(K, [D_2 \oplus O_1]) \\
O_3 &= \mathrm{E}(K, [D_3 \oplus O_2]) \\
&\bullet \\
&\bullet \\
&\bullet \\
O_N &= \mathrm{E}(K, [D_N \oplus O_{N-1}])
\end{aligned}
$$

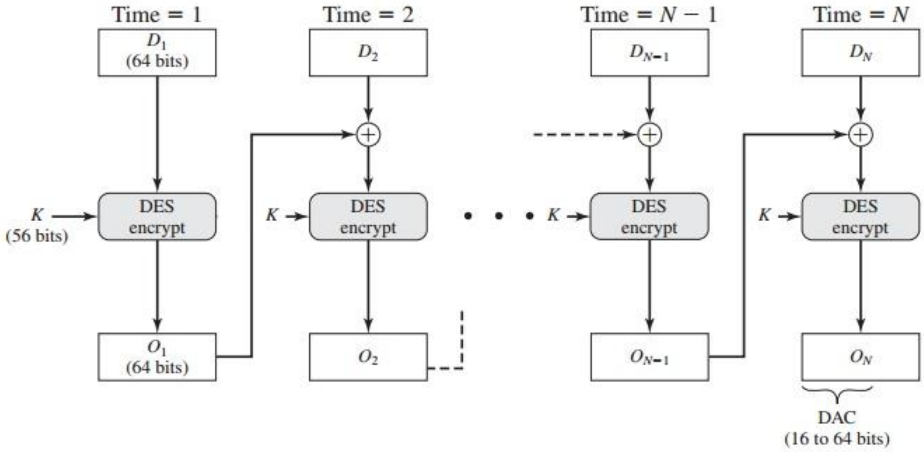


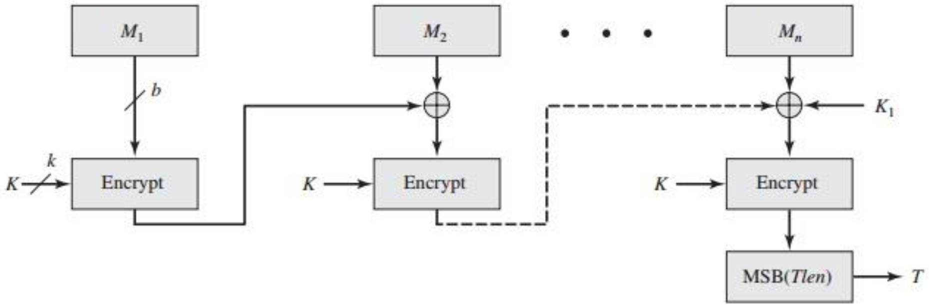
Figure 12.7 Data Authentication Algorithm (FIPS PUB 113)

The DAC consists of either the entire block *ON* or the leftmost *M* bits of the block, with 16 <= *M* <= 64.

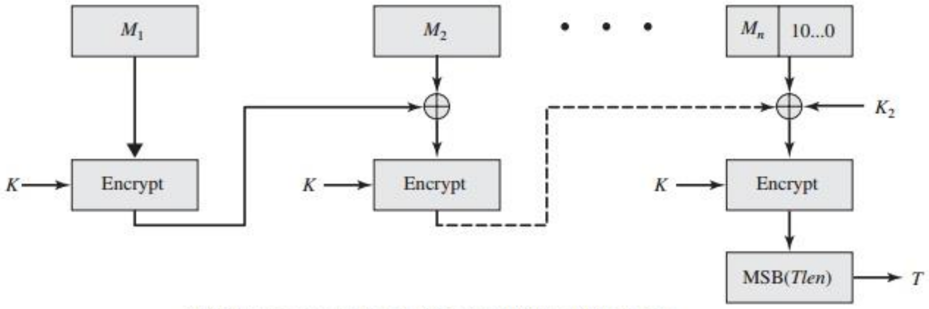
Cipher-Based Message Authentication Code (CMAC)

As was mentioned, DAA has been widely adopted in government and industry. [BELL00] demonstrated that this MAC is secure under a reasonable set of security criteria, with the following restriction. Only messages of one fixed length of *mn* bits are processed, where *n* is the cipher block size and *m* is a fixed positive integer. As a simple example, notice that given the CBC MAC of a one-block message *X*, say *T* = MAC(*K*, *X*), the adversary immediately knows the CBC MAC for the two- block message *X* || (*X* {*T*) since this is once again *T*.
Black and Rogaway [BLAC00] demonstrated that this limitation could be overcome using three keys: one key of length *K* to be used at each step of the cipher block chaining and two keys of length *n*, where *k* is the key length and *n* is the cipher block length. This proposed construction was refined by Iwata and Kurosawa so that the two *n*-bit keys could be derived from the encryption key, rather than being provided separately [IWAT03]. This refinement, adopted by NIST, is the **Cipher-based Message Authentication Code** (CMAC) mode of operation for use with AES and triple DES. It is specified in NIST Special Publication 800-38B.

First, let us define the operation of CMAC when the message is an integer multiple *n* of the cipher block length *b*. For AES, *b* = 128, and for triple DES, *b* = 64. The message is divided into *n* blocks (*M*1, *M*2, . . . , *Mn*). The algorithm makes use of a *k*-bit encryption key *K* and an *n*-bit constant, *K*1. For AES, the key size *k* is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits. CMAC is calculated as follows (Figure 12.8).



(a) Message length is integer multiple of block size



(b) Message length is not integer multiple of block size

Figure 12.8 Cipher-Based Message Authentication Code (CMAC)

$$
\begin{aligned}
C_1 &= E(K, M_1)\\
C_2 &= E(K, [M_2 \oplus C_1])\\
C_3 &= E(K, [M_3 \oplus C_2])\\
&\bullet\\
&\bullet\\
&\bullet\\
C_n &= E(K, [M_n \oplus C_{n-1} \oplus K_1])\\
T &= MSB_{Tlen}(C_n)
\end{aligned}
$$

where

T = message authetication code, also referred to as the tag
$Tlen$ = bit length of T
$MSB_s(X)$ = the s leftmost bits of the bit string X

If the message is not an integer multiple of the cipher block length, then the final block is padded to the right (least significant bits) with a 1 and as many 0s as necessary so that the final block is also of length b. The CMAC operation then proceeds as before, except that a different n-bit key $K2$ is used instead of $K1$.

The two n-bit keys are derived from the k-bit encryption key as follows.

$$
\begin{aligned}
L &= E(K, 0^n)\\
K_1 &= L \cdot x\\
K_2 &= L \cdot x^2 = (L \cdot x) \cdot x
\end{aligned}
$$

where multiplication (.) is done in the finite field GF(2n) and x and $x2$ are first- and

second order polynomials that are elements of GF(2n). Thus, the binary representation of x

consists of n - 2 zeros followed by 10; the binary representation of $x2$ consists of n - 3 zeros followed by 100. The finite field is defined with respect to an irreducible polynomial that is lexicographically first among all such polynomials with the minimum possible number of nonzero terms. For the two approved block sizes, the polynomials are $x64 + x4 + x3 + x + 1$ and $x128 + x7 + x2 + x + 1$.

To generate $K1$ and $K2$, the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting ciphertext by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. This property of finite fields of the form GF($2n$) was explained in the discussion of MixColumns in Chapter 5.

4.8 AUTHENTICATED ENCRYPTION: CCM AND GCM

Authenticated encryption (AE) is a term used to describe encryption systems that simultaneously protect confidentiality and authenticity (integrity) of communications. Many applications and protocols require both forms of security, but until recently the two services have been designed separately.

- **HtE: Hash-then-encrypt.** First compute the cryptographic hash function over M as $h = H(M)$. Then encrypt the message plus hash function: E(K, (M||h)).

- **MtE: MAC-then-encrypt.** Use two keys. First authenticate the plaintext by computing the MAC value as T = MAC($K1$, M). Then encrypt the message plus tag: E($K2$, (M || T). This approach is taken by the SSL/TLS protocols (Chapter 16).
- **EtM: Encrypt-then-MAC.** Use two keys. First encrypt the message to yield the ciphertext C = E($K2$, M). Then authenticate the ciphertext with T = MAC($K1$, C) to yield the pair (C, T). This approach is used in the IPsec protocol (Chapter 19).

E&M: Encrypt-and-MAC.

Use two keys. Encrypt the message to yield the ciphertext C = E($K2$, M). Authenticate the plaintext with T = MAC($K1$, M) to yield the pair (C, T). These operations can be performed in either order. This approach is used by the SSH protocol (Chapter 16).

Both decryption and verification are straightforward for each approach. For HtE, MtE, and E&M, decrypt first, then verify. For EtM, verify first, then decrypt. There are security vulnerabilities with all of these approaches. The HtE approach isused in the Wired Equivalent Privacy (WEP) protocol to p rotect WiFi networks. This approach had fundamental weaknesses and led to the replacement of the WE P proto- col. [BLAC05] and [BELL00] point out that there security concerns in each of the three encryption/MAC approaches listed above. Nevertheless, with proper design, any of these approach es can provide a high level of security. This is the goal of the two approaches discussed in this section, both of which have been standardized by NIST.

Counter with Cipher Block Chaining-Message Authentication CodeThe CCM mode of operation was standardized by NISTspecifically to support the securityrequirements of IEEE 802.11 WiFi wireless local area networks (Chapter 17), but can be used in any networking application requiring authenticated encryption. CCM is a variation of the encrypt-and-MAC approach to authenticated encryption. It is defined in NIST SP 800-38C.

The key algorithmic ingredients of CCM are the AES encryption algorithm (Chapter 5), the CTR mode of operation (Chapter 6), and the CMAC authentica- tion algorithm (Section 12.6). A single key *K* is used for both encryption and MAC algorithms. The i nput to the CCM encryption process consists of three elements.

1. Data that will be both authenticated and encrypted. This is the plaintext message *P* of

   data block.

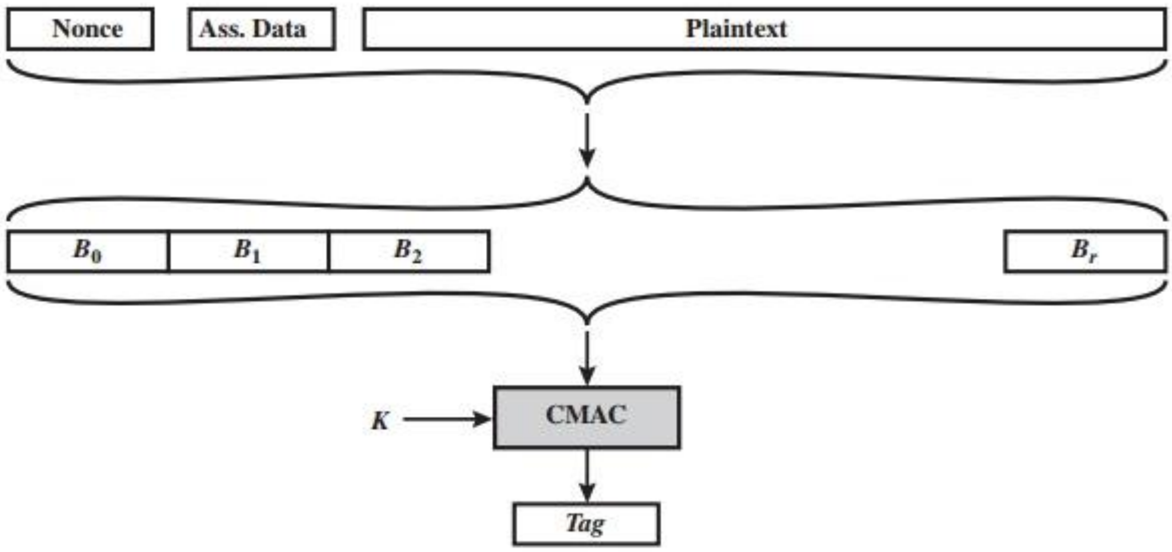
2. Associated data *A* that will be authenticated but not encrypted. An example is a protocol header that must be transmitted in the clear for proper protocol operation but which needs to be authenticated.
3. A nonce *N* that is assigned to the payload and the associated data. This is a unique value that is different for every instance during the lifetime of a proto- col association and is intended to prevent replay attacks and certain other types ofattacks.

   Figure 12.9 illustrates the operation of CCM. For authentication, the input includes the nonce, the associated data, and the plaintext. This input is formatted as a sequence of blocks *B*0 through *Br*. The first block contains the nonce plus some formatting bits that indicate the lengths of the *N*, *A*, and *P* elements. This is fol- lowed by zero or more blocks that contain *A*, followed by zero of more blocks that contain *P*. The resulting sequence of blocks serves as input to the CMAC algorithm, which produces a MAC value with length *Tlen*, which is less than or equal to the block length (Figure 12.9a).

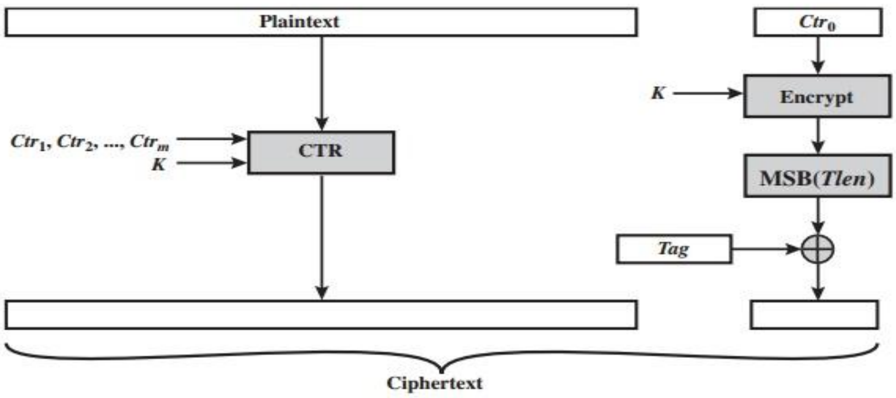
For encryption, a sequence of counters is generated that must be independent of the nonce. The authentication tag is encrypted in CTR mode using the single counter *Ctr*0. The *Tlen* most significant bits of the output are XORed with the tag to produ ce an encrypted tag. The remaining counters are used for the CTR mode encryption of the plaintext (Figure 6.7). The encrypted plaintext is concatenated with the encrypted tag to form the ciphertext output (Figure 12.9b).

SP 800-38C defines the authentication/encryption process as follows.

1. Apply the formatting function to (N, A, P) to produce the blocks $B_0, B_1, \ldots, B_r$.
2. Set $Y_0 = E(K, B_0)$.



(a) Authentication



(b) Encryption

Figure 12.9 Counter with Cipher Block Chaining-Message Authentication Code (CCM)

3. For $i = 1$ to r, do $Y_i = E(K, (B_i \oplus Y_{i-1}))$.
4. Set $T = MSB_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, \ldots, Ctr_m$, where $m = \lceil Plen/128 \rceil$.
6. For $j = 0$ to m, do $S_j = E(K, Ctr_j)$.
7. Set $S = S_1 \| S_2 \| \cdots \| S_m$.
8. Return $C = (P \oplus MSB_{Plen}(S)) \| (T \oplus MSB_{Tlen}(S_0))$.

For decryption and verification, the recipient requires the following input: the ciphertext C, the nonce N, the associated data A, the key K, and the initial counter $Ctr0$. The steps are as follows.

1. If $Clen$ £ $Tlen$, then return INVALID.
2. Apply the counter generation function to generate the counter blocks $Ctr0, Ctr1,$ Á, $Ctrm$, where $m = < Clen/128 >$.
3. For $j = 0$ to m, do $Sj = E(K, Ctrj)$.
4. Set $S = S1 \| S2 \|$ Á $\| Sm$.
5. Set P = MSB$Clen$-$Tlen$(C) {MSB$Clen$-$Tlen$(S).
6. Set T = LSB$Tlen$(C) NOR MSB$Tlen$($S0$).
7. Apply the formatting function to (N, A, P) to produce the blocks $B0, B1,$ Á , Br.
8. Set $Y0 = E(K, B0)$.
9. For i = 1 to r, do $Yi = E(K, (Bi$ NOR Yi-1)).
10. If T |=MSB$Tlen$ (Yr), then return INVALID, else return P

CCM is a relatively complex algorithm. Note that it requires two complete passes through the plaintext, once to generate the MAC value, and once for encryp- tion. Further, the details of the specification require a tradeoff between the length of the nonce and the length of the tag, which is an unnecessary restriction. Also note that the encryption key is used twice with the CTR encryption mode: once to generate the tag and once to en crypt the plaintext plus tag. Whether these complexities add to the security of the algorithm is not clear. In any case, two analyses of the algorithm ([JONS02] and [ROGA03]) conclude that CCM provides a high level of security.

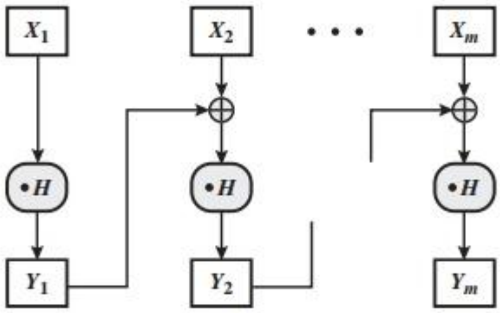
Galois/Counter Mode

The GCM mode of operation, standardized by NIST in NIST SP 800-38D, is designed to be parallelizable so that it can provide high throughput with low cost and low latency. In essence, the message is encrypted in variant of CTR mode. The resulting cipherte xt is multiplied with key material and message length informationover GF(2128) to generate the auth enticator tag. The standard also specifies a modeof operation that supplies the MAC only, known as GMAC.

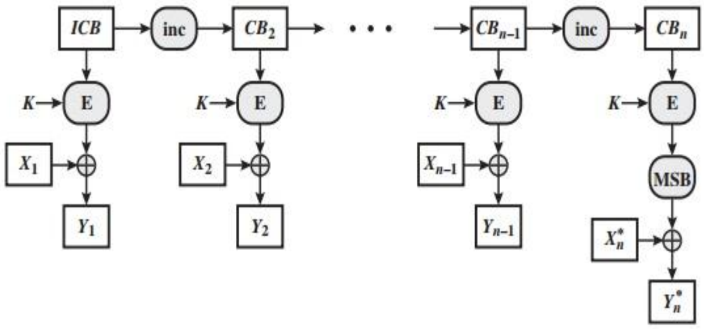
The GCM mode makes use of two functions: GHASH, which is a keyed hash function, and GCTR, which is essentially the CTR mode with the counters deter- mined by a simple increment by one operation.

GHASHH(X) takes a input the hash key H and a bit string X such that len(X) = 128m bits for some positive integer m and produces a 128bit MAC value. The function may be specified as follows (Figur e 12.10a).

1. Let $X_1, X_2, \ldots, X_{m-1}, X_m$ denote the unique sequence of blocks such that $X = X_1 \| X_2 \| \ldots \| X_{m-1} \| X_m$.
2. Let Y_0 be a block of 128 zeros, designated as 0^{128}.
3. For $i = 1, \ldots, m$, let $Y_i = (Y_{i-1} \oplus X_i) \cdot H$, where $\cdot$ designates multiplication in GF(2^{128}).
4. Return Y_m.



(a) $\text{GHASH}_H(X_1 \| X_2 \| \ldots \| X_m) = Y_m$



(b) $\text{GCTR}_K(ICB, X_1 \| X_2 \| \ldots \| X_n^*) = Y_n^*$

Figure 12.10 GCM Authentication and Encryption Functions

The $\text{GHASH}_H(X)$ function can be expressed as

$$(X_1 \cdot H^m) \oplus (X_2 \cdot H^{m-1}) \oplus \cdots \oplus (X_{m-1} \cdot H^2) \oplus (X_m \cdot H)$$

This formulation has desirable performance implications. If the same hash key is to be used to authenti cate multiple messages, then the values $H2$, $H3$, Á can be precalculated one time for use with each message to be authenticated. Then, the blocks of the data to be authenticated (X_1, X_2, Á , Xm) can be processed in parallel, because the computations are independent of one another.

GCTRK(ICB, X) takes a input a secret key K and a bit string X arbitrary length and returns a cipher text Y of bit length len(X). The function may be specified as follows (Figure 12.10b).

1. X is the empty string, then return the empty string as Y.
2. Let n = <(len(X)/128)>. That is, n is the smallest integer greater than or equal to len(X)/128.

3. Let $X_1, X_2, \ldots, X_{n-1}, X_n^*$ denote the unique sequence of bit strings such that

$$X = X_1 \| X_2 \| \ldots \| X_{n-1} \| X_n^*;$$
$$X_1, X_2, \ldots, X_{n-1} \quad \text{are complete 128-bit blocks.}$$

4. Let $CB_1 = ICB$.
5. For, $i = 2$ to n let $CB_i = \text{inc}_{32}(CB_{i-1})$, where the $\text{inc}_{32}(S)$ function increments the rightmost 32 bits of S by 1 mod 2^{32}, and the remaining bits are unchanged.
6. For $i = 1$ to $n - 1$, do $Y_i = X_i \oplus E(K, CB_i)$.
7. Let $Y_n^* = X_n^* \oplus \text{MSB}_{\text{len}(X_n^*)}(E(K, CB_n))$.
8. Let $X = X_1 \| X_2 \| \cdots \| X_{n-1} \| Y_n^*$
9. Return Y.

Note that the counter values can be quickly generated and that the encryption operations can be performed in parallel.

We can now define the overall authenticated encryption function (Figure 12.11). The input consists of a secret key K, an initialization vector IV, a plaintext P, and

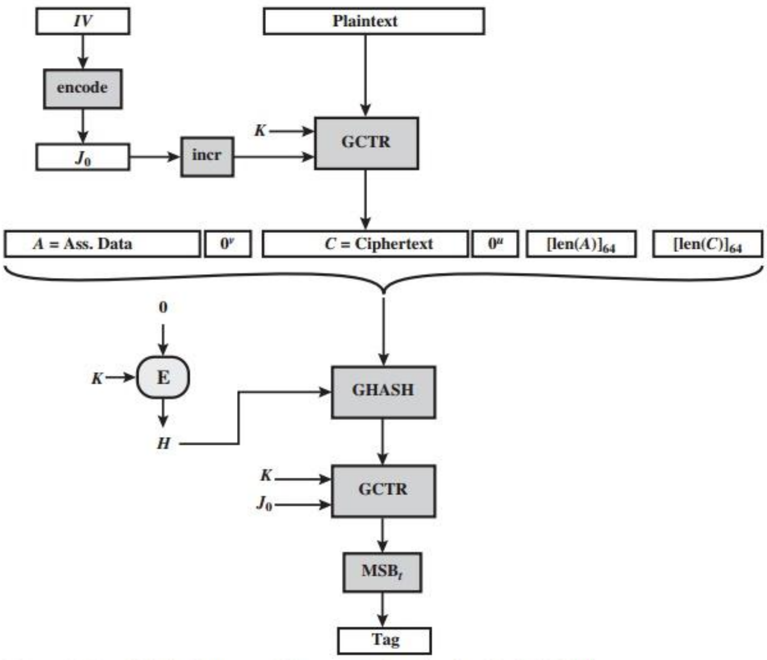


Figure 12.11 Galois Counter— Message Authentication Code (GCM)

additional authenticated data A. The notation $[x]s$ means the s-bit binary representation of the nonnegative integer x. The steps are as follows.

1. Let $H = E(K, 0128)$.

2. Define a block, $J0$, as If len(IV) = 96, then let $J0 = IV \| 031 \|$

1.If len (IV) |=96, then let $s = 128 <$ len(IV)/128 = -len(IV), and let

$J0$ = GHASHH($IV \| 0s + 64 \|$ [len(IV)]64).

3. Let C = GCTRK(inc32($J0$), P).

4. Let *u* = 128 <len(*C*)/128> - len(*C*) and let *v* = 128 <len(*A*)/128> -len(*A*).

5. Define a block, S, as*S* = GHASH*H*(*A* || 0*v* || *C* || 0*u* || [len(*A*)]64 || [len(*C*)]64)

6. Let *T* = *MSBt*(*GCTRK*1*J*0, *S*)), where *t* is the supported tag length.

7. Return (*C*, *T*).

In step 1, the hash key is generated by encrypting a block of all zeros with the secret key *K*. In step 2, the pre-counter block (*J*0) is generated from the *IV*. In par- ticular, when the length of the *IV* is 96 bits, then the padding string 031 || 1 is appended to the *IV* to form the pre-counter block. Otherwise,
the *IV* is padded with the minimum number of 0 bits, possibly none, so that the length of the result- ing string is a multiple of 128 bits (the block size); this string in turn is appended with 64 additional 0 bits, followed by the 64-bit representation of the length of the *IV*, and the GHASH function is applied to the resulting string to form the pre- counter block.

Thus, GCM is based on the CTR mode of operation and adds a MAC that authenticates both the message and additional data that requires only authentica- tion. The function that computes the hash uses only multiplication in a Galois field. This choice was made because the operation of multiplication is easy to perform within a Galois field and is easily implemented in hardware [MCGR05].

[MCGR04] examines the available block cipher modes of operation and shows that a CTR-based authenticated encryption approach is the most efficient mode of operation for high-speed packet networks. The paper further demonstrates
that GCM meets a high level of security requirements.

