

PART FOUR: Cryptographic Algorithms

CHAPTER

20

SYMMETRIC ENCRYPTION AND MESSAGE CONFIDENTIALITY

20.1 Symmetric Encryption Principles

- Cryptography
- Cryptanalysis
- Feistel Cipher Structure

20.2 Data Encryption Standard

- Data Encryption Standard
- Triple DES

20.3 Advanced Encryption Standard

- Overview of the Algorithm
- Algorithm Details

20.4 Stream Ciphers and RC4

- Stream Cipher Structure
- The RC4 Algorithm

20.5 Cipher Block Modes of Operation

- Electronic Codebook Mode
- Cipher Block Chaining Mode
- Cipher Feedback Mode
- Counter Mode

20.6 Key Distribution

20.7 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Explain the basic principles of symmetric encryption.
- ◆ Understand the significance of the Feistel cipher structure.
- ◆ Describe the structure and function of DES.
- ◆ Distinguish between two-key and three-key triple DES.
- ◆ Describe the structure and function of AES.
- ◆ Compare and contrast stream encryption and block cipher encryption.
- ◆ Distinguish among the major block cipher modes of operation.
- ◆ Discuss the issues involved in key distribution.

Symmetric encryption, also referred to as conventional encryption, secret-key, or single-key encryption, was the only type of encryption in use prior to the development of public-key encryption in the late 1970s.¹ It remains by far the most widely used of the two types of encryption.

This chapter begins with a look at a general model for the symmetric encryption process; this will enable us to understand the context within which the algorithms are used. Then, we look at three important block encryption algorithms: DES, triple DES, and AES. Next, the chapter introduces symmetric stream encryption and describes the widely used stream cipher RC4. We then examine the application of these algorithms to achieve confidentiality.

20.1 SYMMETRIC ENCRYPTION PRINCIPLES

At this point the reader should review Section 2.1. Recall that a symmetric encryption scheme has five ingredients (see Figure 2.1):

- **Plaintext:** This is the original 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 algorithm. 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.
- **Decryption algorithm:** This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the same secret key and produces the original plaintext.

¹Public-key encryption was first described in the open literature in 1976; the US National Security Agency (NSA) and the (then) UK CESG claim to have discovered it some years earlier.

Cryptography

Cryptographic systems are generically classified along three independent dimensions:

1. **The type of operations used for transforming plaintext to ciphertext.** All encryption algorithms are based on two general principles: substitution, in which each element in the plaintext (bit, letter, group of bits or letters) is mapped into another element, and transposition, in which elements in the plaintext are rearranged. The fundamental requirement is that no information be lost (i.e., that all operations be reversible). Most systems, referred to as product systems, involve multiple stages of substitutions and transpositions.
2. **The number of keys used.** If both sender and receiver use the same key, the system is referred to as symmetric, single-key, secret-key, or conventional encryption. If the sender and receiver each use a different key, 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

The process of attempting to discover the plaintext or key is known as **cryptanalysis**. The strategy used by the cryptanalyst depends on the nature of the encryption scheme and the information available to the cryptanalyst.

Table 20.1 summarizes the various types of cryptanalytic attacks, based on the amount of information known to the cryptanalyst. The most difficult problem is presented when all that is available is the *ciphertext only*. In some cases, not even the encryption algorithm is known, but in general, we can assume 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.

The ciphertext-only attack is the easiest to defend against because the opponent has the least amount of information with which to work. In many cases, however, the analyst has more information. The analyst may be able to capture one or more plaintext messages as well as their encryptions. Or the analyst may know that certain plaintext patterns will appear in a message. For example, a file that is encoded in the Postscript format always begins with the same pattern, or there may be a standardized header or banner to an electronic funds transfer message, and so on. All these are examples of *known plaintext*. With this knowledge, the analyst may be able to deduce the key on the basis of the way in which the known plaintext is transformed.

Closely related to the known-plaintext attack is what might be referred to as a probable-word attack. If the opponent is working with the encryption of some general prose message, he or she may have little knowledge of what is in the message.

Table 20.1 Types of Attacks on Encrypted Messages

Type of Attack	Known to Cryptanalyst
Ciphertext only	<ul style="list-style-type: none"> • Encryption algorithm • Ciphertext to be decoded
Known plaintext	<ul style="list-style-type: none"> • Encryption algorithm • Ciphertext to be decoded • One or more plaintext–ciphertext pairs formed with the secret key
Chosen plaintext	<ul style="list-style-type: none"> • Encryption algorithm • Ciphertext to be decoded • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key
Chosen ciphertext	<ul style="list-style-type: none"> • Encryption algorithm • Ciphertext to be decoded • Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key
Chosen text	<ul style="list-style-type: none"> • Encryption algorithm • Ciphertext to be decoded • Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key • Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key

However, if the opponent is after some very specific information, then parts of the message may be known. For example, if an entire accounting file is being transmitted, the opponent may know the placement of certain key words in the header of the file. As another example, the source code for a program developed by a corporation might include a copyright statement in some standardized position.

If the analyst is able somehow to get the source system to insert into the system a message chosen by the analyst, then a *chosen-plaintext* attack is possible. In general, if the analyst is able to choose the messages to encrypt, the analyst may deliberately pick patterns that can be expected to reveal the structure of the key.

Table 20.1 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.

An encryption scheme is **computationally secure** if the ciphertext generated by the scheme 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.

Unfortunately, it is very difficult to estimate the amount of effort required to cryptanalyze ciphertext successfully. However, assuming there are no inherent mathematical weaknesses in the algorithm, then a brute-force approach is indicated, and here we can make some reasonable estimates about costs and time.

A brute-force approach 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. This type of attack is discussed in Section 2.1.

Feistel Cipher Structure

Many symmetric block encryption algorithms, including DES, have a structure first described by Horst Feistel of IBM in 1973 [FEIS73] and shown in Figure 20.1. The inputs to the encryption algorithm are a plaintext block of length $2w$ bits and a key K . The plaintext block is divided into two halves, L_0 and R_0 . The two halves of the data pass through n rounds of processing and then combine to produce the ciphertext block. Each round i has as inputs L_{i-1} and R_{i-1} , derived from the previous round, as well as a subkey K_i , derived from the overall K . In general, the subkeys K_i are different from K and from each other, and are generated from the key by a subkey generation algorithm.

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 (XOR) of the output of that function and the left half of the data. The round function has the same general structure for each round but is parameterized by the round subkey K_i . Following this substitution, a permutation is performed that consists of the interchange of the two halves of the data.

The Feistel structure is a particular example of the more general structure used by all symmetric block ciphers. In general, a symmetric block cipher consists of a sequence of rounds, with each round performing substitutions and permutations conditioned by a secret key value. The exact realization of a symmetric block cipher 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. A block size of 128 bits is a reasonable tradeoff and is nearly universal among recent block cipher designs.
- **Key size:** Larger key size means greater security but may decrease encryption/decryption speed. The most common key length in modern algorithms is 128 bits.
- **Number of rounds:** The essence of a symmetric block cipher is that a single round offers inadequate security but that multiple rounds offer increasing security. A typical size is 16 rounds.
- **Subkey generation algorithm:** Greater complexity in this algorithm should lead to greater difficulty of cryptanalysis.
- **Round function:** Again, greater complexity generally means greater resistance to cryptanalysis.

There are two other considerations in the design of a symmetric block 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.

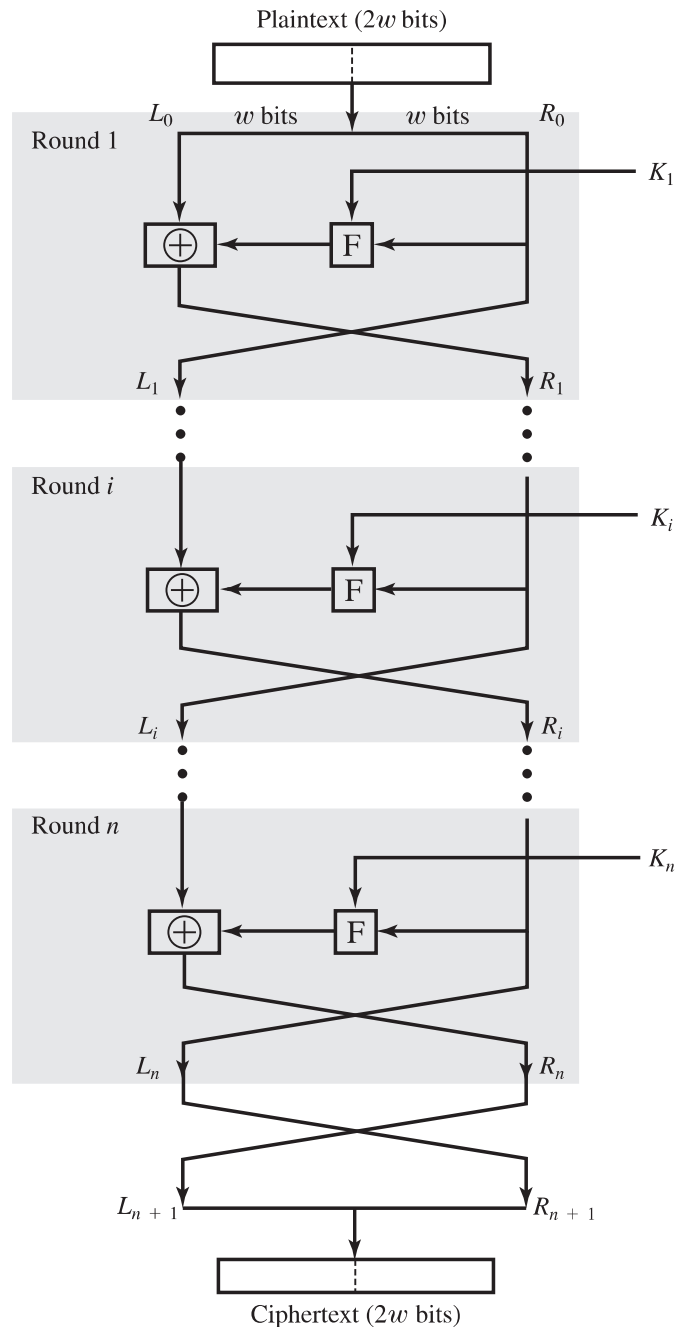


Figure 20.1 Classical Feistel Network

- **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.