4.9 DIGITAL SIGNATURES

- **Digital Signatures**
  - Properties
  - Attacks and Forgeries
  - Digital Signature Requirements Direct Digital Signature
- **ElGamal Digital Signature Scheme**
- **Schnorr Digital Signature Scheme**
- **Digital Signature Standard**
  - The DSS Approach
  - The Digital Signature Algorithm

4.10 ELGAMAL DIGITAL SIGNATURE SCHEME

Before examining the NIST Digital Signature standard, it will be helpful to under-
stand the ElGamal and Schnorr signature schemes. Recall from Chapter 10, that the
ElGamal encryption scheme is designed to enable encryption by a user's public key with decryption

by the user's private key. The ElGamal signature scheme involves the use of the private key for encryption and the public key for decryption [ELGA84, ELGA85].

Before proceeding, we need a result from number theory. Recall from Chapter 8
that for a prime number *q*, if *a* is a primitive root of *q*, then

$$\alpha, \alpha^2, \ldots, \alpha^{q-1}$$

are distinct (mod *q*). It can be shown that, if *a* is a primitive root of *q*, then

1. For any integer m, $\alpha^m \equiv 1 \pmod{q}$ if and only if $m \equiv 0 \pmod{q-1}$.
2. For any integers, i, j, $\alpha^i \equiv \alpha^j \pmod{q}$ if and only if $i \equiv j \pmod{q-1}$.

As with ElGamal encryption, the global elements of **ElGamal digital signature** are a prime number *q* and a, which is a primitive root of *q*. User A generates a private/public key pair as follows.

1.Generate a random integer *XA*, such that 1 6 *XA*<*q* - 1.
2.Compute *YA* = *aXA* mod *q*.
3.A's private key is *XA*; A's pubic key is {*q*, *a*, *YA*}.

To sign a message *M*, user A first computes the hash *m* = H(*M*), such that *m* is an integer in the range 0 <= *m* <= *q* - 1. A then forms a digital signature as follows.

1. Choose a random integer *K* such that 1 <= *K* <= *q* - 1 and gcd(*K*, *q* - 1) = 1. That is, *K* is relatively prime to *q* - 1.
2. Compute *S*1 = *aK*mod *q*. Note that this is the same as the computation of *C*1 for ElGamal encryption.
3. Compute*K*-1mod(*q*-1).Thatis,computetheinverseof*K*modulo*q*-1.
4. Compute *S*2 = *K*-1(*m* -*XAS*1)mod (*q* - 1).

The signature consists of the pair (*S*1, *S*2). Any user B can verify the signature as follows.
Compute *V*1 = *am* mod *q*.

*S*1 *S*2

Compute *V*2 = (*YA*) 1(*S*1) mod *q*.
The signature is valid if *V*1 = *V*2. Let us demonstrate that this is so. Assume that the equality is true. Then we have

$\alpha^m \bmod q = (Y_A)^{S_1}(S_1)^{S_2} \bmod q$	assume $V_1 = V_2$
$\alpha^m \bmod q = \alpha^{X_A S_1}\alpha^{K S_2} \bmod q$	substituting for Y_A and S_1
$\alpha^{m - X_A S_1} \bmod q = \alpha^{K S_2} \bmod q$	rearranging terms
$m - X_A S_1 \equiv K S_2 \bmod (q-1)$	property of primitive roots
$m - X_A S_1 \equiv K K^{-1}(m - X_A S_1) \bmod (q-1)$	substituting for S_2

For example, let us start with the prime field GF(19); that is, $q = 19$. It has primitive roots {2, 3, 10, 13, 14, 15}, as shown in Table 8.3. We choose $a = 10$.

Alice generates a key pair as follows:

1. Alice chooses $XA = 16$.
2. Then $YA = aXA \bmod q = a16 \bmod 19 = 4$.
3. Alice's private key is 16; Alice's pubic key is $\{q, a, YA\} = \{19, 10, 4\}$.
Suppose Alice wants to sign a message with hash value $m = 14$.
1. Alice chooses $K = 5$, which is relatively prime to $q - 1 = 18$.
2. $S1 = aK \bmod q = 105 \bmod 19 = 3$ (see Table 8.3).

3. $K^{-1} \bmod (q - 1) = 5^{-1} \bmod 18 = 11$.
4. $S_2 = K^{-1}(m - X_A S_1) \bmod (q - 1) = 11(14 - (16)(3)) \bmod 18 = -374 \bmod 18 = 4$.

Bob can verify the signature as follows.

1. $V_1 = \alpha^m \bmod q = 10^{14} \bmod 19 = 16$.
2. $V_2 = (Y_A)^{S_1}(S_1)^{S_2} \bmod q = (4^3)(3^4) \bmod 19 = 5184 \bmod 19 = 16$.

Thus, the signature is valid.

4.11 SCHNORR DIGITAL SIGNATURE SCHEME

As with the ElGamal digital signature scheme, the Schnorr signature scheme is based on discrete logarithmsThe Schnorr scheme minimizes the message-dependent amount of computation required to generate signature. The main work for signature generation does not depend on the message and can be done during the idle time of the processor. The message-dependent part of the signature generation requires multiplying a $2n$-bit integer with an n-bit integer.
The scheme is based on using a prime modulus p, with $p - 1$ having a prime factor q of appropriate size; that is, $p - 1 == (\bmod q)$. Typically, we use $p \sim\sim 21024$ and $q \sim 2160$. Thus, p is a 1024-bit number, and q is a 160-bit number, which is also the length of the SHA-1 hash value.

The first part of this scheme is the generation of a private/public key pair, which consists of the following steps.

1. Choose primes p and q, such that q is a prime factor of $p - 1$.

2. Choose an integer a, such that $aq = 1 \bmod p$. The values a, p, and q comprise a global public key that can be common to a group of users.

3. Choose a random integer s with $0 < s < q$. This is the user's private key.

4. Calculate $v = a - s \bmod p$. This is the user's public key.
A user with private key s and public key v generates a signature as follows.

1. Choose a random integer r with $0 < r < q$ and compute $x = a^r \bmod p$. This computation is a preprocessing stage independent of the message M to be signed.

2. Concatenate the message with x and hash the result to compute the value e:

$$e = \mathrm{H}(M \| x)$$

3. Compute $y = (r + se) \bmod q$. The signature consists of the pair (e, y).

Any other user can verify the signature as follows.

1. Compute $x' = a^y v^e \bmod p$.

2. Verify that $e = \mathrm{H}(M \| x')$.

To see that the verification works, observe that

$$x' \equiv a^y v^e \equiv a^y a^{-se} \equiv a^{y-se} \equiv a^r \equiv x \pmod{p}$$

Hence, $\mathrm{H}(M \| x') = \mathrm{H}(M \| x)$.

4.12 DIGITAL SIGNATURE STANDARD

The National Institute of Standards and Technology (NIST) has published Federal Information Processing Standard FIPS 186, known as the Digital Signature Standard (DSS). The DSS makes use of the Secure Hash Algorithm (SHA) described in Chapter 12 and presents a new digital signature technique, the **DigitalSignature Algorithm (DSA)**. The DSS was originally proposed in 1991 and revisedin 1993 in response to public feedback concerning the security of the scheme. There was a further minor revision in 1996. In 2000, an expanded version of the standard was issued as FIPS 186-2, subsequently updated to FIPS 186-3 in 2009. This latest version also incorporates digital signature algorithms based on RSA and on ellipticcurve cryptography. In this section, we discuss the original DSS algorithm.

The DSS Approach

The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique.

Figure 13.3 contrasts the DSS approach for generating digital signatures to that used with RSA. In the RSA approach, the message to be signed is input to a hash function that produces a secure hash code of fixed length. This hash code is then encrypted using the sender's private key to form the signature. Both the message and the signature are then transmitted. The recipient takes the message and produces a hash code. The recipient also decrypts the signature using the sender's public key. If the calculated hash code matches the decrypted signature, the signature is accepted as valid. Because only the sender knows the private key, only the sender could have produced a valid signature.

The DSS approach also makes use of a hash function. The hash code is provided as input to a signature function along with a random number k generated for this particular signature. The signature function also depends on the sender's private key (PRa) and a set of parameters known to a group of communicating principals.

We can consider this set to constitute a global public key (PUG).1 The result is a signature consisting of two components, labeled s and r.

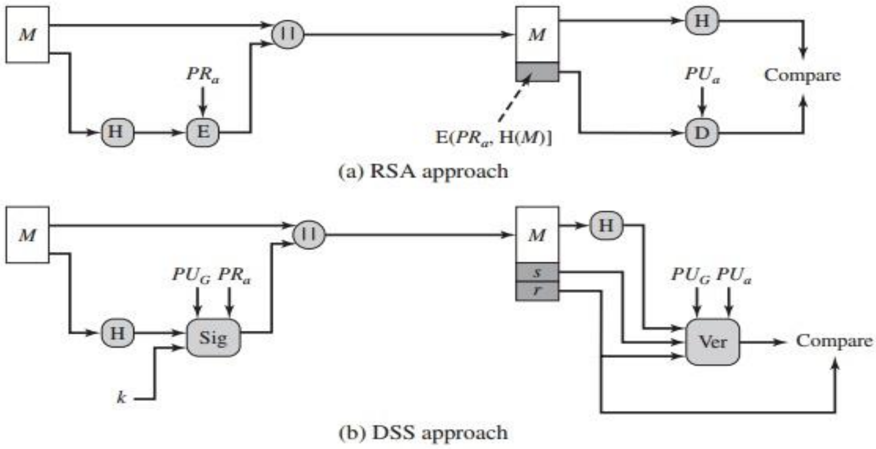


Figure 13.3 Two Approaches to Digital Signatures

At the receiving end, the hash code of the incoming message is generated. This plus the signature is input to a verification function. The verification function also depends on the global public key as well as the sender's public key (PUa), which is paired with the sender's private key. The output of the verification function is a value that is equal to the signature component r if the signature is valid. The signature function is suc h that only the sender, with knowledge of the private key, could have produced the valid signature.

The Digital Signature Algorithm

The DSA is based on the difficulty of computing discrete logarithms (see Chapter 8) and is based on schemes originally presented by ElGamal [ELGA85] and Schnorr [SCHN91].

Figure 13.4 summarizes the algorithm. There are three parameters that are public and can be common to a group of users. A 160-bit prime number q is chosen. Next, a prime number p is selected with a length between 512 and 1024 bits such that $q$divides (p - 1). Finally, g is chosen to be of the form $h(p-1)/q \bmod p$, where h is an integer between 1 and (p - 1) with the restriction that g must be greater than 1.2 Thus, the global public-key components of DSA have the same for as in the Schnorr signature scheme.

With these numbers in hand, each user selects a private key and generates a public key. The private key x must be a number from 1 to (q - 1) and should be chosen randomly or pseudorandomly. The public key is calculated from the private key as $y = gx \bmod p$. The calculation of y given x is relatively straightforward. However, given the public key y, it is believed to be computat ionally infeasible to determine x, which is the discrete logarithm of y to the base g, mod$p$ (see Chapter 8).

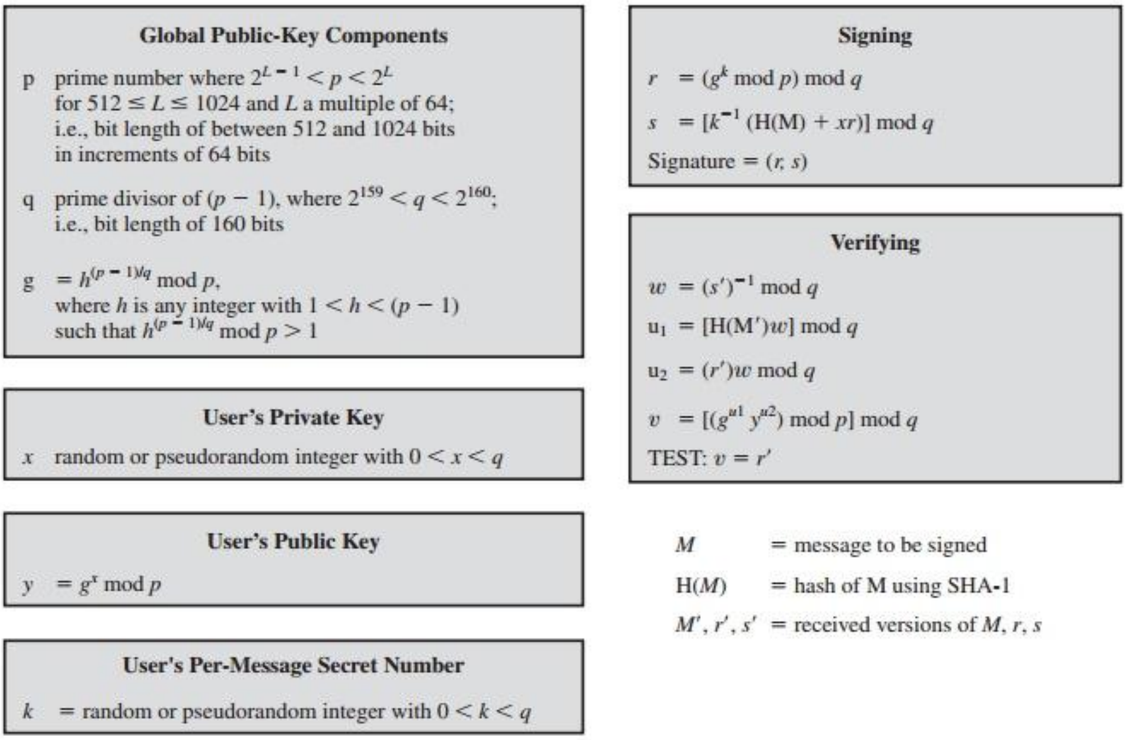


Figure 13.4 The Digital Signature Algorithm (DSA)

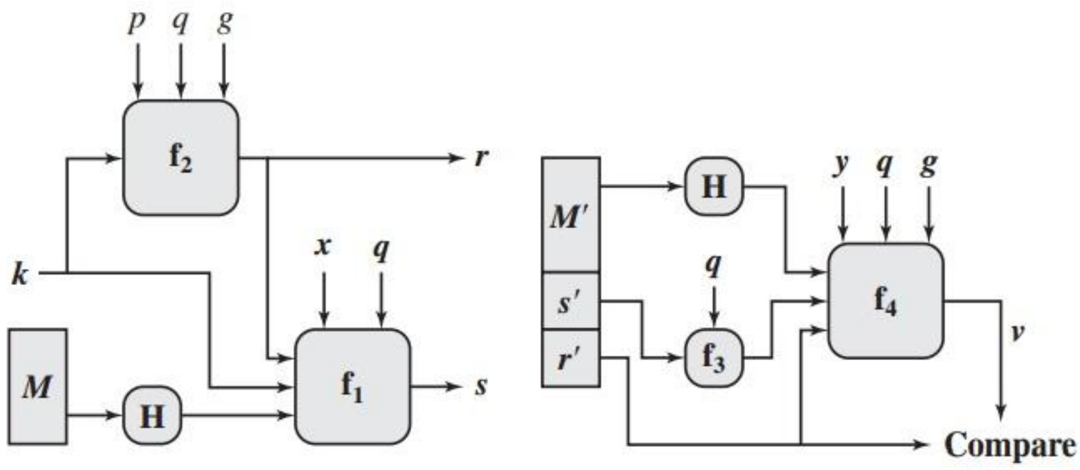
To create a signature, a user calculates two quantities, *r* and *s*, that are functions of the public key components (p, q, g), the user's private key (x), the hash code of the message H(M), and an additional integer *k* that should be generated randomly or pseudorandomly and be unique for each signing.

At the receiving end, verification is performed using the formulas shown in Figure 13.4.The receiver generates a quantity *v* that is a function of the public key components, the sender's public key, and the hash code of the incoming message. If this quantity matches the *r* component of the signature, then the signature is validated.

Figure 13.5 depicts the functions of signing and verifying.

The structure of the algorithm, as revealed in Figure 13.5, is quite interesting. Note that the test at the end is on the value *r*, which does not depend on the message at all. Instead, *r* is a function of *k* and the three global public-key components. The multiplicative inverse of *k* (mod *q*) is passed to a function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover *r* using the incoming message and signature, the public key of the user, and the global public key. It is certainly not obvious from Figure 13.4 or Figure 13.5 that such a scheme would work. A proof is provided in Appendix K. Given the difficulty of taking discrete logarithms, it is infeasible for an opponent to recover *k* from *r* or to recover *x* from *s*.

Another point worth noting is that the only computationally demanding task in signature generation is the exponential calculation *gk* mod *p*. Because this value does not depend on the message to be signed, it can be computed ahead of time.



$s = f_1(H(M), k, x, r, q) = (k^{-1}\,(H(M) + xr)) \bmod q$

$r = f_2(k, p, q, g) = (g^k \bmod p) \bmod q$

(a) Signing

$w = f_3(s', q) = (s')^{-1} \bmod q$

$v = f_4(y, q, g, H(M'), w, r')$

$= ((g^{(H(M')w) \bmod q}\, y^{r'w \bmod q}) \bmod p) \bmod q$

(b) Verifying

Figure 13.5 DSS Signing and Verifying

Indeed, a user could precalculate a number of values of *r* to be used to sign documents as needed. The only other somewhat demanding task is the determination of a multiplicative inverse, *k*- 1. Again, a number of these values can be precalculated.

UNIT V-TRANSPORT LEVEL SECURITY

5.1 TRANSPORT LEVEL SECURITY

KEY POINTS

◆ Secure Socket Layer (SSL) provides security services between TCP and applications that use TCP. The Internet standard version is called Transport Layer Service (TLS).

◆ SSL/TLS provides confidentiality using symmetric encryption and message integrity using a message authentication code.

◆ SSL/TLS includes protocol mechanisms to enable two TCP users to deter- mine the security mechanisms and services they will use.

◆ HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server.

◆ Secure Shell (SSH) provides secure remote logon and other secure client/server facilities.

Virtually all businesses, most government agencies, and many individuals now have Web sites. The num ber of individuals and companies with Internet access is expanding rapidly and all of these have graphical Web browsers. As a result, businesses are enthsiastic about setting up facilities on the Web for electronic Commerce. But the reality is that the Internet and the Web are extremely vulnerable to compromises of various sorts. As businesses wake up to this reality, the demand for secure Web service s grows. The topic of Web security is a broad one and can easily fill a book. In this chapter, we begin with a discussion of the general requirements for Web security and then focus on three standardized schemes that are becoming increasingly important as part of Web commerce and that focus on security at the transport layer: SSL/TLS, HTTPS, and SSH.

5.2 WEB SECURITY CONSIDERATIONS

The World Wide Web is fundamentally a client/server application running over the Internet and TCP/IP intranets. As such, the security tools and approaches discussed so far in this book are relevant to the issue of Web security. But, as pointed out in [GARF02], the Web presents new challenges not generally appreciated in the context of computer and network security.

The Internet is two-way. Unlike traditional publishing environments—even electronic publishing systems involving teletext, voice response, or fax-back— the Web is vulnerable to attacks on the Web servers over the Internet.

- The Web is increasingly serving as a highly visible outlet for corporate and product information and as the platform for business transactions. Reputations can be damaged and money can be lost if the Web servers are subverted.
- Although Web browsers are very easy to use, Web servers are relatively easy to configure and manage, and Web content is increasingly easy to develop, the underlying software is extraordinarily complex. This complex software may hide many potential security flaws. The short history of the Web is filled with examples of new and upgraded systems, properly installed, that are vulnerable to a variety of security attacks.
- A Web server can be exploited as a launching pad into the corporation's or agency's entire computer complex. Once the Web server is subverted, an attacker may be able to gain access to data and systems not part of the Web itself but connected to the server at the local site.
- Casual and untrained (in security matters) users are common clients for Web-based services. Such users are not necessarily aware of the security risks that exist and do not have the tools or knowledge to take effective countermeasures.

Web Security Threats

Table 16.1 provides a summary of the types of security threats faced when using the Web. One way to group these threats is in terms of passive and active attacks. Passive attacks include eavesdropping on network traffic between browser and server and gaining access to information on a Web site that is supposed to be restricted. Active attacks include impersonating another user, altering messages in transit between client and server, and altering information on a Web site.

Another way to classify Web security threats is in terms of the location of the threat: Web server, Web browser, and network traffic between browser and server. Issues of server and browser security fall into the category of computer system security; Part Four of this book addresses the issue of system security in general but is also applicable to Web system security. Issues of traffic security fall into the category of network security and are addressed in this chapter.

-

Web Traffic Security Approaches

A number of approaches to providing Web security are possible. The various approaches that have been considered are similar in the services they provide and, to some extent, in the mechanisms that they use, but they differ with respect to their scope of applicability and their relative location within the TCP/IP protocol stack.

Figure 16.1 illustrates this difference. One way to provide Web security is to use IP security (IPsec) (Figure 16.1a). The advantage of using IPsec is that it is trans- parent to end users and applications and provides a general-purpose solution. Furthermore, IPsec includes a filtering capability so that only selected traffic need incur the overhead of IPsec processing.

Table 16.1 A Comparison of Threats on the Web

	Threats	Consequences	Countermeasures
Integrity	• Modification of user data • Trojan horse browser • Modification of memory • Modification of message traffic in transit	• Loss of information • Compromise of machine • Vulnerabilty to all other threats	Cryptographic checksums
Confidentiality	• Eavesdropping on the net • Theft of info from server • Theft of data from client • Info about network configuration • Info about which client talks to server	• Loss of information • Loss of privacy	Encryption, Web proxies
Denial of Service	• Killing of user threads • Flooding machine with bogus requests • Filling up disk or memory • Isolating machine by DNS attacks	• Disruptive • Annoying • Prevent user from getting work done	Difficult to prevent
Authentication	• Impersonation of legitimate users • Data forgery	• Misrepresentation of user • Belief that false information is valid	Cryptographic techniques

5.3 SECURE SOCKET LAYER AND TRANSPORT LAYER SECURITY

Netscape originated SSL. Version 3 of the protocol was designed with public review and input from industry and was published as an Internet draft document. Subsequently, when a consensus was reached to submit the protocol for Internet standardization, the TLS working group was formed within IETF to develop a common standard. This first published version of TLS can be viewed as essentially an SSLv3.1 and is very close to and backward compatible with SSLv3.

SSL Architecture

SSL is designed to make use of TCP to provide a reliable end-to-end secure service.

SSL is not a singleprotocol but rather two layers of protocols, as illustrated in Figure 16.2.

The SSL Record Protocol provides basic security services to various higher- layer protocols. In particular, the Hypertext Transfer Protocol (HTTP), which provides the transfer service for Web client/server interaction, can operate on top of SSL. Three higher-layer protocols are defined as part of SSL: the Handshake Protocol, The Change Cipher Spec Proto col, and the Alert Protocol. These SSL specific protocols are used in the management of SSL exchanges and are examined later in this section.

Two important SSL concepts are the SSL session and the SSL connection, which are defined in the specification as follows.

- **Connection:** A connection is a transport (in the OSI layering model definition) that provides a suitable type of service. For SSL, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.
- **Session:** An SSL session is an association between a client and a server. Sessions are created by the Handshake Protocol. Sessions define a set of cryptographic

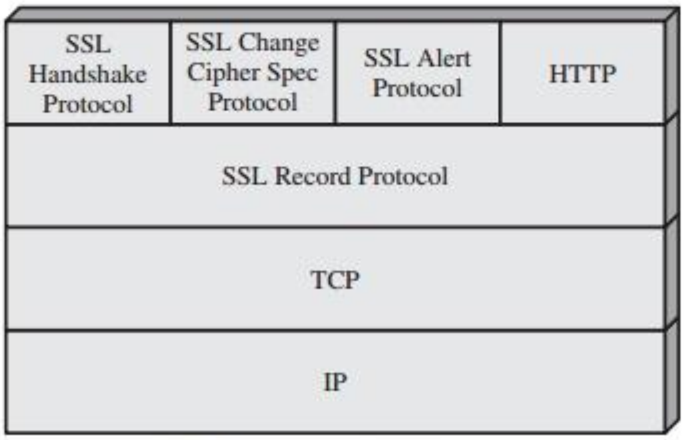


Figure 16.2 SSL Protocol Stack

security parameters which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

Between any pair of parties (applications such as HTTP on client and server), there may be multiple secure connections. In theory, there may also be multiple simultaneous sessions between parties, but this feature is not used in practice.

There are a number of states associated with each session. Once a session is established, there is a current operating state for both read and write (i.e., receive and send). In addition, during the Handshake Protocol, pending read and write states are created. Upon successful conclusion of the Handshake Protocol, the pending states become the current states.

A session state is defined by the following parameters.

- **Session identifier:** An arbitrary byte sequence chosen by the server to identify an active or resumable session state.
- **Peer certificate:** An X509.v3 certificate of the peer. This element of the state may be null.
- **Compression method:** The algorithm used to compress data prior to encryption.
- **Cipher spec:** Specifies the bulk data encryption algorithm (such as null, AES, etc.) and a has h algorithm (such as MD5 or SHA-1) used for MAC calculation. It also defines cryptographic attributes such as the hash_size.
- **Master secret:** 48-byte secret shared between the client and server.
- **Is resumable:** A flag indicating whether the session can be used to initiate new connections.

A connection state is defined by the following parameters.

- **Server and client random:** Byte sequences that are chosen by the server and client for each connection.
- **Server write MAC secret:** The secret key used in MAC operations on data sent by the server.
- **Client write MAC secret:** The secret key used in MAC operations on data sent by the client.
- **Server write key:** The secret encryption key for data encrypted by the server and decrypted by the client.
- **Client write key:** The symmetric encryption key for data encrypted by the client and decrypted by the server.

Initialization vectors: When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL Handshake Protocol. Thereafter, the final ciphertext block from each record is preserved for use as the IV with the following record.

Sequence numbers: Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec message, the appropriate sequence number is set to zero.

Sequence numbers may not exceed 264 – 1.

SSL Record Protocol

The SSL Record Protocol provides two services for SSL connections

- **Confidentiality:** The Handshake Protocol defines a shared secret key that is used for conventional encryption of SSL payloads.
- **Message Integrity:** The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC).

Figure 16.3 indicates the overall operation of the SSL Record Protocol. The Record Protocol takes an application message to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, adds a header, and transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled before being delivered to higher-level users.

The first step is **fragmentation**. Each upper-layer message is fragmented into blocks of 214 bytes (16384bytes) or less. Next, **compression** is optionallyapplied. Compression must be lossless and may not increase the content length by more than 1024 bytes.
1In SSLv3 (as well as the current version of TLS), no compression algorithm is specified, so the default compression algorithm is null.

The next step in processing is to compute a **message authentication code** over the compressed data. For this purpose, a shared secret key is used. The calculation is defined as

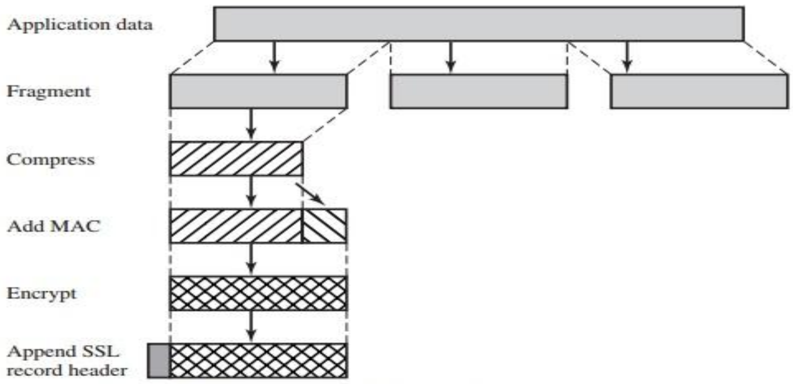


Figure 16.3 SSL Record Protocol Operation

```
hash(MAC_write_secret || pad_2||
    hash(MAC_write_secret || pad_1||seq_num ||
    SSLCompressed.type || SSLCompressed.length ||
    SSLCompressed.fragment))
```

where

||	=	concatenation
MAC_write_secret	=	shared secret key
hash	=	cryptographic hash algorithm; either MD5 or SHA-1
pad_1	=	the byte 0x36 (0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1
pad_2	=	the byte 0x5C (0101 1100) repeated 48 times for MD5 and 40 times for SHA-1
seq_num	=	the sequence number for this message
SSLCompressed.type	=	the higher-level protocol used to process this fragment
SSLCompressed.length	=	the length of the compressed fragment
SSLCompressed.fragment	=	the compressed fragment (if compression is not used, this is the plaintext fragment)

Note that this is very similar to the HMAC algorithm defined in Chapter 12. The difference is that the tw o pads are concatenated in SSLv3 and are XORed in HMAC. The SSLv3 MAC algorithm is based on the original Internet draft for HMAC, which used concatenation. The final version of HMAC (defined in RFC 2104) uses the XOR. Next, the compressed message plus the MAC are **encrypted** usi ng symmetric encryption. Encryption may not increase the content length by more than 1024 bytes, so that the total length may not exceed 214 + 2048. The following encryption algorithms are permitted:

Block Cipher		Stream Cipher	
Algorithm	**Key Size**	**Algorithm**	**Key Size**
AES	128, 256	RC4-40	40
IDEA	128	RC4-128	128
RC2-40	40		
DES-40	40		
DES	56		
3DES	168		
Fortezza	80		

Fortezza can be used in a smart card encryption scheme.

For stream encryption, the compressed message plus the MAC are encrypted. Note that the MAC is computed before encryption takes place and that the MAC is then encrypted along with the plaintext or compressed plaintext.

For block encryption, padding may be added after the MAC prior to encryption.

The padding is in the form of a number of padding bytes followed by a one-byte indication of the

length of the padding. The total amount of padding is the smallestamount such that the total size of the data to be encrypted (plaintext plus MAC plus padding) is a multiple of the cipher's block length. An example is a plaintext of 58 bytes, with a MAC of 20 bytes (using SHA-1), that is encrypted using a block length of 8 bytes (e.g., DES). With the padding-length byte, this yields a total of 79 bytes. To make the total an integer multiple of 8, one byte of padding is added.

The final step of SSL Record Protocol processing is to prepare a header consisting of the following fields:

- **Content Type (8 bits):** The higher-layer protocol used to process the enclosed fragment.
- **Major Version (8 bits):** Indicates major version of SSL in use. For SSLv3, the value is 3.
- **Minor Version (8 bits):** Indicates minor version in use. For SSLv3, the value is 0.
- **Compressed Length (16 bits):** The length in bytes of the plaintext fragment (or compressed fragment if compression is used). The maximum value is 214 + 2048.

Change Cipher Spec Protocol

The Change Cipher Spec Protocol is one of the three SSL-specific protocols that use the SSL Record Protocol, and it is the simplest. This protocol consists of a single message (Figure 16.5a), which consists of a single byte with the value 1. The sole purpose of this message is to cause the pending state to be copied into the current state, which updates the cipher suite to be used on this connection.

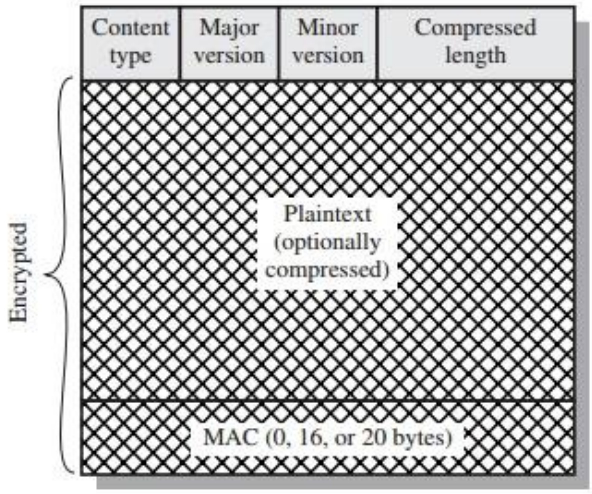


Figure 16.4 SSL Record Format

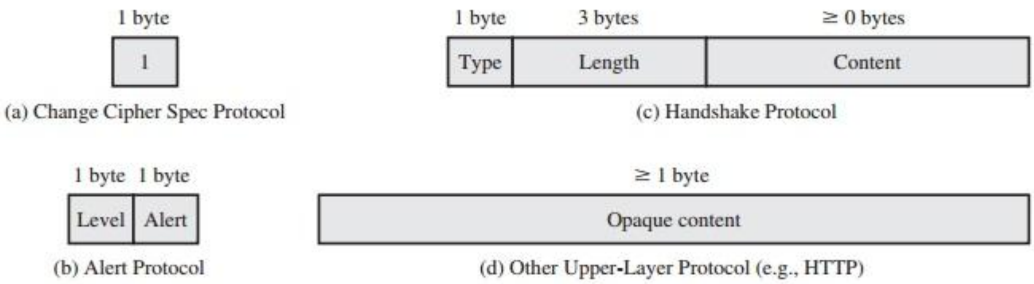


Figure 16.5 SSL Record Protocl Payload

Alert Protocol

The Alert Protocol is used to convey SSL-related alerts to the peer entity. As with other applications that use SSL, alert messages are compressed and encrypted, as specified by the current state.

Each message in this protocol consists of two bytes (Figure 16.5b). The first byte takes the value warning (1) or fatal (2) to convey the severity of the message. If the level is fatal, SSL immediately terminates the connection. Other connections on the same session may continue, but no new connections on this session may be established. The second byte contains a code that indicates the specific alert. First, we list those alerts that are always fatal (definitions from the SSL specification):

- **unexpected_message:** An inappropriate message was received.

- **bad_record_mac:** An incorrect MAC was received.
- **decompression_failure:** The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
- **handshake_failure:** Sender was unable to negotiate an acceptable set of security parameters given the options available.
- **illegal_parameter:** A field in a handshake message was out of range or inconsistent with other fields.

The remaining alerts are the following.

- **close_notify:** Notifies the recipient that the sender will not send any more messages on thisconnection. Each party is required to send a **close_notify** alert before closing the write side of a connection.

- **no_certificate:** May be sent in response to a certificate request if no appropriate certificate is available.
- **bad_certificate:** A received certificate was corrupt (e.g., contained a signature that did not verify).
- **unsupported_certificate:** The type of the received certificate is not supported.
- **certificate_revoked:** A certificate has been revoked by its signer.
- **certificate_expired:** A certificate has expired.
- **certificate_unknown:** Some other unspecified issue arose in processing the certificate, rendering it unacceptable.

Handshake Protocol

The most complex part of SSL is the Handshake Protocol. This protocol allows the server and client to authenticate each other and to negotiate an encryption and MAC algorithm and cryptographic keys to be used to protect data sent in an SSL record. The Handshake Protocol is used before any application data is transmitted.

The Handshake Protocol consists of a series of messages exchanged by client and server. All of these have the format shown in Figure 16.5c. Each message has three fields:

(f) **Type (1 byte):** Indicates one of 10 messages. Table 16.2 lists the defined message types.

(g) **Length (3 bytes):** The length of the message in bytes.

(h) **Content (Ú 0 bytes):** The parameters associated with this message; these are listed in Table 16.2.

Figure 16.6 shows the initial exchange needed to establish a logical connection between client and server. The exchange can be viewed as having four phases.

PHASE 1. ESTABLISH SECURITY CAPABILITIES This phase is used to initiate a logical connection and to establish the security capabilities that will be associated with it. The exchange is initiated by the client, which sends a **client_hellomessage** with the following parameters:

(i) **Version:** The highest SSL version understood by the client.

(j) **Random:** A client-generated random structure consisting of a 32-bit timestamp and 28 bytes generated by a secure random number generator. These values serve as nonces and are used during key exchange to prevent replay attacks.

Table 16.2 SSL Handshake Protocol Message Types

Message Type	Parameters
`hello_request`	null
`client_hello`	version, random, session id, cipher suite, compression method
`server_hello`	version, random, session id, cipher suite, compression method
`certificate`	chain of X.509v3 certificates
`server_key_exchange`	parameters, signature
`certificate_request`	type, authorities
`server_done`	null
`certificate_verify`	signature
`client_key_exchange`	parameters, signature
`finished`	hash value

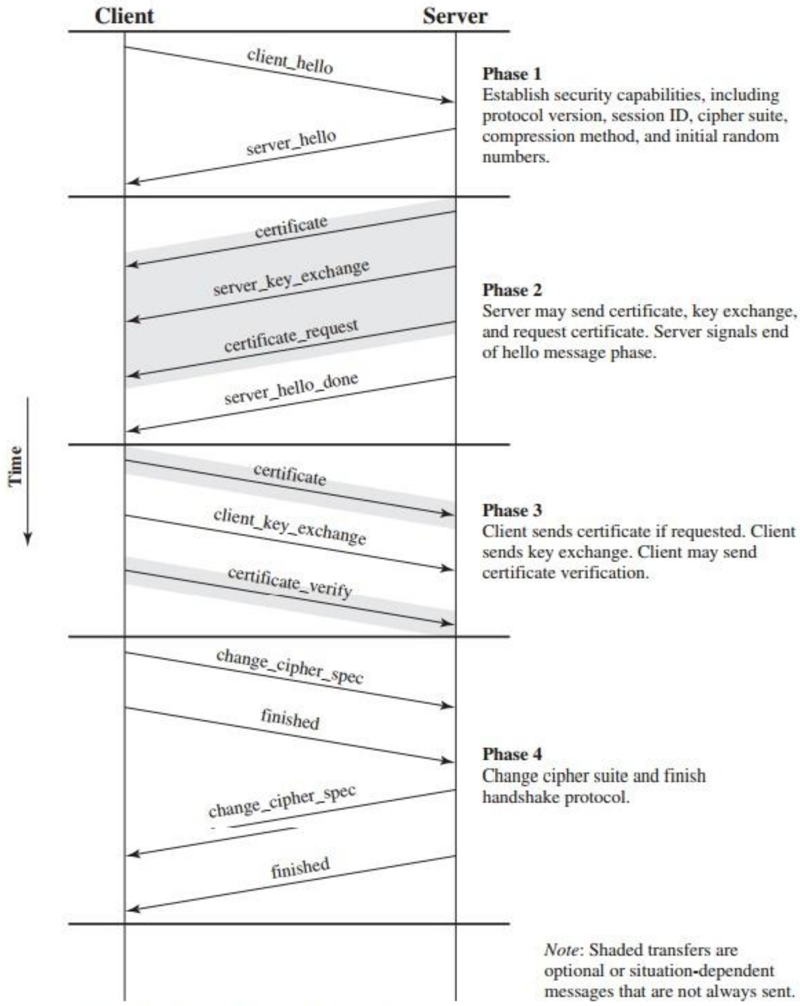


Figure 16.6 Handshake Protocol Action

Session ID: A variable-length session identifier. A nonzero value indicates that the client wishes to update the parameters of an existing connection or to create a new connection on this session. A zero value indicates that the client wishes to establish a new connection on a new session.

CipherSuite: This is a list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each element of the list (each cipher suite) defines both a key exchange algorithm and a CipherSpec; these are discussed subsequently.

Compression Method: This is a list of the compression methods the client supports.
After sending the client_hello message, the client waits for the server_hello message, which contains the same parameters as the client_hellomessage. For the server_hellomessage, the following conventions apply.
The Version field contains the lower of the versions suggested by the client and the highest supported by the server. The Random field is generated by the server and is independent of the client's Random field. If the SessionID field ofthe client was non zero, the same value is used by the server; otherwise the server's Session ID field contains the value for a new session. The Cipher Suite field contains the single cipher suite selected by the server from

those proposed by the client. The Compression field contains the compression method selected by the server from those proposed by the client.

The first element of the CipherSuite parameter is the key exchange method (i.e., the means by which the cryptographic keys for conventional encryption and MAC are exchanged). The following key exchange methods are supported.

RSA: The secret key is encrypted with the receiver's RSA public key. A public-key certificate for the receiver's key must be made available.

Fixed Diffie-Hellman: This is a Diffie-Hellman key exchange in which the server's certificate contains the Diffie-Hellman public parameters signed by the certificate authority (CA). That is, the public-key certificate contains the Diffie-Hellman public-key parameters. The client provides its Diffie-Hellman public-key parameters either in a certificate, if client authentication is required, or in a key exchange message. This method results in a fixed secret key between two peers based on the Diffie-Hellman calculation using the fixed public keys.

Ephemeral Diffie-Hellman: This technique is used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie-Hellman public keys are exchanged, signed using the sender's private RSA or DSS key. The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys. This would appear to be the most secure of the three Diffie-Hellman options, because it results in a temporary, authenticated key.

Anonymous Diffie-Hellman: The base Diffie-Hellman algorithm is used with no authentication. That is, each side sends its public Diffie-Hellman parameters to the other with no authentication. This approach is vulnerable to man-in-the- middle attacks, in which the attacker conducts anonymous Diffie-Hellman with both parties.

Fortezza: The technique defined for the Fortezza scheme.

Following the definition of a key exchange method is the CipherSpec, which includes the following fields.

CipherAlgorithm: Any of the algorithms mentioned earlier: RC4, RC2, DES, 3DES, DES40, IDEA, or Fortezza

MACAlgorithm: MD5 or SHA-1

CipherType: Stream or Block

IsExportable: True or False

HashSize: 0, 16 (for MD5), or 20 (for SHA-1) bytes

Key Material: A sequence of bytes that contain data used in generating the write keys

IV Size: The size of the Initialization Value for Cipher Block Chaining (CBC) encryption

PHASE 2. SERVER AUTHENTICATION AND KEY EXCHANGE

The server begins this phase by sending its certificate if it needs to be authenticated; the message contains one or a chain of X.509 certificates. The **certificate message** is required for any agreed-on key exchange method except anonymous Diffie-Hellman. Note that if fixed Diffie- Hellman is used, this certificate message functions as the server's key exchange message because it contains the server's public Diffie-Hellman parameters.

Next, a **server_key_exchangemessage** may be sent if it is required. It is not required in two instan ces: (1) The server has sent a certificate with fixed Diffie-Hellman parameters or (2) a RSA key exchange is to be used. The server_key_exchangemessage is needed for the following:

Anonymous Diffie-Hellman: The message content consists of the two global Diffie-Hellman values (a prime number and a primitive root of that number) plus the server's public Diffie-Hellman key (see Figure 10.1).

Ephemeral Diffie-Hellman: The message content includes the three Diffie-Hellman parameters provided for anonymous Diffie-Hellman plus a signature of those parameters.

RSA key exchange (in which the server is using RSA but has a signature-onlyRSA key):

Accordingly, the client cannot simply send a secret key encrypted with the server's public key. Instead, the server must create a temporary RSA public/private key pair and use the server_key_exchangemessage to send the public key.

The message content includes the two parameters of the temporary RSA public key plus a signature of those parameters.

Fortezza

Some further details about the signatures are warranted. As usual, a signature is created by taking the hash of a message and encrypting it with the sender's private key. In this case, t he hash is defined as

hash(ClientHello.random | ServerHello.random | ServerParams)

So the hash covers not only the Diffie-Hellman or RSA parameters but also the two nonces from the initial hello messages. This ensures against replay attacks and misrepresentation. In the case of a DSS signature, the hash is performed using the SHA-1 algorithm. In the case of an RSA signature, both an MD5 and an SHA-1 hash are calculated, and the concatenation of the two hashes (36 bytes) is encrypted with the server's private key.

Next, a nonanonymous server (server not using anonymous Diffie-Hellman) can request a certificate from the client. The **certificate_requestmessage** includes two parameters:

certificate_typeand certificate_authorities. The certificate type indicates the public-key algorithm and its use:

- RSA, signature only
- DSS, signature only

- RSA for fixed Diffie-Hellman; in this case the signature is used only for authentication, by sending a certificate signed with RSA
- DSS for fixed Diffie-Hellman; again, used only for authentication
- RSA for ephemeral Diffie-Hellman
- DSS for ephemeral Diffie-Hellman
- Fortezza

The second parameter in the certificate_requestmessage is a list of the distinguished names of acceptable certificate authorities.

The final message in phase 2, and one that is always required, is the server_done message, which is sent by the server to indicate the end of the server hello and associated messages. After sending this message, the server will wait for a client response. This message has no parameters.

PHASE 3. CLIENT AUTHENTICATION AND KEY EXCHANGE Upon receipt of the server_donemessage, the client should verify that the server provided a valid certificate (if required) and check that the server_helloparameters are acceptable. If all is satisfactory , the client sends one or more messages back to the server.

If the server has requested a certificate, the client begins this phase by sending a **certificate message**. If no suitable certificate is available, the client sends a no_certificatealert instead.

Next is the **client_key_exchange message**, which must be sent in this phase. The content of the message depends on the type of key exchange, as follows.

- **RSA:** The client generates a 48-byte *pre-master secret* and encrypts with the public key from the server'scertificate or temporary RSA key from a server_key_exchangemessage.
- **Ephemeral or Anonymous Diffie-Hellman:** The client's public Diffie-Hellman parameters are sent.
- **Fixed Diffie-Hellman:** The client's public Diffie-Hellman parameters were sent in a certificate message, so the content of this message is null.
- **Fortezza:** The client's Fortezza parameters are sent.

Finally, in this phase, the client may send a **certificate_verifymessage** to provide explicit verification of a client certificate. This message is only sent fol- lowing any client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie-Hellman parameters). This message signs a hash code based on the preceding messages, defined as

CertificateVerify.signature.md5_hash=MD5(master_secret | pad_2 | MD5(handshake_messages | master_secret | pad_1));

CertificateVerify.signature.sha_hash=SHA(master_secret | pad_2 | SHA(handshake_messages | mas ter_secret | pad_1));

where pad_1 and pad_2 are the values defined earlier for the MAC,**handshake_messages** refers to all Handshake Protocol messages sent or received starting at client_hello but not including this message, and master_secretis the calculated secret whose construction is explained later in this section. If the user's private key is DSS, then it is used to encrypt the SHA-1 hash. If the user's private key is RSA, it is used to encrypt the concatenation of the MD5 and SHA-1 hashes. In either case, the purpose is to verify the client's owner- ship of the private key for the client certificate. Even if someone is misusing the client's certificate, he or she would be unable to send this message.

PHASE 4. FINISH This phase completes the setting up of a secure connection. The client sends a change_cipher_spec message and copies the pending CipherSpec into the current CipherSpec. Note that this message is not considered part of the Handshake Protocol but is sent using the Change Cipher Spec Protocol. The client then immediately sends the **finished message** under the new algorithms, keys, and secrets. The finished message verifies that the key exchange and authentication processes were successful. The content of the finished message is the concatenation of two hash values:

MD5(master_secret | pad2 | MD5(handshake_messages | Sender | master_secret | pad1))

SHA(master_secret | pad2 | SHA(handshake_messages | Sender | master_secret | pad1))

where Sender is a code that identifies that the sender is the client and handshake_messagesis all of the data from all handshake messages up to butnot including this message. In response to these two messages, the server sends its own change_cipher_specmessage, transfers the pending to the current CipherSpec, and sends its finished message. At this point, the handshake is complete and the client and server may begin to exchange application-layer data.

Cryptographic Computations

Two further items are of interest: (1) the creation of a shared master secret by means of the key exchange and (2) the generation of cryptographic parameters from the master secret.

MASTER SECRET CREATION The shared master secret is a one-time 48-byte value (384 bits) generated for this session by means of secure key exchange. The creation is in two stages. First, a pre_master_secret is exchanged.
Second, the master_secret is calculated by both parties. For pre_master_secret exchange, there are two possibilities.

RSA: A 48-byte pre_master_secretis generated by the client, encrypted with the server's public RSA key, and sent to the server. The serverdecrypts the ciphertext using its private key to recover the pre_master_secret.

Diffie-Hellman: Both client and server generate a Diffie-Hellman public key. After these are exchanged, each side performs the Diffie-Hellman calculation to create the shared pre_master_secret.

Both sides now compute the master_secretas master_secret=MD5(pre_master_secret | SHA('A' |pre_master_secret | ClientHello.random | Server Hello.random) |MD5(pre_master_secret | SHA('BB' | pre_master_secret | ClientHello.random | ServerHello.random)) |MD5(pre_master_secret | SHA('CCC' | pre_master_secret | ClientHello.random | ServerHello.random))

where ClientHello.random and ServerHello.randomare the two nonce values exchanged in the initial hello messages.

GENERATION OF CRYPTOGRAPHIC PARAMETERS CipherSpecs require a client write

MAC secret, a server write MAC secret, a client write key, a server write key, a client write IV, and a server write IV, which are generated from the master secret in that order.These parameters are

generated from the master secret by hashing the master secret into a sequence of secure bytes

of sufficient length for all needed parameters.The generation of the key material from themaster

secret uses the same format for generation of the master secret from the pre-master secret as

key_block=MD5(master_secret | SHA('A' | master_secret | ServerHello.random | ClientHello.random)|MD5(master_secret | SHA('BB' | master_secret | ServerHello.random | ClientHello.random)|

MD5(master_secret | SHA('CCC' | master_secret | ServerHello.random | ClientHello.random)| . . .

until enough output has been generated. The result of this algorithmic structure is a pseudorandom function. We can view the master_secretas the pseudorandom seed value to the function. The client and server random numbers can be viewed as salt values to complicate cryptanalysis (see Chapter 20 for a discussion of the use of salt values).

5.4 TRANSPORT LAYER SECURITY

TLS is an IETF standardization initiative whose goal is to produce an Internet standard version of SSL. TLS is defined as a Proposed Internet Standard in RFC 5246. RFC 5246 is very similar to SSLv3. In this section, we highlight the differences.

Version Number

The TLS Record Format is the same as that of the SSL Record Format (Figure 16.4), and the fields in the header have the same meanings. The one difference is in version values. For the current version of TLS, the major version is 3 and the minor version is 3.

Message Authentication Code

There are two differences between the SSLv3 and TLS MAC schemes: the actual algorithm and the scope of the MAC calculation. TLS makes use of the HMAC algorithm defined in RFC 2104. Recall from Chapter 12 that HMAC is defined as

$$\text{HMAC}_K(M) = \text{H}[(K^+ \oplus \text{opad}) \| \text{H}[(K^+ \oplus \text{ipad}) \| M]]$$

where

H = embedded hash function (for TLS, either MD5 or SHA-1)

M = message input to HMAC

K^+ = secret key padded with zeros on the left so that the result is equal to the block length of the hash code (for MD5 and SHA-1, block length = 512 bits)

ipad = 00110110 (36 in hexadecimal) repeated 64 times (512 bits)

opad = 01011100 (5C in hexadecimal) repeated 64 times (512 bits)

SSLv3 uses the same algorithm, except that the padding bytes are concatenated with the secret key rather than being XORed with the secret key padded to the block length. The level of security should be about the same in both cases.

For TLS, the MAC calculation encompasses the fields indicated in the following expression:

MAC(MAC_write_secret,seq_num | TLSCompressed.type | TLSCompressed.version | TLSCompressed.length | TLSCompressed.fragment)

The MAC calculation covers all of the fields covered by the SSLv3 calculation, plus the field TLSCompressed.version, which is the version of the protocol being employed.