Decryption with a symmetric block cipher is essentially the same as the encryption process. The rule is as follows: Use the ciphertext as input to the algorithm, but use the subkeys K_i in reverse order. That is, use K_n in the first round, K_{n-1} in

the second round, and so on until K_1 is used in the last round. This is a nice feature because it means we need not implement two different algorithms, one for encryption and one for decryption.

20.2 DATA ENCRYPTION STANDARD

The most commonly used symmetric encryption algorithms are block ciphers. A block cipher processes the plaintext input in fixed-size blocks and produces a block of ciphertext of equal size for each plaintext block. This section and the next focus on the three most important symmetric block ciphers: the Data Encryption Standard (DES), triple DES (3DES), and the Advanced Encryption Standard (AES).

Data Encryption Standard

The most widely used encryption scheme is based on the Data Encryption Standard (DES) adopted in 1977 by the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as FIPS 46 (*Data Encryption Standard*, January 1977). The algorithm itself is referred to as the Data Encryption Algorithm (DEA).²

The DES algorithm can be described as follows. The plaintext is 64 bits in length and the key is 56 bits in length; longer plaintext amounts are processed in 64-bit blocks. The DES structure is a minor variation of the Feistel network shown in Figure 20.1. There are 16 rounds of processing. From the original 56-bit key, 16 subkeys are generated, one of which is used for each round.

The process of decryption with DES is essentially the same as the encryption process. The rule is as follows: Use the ciphertext as input to the DES algorithm, but use the subkeys K_i in reverse order. That is, use K_{16} on the first iteration, K_{15} on the second iteration, and so on until K_1 is used on the sixteenth and last iteration.

Triple DES

Triple DES (3DES) was first standardized for use in financial applications in ANSI standard X9.17 in 1985. 3DES was incorporated as part of the Data Encryption Standard in 1999, with the publication of FIPS 46-3.

3DES uses three keys and three executions of the DES algorithm. The function follows an encrypt-decrypt-encrypt (EDE) sequence (see Figure 20.2a):

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

where

C = ciphertext

P = plaintext

²The terminology is a bit confusing. Until recently, the terms *DES* and *DEA* could be used interchangeably. However, the most recent edition of the DES document includes a specification of the DEA described here plus the triple DEA (3DES) described subsequently. Both DEA and 3DES are part of the Data Encryption Standard. Further, until the recent adoption of the official term *3DES*, the triple DEA algorithm was typically referred to as *triple DES* and written as 3DES. For the sake of convenience, we will use 3DES.

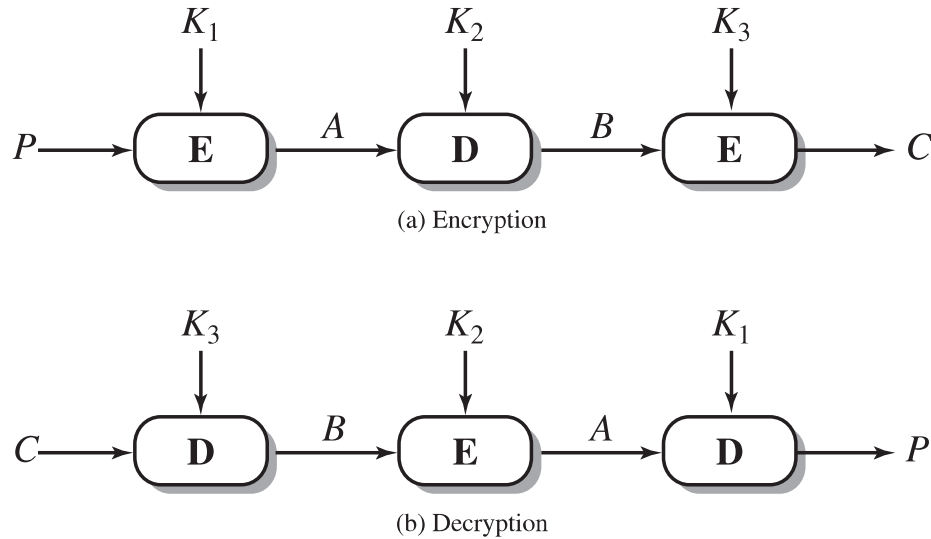


Figure 20.2 Triple DES

$E(K, X)$ = DES encryption of X using key K

$D(K, Y)$ = DES decryption of Y using key K

Decryption is simply the same operation with the keys reversed (see Figure 20.2b):

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

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

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

With three distinct keys, 3DES has an effective key length of 168 bits. FIPS 46-3 also allows for the use of two keys, with $K_1 = K_3$; this provides for a key length of 112 bits. FIPS 46-3 includes the following guidelines for 3DES:

- 3DES is the FIPS approved symmetric encryption algorithm of choice.
- The original DES, which uses a single 56-bit key, is permitted under the standard for legacy systems only. New procurements should support 3DES.
- Government organizations with legacy DES systems are encouraged to transition to 3DES.
- It is anticipated that 3DES and the Advanced Encryption Standard (AES) will coexist as FIPS-approved algorithms, allowing for a gradual transition to AES.

It is easy to see that 3DES is a formidable algorithm. Because the underlying cryptographic algorithm is DEA, 3DES can claim the same resistance to cryptanalysis based on the algorithm as is claimed for DEA. Further, with a 168-bit key length, brute-force attacks are effectively impossible.