Pseudorandom Function

TLS makes use of a pseudorandom function referred to as PRF to expand secrets into blocks of data for purposes of key generation or validation. The objective is to make use of a relatively small shared secret value but to generate longer blocks of data in a way that is secure from the kinds of attacks made on hash functions and MACs. The PRF is based on the data expansion function (Figure 16.7) given as

```
P_hash(secret, seed) = HMAC_hash(secret,A(1)  || seed) ||
                       HMAC_hash(secret, A(2) || seed) ||
                       HMAC_hash(secret, A(3) || seed) || .
```

where A() is defined as

A(0) = seed

A(i) = HMAC_hash(secret, A($i-1$))

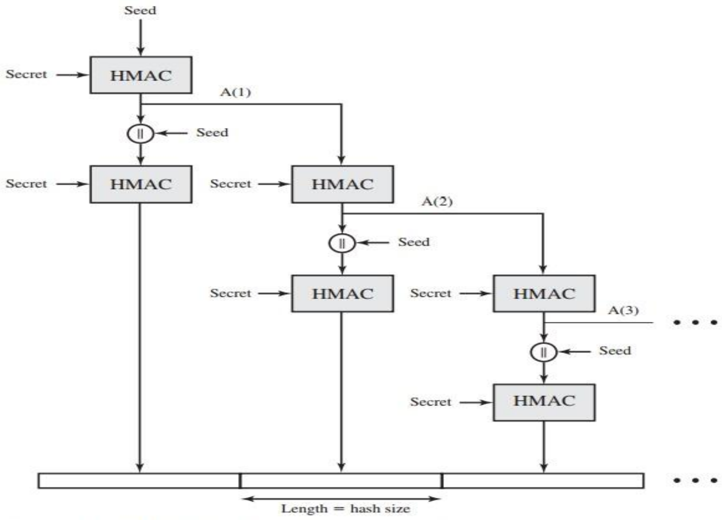


Figure 16.7 TLS Function P_hash(secret, seed)

The data expansion function makes use of the HMAC algorithm with either MD5or SHA-1 as the underlying hash function. As can be seen, P_hash can be iterated as many times as necessary to produce the required quantity of data. For example, if P_SHA-1was used to generate 64 bytes of data, it would have to be iterated four times, producing 80 bytes of data of which the last 16 would bedis- carded. In this case, P_MD5would also have to be iterated four times, producing exactly 64 bytes of data.Notethat each iteration involves two executions of HMAC—each of which in turn involves two executions of the underlying hash algorithm.
To make PRF as secure as possible, it uses two hash algorithms in a way that should guarantee its security if either algorithm remains secure. PRF is defined as

PRF(secret,label,seed)=P_hash(S1,label | seed)

PRF takes as input a secret value, an identifying label, and a seed value and produces an output of arbitrary length.

Alert Codes

TLS supports all of the alert codes defined in SSLv3 with the exception of no_certificate.

A number of additional codes are defined in TLS; of these, the following are always fatal.

record_overflow: A TLS record was received with a payload (ciphertext) whose length exceeds 214+2048 bytes, or the ciphertext decrypted to a length of greater than 214+1024 bytes.

unknown_ca: A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA.

access_denied: A valid certificate was received, but when access control was applied, the sender decided not to proceed with the negotiation.

decode_error: A message could not be decoded, because either a field was out of its specified range or the length of the message was incorrect.

protocol_version: The protocol version the client attempted to negotiate is recognized but not supported.

insufficient_security: Returnedinstead of handshake_failure when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client.

unsupported_extension: Sent by clients that receive an extended server hello containing an extension not in the corresponding client hello.

internal_error: An internal error unrelated to the peer or the correct- ness of the protocol makes it impossible to continue.

decrypt_error:A handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a fin- ished message.

The remaining alerts include the following.

user_canceled: This handshake is being canceled for some reason unre- lated to a protocol failure.

no_renegotiation: Sent by a client in response to a hello request or by the server in response to a cli ent hello after initial handshaking. Either of these messages would normally result in renegotiation, but this alert indicates that the sender is not able to renegotiate. This message is always a warning.

Cipher Suites

There are several small differences between the cipher suites available under SSLv3 and under TLS:

Key Exchange: TLS supports all of the key exchange techniques of SSLv3 with the exception of Fortezza.

Symmetric Encryption Algorithms: TLS includes all of the symmetric encryp- tion algorithms found in SSLv3, with the exception of Fortezza.

Client Certificate Types

TLS defines the following certificate types to be requested in a

certificate_request message: rsa_sign, dss_sign, rsa_fixed_dh, and dss_fixed_dh. These are all defined in SSLv3. In addition, SSLv3 includes rsa_ephemeral_dh, dss_ephemeral_dh, and fortezza_kea. Ephemeral Diffie-Hellman involves signing the Diffie-Hellman parameters with either RSA or DSS. For TLS, the rsa_sign and dss_sign types are used for that function; a separate signing type is not needed to sign Diffie-Hellman parameters. TLS does not include the Fortezza scheme.

certificate_verifyand Finished Messages

In the TLS certificate_verifymessage, the MD5 and SHA-1 hashes are calculated only over handshake_messages. Recall that for SSLv3, the hash calculation also included the master secret a nd pads. These extra fields were felt to add no additional security.

As with the finished message in SSLv3, the finished message in TLS is a hash based on the shared master_secret,the previous handshake messages, and a label that identifies client or server.

The calculation is somewhat different. For TLS, we have

PRF(master_secret,finished_label,MD5(handshake_messages)| SHA-1(handshake_messages))

where finished_labelis the string "client finished" for the client and "server finished" for the server.

Cryptographic Computations

The pre_master_secretfor TLS is calculated in the same way as in SSLv3. As in SSLv3, the master_secret in TLS is calculated as a hash function of the pre_master_secret and the two hello random numbers. The form of the TLS calculation is different from that of SSLv3 and is defined as

master_secret= PRF(pre_master_secret,"master secret", ClientHello.random||ServerHello.random)

The algorithm is performed until 48 bytes of pseudorandom output are produced. The calculation of the key block material (MAC secret keys, session encryption keys, and IVs) is defined as

key_block = PRF(master_secret, "key expansion", SecurityParameters.server_random|| SecurityParameters.client_random)

until enough output has been generated. As with SSLv3, the key_blockis a func- tion of the master_secret and the client and server random numbers, but for TLS, the actual algorithm is different.

Padding

In SSL, the padding added prior to encryption of user data is the minimum amount required so that the total size of the data to be encrypted is a multiple of the cipher's block length. In TLS, the padding can be any amount that results in a total that is a multiple of the cipher's block length, up to a maximum of 255 bytes. For example, if the plaintext (or compressed text if compression is used) plus MAC plus padding.length byte is 79 bytes long, then the padding length (in bytes) can be 1, 9, 17, and so on, up to 249.
A variable padding length may be used to frustrate attacks based on an analysis of the lengths of exchanged messages.

5.5 ELECTRONIC MAIL SECURITY

- PGP is an open-source, freely available software package for e-mail secu- rity. It provides authentication through the use of digital signature, confidentiality through the use of symmetric block encryption, compression using the ZIP algorithm, and e-mail compatibility using the radix-64 encoding scheme.

- PGP incorporates tools for developing a public-key trust model and public-key certificate management.

- S/MIME is an Internet standard approach to e-mail security that incorporates the same functionality as PGP.

◆ DKIM is a specification used by e-mail providers for cryptographically signing e-mail messages on behalf of the source domain.

In virtually all distributed environments, electronic mail is the most heavily used network-based application. Users expect to be able to, and do, send e-mail to others who are connected directly or indirectly to the Internet, regardless of host operating system or communications suite. With the explo- sively growing reliance on e-mail, there grows a demand for authentication and confidentiality services. Two schemes stand out as approaches that enjoy widespread use: Pretty Good Privacy (PGP) and S/MIME. Both are examined in this chapter. The chapter closes with a discussion of DomainKeys Identified Mail.

5.6 PRETTY GOOD PRIVACY

PGP is a remarkable phenomenon. Largely the effort of a single person, Phil Zimmermann, PGP provides a confidentiality and authentication service that can be used for electronic mail and file storage applications. In essence, Zimmermann has done the following:

1. Selected the best available cryptographic algorithms as building blocks.

2. Integrated these algorithms into a general-purpose application that is inde- pendent of operating system and processor and that is based on a small set of easy-to- use commands.

3. Made the package and its documentation, including the source code, freely available via the Internet, bulletin boards, and commercial networks such as AOL (America On Line).

4. Entered into an agreement with a company (Viacrypt, now Network Associates) to provide a fully compatible, low-cost commercial version of PGP.

PGP has grown explosively and is now widely used. A number of reasons can be cited for this growth.

1. It is available free worldwide in versions that run on a variety of platforms, including Windows, UNIX, Macintosh, and many more. In addition, the commer- cial version satisfies users who want a product that comes with vendor support.

2. It is based on algorithms that have survived extensive public review and are con- sidered extremely secure. Specifically, the package includes RSA, DSS, and Diffie-Hellman for public-key encryption; CAST-128, IDEA, and 3DES for sym- metric encryption; and SHA-1 for hash coding.

3. It has a wide range of applicability, from corporations that wish to select and enforce a standardized scheme for encrypting files and messages to individuals who wish to comm unicate securely with others worldwide over the Internet and other networks.

4. It was not developed by, nor is it controlled by, any governmental or standards organization.

For those with an instinctive distrust of "the establishment," this makes PGP attractive.

5. PGP is now on an Internet standards track (RFC 3156; *MIME Security with OpenPGP*). Nevertheless, PGP still has an aura of an antiestablishment endeavor.

We begin with an overall look at the operation of PGP. Next, we examine how cryptographic keys are created and stored. Then, we address the vital issue of public-key management.

Notation

Most of the notation used in this chapter has been used before, but a few terms are new. It is perhaps best t o summarize those at the beginning. The following symbols are used.

K_s = session key used in symmetric encryption scheme
PR_a = private key of user A, used in public-key encryption scheme
PU_a = public key of user A, used in public-key encryption scheme
EP = public-key encryption
DP = public-key decryption
EC = symmetric encryption
DC = symmetric decryption

H = hash function
|| = concatenation
Z = compression using ZIP algorithm
R64 = conversion to radix 64 ASCII format[1]

The PGP documentation often uses the term *secret key* to refer to a key paired with a public key in a public-key encryption scheme. As was mentioned earlier, this practice risks confusion with a secret key used for symmetric encryption. Hence, we use the term *private key* instead.

Operational Description

The actual operation of PGP, as opposed to the management of keys, consists of four services: authentication, confidentiality, compression, and e-mail compatibility (Table 18.1). We examine each of these in turn.

AUTHENTICATION Figure 18.1a illustrates the digital signature service provided by PGP. This is the digital signature scheme discussed in Chapter 13 and illustrated in Figure 13.2. The sequence is as follows.

1. The sender creates a message.

2. SHA-1 is used to generate a 160-bit hash code of the message.

3. The hash code is encrypted with RSA using the sender's private key, and the result is prepended to the message.

4. The receiver uses RSA with the sender's public key to decrypt and recover the hash code.

5. The receiver generates a new hash code for the message and compares it with the Decrypted hash code. If the two match, the message is accepted as authentic.

Table 18.1 Summary of PGP Services

Function	Algorithms Used	Description
Digital signature	DSS/SHA or RSA/SHA	A hash code of a message is created using SHA-1. This message digest is encrypted using DSS or RSA with the sender's private key and included with the message.
Message encryption	CAST or IDEA or Three-key Triple DES with Diffie-Hellman or RSA	A message is encrypted using CAST-128 or IDEA or 3DES with a one-time session key generated by the sender. The session key is encrypted using Diffie-Hellman or RSA with the recipient's public key and included with the message.
Compression	ZIP	A message may be compressed for storage or transmission using ZIP.
E-mail compatibility	Radix-64 conversion	To provide transparency for e-mail applications, an encrypted message may be converted to an ASCII string using radix-64 conversion.

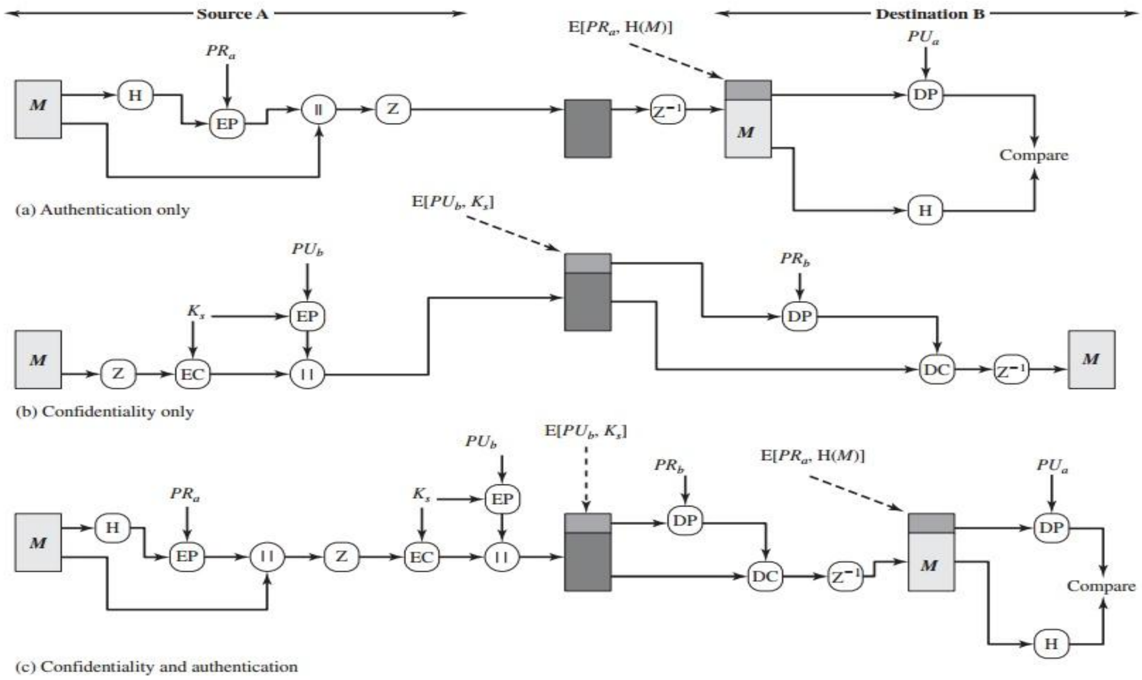


Figure 18.1 PGP Cryptographic Functions

The combination of SHA-1 and RSA provides an effective digital signature scheme. Because of the strength of RSA, the recipient is assured that only the possessor of the matching private key can generate the signature. Because of the strength of SHA-1, the recipient is assured that no one else could generate a new message that matches the hash code and, hence, the signature of the original message.

As an alternative, signatures can be generated using DSS/SHA-1.

Although signatures normally are found attached to the message or file that they sign, this is not always the case: Detached signatures are supported. A detached signature may be stored and transmitted separately from the message it signs. This is useful in several contexts. A user may wish to maintain a separate signature log of all messages sent or received. A detached signature of an executable program can detect subsequent virus infection. Finally, detached signatures can be used when more than one party must sign a document, such as a legal contract. Each person's signature is independent and therefore is applied only to the document. Otherwise, signatures would have to be nested, with the second signer signing both the document and the first signature, and so on.

CONFIDENTIALITY Another basic service provided by PGP is confidentiality, which is provided by encrypting messages to be transmitted or to be stored locally as files. In both cases, the symmetric encryption algorithm CAST-128 may be used. Alternatively, IDEA or 3DES may be used. The 64-bit cipher feedback (CFB) mode is used.

As always, one must address the problem of key distribution. In PGP, each symmetric key is used only once. That is, a new key is generated as a random 128-bit number for each message. Thus, although this is referred to in the documentation as a session key, it is in reality a one-time key. Because it is to be used only once, the session key is bound to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key.

Figure 18.1b illustrates the sequence, which can be described as follows.

1. The sender generates a message and a random 128-bit number to be used as a session key for this message only.
2. The message is encrypted using CAST-128 (or IDEA or 3DES) with the session key.
3. The session key is encrypted with RSA using the recipient's public key and is prepended to the message.
4. The receiver uses RSA with its private key to decrypt and recover the session key.
5. The session key is used to decrypt the message.

As an alternative to the use of RSA for key encryption, PGP provides an option referred to as *Diffie-Hellman*. As was explained in Chapter 10, Diffie- Hellman is a key exchange algorithm. In fact, PGP uses a variant of Diffie-Hellman that does provide encryption/decryption, known as ElGamal .

Several observations may be made. First, to reduce encryption time, the combi- nation of symmetric and public-key encryption is used in preference to simply using RSA or ElGamal to encrypt the message directly: CAST-128 and the other symmetricalgorithms are substantially faster than RSA or ElGamal. Second, the use of the public-key algorithmsolves the session-keydistribution problem, because only the recipient is able to recover the session key that is bound to the message. Note that wedonotneedasession-keyexchangeprotocolofthetypediscussedin Chapter 14, because we are not beginning an ongoing session. Rather, each message is a one-time independent event with its own key. Furthermore, given the store-and-forward nature of electronic mail, the use of handshaking to assure that both sides have the same session key is not practical. Finally, the use of one-time symmetric keys strengthens what is already a strong symmetric encryption approach. Only a small amount of plaintext is encrypted with each key, and there is no relationship

among the keys. Thus, to the extent that the public-key algorithm is secure, the entire scheme is secure.
To this end, PGP provides the user with a range of key size options from 768to 3072 bits (the DSS key for signatures is limited to 1024 bits).

CONFIDENTIALITY AND AUTHENTICATION As Figure 18.1c illustrates, both services may be used for the same message. First, a signature is generated for the plaintext message and prepended to the message. Then the plaintext message plus signature is encrypted using CAST-128 (or IDEA or 3DES), and the session key is encrypted using RSA (or ElGamal). This sequence is preferable to the opposite: encrypting the message and then generating a signature for the encrypted message. It is generally more convenient to store a signature with a plaintext version of a message. Furthermore, for purposes of third-party verification, if the signature is performed first, a third party need not be concerned with the symmetric key when verifying the signature.

In summary, when both services are used, the sender first signs the message with its own private key, then encrypts the message with a session key, and finally encrypts the session key with the recipient's public key.

COMPRESSION Asa default, PGP compresses the messageafter applying the signature but before encryption.This has the benefit of saving space both for e-mail transmission and for file storage.The placement of the compression algorithm, indicated by Z for compressionand Z–1 for decompression in Figure 18.1, is critical.

1. The signature is generated before compression for two reasons:

a. It is preferable to sign an uncompressed message so that one can store only the uncompressed message together with the signature for future verification. If one signed a compressed document, then it would be necessary either to store a compressed version of the message for later verification or to recompress the message when verification is required.Even if one were willing to generate dynamically a recompressed messagefor verification, PGP's compression algorithmpresents a difficulty. The algorithm is not deterministic; various implementations of thealgorithm achieve different tradeoffs in running speed versus compression ratio and, as a result,produce different compressed forms. However, these different compression algorithms are interoperable because any version of the algorithm can correctly decompress the output of any other version. Applying the hash function and signature after compression would constrain all PGP implementations to the same version of the compression algorithm.

2. Message encryption is applied after compression to strengthen cryptographic security. Because the compressed message has less redundancy than the original plaintext, cryptanalysis is more difficult.

The compression algorithm used is ZIP, which is described in Appendix O.

E-MAIL COMPATIBILITY When PGP is used, at least part of the block to be transmitted is encrypted. If only the signature service is used, then the message digest is encrypted

(with the sender's private key).
If the confidentiality service is used, themessage plus signature (if present) are encrypted (With a one-time symmetric key). Thus, part or all of the resulting block consists of a stream of arbitrary8-bit octets. However, many electronic mail systems only permit the use of blocks consisting of ASCII text. To accommodate this restriction, PGP provides the service of converting the raw 8-bit binary stream to a stream of printable ASCII characters.
The scheme used for this purpose is radix-64 conversion.

Each group of three octets of binary data is mapped into four ASCII characters.

This format also appends a CRC to detect transmission errors. See Appendix 18A for a description. The use of radix 64 expands a message by 33%. Fortunately, the session key and signature portions of the message are relatively compact, and the plaintext message has been compressed. In fact, the compression should be more than enough to compensate for the radix-64 expansion. For example, [HELD96] reports an average compression ratio of about 2.0 using ZIP. If we ignore the relatively small signature and key components, the typical overall effect of compression and expansion of a file of length *X* would be 1.33 * 0.5 * *X* = 0.665 * *X*. Thus, there is still an overall compression of about one-third.
One noteworthy aspect of the radix-64 algorithm is that it blindly converts the input stream to radix-64 format regardless of content, even if the input happens to be ASCII text. Thus, if a message is signed but not encrypted and the conversion is applied to the entire block, the output will be unreadable to the casual observer, which provides a certain level of confidentiality. As an option, PGP can be configured to convert to radix-64 format only the signature portion of signed plaintext messages. This enables the human recipient to read the message without using PGP. PGP would still have to be used to verify the signature.

Figure 18.2 shows the relationship among the four services so far discussed. On transmission (if it is required), a signature is generated using a hash code of the uncompressed plaintext. Then the plaintext(plussignature if present) is com- pressed. Next, if confidentiality is required, the block (compressed plaintext or com- pressed signature plus plaintext) is encrypted and prepended with the public-key- encrypted symmetric encryption key. Finally, the entire block is converted to radix-64 format.

On reception, the incoming block is first converted back from radix-64 format to binary. Then, if the message is encrypted, the recipient recovers the session key and decrypts the message. The resulting block is then decompressed. If the message is signed, the recipient recovers the transmitted hash code and compares it to its own calculation of the hash code.

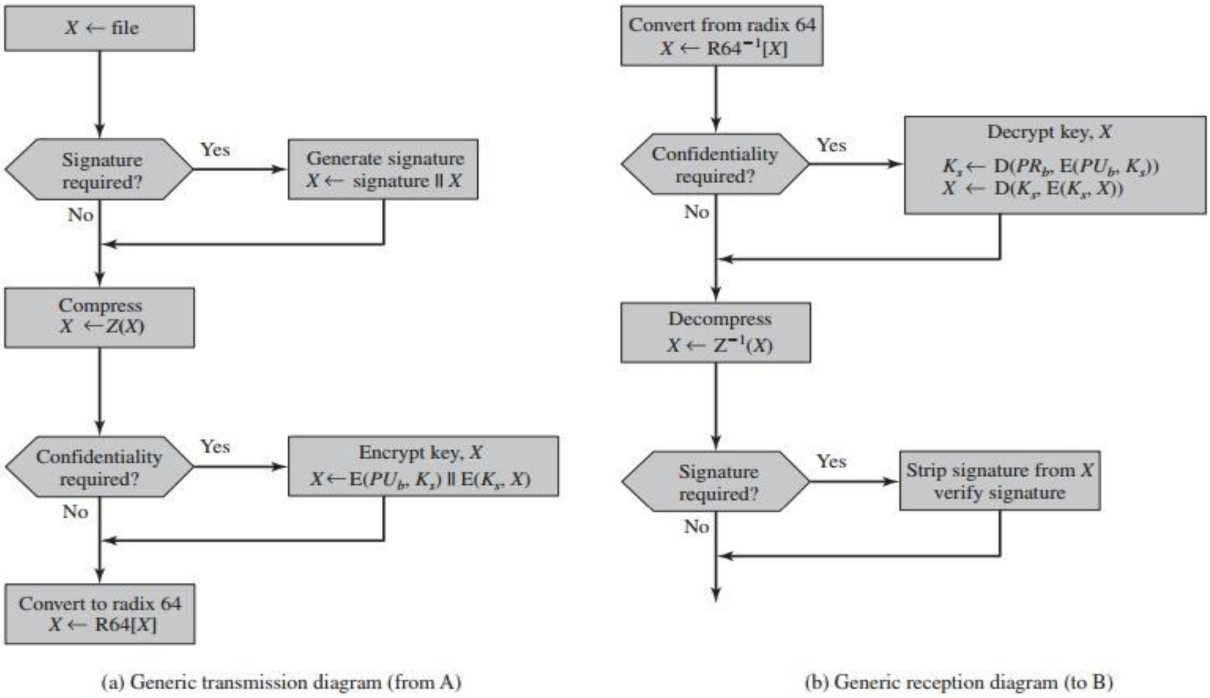


Figure 18.2 Transmission and Reception of PGP Messages

Cryptographic Keys and Key Rings

PGP makes use of four types of keys: one-time session symmetric keys, public keys, private keys, and passphrase-based symmetric keys (explained subsequently). Three separate requirements can be identified with respect to these keys.

1. A means of generating unpredictable session keys is needed.
2. We would like to allow a user to have multiple public-key/private-key pairs. One reason is that the user may wish to change his or her key pair from time to time. When this happens, any messages in the pipeline will be constructed with an obsolete key. Furthermore, recipients will know only the old public key until an update reaches them. In addition to the need to change keys over time, a user may wish to have multiple key pairs at a given time to interact with different groups of correspondents or simply to enhance security by limiting the amount of material encrypted with any one key. The upshot of all this is that there is not a one-to-one correspondence between users and their public keys. Thus, some means is needed for identifying particular keys.
3. Each PGP entity must maintain a file of its own public/private key pairs as well as a file of public keys of correspondents.

SESSION KEY GENERATION Each session key is associated with a single message and is used only for the purpose of encrypting and decrypting that message. Recall that message encryption/decryption is done with a symmetric encryption algorithm. CAST-

128 and IDEA use 128-bit keys; 3DES uses a 168-bit key. For the following discussion, we assume CAST-128.

Random 128-bit numbers are generated using CAST-128 itself. The input to the random number generator consists of a 128-bit key and two 64-bit blocks that are treated as plaintext to be encrypted. Using cipher feedback mode, the CAST-128 encrypter produces two 64-bit cipher text blocks, which are concatenated to form the 128-bit session key. The algorithm that is used is based on the one specified in ANSI X12.17.

The "plaintext" input to the random number generator, consisting of two 64-bit blocks, is itself derived from a stream of 128-bit randomized numbers. These numbers are based on keystroke input from the user. Both the keystroke timing and the actual keys struck are used to generate the randomized stream. Thus, if the user hits arbitrary keys at his or her normal pace, a reasonably "random" input will be generated. This random input is also combined with previous session key output from CAST-128 to form the key input to the generator. The result, given the effective scrambling of CAST128, is to produce a sequence of session keys that is effectiv ely unpredictable.

KEY IDENTIFIERS As we have discussed, an encrypted message is accompanied by an encrypted form of the session key that was used for message encryption. The session key itself is encrypted with the recipient's public key. Hence, only the recipient will be able to recover the session key and therefore recover the message. If each user employed a single public/private key pair, then the recipient would automatically know which key to use to decrypt the session key: the recipient's unique private key. However, we have stated a requirement that any given user may have multiple public/private key pairs.How, then, does the recipient know which of its public keys was used to encrypt the session key? One simple solution would be to transmit the public key with the message. The recipient could then verify that this is indeed one of its public keys, and proceed. This scheme would work, but it is unnecessarily wasteful o f space.An RSA public key may be hundreds of decimal digits in length. Another solution would be to associate an identifier with each public key that is unique at least within one user. That is, the combination of user ID and key ID would be sufficient to iden- tify a key uniquely.

Then only the much shorter key ID would need to be transmitted. This solution, however, raises a management and overhead problem: Key IDs must be assigned and stored so that both sender and recipient could map from key ID to public key. This seems unnecessarily burdensome.

The solution adopted by PGP is to assign a key ID to each public key that is, with very high probability, unique within a user ID. The key ID associated with each public key consists of its least significant 64 bits. That is, the key ID of public key PUa is (PUa mod 264). This is a sufficient length that the probability of duplicate key IDs is very small.

A key ID is also required for the PGP digital signature. Because a sender may use one of a number of private keys to encrypt the message digest, the recipient must know which public key is intended for use. Accordingly, the digital signature component of a message includes the 64-bit key

ID of the required public key. When the message is received, the recipient verifies that the key ID is for a public key that it knows for that sender and then proceeds to verify the signature.

Now that the concept of key ID has been introduced, we can take a more detailed look at the

format of a transmitted message, which is shown in Figure 18.3. A message consists of three components: the messagecomponent, a signature (optional), and a session key component.

The **message component** includes the actual data to be stored or transmitted, as well as a filename and a timestamp that specifies the time of creation.

The **signature component** includes the following.

Timestamp: The time at which the signature was made.

Message digest: The 160-bit SHA-1 digest encrypted with the sender's private signature key. The digest is calculated over the signature timestamp concatenated with the data portion of the message component. The inclusion of the signature timestamp in the digest insures against replay types of attacks. The exclusion of the filename and timestamp portions of the message component ensures that detached signatures are exactly the same as attached signatures

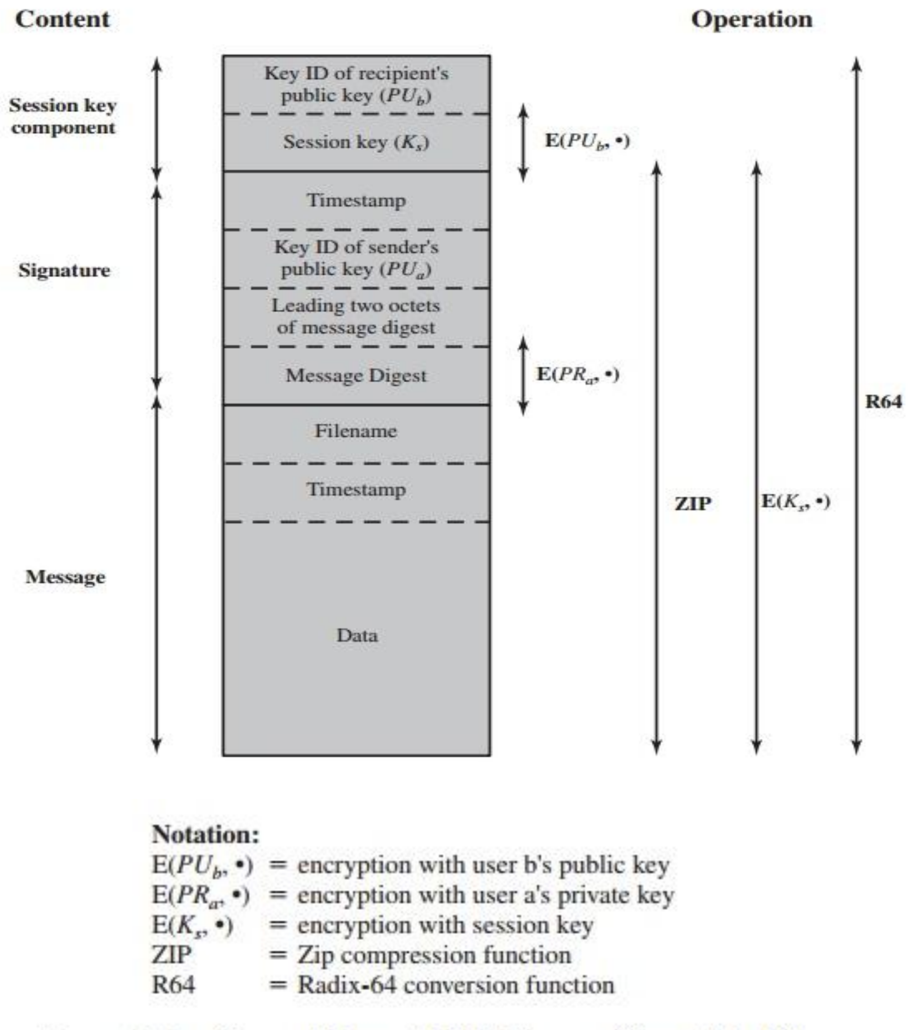


Figure 18.3 General Format PGP Message (from A to B)

prefixed to the message. Detached signatures are calculated on a separate file that has none of the message component header fields.

Leading two octets of message digest: Enables the recipient to determine if the correct public key was used to decrypt the message digest for authentication by comparing this plaintext copy of the first two octets with the first two octets of the decrypted digest. These octets also serve as a 16-bit frame check sequence for the message.

Key ID of sender's public key: Identifies the public key that should be used to decrypt the message digest and,hence, identifies the private key that was used to encrypt the message digest.The message component and optional signature component may be com- pressed using ZIP and may be encrypted using a session key.

The **session key component** includes the session key and the identifier of the recipient's public key that was used by the sender to encrypt the session key.

The entire block is usually encoded with radix-64 encoding.

KEY RINGS We have seen how key IDs are critical to the operation of PGP and that two key IDs are included in any PGP message that provides both confidentiality and authentication. These keys need to be stored and organized in a systematic way for efficient and effective use by all parties. The scheme used in PGP is to provide a pair of data structures at each node, one to store the public/private key pairs owned by that node and one to store the public keys of other users known at this node. These data structures are referred to, respectively, as the private-key ring and the public-key ring.

Figure 18.4 shows the general structure of a **private-key ring.** We can view the ring as a table in which each row represents one of the public/private key pairs owned by this user.

Each row contains the entries:

Timestamp: The date/time when this key pair was generated.

Key ID: The least significant 64 bits of the public key for this entry.

Public key: The public-key portion of the pair.

Private key: The private-key portion of the pair; this field is encrypted.

User ID: Typically, this will be the user's e-mail address (e.g., stallings@acm.org). However, the user may choose to associate a different name with each pair (e.g., Stallings, WStallings, WilliamStallings, etc.) or to reuse the same User ID more than once.

The private-key ring can be indexed by either User ID or Key ID; later we will see the need for both means of indexing.Although it is intended that the private-key ring be stored only on the machine of the user that created and owns the key pairs and that it be accessible only to that user, it makes sense to make the value of the private key as secure as possible. Accordingly, the private key itself is not stored in the key ring. Rather, this key is encrypted using CAST-128 (or IDEA or 3DES).

Private-Key Ring

Timestamp	Key ID*	Public Key	Encrypted Private Key	User ID*
• • •	• • •	• • •	• • •	• • •
T_i	$PU_i \bmod 2^{64}$	PU_i	$E(H(P_i), PR_i)$	User i
• • •	• • •	• • •	• • •	• • •

Public-Key Ring

Timestamp	Key ID*	Public Key	Owner Trust	User ID*	Key Legitimacy	Signature(s)	Signature Trust(s)
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
T_i	$PU_i \bmod 2^{64}$	PU_i	$trust\_flag_i$	User i	$trust\_flag_i$		
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •

* = field used to index table

Figure 18.4 General Structure of Private- and Public-Key Rings

The procedure is as follows:

1.The user selects a passphrase to be used for encrypting private keys.

2.When the system generates a new public/private key pair using RSA, it asks the user for the passphra se. Using SHA-1, a 160bit hash code is generated from the passphrase, and the passphrase is discarded.

3.The system encrypts the private key using CAST-128 with the 128 bits of the hash code as the key. The hash code is then discarded, and the encrypted private key is stored in the private-key ring.

Figure 18.4 also shows the general structure of a **public-key ring.** This data structure is used to store public keys of other users that are known to this user. For the moment, let us ignore some fields shown in the figure and describe the following fields.

- **Timestamp:** The date/time when this entry was generated.
- **Key ID:** The least significant 64 bits of the public key for this entry.
- **Public Key:** The public key for this entry.
- **User ID:** Identifies the owner of this key. Multiple user IDs may be associated with a single public key.

The public-key ring can be indexed by either User ID or Key ID.

We are now in a position to show how these key rings are used in message transmission and reception. For simplicity, we ignore compression and radix-64 con- version in the following discussion. First consider message transmission (Figure 18.5) and assume that the message is to be both signed and encrypted. The sending PGP entity performs the following steps.

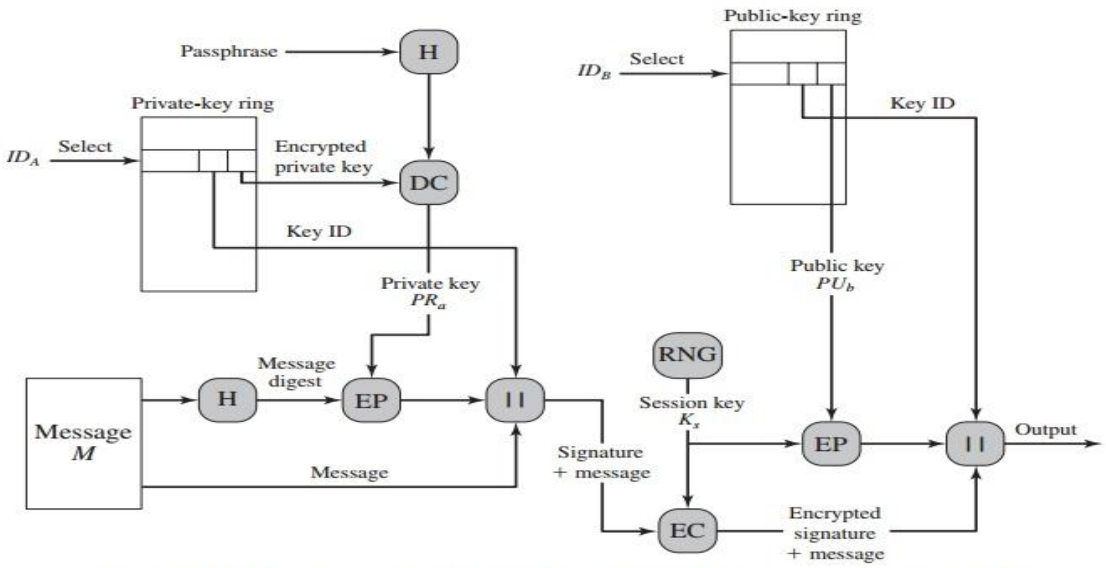


Figure 18.5 PGP Message Generation (from User A to User B: no compression or radix-64 conversion)

Figure 18.5 PGP Message Generation (from User A to User B: no compression or radix-64 conversion)

1. Signing the message:

a. PGP retrieves the sender's private key from the private-key ring using your_userid as an index. If your_useridwas not provided in the command, the first private key on the ring is retrieved.

b. PGP prompts the user for the passphrase to recover the unencrypted private key.

c. The signature component of the message is constructed.

2. Encrypting the message:

a. PGP generates a session key and encrypts the message.

b. PGP retrieves the recipient's public key from the public-key ring usingher_useridas an index.

c. The session key component of the message is constructed.

The receiving PGP entity performs the following steps (Figure 18.6).

1. Decrypting the message:

a. PGP retrieves the receiver's private key from the private-key ring using the Key ID field in the session key component of the message as an index.

b. PGP prompts the user for the passphrase to recover the unencrypted private key.

c. PGP then recovers the session key and decrypts the message.

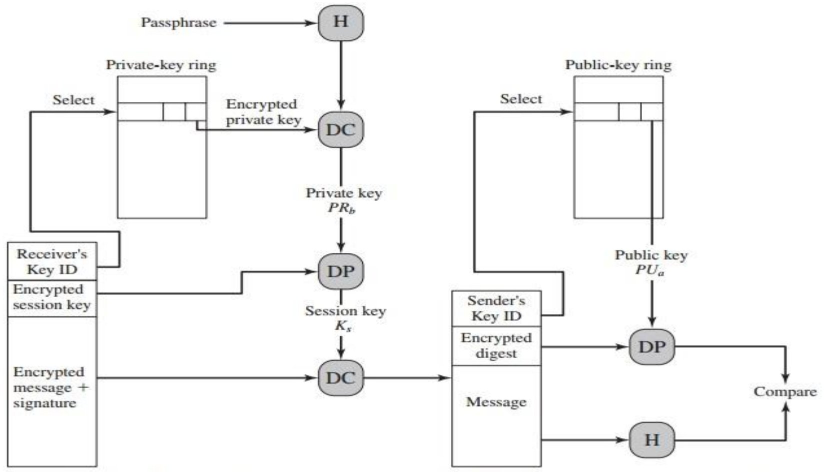


Figure 18.6 PGP Message Reception (from User A to User B; no compression or radix-64 conversion)

2. Authenticating the message:

a. PGP retrieves the sender's public key from the public-key ring using the Key ID field in the signature key component of the message as an index.

b. PGP recovers the transmitted message digest.

c. PGP computes the message digest for the received message and compares it to the transmitted message digest to authenticate.

Public-Key Management

As can be seen from the discussion so far, PGP contains a clever, efficient, interlock- ing set of functions and formats to provide an effective confidentiality and authentication service. To complete the system, one final area needs to be addressed, that of public-key management. The PGP documentation captures the importance of this area:

This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. Itis the "Achilles heel" of public key cryptography, and a lot of software complexity is tied up in solving this one problem.

PGP provides a structure for solving this problem with several suggested options that may be used. Because PGP is intended for use in a variety of formal and informal environments, no rigid public-key management scheme is set up.

5.7 S/MIME

Secure/Multipurpose Internet Mail Extension (S/MIME) is a security enhancement to the MIME Internet e-mail format standard based on technology from RSA Data Security. Although both PGP and S/MIME are on an IETF standards track, it appears likely that S/MIME will emerge as the industry standard for commercial and organizational use, while PGP will remain the choice for personal e-mail security for many users. S/MIME is defined in a number of documentsmost importantly RFCs 3370, 3850, 3851, an d 3852.

To understand S/MIME, we need first to have a general understanding of the underlying e-mail format that it uses, namely MIME. But to understand the significance of MIME, we need to go back to the traditional e-mail format standard, RFC 822, which is still in common use.

RFC 5322

RFC 5322 defines a format for text messages that are sent using electronic mail. It has been the standard for Internet-based text mail messages and remains in common use. In the RFC 5322 context, messages are viewed as having an envelope and contents. The envelope contains whatever information is needed to accomplish transmission and delivery. The contents compose the object to be delivered to the recipient. The RFC 5322 standard applies only to the contents. However, the content standard includes a set of header fields that may be used by the mail system to create the envelope, and the standard is intended to facilitate the acquisition of such information by programs.

The overall structure of a message that conforms to RFC 5322 is very simple. A message consists of some number of header lines (*the header*) followed by unrestricted text (*the body*). The header is separated from the body by a blank line. Put differently, a message is ASCII text, and all lines up to the first blank line are assumed to be header lines used by the user agent part of the mail system. A header line usually consists of a keyword, followed by a colon, followed by the keyword's arguments; the format allows a long line to be broken up into several lines. The most frequently used keywords are *From*, *To*, *Subject*, and *Date*. Here is an example message:

Date: October 8, 2009 2:15:49 PM EDT

From: "William Stallings"<ws@shore.net>

Subject: The Syntax in RFC 5322 To: Smith@Other-host.com

Cc: Jones@Yet-Another-Host.com

Multipurpose Internet Mail Extensions

Multipurpose Internet Mail Extension (MIME) is an extension to the RFC 5322 framework that is intended to address some of the problems and limitations of the use of Simple Mail Transfer Protocol (SMTP), defined in RFC 821, or some other mail transfer protocol and RFC 5322 for electronic mail. [PARZ06] lists the following limitations of the SMTP/5322 scheme.

1. SMTP cannot transmit executable files or other binary objects. A number of schemes are in use for converting binary files into a text form that can be used by SMTP mail systems, including the popular UNIX UUencode/UUdecode scheme.

However, none of these is a standard or even a *de facto* standard.

2. SMTP cannot transmit text data that includes national language characters, because these are represented by 8-bit codes with values of 128 decimal or higher, and SMTP is limited to 7-bit ASCII.

3. SMTP servers may reject mail message over a certain size.

4. SMTP gateways that translate between ASCII and the character code EBCDIC do not use a consistent set of mappings, resulting in translation problems.

5. SMTP gateways to X.400 electronic mail networks cannot handle nontextual data included in X.40 0 messages.

6. Some SMTP implementations do not adhere completely to the SMTP standards defined in RFC 821. Common problems include:

- Deletion, addition, or reordering of carriage return and linefeed
- Truncating or wrapping lines longer than 76 characters
- Removal of trailing white space (tab and space characters)
- Padding of lines in a message to the same length
- Conversion of tab characters into multiple space characters

MIME is intended to resolve these problems in a manner that is compatible with existing RFC 5322 implementations. The specification is provided in RFCs 2045 through 2049.

The MIME specification includes the following elements.

1. Five new message header fields are defined, which may be included in an RFC 5322 header. These fields provide information about the body of the message.

2. A number of content formats are defined, thus standardizing representations that support multimedia electronic mail.

3. Transfer encodings are defined that enable the conversion of any content for-mat into a form that is protected from alteration by the mail system.

In this subsection, we introduce the five message header fields. The next two subsections deal with content formats and transfer encodings.

The five header fields defined in MIME are

MIME-Version: Must have the parameter value 1.0. This field indicates that the message

conforms to RFCs 2045 and 2046.

Content-Type: Describes the data contained in the body with sufficient detail that the receiving user agent can pick an appropriate agent or mechanism to represent the data to the user or otherwise deal with the data in an appropriate manner.

ContentTransferEncoding: Indicates the type of transformation that has been used to represent th e body of the message in a way that is acceptable for mail transport.

- **Content-ID:** Used to identify MIME entities uniquely in multiple contexts.
- **Content-Description:** A text description of the object with the body; this is useful when the object is not readable (e.g., audio data).

Any or all of these fields may appear in a normal RFC 5322 header. A compliant implementation must support the MIME-Version, Content-Type, and Content- Transfer-Encoding fields; the Content-ID and Content-Description fields are optional and may be ignored by the recipient implementation.

MIME CONTENT TYPES The bulk of the MIME specification is concerned with the definition of a variety of content types.This reflects the need to provide standardized ways of dealing with a wid e variety of information representations in a multimedia environment.

Table 18.3 lists the content types specified in RFC 2046. There are seven different major types of content a nd a total of 15 subtypes. In general, a content type declares the general type of data, and the subtype speci fies a particular format for that type of data.

For the **text type** of body, no special software is required to get the full meaning of the text aside from support of the indicated character set. The primary subtype is *plain text*, which is simply a string of ASCII characters or ISO 8859 characters. The *enriched* subtype allows greater formatting flexibility The **multipart type** indicates that the body contains multiple, independent parts. The Content-Type header field includes a parameter (called a boundary) that defines the delimiter between body parts. This boundary should not appear in any parts of the message. Each boundary starts on a new line and consists of two hyphens followed by the boundary value. The final boundary, which indicates the end of the last part, also has a suffix of two hyphens. Within each part, there may be an optional ordinary MIME header.

Table 18.3 MIME Content Types

Table 18.3 MIME Content Types

Type	Subtype	Description
Text	Plain	Unformatted text; may be ASCII or ISO 8859.
	Enriched	Provides greater format flexibility.
Multipart	Mixed	The different parts are independent but are to be transmitted together. They should be presented to the receiver in the order that they appear in the mail message.
	Parallel	Differs from Mixed only in that no order is defined for delivering the parts to the receiver.
	Alternative	The different parts are alternative versions of the same information. They are ordered in increasing faithfulness to the original, and the recipient's mail system should display the "best" version to the user.
	Digest	Similar to Mixed, but the default type/subtype of each part is message/rfc822.
Message	rfc822	The body is itself an encapsulated message that conforms to RFC 822.
	Partial	Used to allow fragmentation of large mail items, in a way that is transparent to the recipient.
	External-body	Contains a pointer to an object that exists elsewhere.
Image	jpeg	The image is in JPEG format, JFIF encoding.
	gif	The image is in GIF format.
Video	mpeg	MPEG format.
Audio	Basic	Single-channel 8-bit ISDN mu-law encoding at a sample rate of 8 kHz.
Application	PostScript	Adobe Postscript format.
	octet-stream	General binary data consisting of 8-bit bytes.

There are four subtypes of the multipart type, all of which have the same overall syntax. The **multipart/mixed subtype** is used when there are multiple indepen- dent body parts that need to be bundled in a particular order. For the **multipart/parallel subtype,** the order of the parts is not significant. If the r ecipient's system is appropriate, the multiple parts can be presented in parallel. For example, a picture or text part could be accompanied by a voice commentary that is played while the picture or text is displayed.

For the **multipart/alternative subtype,** the various parts are different represen-tations of the same information. The following is an example:

From: Nathaniel Borenstein <nsb@bellcore.com> To: Ned Freed <ned@innosoft.com>

Subject: Formatted text mail MIME-Version: 1.0

Content-Type: multipart/alternative; boundary=boundary42

—boundary42

Content-Type: text/plain; charset=us-ascii

...plain text version of message goes here....

—boundary42

Content-Type: text/enriched

.... RFC 1896 text/enriched version of same message goes here ...

—boundary42—

In this subtype, the body parts are ordered in terms of increasing preference. For this example, if the recipient system is capable of displaying the message in the text/enriched format, this is done; otherwise, the plain text format is used.

The **multipart/digest subtype** is used when each of the body parts is interpreted as an RFC 5322 message with headers. This subtype enables the construction of a message whose parts are individual messages. For example, the moderator of a group might collect e-mail messages from participants, bundle these messages, and send them out in one encapsulating MIME message.

The **message type** provides a number of important capabilities in MIME. The **message/rfc822 subtype** indicates that the body is an entire message, including header and body. Despite the name of this subtype, the encapsulated message may be not only a simple RFC 5322 message but also any MIME message.

The **message/partial subtype** enables fragmentation of a large message into a number of parts, which must be reassembled at the destination. For this subtype, three parameters are specified in the Content-Type: Message/Partial field: an *id* common to all fragments of the same message, a *sequence number* unique to each fragment, and the *total* number of fragments.

The **message/externalbody subtype** indicates that the actual data to be conveyed in this message are not contained in the body. Instead, the body contains the information needed to access the data. As with the other message types, the message/external-body subtype has an outer header and an encapsulated message with its own header. The only necessary field in the outer header is the Content-Type field, which identifies this as a message/external-body subtype. The inner header is the message header for the encapsulated message. The Content-Type field in the outer header must include an access-type parameter, which indicates the method of access, such as FTP (file transfer protocol).

The **application type** refers to other kinds of data, typically either uninterpreted binary data or information to be processed by a mail-based application.

MIME TRANSFER ENCODINGS The other major component of the MIME specification, in addition to content type specification, is a definition of transfer encodings for message bodies. The objective is to provide reliable delivery across the largest range of environments.