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

20.3 ADVANCED ENCRYPTION STANDARD

The Advanced Encryption Standard (AES) was issued as a federal information processing standard FIPS 197 (*Advanced Encryption Standard*, November 2001). It is intended to replace DES and triple DES with an algorithm that is more secure and efficient.

Overview of the Algorithm

AES uses a block length of 128 bits and a key length that can be 128, 192, or 256 bits. In the description of this section, we assume a key length of 128 bits, which is likely to be the one most commonly implemented.

Figure 20.3 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS 197, this block is depicted as a square matrix of bytes. This block is copied into the **State** array, which is modified at each stage of encryption or decryption. After the final stage, **State** is copied to an output matrix. Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is 4 bytes and the total key schedule is 44 words for the 128-bit key. The ordering of bytes within a matrix is by column. So, for example, the first 4 bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the **in** matrix, the second 4 bytes occupy the second column, and so on. Similarly, the first 4 bytes of the expanded key, which form a word, occupy the first column of the **w** matrix.

The following comments give some insight into AES:

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, then the halves are swapped. AES does not use a Feistel structure but processes the entire data block in parallel during each round using substitutions and permutation.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, $w[i]$. Four distinct words (128 bits) serve as a round key for each round.
3. Four different stages are used, one of permutation and three of substitution:
 - **Substitute Bytes:** Uses a table, referred to as an S-box,³ to perform a byte-by-byte substitution of the block
 - **Shift Rows:** A simple permutation that is performed row by row
 - **Mix Columns:** A substitution that alters each byte in a column as a function of all of the bytes in the column
 - **Add Round key:** 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 Add Round Key stage, followed by nine rounds that each includes

³The term *S-box*, or substitution box, is commonly used in the description of symmetric ciphers to refer to a table used for a table-lookup type of substitution mechanism.

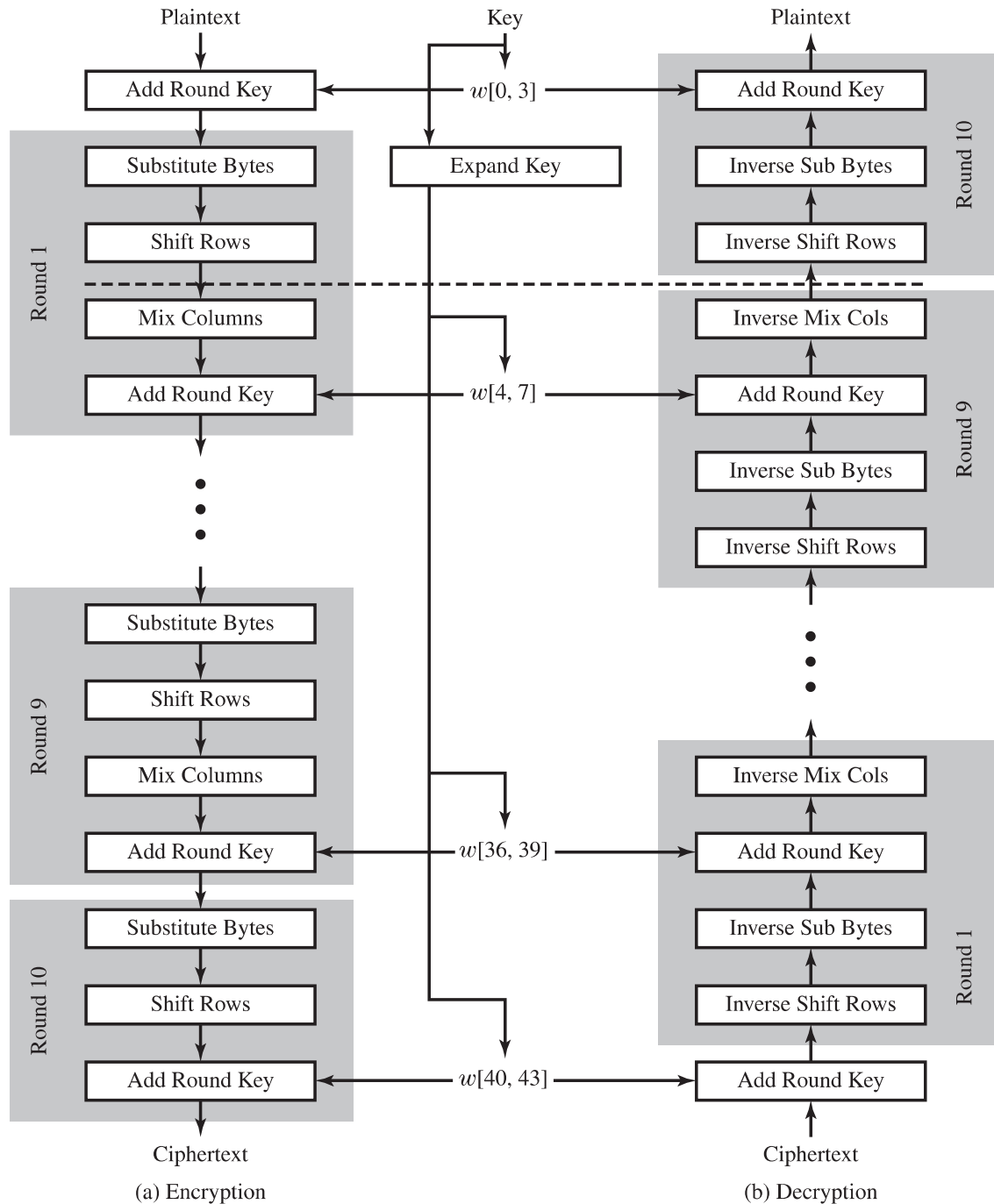


Figure 20.3 AES Encryption and Decryption

all four stages, followed by a tenth round of three stages. Figure 20.4 depicts the structure of a full encryption round.

5. Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.
6. The Add Round Key stage by itself would not be formidable. The other three stages together scramble the bits, but by themselves would provide no security

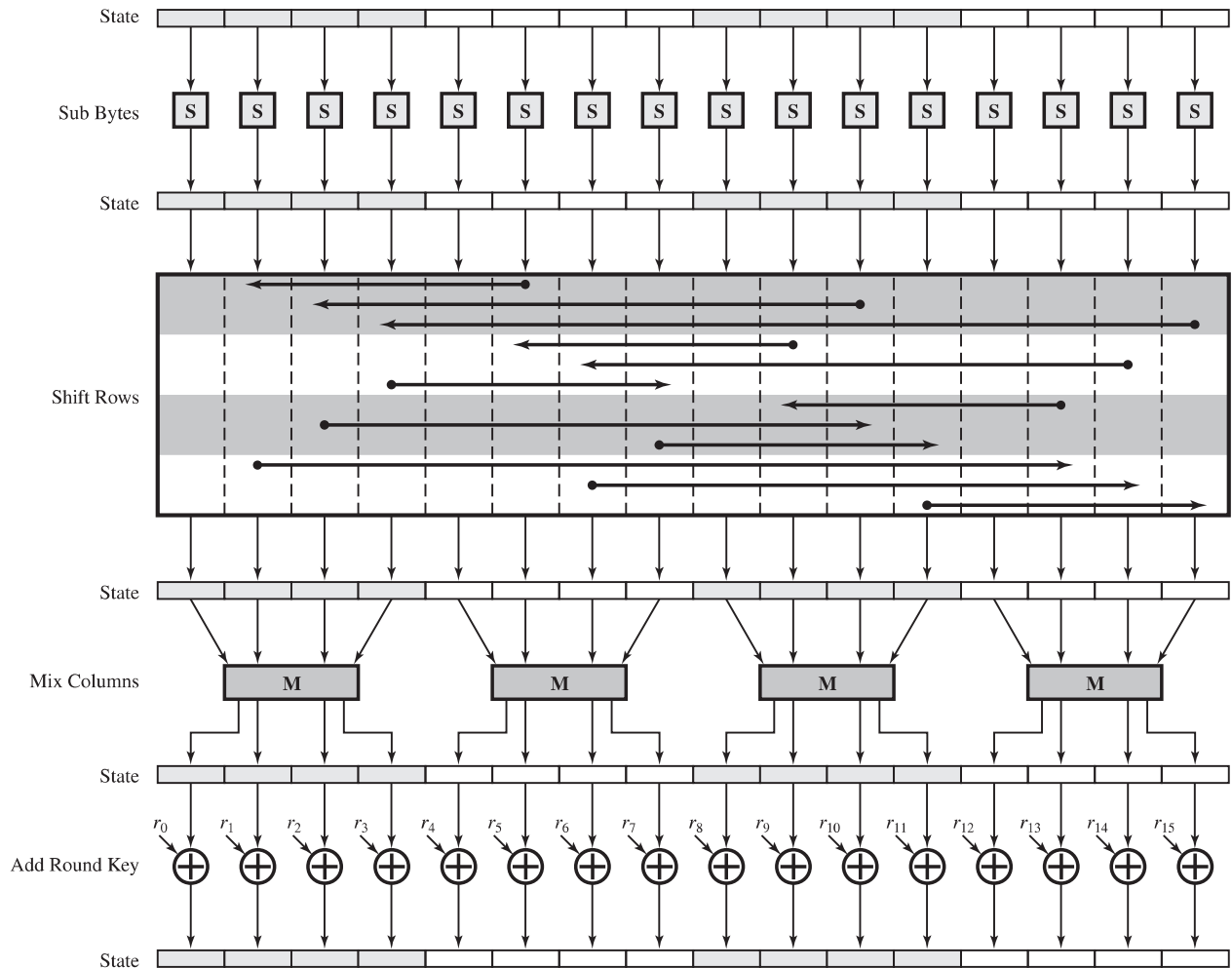


Figure 20.4 AES Encryption Round

because they do not use the key. We can view the cipher as alternating operations of XOR encryption (Add Round Key) 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, Shift Row, and Mix Columns stages, an inverse function is used in the decryption algorithm. For the Add Round Key stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus A \oplus B = B$.
8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.
9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 20.3 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 and 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.

Algorithm Details

We now look briefly at the principal elements of AES in more detail. A more detailed description is given in [STAL17].

SUBSTITUTE BYTES TRANSFORMATION The **forward substitute byte transformation**, called SubBytes, is a simple table lookup. AES defines a 16·16 matrix of byte values, called an S-box (see Table 20.2a), 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 value⁴ {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}.