The MIME standard defines two methods of encoding data. The Content- Transfer-Encoding field can actually take on six values, as listed in Table 18.4. However, three of these values (7bit, 8bit, and binary) indicate that no encoding has been done but provide some information about the nature of the data. For SMTP transfer, it is safe to use the 7bit form. The 8bit and binary forms may be usable in other mail transport contexts. Another Content-Transfer-Encoding value is x-token,

Table 18.4 MIME Transfer Encodings

7bit	The data are all represented by short lines of ASCII characters.
8bit	The lines are short, but there may be non-ASCII characters (octets with the high-order bit set).
binary	Not only may non-ASCII characters be present, but the lines are not necessarily short enough for SMTP transport.
quoted-printable	Encodes the data in such a way that if the data being encoded are mostly ASCII text, the encoded form of the data remains largely recognizable by humans.
base64	Encodes data by mapping 6-bit blocks of input to 8-bit blocks of output, all of which are printable ASCII characters.
x-token	A named nonstandard encoding.

which indicates that some other encoding scheme is used for which a name is to be supplied. Thi s could be a vendor-specific or application-specific scheme. The two actual encoding schemes defined are quoted-printable and base64. Two schemes are defined to provide a choice between

a transfer technique that is essentially human readable and one that is safe for all types of data in a way that is reasonably compact. The **quoted-printable** transfer encoding is useful when the data consists largely of octets that correspond to printable ASCII characters.

In essence,it represents nonsafe characters by the hexadecimal representation of their code andi ntroduces reversible (soft) line breaks to limit message lines to 76 characters.

The **base64 transfer encoding,** also known as radix-64 encoding, is a common one for encoding arbitrary binary data in such a way as to be invulnerable to the processing by mail-transport programs. It is also used in PGP and is described in Appendix 18A.

CANONICAL FORM An important concept in MIME and S/MIME is that of canonical form.

Canonical form is a format, appropriate to the content type, that isstandardized for use between syste ms. This is in contrast to native form, which is a format that may be peculiar to a particular system. Table 18.5, from RFC 2049, should help clarify this matter.

S/MIME Functionality

In terms of general functionality, S/MIME is very similar to PGP. Both offer the ability to sign and/or encrypt messages. In this subsection, we briefly summarize S/MIME capability. We then look in more detail at this capability by examining mes-
sage formats and message preparation.

MIME-Version: 1.0

From: Nathaniel Borenstein <nsb@bellcore.com> To: Ned Freed <ned@innosoft.com>

Subject: A multipart example Content-Type: multipart/mixed;

boundary=unique-boundary-1

This is the preamble area of a multipart message. Mail readers that understand multipart format should ignore this preamble. If you are reading this text, you might want to consider changing to a mail reader that understands how to properly display multipart messages.

--unique-boundary-1

...Some text appears here...

[Note that the preceding blank line means no header fields were given and this is text, with charset US ASCII. It could have been done with explicit typing as in the next part.]

From: (mailbox in US-ASCII) To: (address in US-ASCII) Subject: (subject in US-ASCII)

Content-Type: Text/plain; charset=ISO-8859-1 Content-Transfer-Encoding: Quoted-printable

... Additional text in ISO-8859-1 goes here ...

--unique-boundary-1--

Table 18.5 Native and Canonical Form

Native Form	The body to be transmitted is created in the system's native format. The native character set is used and, where appropriate, local end-of-line conventions are used as well. The body may be a UNIX-style text file, or a Sun raster image, or a VMS indexed file, or audio data in a system-dependent format stored only in memory, or anything else that corresponds to the local model for the representation of some form of information. Fundamentally, the data is created in the "native" form that corresponds to the type specified by the media type.
Canonical Form	The entire body, including "out-of-band" information such as record lengths and possibly file attribute information, is converted to a universal canonical form. The specific media type of the body as well as its associated attributes dictate the nature of the canonical form that is used. Conversion to the proper canonical form may involve character set conversion, transformation of audio data, compression, or various other operations specific to the various media types. If character set conversion is involved, however, care must be taken to understand the semantics of the media type, which may have strong implications for any character set conversion (e.g., with regard to syntactically meaningful characters in a text subtype other than "plain").

S/MIME provides the following functions.

- **Enveloped data:** This consists of encrypted content of any type and encrypted-content encryption keys for one or more recipients.
- **Signed data:** A digital signature is formed by taking the message digest of the content to be signed and then encrypting that with the private key of the signer. The content plus signature are then encoded using base64 encoding. A signed data message can only be viewed by a recipient with S/MIME capability.
- **Clear-signed data:** As with signed data, a digital signature of the content is formed. However, in this case, only the digital signature is encoded using base64. As a result, recipients without S/MIME capability can view the mes sage content, although they cannot verify the signature.
- **Signed and enveloped data:** Signed-only and encrypted-only entities may be nested, so that encrypted data may be signed and signed data or clear-signed data may be encrypted.

CRYPTOGRAPHIC ALGORITHMS Table 18.6 summarizes the cryptographic algorithms used in S/MIME. S/MIME uses the following terminology taken from RFC 2119 (*Key Words for use in RFCs to Ind icate Requirement Levels*) to specify the requirement level:

- **MUST:** The definition is an absolute requirement of the specification. An implementation must include this feature or function to be in conformance with the specification.
- **SHOULD:** There may exist valid reasons in particular circumstances to ignore this feature or function, but it is recommended that an implementation include the feature or function.

S/MIME incorporates three public-key algorithms. The Digital Signature Standard (DSS) described in Chapter 13 is the preferred algorithm for digital signature. S/MIME lists Diffie-Hellman as the preferred algorithm for encrypting session keys; in fact, S/MIME uses a variant of Diffie-Hellman that does provide

Table 18.6 Cryptographic Algorithms Used in S/MIME

Function	Requirement
Create a message digest to be used in forming a digital signature.	MUST support SHA-1. Receiver SHOULD support MD5 for backward compatibility.
Encrypt message digest to form a digital signature.	Sending and receiving agents MUST support DSS. Sending agents SHOULD support RSA encryption. Receiving agents SHOULD support verification of RSA signatures with key sizes 512 bits to 1024 bits.
Encrypt session key for transmission with a message.	Sending and receiving agents SHOULD support Diffie-Hellman. Sending and receiving agents MUST support RSA encryption with key sizes 512 bits to 1024 bits.
Encrypt message for transmission with a one-time session key.	Sending and receiving agents MUST support encryption with tripleDES. Sending agents SHOULD support encryption with AES. Sending agents SHOULD support encryption with RC2/40.
Create a message authentication code.	Receiving agents MUST support HMAC with SHA-1. Sending agents SHOULD support HMAC with SHA-1.

encryption/decryption.

S/MIME Messages

S/MIME makes use of a number of new MIME content types, which are shown in Table 18.7. All of the new application types use the designation PKCS. This refers to a set of public-key cryptography specifications issued by RSA Laboratories and made available for the S/MIME effort.

SECURING A MIME ENTITY

S/MIME secures a MIME entity with a signature, encryption, or both. A MIME entity may be an entire message (except for the RFC 5322 headers), or if the MIME content type is multipart, then a MIME entity is one or more of the subparts of the message. The MIME entity is prepared according to the normal rules for MIME message preparation. Then the MIME entity plus some security-related data, such as algorithm identifiers and certificates, are processed by S/MIME to produce what is known as a PKCS object. A PKCS object is then treated as message content and wrapped in MIME (provided with appropriate MIME headers).

In all cases, the message to be sent is converted to canonical form. In particular, for a given type and subtype, the appropriate canonical form is used for the message content. For a multipart message, the appropriate canonical form is used for each subpart.

The use of transfer encoding requires special attention. For most cases, the result of applying the security algorithm will be to produce an object that is partially or totally represented in arbitrary binary data. This will then be wrapped in an outer MIME message, and transfer encoding can be applied

at that point, typically base64. However, in the case of a multipart signed message (described in more det ail later), the message content in one of the subparts is unchanged by the security process. Unless that content is 7bit, it should be transfer encoded using base64 or quotedprintable so that there is no danger of altering the content to which the signature was applied.
We now look at each of the S/MIME content types.

Table 18.7 S/MIME Content Types

Type	Subtype	smime Parameter	Description
Multipart	Signed		A clear-signed message in two parts: one is the message and the other is the signature.
Application	pkcs7-mime	signedData	A signed S/MIME entity.
	pkcs7-mime	envelopedData	An encrypted S/MIME entity.
	pkcs7-mime	degenerate signedData	An entity containing only public-key certificates.
	pkcs7-mime	CompressedData	A compressed S/MIME entity.
	pkcs7-signature	signedData	The content type of the signature subpart of a multipart/signed message.

S/MIME Certificate Processing

S/MIME uses public-key certificates that conform to version 3 of X.509 . The key-management scheme used by S/MIME is in some ways a hybrid between a strict X.509 certification hierarchy and PGP's web of trust. As with the PGP model, S/MIME managers and /or users must configure each client with a list of trusted keys and with certificate revocation lists.

That is, the responsi- bility is local for maintaining the certificates needed to verify incoming signatures and to encrypt outgoing messages.

On the other hand, the certificates are signed by certification authorities.

USER AGENT ROLE An S/MIME user has several key-management functions to perform.

Key generation: The user of some related administrative utility MUST be capable of generating sepa rate Diffie-Hellman and DSS key pairs and SHOULD be capable of generating RSA key pairs. Each key pair MUST be generated from a good source of non- deterministic random

input and be protected in a secure fashion. A user agentSHOULD generate RSA key pairs with a lengt h in the range of 768 to 1024 bits and MUST NOT generate a length of less than 512 bits.

- **Registration:** A user's public key must be registered with a certification authority in order to receive an X.509 public-key certificate.
- **Certificate storage and retrieval:** A user requires access to a local list of certifi- cates in order to verify incoming signatures and to encrypt outgoing messages. Such a list could be maintained by the user or by some local administrative entity on behalf of a number of users.

VERISIGN CERTIFICATES

VeriSign provides a CA service that is intended to be compatible with S/MIME and a variety of other applications. VeriSign issues X.509 certificates with the product name VeriSign Digital ID. As of early 1998, over 35,000 commercial Web sites were using VeriSign Server Digital IDs, and over a million consumer Digital IDs had been issued to users of Netscape and Microsoft browsers.

The information contained in a Digital ID depends on the type of Digital ID and its use. At a minimum, each Digital ID contains

- Owner's public key
- Owner's name or alias
- Expiration date of the Digital ID
- Serial number of the Digital ID
- Name of the certification authority that issued the Digital ID
- Digital signature of the certification authority that issued the Digital ID

Digital IDs can also contain other user-supplied information, including

- Address
- E-mail address
- Basic registration information (country, zip code, age, and gender)

VeriSign provides three levels, or classes, of security for public-key certificates, as summarized in Table 18.8. A user requests a certificate online at VeriSign's Web site or other participating Web sites. Class 1 and Class 2 requests are processed on line, and in most cases take only a few seconds to approve. Bri efly, the following procedures are used.

- For Class 1 Digital IDs, VeriSign confirms the user's e-mail address by sending a PIN and Digital ID pick-up information to the e-mail address provided in the application.
- For Class 2 Digital IDs, VeriSign verifies the information in the application through an automated comparison with a consumer database in addition to performing all of the checking associated with a Class 1 Digital ID.

Finally, confirmation is sent to the specified postal address alerting the user that a Digital ID has been issued in his or her name.

- For Class 3 Digital IDs, VeriSign requires a higher level of identity assurance. An individual must prove his or her identity by providing notarized credentials or applying in person.

5.8 DOMAINKEYS IDENTIFIED MAIL

DomainKeys Identified Mail (DKIM) is a specification for cryptographically signing e-mail messages, permitting a signing domain to claim responsibility for a message in the mail stream. Message recipients (or agents acting in their behalf) can verify the signature by querying the signer's domain directly to retrieve the appropriate public key and thereby can confirm that the message was attested to by a party in possession of the private key for the signing domain. DKIM is a proposed Internet Standard (RFC 4871: *DomainKeys Identified Mail (DKIM) Signatures*). DKIM has been widely adopted by a range of e-mail providers, including corporations, government agencies, gmail, yahoo, and many Internet Service Providers (ISPs).

This section provides an overview of DKIM. Before beginning our discussion of DKIM, we introduce the standard Internet mail architecture. Then we look at the threat that DKIM is intended to address, and finally provide an overview of DKIM operation.

Internet Mail Architecture

To understand the operation of DKIM, it is useful to have a basic grasp of the Internet mail architecture, which is currently defined in [CROC09]. This subsectionprovides an overview of the basic concepts.At its most fundamental level, the Internet mail architecture consists of a user world in the form of Message User Agents (MUA), and the transfer world, in the form of the Message Handling Service (MHS), which is composed of Message Transfer Agents (MTA).

The MHS accepts a message from one user and delivers it to one or more other users, creating a virtual MUA-to-MUA exchange environment. This architecture involves three types of interoperability. One is directly between users: messages must be formatted by the MUA on behalf of the message author so that the message can be displayed to the message recipient by the destination MUA. There are also interoperability requirements between the MUA and the MHS—first when a message is posted from an MUA to the MHS and later when it is delivered from the MHS to the destination MUA. Interoperability is required among the MTA components along the transfer path through the MHS.

Figure 18.9 illustrates the key components of the Internet mail architecture, which include the following.

• **Message User Agent (MUA):** Works on behalf of user actors and user applications. It is their representative within the e-mail service. Typically, this function is housed in the user's computer and is referred to as a client e-mail program or a local network e-mail server.

The author MUA formats a message and performs initial submission into the MHS via a MSA. The recipient MUA processes received mail for storage and/or display to the recipient user.

• **Mail Submission Agent (MSA):** Accepts the message submitted by an MUA and enforces the policies of the hosting domain and the requirements of Internet standards.

This function may be located together with the MUA or as a separate functional model. In the latter case, the Simple Mail Transfer Protocol (SMTP) is used betweenthe MUA and the MSA.

• **Message Transfer Agent (MTA):** Relays mail for one application-level hop. It is like a packet switch or IP router in that its job is to make routing assessments and to move the message closer to the recipients. Relaying is performed by a sequence of MTAs until the message reaches a destination MDA. An MTA also adds trace information to the message header. SMTP is used between MTAs and between an MTA and an MSA or MDA.

• **Mail Delivery Agent (MDA):** Responsible for transferring the message from the MHS to the MS.

• **Message Store (MS):** An MUA can employ a long-term MS. An MS can be located on a remote server or on the same machine as the MUA. Typically, an MUA retrieves messages from a remote server using POP (Post Office Protocol) or IMAP (Internet Message Access Protocol).

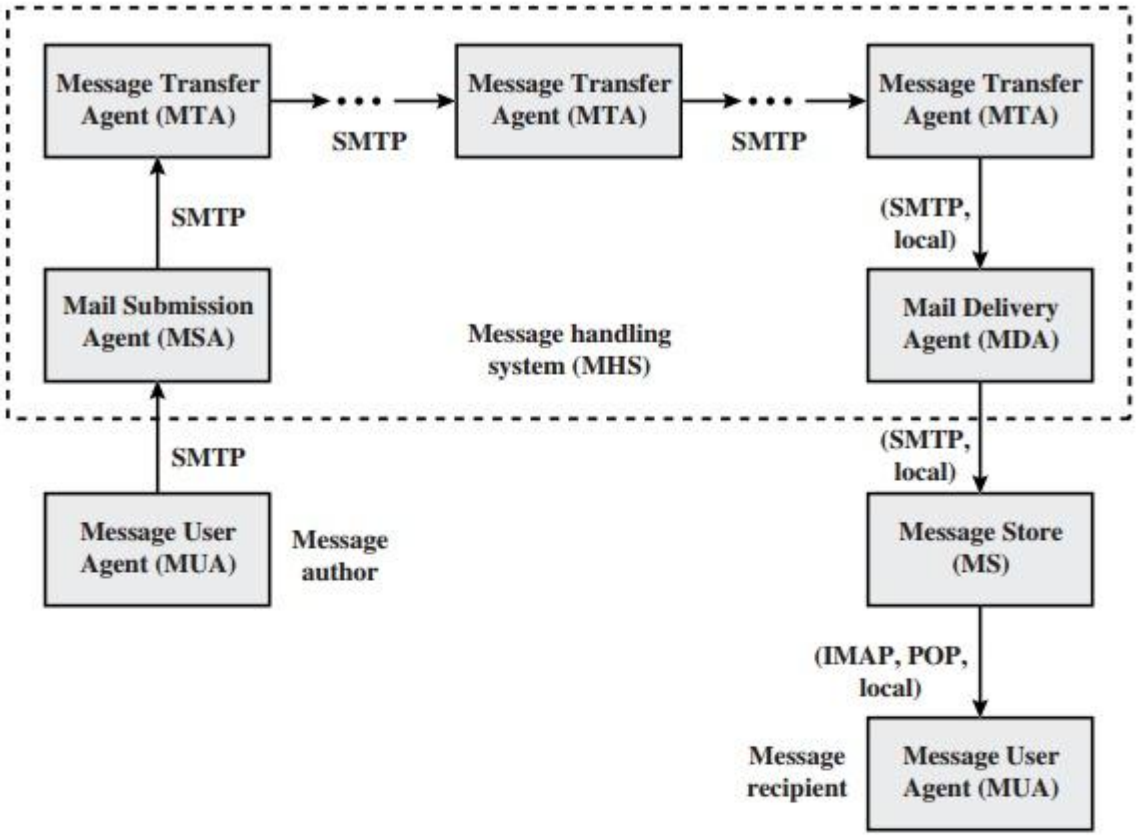


Figure 18.9 Function Modules and Standardized Protocols for the Internet

Two other concepts need to be defined. An **administrative management domain (ADMD)** is an Internet e-mail provider. Examples include a department that operates a local mail relay (MTA), an IT department that operates an enterprise mail relay, and an ISP that operates a public shared e-mail service. Each ADMD can have different operating policies and trust-based decision making. One obvious example is the distinction between mail that is exchanged within an organization and mail that is exchanged between independent organizations. The rules for handling the two types of traffic tend to be quite different.

The **Domain Name System (DNS)** is a directory lookup service that provides a mapping between the name of a host on the Internet and its numerical address.

E-mail Threats

RFC 4684 (*Analysis of Threats Motivating DomainKeys Identified Mail*) describes the threats being addressed by DKIM in terms of the characteristics, capabilities, and location of potential attackers.

CHARACTERISTICS RFC characterizes the range of attackers on a spectrum of three levels of threat.

At the low end are attackers who simply want to send e-mail that a recipient does not want to receive. The attacker can use one of a number of commercially available tools that allow the sender to falsify the origin address of messages. This makes it difficult for the receiver to filter spam on the basis of originating address or domain.

1. At the next level are professional senders of bulk spam mail. These attackers often operate as commercial enterprises and send messages on behalf of third parties. They employ more comprehensive tools for attack, including Mail Transfer Agents (MTAs) and registered domains and networks of compromisedcomputers (zombies) to send messages and (in some cases) to harvest addresses to which to send.
2. The most sophisticated and financially motivated senders of messages are those who stand to receive substantial financial benefit, such as from an e-mail-based fraud scheme. These attackers can be expected to employ all of the above mechanisms and additionally may attack the Internet infrastructure itself, including DNS cache-poisoning attacks and IP routing attacks.

CAPABILITIES RFC 4686 lists the following as capabilities that an attacker might have.

1. Submit messages to MTAs and Message Submission Agents (MSAs) at multiple locations in the Internet.
2. Construct arbitrary Message Header fields, including those claiming to be mailing lists, resenders, and other mail agents.
3. Sign messages on behalf of domains under their control.
4. Generate substantial numbers of either unsigned or apparently signed messages that might be used to attempt a denial-of-service attack.
5. Resend messages that may have been previously signed by the domain.
6. Transmit messages using any envelope information desired.
7. Act as an authorized submitter for messages from a compromised computer.
8. Manipulation of IP routing. This could be used to submit messages from specific IP addresses or difficu lt-to-trace addresses, or to cause diversion of messages to a specific domain.
9. Limited influence over portions of DNS using mechanisms such as cache poisoning. This might be used to influence message routing or to falsify advertisements of DNSbased keys or signing practic es.
10. Access to significant computing resources, for example, through the conscription of worm-infected "zombie" computers. This could allow the "bad actor" to perform various types of brute-force attacks.

11.Ability to eavesdrop on existing traffic, perhaps from a wireless network.

LOCATION DKIM focuses primarily on attackers located outside of the administrative units of the claimed originator and the recipient. These administrative units frequently correspond to the protected portions of the network adjacent to the originator and recipient. It is in this area that the trust relationships required for authenticated message submission do not exist and do not scale adequately to be practical. Conversely, within these administrative units, there are other mechanisms (such as authenticated message submission) that are easier to deploy and more likely to be used than DKIM. External "bad actors" are usually attempting to exploit the "any-to-any" nature of e-mail that motivates most recipient MTAs to accept messages from anywhere for delivery to their local domain. They may generate messages without signatures, with incorrect signatures, or with correct signatures from domains with little traceability. They may also pose as mailing lists, greeting cards, or other agents that legitimately send or resend messages on behalf of others.

DKIM Strategy

DKIM is designed to provide an e-mail authentication technique that is transparent to the end user. In essence, a user's e-mail message is signed by a private key of the administrative domain from which the e-mail originates. The signature covers all of the content of the message and some of the RFC 5322 message headers. At the receiving end, the MDA can access the corresponding public key via a DNS and ver- ify the signature, thus authenticating that the message comes from the claimed administrative domain. Thus, mail that originates from somewhere else but claims to come from a given domain will not pass the authentication test and can be rejected.
This approach differs from that of S/MIME and PGP, which use the originator's pri- vate key to sign the content of the message. The motivation for DKIM is based on the following reasoning.5

1.S/MIME depends on both the sending and receiving users employing S/MIME. For almost all users, the bulk of incoming mail does not use S/MIME, and the bulk of the mail the user wants to send is to recipients not using S/MIME.

2.S/MIME signs only the message content. Thus, RFC 5322 header information concerning origin can be compromised.

3.DKIM is not implemented in client programs (MUAs) and is therefore transparent to the user; the user need take no action.

4.DKIM applies to all mail from cooperating domains.

5.DKIM allows good senders to prove that they did send a particular message and to prevent forgers from masquerading as good senders.

Figure 18.10 is a simple example of the operation of DKIM. We begin with a message generated by a user and transmitted into the MHS to an MSA that is within the users administrative domain. An e-mail message is generated by an e-mail client program. The content of the message, plus selected RFC 5322 headers, is signed by the e-mail provider using the provider's private key. The signer is associated with a domain, which could be a corporate local network, an ISP, or a public e-mail facility such as gmail. The signed message then passes through the Internet via a sequence of MTAs. At the destination, the MDA retrieves the public key for the incoming signature and verifies the signature before passing the message on to the destination e-mail client. The default signing algorithm is RSA with SHA-256. RSA with SHA-1 also may be used.

DKIM Functional Flow

Figure 18.11 provides a more detailed look at the elements of DKIM operation. Basic message processing is divided between a signing Administrative Management Domain (ADM D) and a verifying ADMD. At its simplest, this is between the

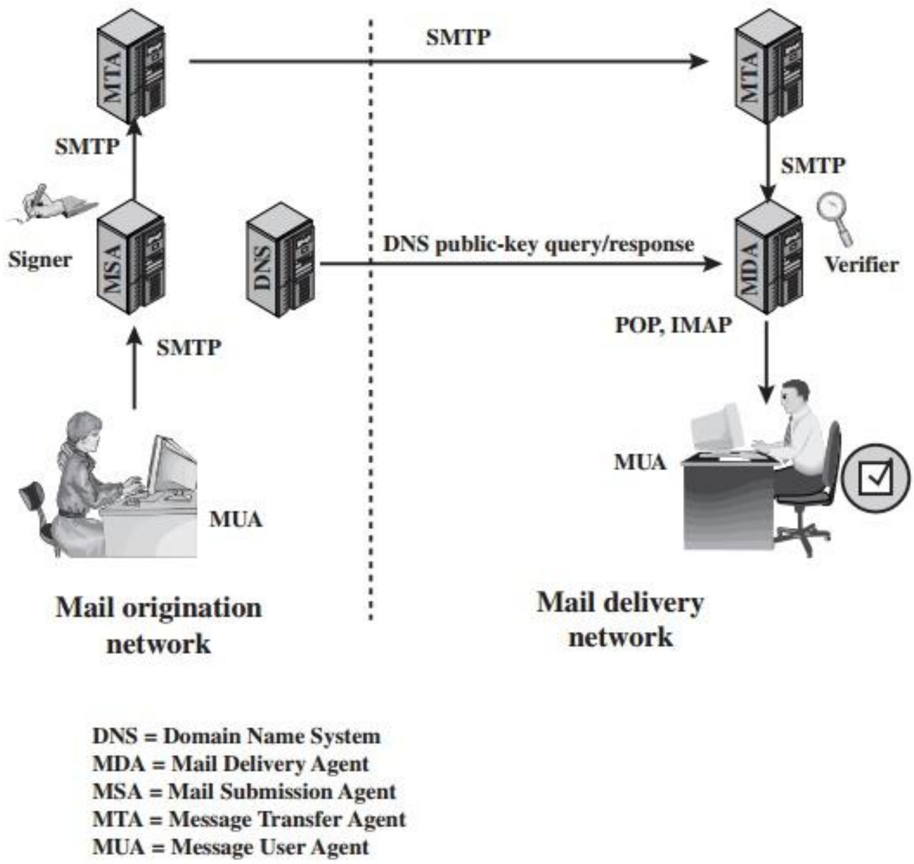


Figure 18.10 Simple Example of DKIM Deployment

originating ADMD and the delivering ADMD, but it can involve other ADMDs in the handling path.

Signing is performed by an authorized module within the signing ADMD and uses private information from a Key Store. Within the originating ADMD, this might be performed by the MUA, MSA, or an MTA. Verifying is performed by anauthorized module within the verifying ADMD. Within a delivering ADMD, verifying might be performed by an MTA, MDA, or MUA. The module verifies the signature or determines whether a particular signature was required. Verifying the signa- ture uses public information from the Key Store. If the signature passes, reputation information is used to assess the signer and that information is passed to the mes- sage filtering system. If the signature fails or there is no signature using the author's domain, information about signing practices related to the author can be retrieved remotely and/or locally, and that information is passed to the message filtering system. For example, if the sender (e.g., gmail) uses DKIM but no DKIM signature is present, then the message may be considered fraudulent.

The signature is inserted into the RFC 5322 message as an additional header entry, starting with the keyword Dkim-Signature. You can view examples from your own incoming mail by using the View Long Headers (or similar wording) option for an incoming message. Here is an example:

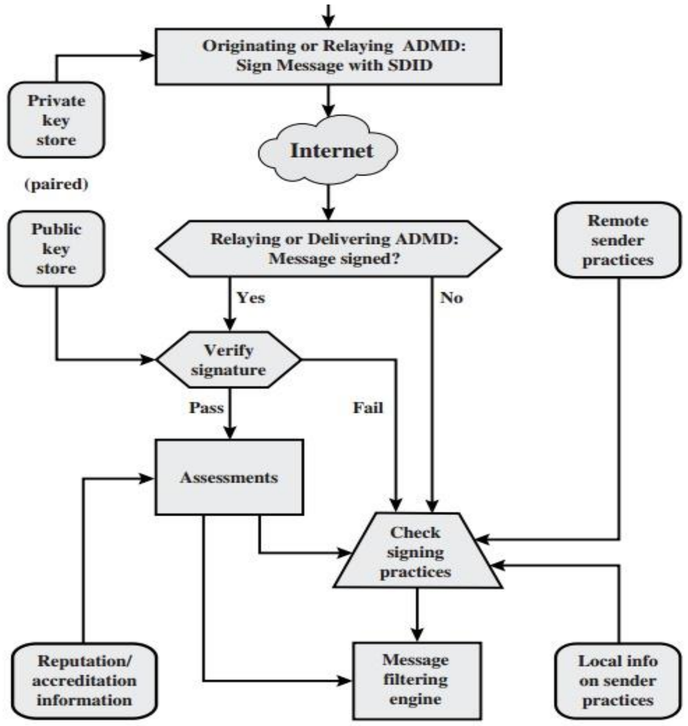


Figure 18.11 DKIM Functional Flow

```
Dkim-Signature:    v=1;  a=rsa-sha256;  c=relaxed/relaxed;
                   d=gmail.com; s=gamma; h=domainkey-signa-
                   ture:mime-version:received:date:message-
                   id:subject   :from:to:content-type:con-
                   tent-transfer-encoding;
                   bh=5mZvQDyCRuyLb1Y28K4zgS2MPOemFToDBgvbJ
                   7GO90s=;
                   b=PcUvPSDygb4ya5Dyj1rbZGp/VyRiScuaz7TTG
                   J5qW5slM+klzv6kcfYdGDHzEVJW+Z
                   FetuPfF1ETOVhELtwH0zjSccOyPkEiblOf6gILO
                   bm3DDRm3Ys1/FVrbhVOlA+/jH9Aei
                   uIIw/5iFnRbSH6qPDVv/beDQqAWQfA/wF7O5k=
```

Before a message is signed, a process known as canonicalization is per- formed on both the header and body of the RFC 5322 message. Canonicalization is necessary to deal with the possibility of minor changes in the message made en route, including character encoding, treatment of trailing white space in message lines, and the “folding” and “unfolding” of header lines. The intent of canonicalization is to make a minimal transformation of the message (for the purpose of signing; the message itself is not changed, so the canonicalization must be per- formed again by the verifier) that will give it its best chance of producing the same canonical value at the receiving end. DKIM defines two header canonicalization algorithms ("simple" and "relaxed") and two for the body (with the same names). The simple algorithm tolerates almost no modification, while the relaxed tolerates common modifications.

The signature includes a number of fields. Each field begins with a tag consisting of a tag code followed by an equals sign and ends with a semicolon. The fields include the following:

- **v =** DKIM version.
- **a =** Algorithm used to generate the signature; must be either rsa-sha1 or rsa-sha256.
- **c =** Canonicalization method used on the header and the body.
- **d =** A domain name used as an identifier to refer to the identity of a responsible person or organization. In DKIM, this identifier is called the Signing Domain IDentifier (SDID). In our example, this field indicates that the sender is using a gmail address.
- **s =** In order that different keys may be used in different circumstances for the same signing domain (allowing expiration of old keys, separate departmental signing, or the like), DKIM defines a selector (a name associated with a key), which is used by the verifier to retrieve the proper key during signature verification.
- **h =** Signed Header fields. A colon-separated list of header field names that identify the header fields presented to the signing algorithm. Note that in our example above, the signature covers the domainkey-signature field. This refers to an older algorithm (since replaced by DKIM) that is still in use.
- **bh =** The hash of the canonicalized body part of the message. This provides additional information for diagnosing signature verification failures.
- **b =** The signature data in base64 format; this is the encrypted hash code.

IP SECURITY

There are application-specific security mechanisms for a number of application areas, including electronic mail (S/MIME, PGP), client/server (Kerberos), Web access (Secure Sockets Layer), and others. However, users have security concerns that cut across protocol layers. For example, an enterprise can run a secure, private IP network by disallowing links to untrusted sites, encrypting packets that leave the premises, and authenticating packets that enter the premises. By implementing security at the IP level, an organization can ensure secure networking not only for applications that have security mechanisms but also for the many security-ignorant applications.

IPlevel security encompasses three functional areas: authentication, confidentiality, and key managem ent. The authentication mechanism assures that a received packet was, in fact, transmitted by the party identified as the source in the packet header. In addition, this mechanism assures that the packet has not been altered in transit. The confidentiality facility enables communicating nodes to encrypt Messages to prevent eavesdropping by third parties. The key management facility is concernedwith the secure exchange of keys. We begin this chapter with an overview of IP security (IPsec) and an introduction to the IPsec architecture. We then look at each of the three functional areas in detail. Appendix L reviews Internet protocols.

IP SECURITY OVERVIEW

In 1994, the Internet Architecture Board (IAB) issued a report titled "Security in the Internet Architecture" (RFC 1636). The report identified key areas for security mechanisms. Among these were the need to secure the network infrastructure from unauthorized monitoring and control of network traffic and the need to secure end-user-to-end-user traffic using authentication and encryption mechanisms.

To provide security, the IAB included authentication and encryption as necessary security features in the next-generation IP, which has been issued as IPv6. Fortunately, these security capabilities were designed to be usable both with the current IPv4 and the future IPv6. This means that vendors can begin offering these features now, and many vendors now do have some IPsec capability in their products. The IPsec specification now exists as a set of Internet standards.

Applications of IPsec

IPsec provides the capability to secure communications across a LAN, across private and public WANs, and across the Internet. Examples of its use include:

- **Secure branch office connectivity over the Internet:** A company can build a secure virtual private network over the Internet or over a public WAN. This enables a business to rely heavily on the Internet and reduce its need for private networks, saving costs and network manage ment overhead.
- **Secure remote access over the Internet:** An end user whose system is equipped with IP security protocols can make a local call to an Internet Service Provider (ISP) and gain secure access to a company network. This reduces the cost of toll charges for traveling employees and telecommuters.
- **Establishing extranet and intranet connectivity with partners:** IPsec can be used to secure communication with other organizations, ensuring authentication and confidentiality and providing a key exchange mechanism.
- **Enhancing electronic commerce security:** Even though some Web and electronic commerce applicati ons have built-in security protocols, the use of IPsec enhancesthat security. IPsec guarantees that all traffic designated by the network administrator is both encrypted and authenticated, adding an additional layer of security to whatever is provided at the application layer.

The principal feature of IPsec that enables it to support these varied applica- tions is that it can encrypt and/or authenticate *all* traffic at the IP level. Thus, all dis- tributed applications (including remote logon, client/server, email, file transfer, Web access, and so on) can be secured.

Figure 19.1 is a typical scenario of IPsec usage. An organization maintains LANs at dispersed locations. Nonsecure IP traffic is conducted on each LAN. For traffic offsite, through some sort of private or public WAN, IPsec protocols are used.
These protocols operate in networking devices, such as a router or firewall, that con- nect each LAN to the outside world. The IPsec networking device will typically encrypt and compress all traffic going into the WAN and decrypt and decompress traffic coming from the WAN; these operations are transparent to workstations and servers on the LAN. Secure transmission is also possible with individual users who dial into the WAN. Such user workstations must implement the IPsec protocols to provide security.

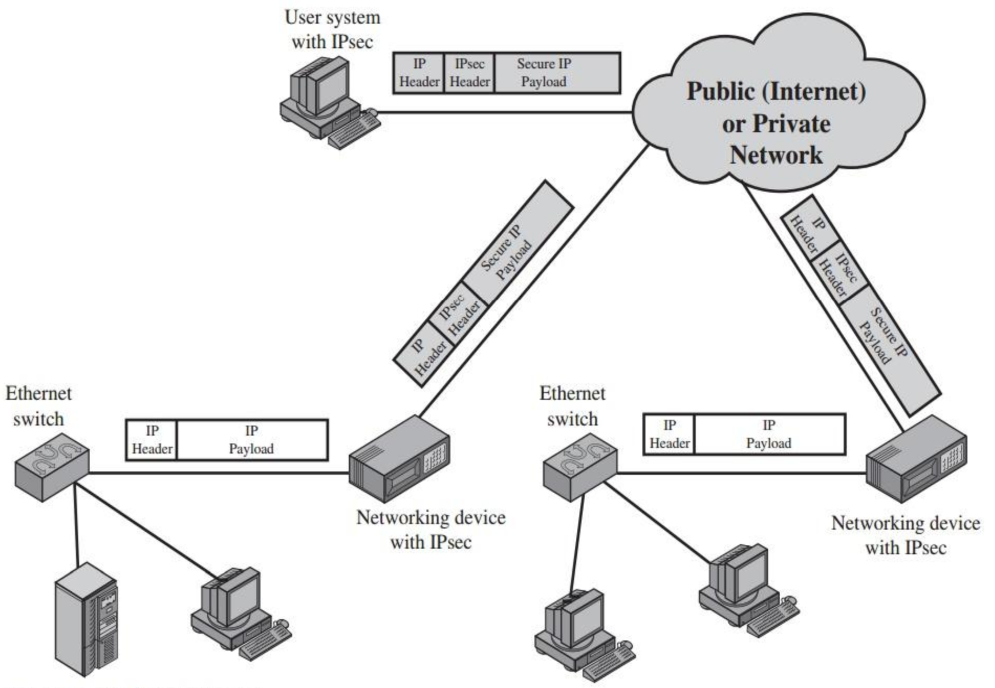


Figure 19.1 An IP Security Scenario

Benefits of IPsec

Some of the benefits of IPsec:

•When IPsec is implemented in a firewall or router, it provides strong security that can be applied to all traffic crossing the perimeter. Traffic within a companyor workgroup does not incur the overhead of security-related processing.

• IPsec in a firewall is resistant to bypass if all traffic from the outside must use IP and the firewall is the only means of entrance from the Internet into the organization.

• IPsec is below the transport layer (TCP, UDP) and sois transparent to applications. There is no need to change software on a user or server system when IPsec is implemented in the firewall or router. Even if IPsec is implemented in end systems, upper-layer software, including applications, is not affected.

• IPsec can be transparent to end users. There is no need to train users on security mechanisms, issue ke ying material on a per-user basis, or revoke keying material when users leave the organization.

• IPsec can provide security for individual users if needed. This is useful for offsite workers and for setting up a secure virtual subnetwork within an organization for sensitive applications.

Routing Applications

In addition to supporting end users and protecting premises systems and networks, IPsec can play a vital role in the routing architecture required for internetworking. [HUIT98] lists the following examples of the use of IPsec. IPsec can assure that

• A router advertisement (a new router advertises its presence) comes from an authorized router.

• A neighbor advertisement (a router seeks to establish or maintain a neighbor relationship with a router in another routing domain) comes from an authorized router.

• A redirect message comes from the router to which the initial IP packet was sent.

• A routing update is not forged.

Without such security measures, an opponent can disrupt communications or divert some traffic. Routing protocols such as Open Shortest Path First (OSPF) should be run on top of security associations between routers that are defined by IPsec.

IPsec Documents

IPsec encompasses three functional areas: authentication, confidentiality, and key management. The totality of the IPsec specification is scattered across dozens of RFCs and draft IETF documents, making this the most complex and difficult to grasp of all IETF specifications. The best way to grasp the scope of IPsec is to consult the latest version of the IPsec document roadmap, which as of this writing is [FRAN09]. The documents can be categorized into the f ollowing groups.

• **Architecture:** Covers the general concepts, security requirements, definitions, and mechanisms defining IPsec technology.

• **Authentication Header (AH):** AH is an extension header to provide message authentication. The c urrent specification is RFC 4302, *IP Authentication Header*. Because message authentication is provided by ESP, the use of AH is deprecated. It is included in IPsecv3 for backward compatibility but should not be used in new applications. We do not discuss AH in this chapter.

• **Encapsulating Security Payload (ESP):** ESP consists of an encapsulating header and trailer used to provide encryption or combined encryption/authentication. The current specification is RFC 4303, *IP Encapsulating Security Payload (ESP)*.

• **Internet Key Exchange (IKE):** This is a collection of documents describing the key management schemes for use with IPsec. The main specification is RFC 4306, *Internet Key Exchange (IKEv2) Protocol*, but there are a number of related RFCs.

- **Cryptographic algorithms:** This category encompasses a large set of documents that define and describe cryptographic algorithms for encryption, message authentication, pseudorandom functions (PRFs), and cryptographic key exchange.
- **Other:** There are a variety of other IPsec-related RFCs, including those dealing with security policy and management information base (MIB) content.

IPsec Services

IPsec provides security services at the IP layer by enabling a system to select required security protocols, determine the algorithm(s) to use for the service(s), and put in place any cryptographic keys required to provide the requested services. Two protocols are used to provide security: an authen tication protocol designated by the header of the protocol, Authentication Header (AH); and a combined encryption/ authentication protocol designated by the format of the packet for that protocol, Encapsulating Security Payload (ESP). RFC 4301 lists the following services:

- Access control
- Connectionless integrity
- Data origin authentication
- Rejection of replayed packets (a form of partial sequence integrity)
- Confidentiality (encryption)
- Limited traffic flow confidentiality

Transport and Tunnel Modes

Both AH and ESP support two modes of use: transport and tunnel mode. The operation of these two modes is best understood in the context of a description of ESP, which is covered in Section 19.3. Here we provide a brief overview.

TRANSPORT MODE Transport mode provides protection primarily for upper-layer protocols. That is, transport mode protection extends to the payload of an IP packet.1 Examples include a TCP or UDP segment or an ICMP packet, all of which operate directly above IP in a host protocol stack. Typically, transport mode is used for end-to-end communication between two hosts (e.g., a client and a server, or two workstations). When a host runs AH or ESP over IPv4, the payload is the data that normally follow the IP header. For IPv6, the payload is the data that normally follow both the IP header and any IPv6 extensions headers that are present, with the possible exception of the destination options header, which may be included in the protection.

ESP in transport mode encrypts and optionally authenticates the IP payload but not the IP header. AH in transport mode authenticates the IP payload and selected portions of the IP header.

TUNNEL MODE Tunnel mode provides protection to the entire IP packet. To achieve this, after the AH or ESP fields are added to the IP packet, the entire packet plus security fields is treated as the payload of new outer IP packet with a new outer IP header. The entire original, inner, packet travels through a tunnel from one point of an IP network to another; no routers along the way are able to examine the inner IP header. Because the original packet is encapsulated, the new, larger packet may have totally different source and destination addresses, adding to the security. Tunnel mode is used when one or both ends of a security association (SA) are a security gateway, such as a firewall or router that implements IPsec. With tunnel mode, a number of hosts on networks behind firewalls may engage in secure communications without implementing IPsec. The unprotected packets generated by such hosts are tunnelled through external networks by tunnel mode SAs set up by the IPsec software in the firewall or secure router at the boundary of the local network. Here is an example of how tunnel mode IPsec operates. Host A on a network generates an IP packet with the destination address of host B on another network. This packet is routed from the originating host to a firewall or secure router at the boundary of A's network. The firewall filters all outgoing packets to determine the need for IPsec processing. If this packet from A to B requires IPsec, the firewall performs IPsec processing and encapsulates the packet with an outer IP header. The source IP address of this outer IP packet is this firewall, and the destination address may be a firewall that forms the boundary to B's local network. This packet is now routed to B's firewall, with intermediate routers examining only the outer IP header. At B's firewall, the outer IP header is stripped off, and the inner packet is delivered to B.

ESP in tunnel mode encrypts and optionally authenticates the entire inner IP packet, including the inner IP header. AH in tunnel mode authenticates the entire inner IP packet and selected portions of the outer IP header.

Table 19.1 summarizes transport and tunnel mode functionality.

Table 19.1 Tunnel Mode and Transport Mode Functionality

	Transport Mode SA	Tunnel Mode SA
AH	Authenticates IP payload and selected portions of IP header and IPv6 extension headers.	Authenticates entire inner IP packet (inner header plus IP payload) plus selected portions of outer IP header and outer IPv6 extension headers.
ESP	Encrypts IP payload and any IPv6 extension headers following the ESP header.	Encrypts entire inner IP packet.
ESP with Authentication	Encrypts IP payload and any IPv6 extension headers following the ESP header. Authenticates IP payload but not IP header.	Encrypts entire inner IP packet. Authenticates inner IP packet.

IP SECURITY POLICY

Fundamental to the operation of IPsec is the concept of a security policy applied to each IP packet that transits from a source to a destination. IPsec policy is determined primarily by the interaction of two databases, the **security association database (SAD)** and the **security policy database (SPD)**. This section provides an overview of these two databases and then summarizes their use during IPsec operation. Figure 19.2 illustrates the relevant relationships.

Security Associations

A key concept that appears in both the authentication and confidentiality mechanisms for IP is the security association (SA). An association is a one-way logical con- nection between a sender and a receiver that affords security services to the traffic carried on it. If a peer relationship is needed for two-way secure exchange, then two security associations are required. Security services are afforded to an SA for the use of AH or ESP, but not both.

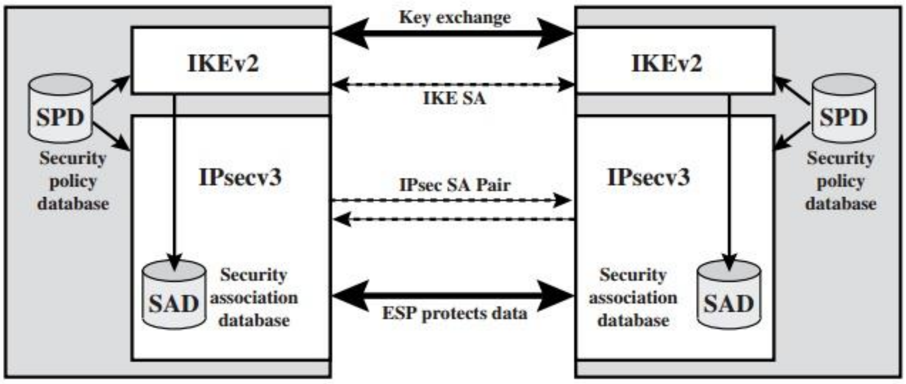


Figure 19.2 IPsec Architecture

A security association is uniquely identified by three parameters.

•**Security Parameters Index (SPI):** A bit string assigned to this SA and having local significance only. The SPI is carried in AH and ESP headers to enable the receiving system to select the SA under which a received packet will be processed.

•**IP Destination Address:** This is the address of the destination endpoint of the SA, which may be an end-user system or a network system such as a firewall or router.

• **Security Protocol Identifier:** This field from the outer IP header indicates whether the association is an AH or ESP security association.Hence, in any IP packet, the security association is uniquely identified by the Destination Address in the IPv4 or IPv6 header and the SPI in the enclosed extension header (AH or ESP).

Security Association Database

In each IPsec implementation, there is a nominal2 Security Association Database that defines the parameters associated with each SA. A security association is normally defined by the following parameters in an SAD entry.

• **Security Parameter Index:** A 32-bit value selected by the receiving end of an SA to uniquely identify the SA. In an SAD entry for an outbound SA, the SPI is used to construct the packet's AH or ESP header. In an SAD entry for an inbound SA, the SPI is used to map traffic to the appropriate SA.

• **Sequence Number Counter:** A 32-bit value used to generate the Sequence Number field in AH or ESP headers, described in Section 19.3 (required for all implementations).

• **Sequence Counter Overflow:** A flag indicating whether overflow of the Sequence Number Counter should generate an auditable event and prevent further transmission of packets on this SA (required for all implementations).

• **Anti-Replay Window:** Used to determine whether an inbound AH or ESP packet is a replay, described in Section 19.3 (required for all implementations).

• **AH Information:** Authentication algorithm, keys, key lifetimes, and related parameters being used with AH (required for AH implementations).

• **ESP Information:** Encryption and authentication algorithm, keys, initialization values, key lifetimes, and related parameters being used with ESP (required for ESP implementations).

• **Lifetime of this Security Association:** A time interval or byte count after which an SA must be replaced with a new SA (and new SPI) or terminated, plus an indication of which of these actions should occur (required for all implementations).

• **IPsec Protocol Mode:** Tunnel, transport, or wildcard.

• **Path MTU:** Any observed path maximum transmission unit (maximum size of a packet that can be transmittedwithoutfragmentation) and aging variables (required for all implementation s).The key management mechanism that is used to distribute keys is coupled to the authentication and privacy mechanisms only by way of the Security ParametersIndex (SPI). Hence, authentication and privacy have been specified independent of any specific key management mechanism.IPsec provides the user with considerable flexibility in the way in which IPsec services are applied to IP traffic. As we will see later, SAs can be combined in a number of ways to yield the desired user configuration. Furthermore, IPsec provides a high degree of granularity in discriminating between traffic that is afforded IPsec protection and traffic that is allowed to bypass IPsec, as in the former case relating IP traffic to specific SAs.

Security Policy Database

The means by which IP traffic is related to specific SAs (or no SA in the case of traffic allowed to bypass IPsec) is the nominal Security Policy Database (SPD). In its simplest form, an SPD contains entries, each of which defines a subset of IP traffic and points to an SA for that traffic. In more complex enviro nments, there may be multiple entries that potentially relate to a single SA or multiple SAs associated with a single SPD entry. The reader is referred to the relevant IPsec documents for a full discussion. Each SPD entry is defined by a set of IP and upperlayer protocol field values, called *selectors*. In effect, these selectors are used to filter outgoing traffic in order to map it into a particular SA. Outb ound processing obeys the following general sequence for each IP packet.

1.Compare the values of the appropriate fields in the packet (the selector fields) against the SPD to find a matching SPD entry, which will point to zero or more SAs.

2.Determine the SA if any for this packet and its associated SPI.

3.Do the required IPsec processing (i.e., AH or ESP processing). The following selectors determine a n SPD entry:

• **Remote IP Address:** This may be a single IP address, an enumerated list or range of addresses, or a wild card (mask) address. The latter two are required to support more than one destination system sharing the same SA (e.g., behind a firewall).

•**Local IP Address:** This may be a single IP address, an enumerated list or range of addresses, or a wildcard (mask) address. The latter two are required to support more than one source system sharing the same SA (e.g., behind a firewall).

•**Next Layer Protocol:** The IP protocol header (IPv4, IPv6, or IPv6 Extension) includes a field (Protocol for IPv4, Next Header for IPv6 or IPv6 Extension) that designates the protocol operating over IP. This is an individual protocol number, ANY, or for IPv6 only, OPAQUE. If AH or ESP is used, then this IP protocol header immediately precedes the AH or ESP header in the packet.

Table 19.2 Host SPD Example

Protocol	Local IP	Port	Remote IP	Port	Action	Comment
UDP	1.2.3.101	500	*	500	BYPASS	IKE
ICMP	1.2.3.101	*	*	*	BYPASS	Error messages
*	1.2.3.101	*	1.2.3.0/24	*	PROTECT: ESP intransport-mode	Encrypt intranet traffic
TCP	1.2.3.101	*	1.2.4.10	80	PROTECT: ESP intransport-mode	Encrypt to server
TCP	1.2.3.101	*	1.2.4.10	443	BYPASS	TLS: avoid double encryption
*	1.2.3.101	*	1.2.4.0/24	*	DISCARD	Others in DMZ
*	1.2.3.101	*	*	*	BYPASS	Internet

• **Name:** A user identifier from the operating system. This is not a field in the IP or upper-layer headers but is available if IPsec is running on the same operat- ing system as the user.

• **Local and Remote Ports:** These may be individual TCP or UDP port values, an enumerated list of ports, or a wildcard port.

Table 19.2 provides an example of an SPD on a host system (as opposed to a network system such as a firewall or router). This table reflects the following config- uration: A local network configuration consists of two networks. The basic corporate network configuration has the IP network number 1.2.3.0/24. The local configuration also includes a secure LAN, often known as a DMZ, that is identified as 1.2.4.0/24. The DMZ is protected from both the outside world and the rest of the corporate LAN by firewalls. The host in this example has the IP address 1.2.3.10, and it is authorized to connect to the server1.2.4.10 in the DMZ.