Here is an example of the SubBytes transformation:

The S-box is constructed using properties of finite fields. The topic of finite fields is beyond the scope of this book; it is discussed in detail in [STAL17].

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 **inverse substitute byte transformation**, called InvSubBytes, makes use of the inverse S-box shown in Table 20.2b. Note, for example, that the input {2A} produces the output {95}, and the input {95} to the S-box produces {2A}.

The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the AES developers sought a design that has a low correlation between input bits and output bits and the property that the output cannot be described as a simple mathematical function of the input.

SHIFT ROW TRANSFORMATION For the **forward shift row transformation**, called ShiftRows, 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 third 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

⁴In FIPS 197, a hexadecimal number is indicated by enclosing it in curly brackets. We use that convention.

Table 20.2 AES S-Boxes

(a) S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	BI	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(b) Inverse S-box

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	FA
	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

The **inverse shift row transformation**, called `InvShiftRows`, performs the circular 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.

The shift row transformation is more substantial than it may first appear. This is because the **State**, as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied to the first column of **State**, and so on. Further, as will be seen, the round key is applied to **State** column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. In addition, note the transformation ensures that the 4 bytes of one column are spread out to four different columns.

MIX COLUMN TRANSFORMATION 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 4 bytes in the column. The mapping makes use of equations over finite fields. The following is an example of `MixColumns`:

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

→

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

The mapping is designed to provide a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits.

ADD ROUND KEY TRANSFORMATION In the **forward add round key transformation**, called `AddRoundKey`, the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. The operation is viewed as a column-wise operation between the four 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
ED	A5	A6	BC

=

EB	59	8B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D2

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.

The add round key transformation is as simple as possible and affects every bit of **State**. The complexity of the round key expansion, plus the complexity of the other stages of AES, ensure security.

AES KEY EXPANSION The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (156 bytes). This is sufficient to provide a 4-word round key for the initial Add Round Key stage and each of the 10 rounds of the cipher.

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word $w[i]$ depends on the immediately preceding word, $w[i - 1]$, and the word four positions back, $w[i - 4]$. A complex finite-field algorithm is used in generating the expanded key.

20.4 STREAM CIPHERS AND RC4

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. Although block ciphers are far more common, there are certain applications in which a stream cipher is more appropriate. Examples are given subsequently in this book. In this section, we look at perhaps the most popular symmetric stream cipher, RC4. We begin with an overview of stream cipher structure, then examine RC4.

Stream Cipher Structure

A typical stream cipher encrypts plaintext 1 byte at a time, although a stream cipher may be designed to operate on 1 bit at a time or on units larger than a byte at a time. Figure 2.3b is a representative diagram of stream cipher structure. In this structure, a key is input to a pseudorandom bit generator that produces a stream of 8-bit numbers that are apparently random. A pseudorandom stream is one that is unpredictable without knowledge of the input key and that has an apparently random character. The output of the generator, called a **keystream**, is combined 1 byte at a time with the plaintext stream using the bitwise exclusive-OR (XOR) operation. For example, if the next byte generated by the generator is 01101100 and the next plaintext byte is 11001100, then the resulting ciphertext byte is:

$$\begin{array}{rcl}
 11001100 & & \text{plaintext} \\
 \oplus \underline{01101100} & & \text{key stream} \\
 10100000 & & \text{ciphertext}
 \end{array}$$

Decryption requires the use of the same pseudorandom sequence:

$$\begin{array}{rcl}
 10100000 & & \text{ciphertext} \\
 \oplus \underline{01101100} & & \text{key stream} \\
 11001100 & & \text{plaintext}
 \end{array}$$

With a properly designed pseudorandom number generator, a stream cipher can be as secure as block cipher of comparable key length. The primary advantage of a stream cipher is that stream ciphers are almost always faster and use far less code

than do block ciphers. The example in this section, RC4, can be implemented in just a few lines of code. Figure 20.5, based on results in [SING11], compares execution times of RC4 with two modes of the symmetric block cipher AES. The advantage of a block cipher is that you can reuse keys. However, if two plaintexts are encrypted with the same key using a stream cipher, then cryptanalysis is often quite simple [DAWS96]. If the two ciphertext streams are XORed together, the result is the XOR of the original plaintexts. If the plaintexts are text strings, credit card numbers, or other byte streams with known properties, then cryptanalysis may be successful.

For applications that require encryption/decryption of a stream of data, such as over a data communications channel or a browser/Web link, a stream cipher might be the better alternative. For applications that deal with blocks of data, such as file transfer, e-mail, and database, block ciphers may be more appropriate. However, either type of cipher can be used in virtually any application.

The RC4 Algorithm

RC4 is a stream cipher designed in 1987 by Ron Rivest for RSA Security. It is a variable-key-size stream cipher with byte-oriented operations. The algorithm is based on the use of a random permutation. Analysis shows that the period of the cipher is overwhelmingly likely to be greater than 10^{100} [ROBS95]. Eight to sixteen machine operations are required per output byte, and the cipher can be expected to run very quickly in software. RC4 is used in the SSL/TLS (Secure Sockets Layer/Transport Layer Security) standards that have been defined for communication between Web browsers and servers. It is also used in the WEP (Wired Equivalent Privacy) protocol and the newer WiFi Protected Access (WPA) protocol that are part of the IEEE 802.11 wireless LAN standard. RC4 was kept as a trade secret by RSA Security. In September 1994, the RC4 algorithm was anonymously posted on the Internet on the Cypherpunks anonymous remailers list.