The entries in the SPD should be self-explanatory. For example, UDP port 500 is the designated port for IKE. Any traffic from the local host to a remote host for purposes of an IKE exchange bypasses the IPsec processing.

IP Traffic Processing

IPsec is executed on a packet-by-packet basis. When IPsec is implemented, each outbound IP packet is processed by the IPsec logic before transmission, and each inbound packet is processed by the IPsec logic after reception and before passing the packet contents on to the next higher layer (e.g., TCP or UDP). We look at the logic of these two situations in turn.

OUTBOUND PACKETS Figure19.3 highlights the main elements of IPsec processing for outbound traffic. A block of data from a higher layer, such as TCP, is passed down to the IP layer and an IP packet is formed, consisting of an IP header and an IP body. Then the following steps occur:

1. IPsec searches the SPD for a match to this packet.
2. If no match is found, then the packet is discarded and an error message is generated.

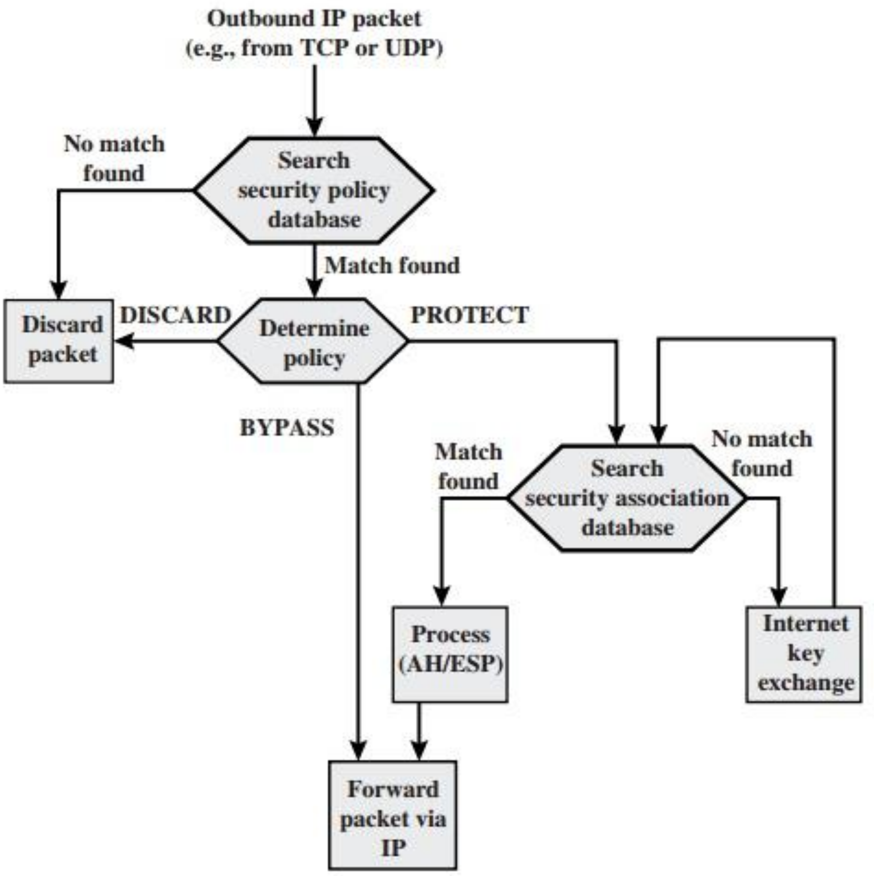


Figure 19.3 Processing Model for Outbound Packets

1. If a match is found, further processing is determined by the first matching entry in the SPD. If th e policy for this packet is DISCARD, then the packet is discarded. If the policy is BYPASS, then

   there is no further IPsec processing; the packet is forwarded to the network for transmission.

2. If the policy is PROTECT, then a search is made of the SAD for a matching entry. If no entry is fou nd, then IKE is invoked to create an SA with the appropriate keys and an entry is made in the SA.

3. The matching entry in the SAD determines the processing for this packet. Either encryption, authentication, or both can be performed, and either transport or tunnel mode can be used. The packet is then forwarded to the network fortransmission.

INBOUND PACKETS Figure 19.4 highlights the main elements of IPsec processing for inbound traffic. An incoming IP packet triggers the IPsec processing. The following steps occur:

IPsec determines whether this is an unsecured IP packet or one that has ESP or AH headers/trailers, by examining the IP Protocol field (IPv4) or Next Header field (IPv6).

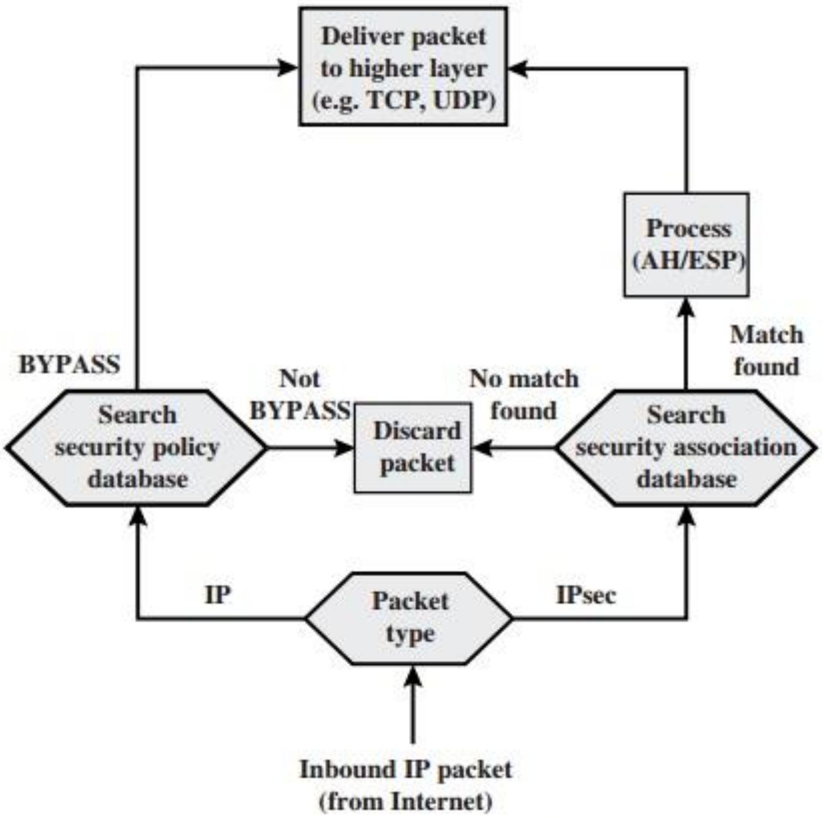


Figure 19.4 Processing Model for Inbound Packets

1. If the packet is unsecured, IPsec searches the SPD for a match to this packet. If the first matching entr y has a policy of BYPASS, the IP header is processed and stripped off and the packet body is delivered to the next higher layer, such as TCP. If the first matching entry has a policy of PROTECT or DISCARD, or if there is no matching entry, the packet is discarded.

2. For a secured packet, IPsec searches the SAD. If no match is found, the packet is discarded. Otherwise, IPsec applies the appropriate ESP or AH processing. Then, the IP header is processed and stripped off and the packet body is delivered to the next higher layer, such as TCP.

ENCAPSULATING SECURITY PAYLOAD

ESP can be used to provide confidentiality,

ESP FORMAT

Figure 19.5a shows the top-level format of an ESP packet. It contains the following fields.

•**Security Parameters Index (32 bits):** Identifies a security association.

•**Sequence Number (32 bits):** A monotonically increasing counter value; this provides an anti-replay function, as discussed for AH.

•**Payload Data (variable):** This is a transport-level segment (transport mode) or IP packet (tunnel mode) that is protected by encryption.

•**Padding (0 – 255 bytes):** The purpose of this field is discussed later.

•**Pad Length (8 bits):** Indicates the number of pad bytes immediately preceding this field.

•**Next Header (8 bits):** Identifies the type of data contained in the payload data field by identifying the first header in that payload (for example, an extension header in IPv6, or an upper-layer protocol such as TCP).

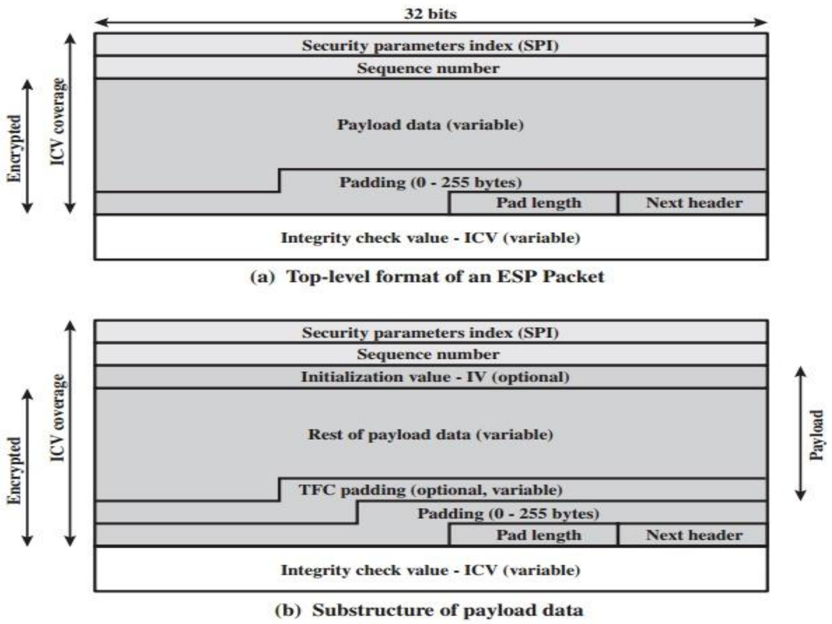


Figure 19.5 ESP Packet Format

•**Integrity Check Value (variable):** A variable-length field (must be an integral number of 32-bit words) that contains the Integrity Check Value computed over the ESP packet minus the Authentication Data field.When any combined mode algorithm is employed, the algorithm itself is expected to return both decrypted plaintext and a pass/fail indication for the integrity check. For combined mode algorithms, the ICV that would normally appear at the end of the ESP packet (when integrity is selected) may be omitted. When the ICV is omitted and integrity is selected, it is the responsibility of the combined mode algorithm to encode within the Payload Data an ICV-equivalent means of verifying the integrity of the packet.

Two additional fields may be present in the payload (Figure 19.5b). An **initialization value (IV)**, or nonce, is present if this is required by the encryption or authenticated encryption algorithm used for ESP. If tunnel mode is being used, then the IPsecimplementation may add

traffic flow confidentiality (TFC) padding afterthe Payload Data and before the Padding field, as ex plained subsequently.

Encryption and Authentication Algorithms

The Payload Data, Padding, Pad Length, and Next Header fields are encrypted by the ESP service. If the algorithm used to encrypt the payload requires crypto- graphic synchronization data, such as an initialization vector (IV), then these data may be carried explicitly at the beginning of the Payload Data field. If included, an IV is usually not encrypted, although it is often referred to as being part of the ciphertext.

The ICV field is optional. It is present only if the integrity service is selected and is provided by either a separate integrity algorithm or a combined mode algorithmthat uses an ICV. The ICV is computed after the encryption is performed. This order of processing facilitates rapid detection and rejection of replayed or bogus packets by the receiver prior to decrypting the packet, hence potentially reducing the impact of denial of service (DoS) attacks.
It also allows for the possibility of parallel processing of packets at the receiver, i.e., decryption can ta ke place in parallelwith integrity checking. Note that because the ICV is not protected by encryption, a keyed integrity algorithm must be employed to compute the ICV.

Padding

The Padding field serves several purposes:

• If an encryption algorithm requires the plaintext to be a multiple of some number of bytes (e.g., the multiple of a single block for a block cipher), the Padding field is used to expand the plaintext (consisting of the Payload Data, Padding, Pad Length, and Next Header fields) to the required length.

• The ESP format requires that the Pad Length and Next Header fields be right aligned within a 32-bit word. Equivalently, the ciphertext must be an integer multiple of 32 bits.

The Padding field is used to assure this alignment.

Additional padding may be added to provide partial trafficflow confidentiality by concealing the actu al length of the payload.

Anti-Replay Service

A **replay attack** is one in which an attacker obtains a copy of an authenticated packet and later transmits it to the intended destination. The receipt of duplicate, authenticated IP packets may disrupt service in some way or may have some other undesired consequence. The Sequence Number field is designed to thwart such attacks. First, we discuss sequence number generation by the sender, and then we look at how it is processed by the recipient.

When a new SA is established, the **sender** initializes a sequence number counter to 0. Each time that a packet is sent on this SA, the sender increments the counter and places the value in the

Sequence Number field. Thus, the first value to be used is 1. If anti-replay is enabled (the default), the sender must not allow thesequence number to cycle past 232 – 1 back to zero. Otherwise, there would be multiple valid packets with the same sequence number. If the limit of 232 – 1 is

reached, the sender should terminate this SA and negotiate a new SA with a new key. Because IP is a c onnectionless, unreliable service, the protocol does not guarantee that packets will be delivered in ord er and does not guarantee that all packets will be delivered. Therefore, the IPsec authentication document dictates that the **receiver** should implement a window of size *W*, with a default of

W = 64. The right edge of the window represents the highest sequence number, *N*, so far received for a valid packet. For any packet with a sequence number in the range from *N* - *W* + 1 to *N* that has been correctly received (i.e., properly authenticated), the corresponding slot in the window is marked (Figure 19.6). Inbound processing proceeds as follows when a packet is received:

1. If the received packet falls within the window and is new, the MAC is checked. If the packet is authenticated, the corresponding slot in the window is marked.
2. If the received packet is to the right of the window and is new, the MAC is checked. If the packet is authenticated, the window is advanced so that this sequence number is the right edge of the window, and the corresponding slot in the window is marked.
3. If the received packet is to the left of the window or if authentication fails, the packet is discarded; this is an auditable event.

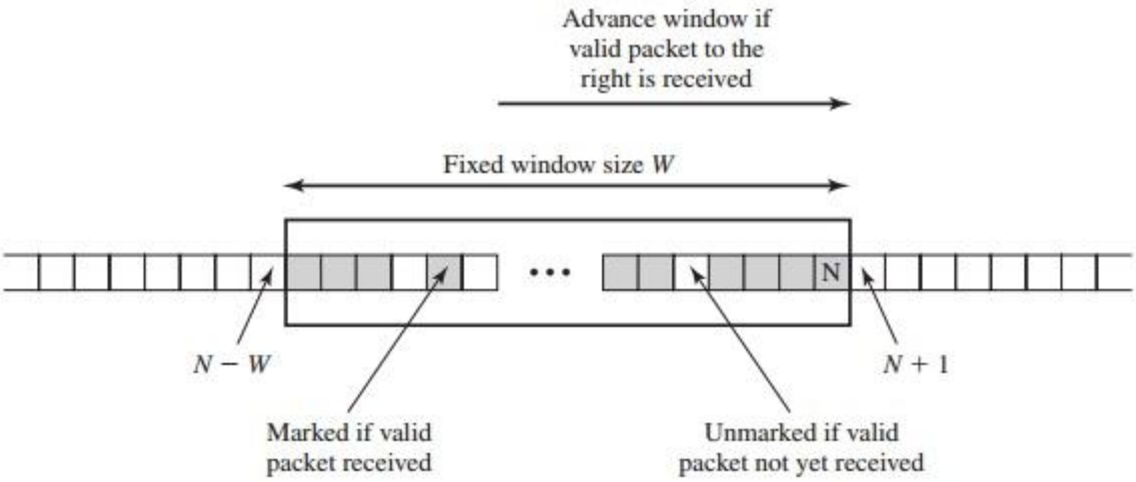


Figure 19.6 Anti-replay Mechanism

Transport and Tunnel Modes

Figure 19.7 shows two ways in which the IPsec ESP service can be used. In the upper part of the figure, encryption (and optionally authentication) is provided directly between two hosts. Figure 19.7b shows how tunnel mode operation can be used to set up a **virtual private network**. In this example, an organization has four private networks interconnected across the Internet. Hosts on the internal networks use the Internet for transport of data but do not interact with other Internet-based hosts. By terminating the tunnels at the security gateway to each internal network, the configuration allows the hosts to avoid implementing the security capability. The former technique is supported by a transport mode SA, while the latter technique uses a tunnel mode SA.

In this section, we look at the scope of ESP for the two modes. The considerations are somewhat different for IPv4 and IPv6. We use the packet formats of Figure 19.8a as a starting point.

<u>TRANSPORT MODE ESP</u> Transport mode ESP is used to encrypt and optionally authenticate the data carried by IP (e.g., a TCP segment), as shown in Figure 19.8b.

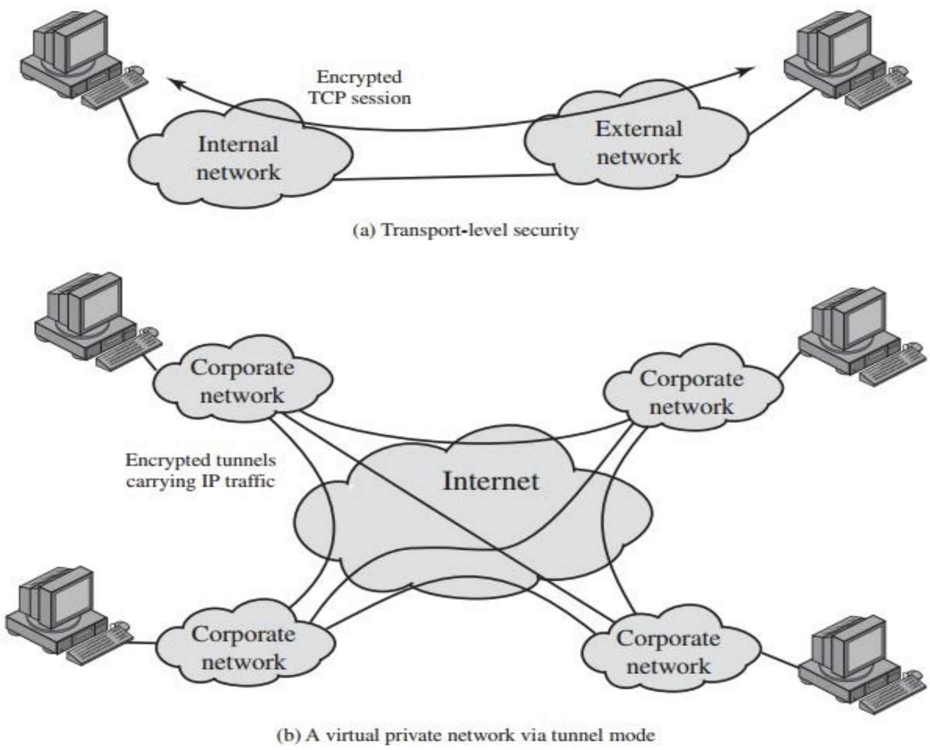
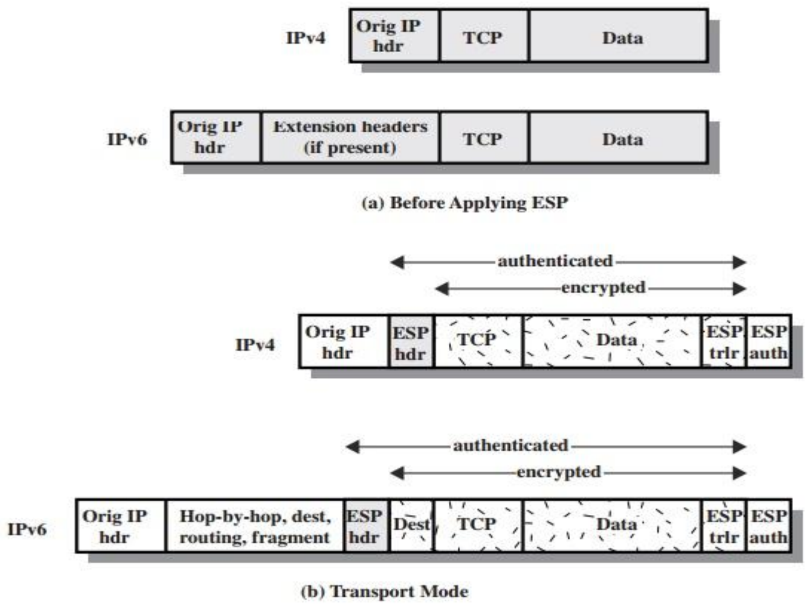
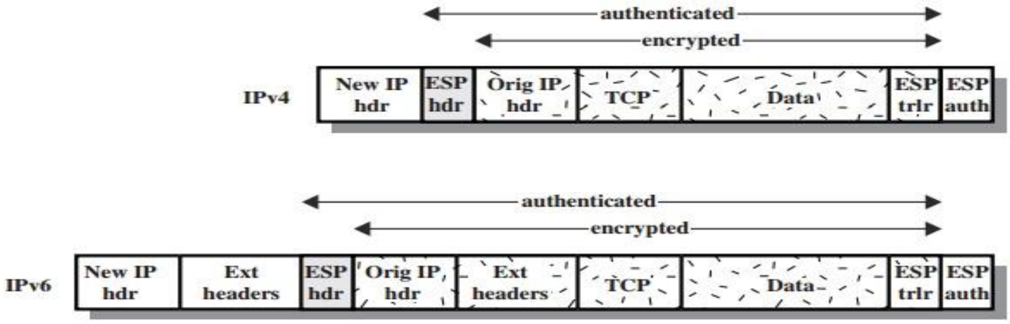


Figure 19.7 Transport-Mode versus Tunnel-Mode Encryption





(c) Tunnel Mode

Figure 19.8 Scope of ESP Encryption and Authentication

For this mode using IPv4, the ESP header is inserted into the IP packet immediately prior to the trans port-layer header (e.g., TCP, UDP, ICMP), and an ESP trailer (Padding, Pad Length, and Next Header fields) is placed after the IP packet. If authentication is selected, the ESP Authentication Data field is added after the ESP trailer. The entire transport-level segment plus the ESP trailer are encrypted. Authentication covers all of the ciphertext plus the ESP header.

In the context of IPv6, ESP is viewed as an end-to-end payload; that is, it is not examined or processed by intermediate routers. Therefore, the ESP header appears after the IPv6 base header and the hop-by-hop, routing, and fragment extension headers. The destination options extension header could appear before or after the ESP header, depending on the semantics desired. For IPv6, encryption covers the entire transport-level segment plus the ESP trailer plus the destination options extension header if it occurs after the ESP header. Again, authentication covers the ciphertext plus the ESP header.

Transport mode operation may be summarized as follows.

1. At the source, the block of data consisting of the ESP trailer plus the entire transport-layer segment is encrypted and the plaintext of this block is replaced with its ciphertext to form the IP packet for transmission. Authentication is added if this option is selected.
2. The packet is then routed to the destination. Each intermediate router needs to examine and process the IP header plus any plaintext IP extension headers but does not need to examine the ciphertext.
3. The destination node examines and processes the IP header plus any plaintext IP extension headers. Then, on the basis of the SPI in the ESP header, the destination node decrypts the remainder of the packet to recover the plaintext transport-layer segment.

Transport mode operation provides confidentiality for any application that uses it, thus avoiding the need to implement confidentiality in every individual application. One drawback to this mode is that it is possible to do traffic analysis on the transmitted packets.

TUNNEL MODE ESP Tunnel mode ESP is used to encrypt an entire IP packet (Figure 19.8c). For this mode, the ESP header is prefixed to the packet and then the packet plus the ESP trailer is encrypted. This method can be used to counter traffic analysis.

Because the IP header contains the destination address and possibly source routing directives and hop-by-hop option information, it is not possible simply to transmit the encrypted IP packet prefixed by the ESP header. Intermediate routers would be unable to process such a packet.

Therefore, it is necessary to encapsulate the entire block (ESP header plus ciphertext plus Authentication Data, if present) with a new IP header that will contain sufficient information for routing but not for traffic analysis. Whereas the transport mode is suitable for protecting connections between hosts that support the ESP feature, the tunnel mode is useful in a configuration that includes a firewall or other sort of security gateway that protects a trusted network from external networks. In this latter case, encryption occurs only between an external host and the security gateway or between two security gateways. This relieves hosts on the internal network of the processing burden of encryption and simplifies the key distribution task by reducing the number of needed keys. Further, it thwarts traffic analysis based on ultimate destination.

Consider a case in which an external host wishes to communicate with a host on an internal network protected by a firewall, and in which ESP is implemented in the external host and the firewalls. The following steps occur for transfer of a transportlayer segment from the external host to the internal host.

1. The source prepares an inner IP packet with a destination address of the target internal host. This packet is prefixed by an ESP header; then the packet and ESP trailer are encrypted and Authentication Data may be added. The resulting block is encapsulated with a new IP header (base header plus optional extensions such as routing and hop-by-hop options for IPv6) whose destination address is the firewall; this forms the outer IP packet.

2. The outer packet is routed to the destination firewall. Each intermediate router needs to examine and process the outer IP header plus any outer IP extension headers but does not need to examine the ciphertext.

3. The destination firewall examines and processes the outer IP header plus any outer IP extension headers. Then, on the basis of the SPI in the ESP header, the destination node decrypts the remainder of the packet to recover the plaintext inner IP packet. This packet is then transmitted in the internal network.

4. The inner packet is routed through zero or more routers in the internal network to the destination host.