The RC4 algorithm is remarkably simple and quite easy to explain. A variable-length key of from 1 to 256 bytes (8 to 2048 bits) is used to initialize a 256-byte state

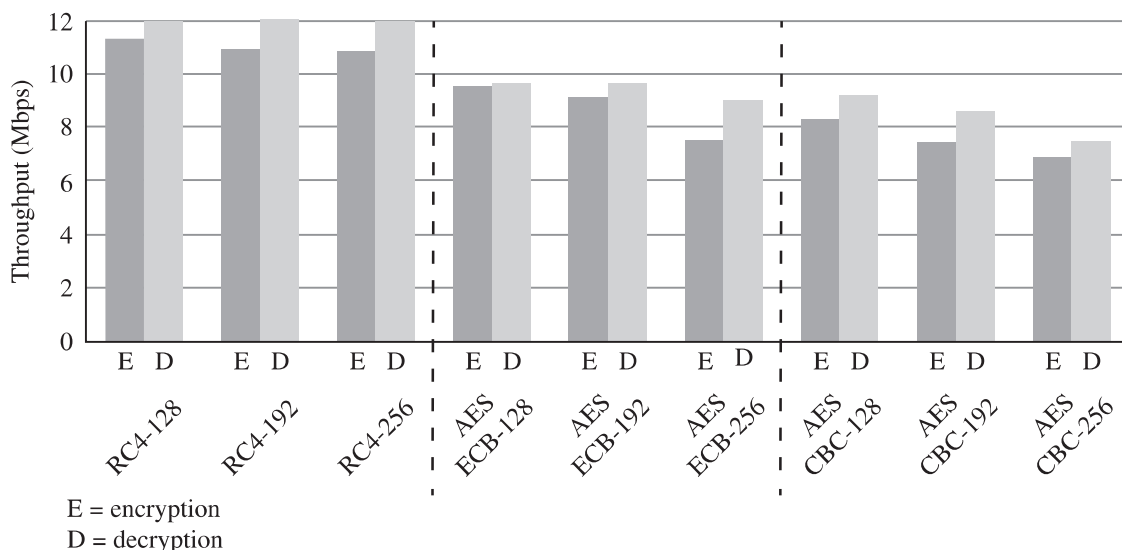


Figure 20.5 Performance Comparison of Symmetric Ciphers on a 3-GHz Processor

vector **S**, with elements **S**[0], **S**[1], . . . , **S**[255]. At all times, **S** contains a permutation of all 8-bit numbers from 0 through 255. For encryption and decryption, a byte k (see Figure 2.3b) is generated from **S** by selecting one of the 255 entries in a systematic fashion. As each value of k is generated, the entries in **S** are once again permuted.

INITIALIZATION OF S To begin, the entries of **S** are set equal to the values from 0 through 255 in ascending order; that is, **S**[0] = 0, **S**[1] = 1, . . . , **S**[255] = 255. A temporary vector, **T**, is also created. If the length of the key **K** is 256 bytes, then **K** is transferred to **T**. Otherwise, for a key of length *keylen* bytes, the first *keylen* elements of **T** are copied from **K** and then **K** is repeated as many times as necessary to fill out **T**. These preliminary operations can be summarized as follows:

```
/* Initialization */
for i = 0 to 255 do
  S[i] = i;
  T[i] = K[i mod keylen];
```

Next we use **T** to produce the initial permutation of **S**. This involves starting with **S**[0] and going through to **S**[255], and, for each **S**[*i*], swapping **S**[*i*] with another byte in **S** according to a scheme dictated by **T**[*i*]:

```
/* Initial Permutation of S */
j = 0;
for i = 0 to 255 do
  j = (j + S[i] + T[i]) mod 256;
  Swap (S[i], S[j]);
```

Because the only operation on **S** is a swap, the only effect is a permutation. **S** still contains all the numbers from 0 through 255.

STREAM GENERATION Once the **S** vector is initialized, the input key is no longer used. Stream generation involves cycling through all the elements of **S**[*i*], and, for each **S**[*i*], swapping **S**[*i*] with another byte in **S** according to a scheme dictated by the current configuration of **S**. After **S**[255] is reached, the process continues, starting over again at **S**[0]:

```
/* Stream Generation */
i, j = 0;
while (true)
  i = (i + 1) mod 256;
  j = (j + S[i]) mod 256;
  Swap (S[i], S[j]);
  t = (S[i] + S[j]) mod 256;
  k = S[t];
```

To encrypt, XOR the value k with the next byte of plaintext. To decrypt, XOR the value k with the next byte of ciphertext.

Figure 20.6 illustrates the RC4 logic.

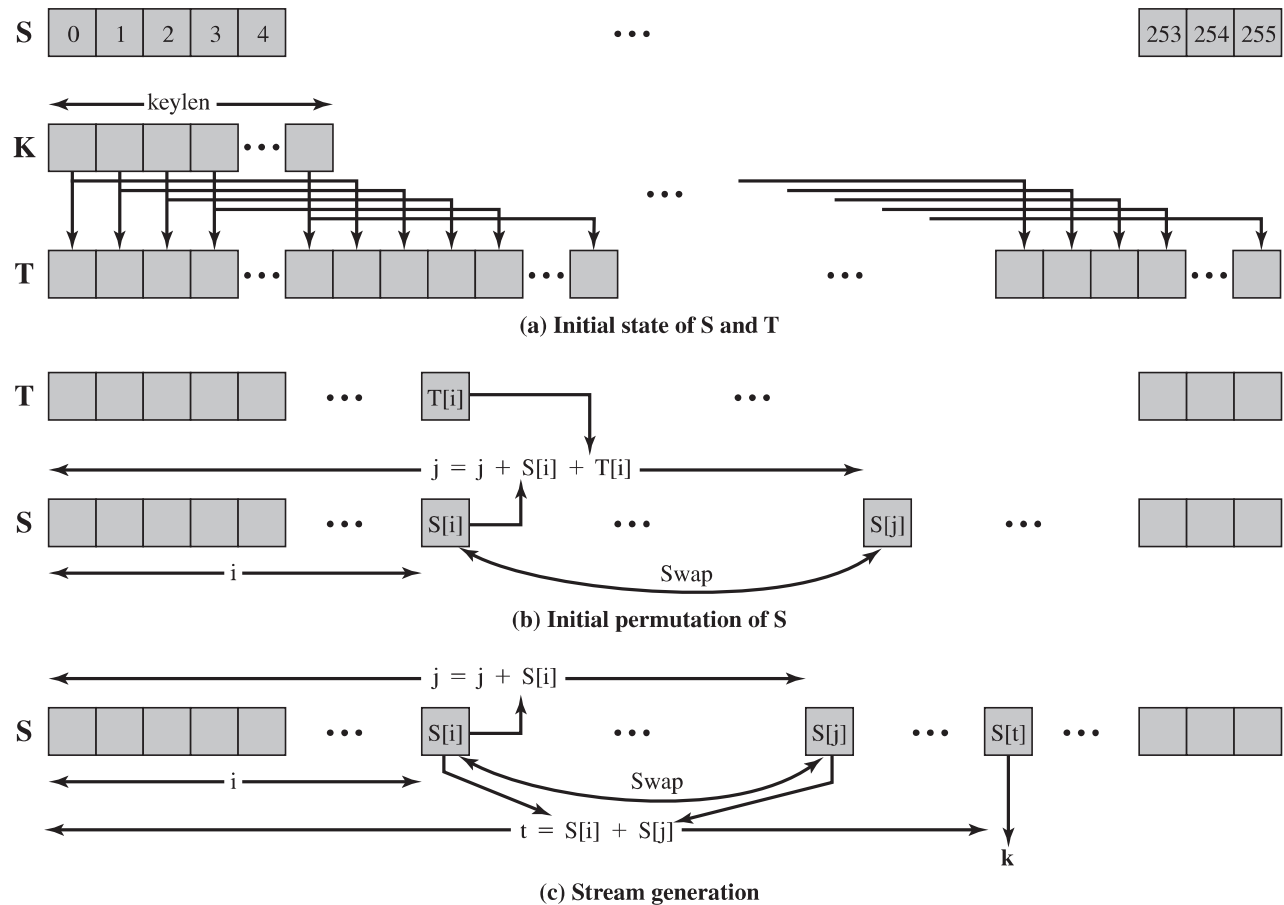


Figure 20.6 RC4

STRENGTH OF RC4 A number of papers have been published analyzing methods of attacking RC4. None of these approaches is practical against RC4 with a reasonable key length, such as 128 bits. A more serious problem is reported in [FLUH01]. The authors demonstrate that the WEP protocol, intended to provide confidentiality on 802.11 wireless LAN networks, is vulnerable to a particular attack approach. In essence, the problem is not with RC4 itself but, the way in which keys are generated for use as input to RC4. This particular problem does not appear to be relevant to other applications using RC4 and can be remedied in WEP by changing the way in which keys are generated. This problem points out the difficulty in designing a secure system that involves both cryptographic functions and protocols that make use of them.

20.5 CIPHER BLOCK MODES OF OPERATION

A symmetric block cipher processes one block of data at a time. In the case of DES and 3DES, the block length is 64 bits. For longer amounts of plaintext, it is necessary to break the plaintext into 64-bit blocks (padding the last block if necessary). To apply a block cipher in a variety of applications, five **modes of operation** have been defined by NIST SP 800-38A (*Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, December 2001). The five modes are intended to cover virtually all the possible applications of encryption for which a block cipher could be used.

Table 20.3 Block Cipher Modes of Operation

Mode	Description	Typical Application
Electronic Code book (ECB)	Each block of 64 plaintext bits is encoded independently using the same key.	<ul style="list-style-type: none"> • Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next 64 bits of plaintext and the preceding 64 bits of ciphertext.	<ul style="list-style-type: none"> • General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed s bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	<ul style="list-style-type: none"> • General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding DES output.	<ul style="list-style-type: none"> • Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	<ul style="list-style-type: none"> • General-purpose block-oriented transmission • Useful for high-speed requirements

These modes are intended for use with any symmetric block cipher, including triple DES and AES. The modes are summarized in Table 20.3, and the most important are described briefly in the remainder of this section.

Electronic Codebook Mode

The simplest way to proceed is what is known as electronic codebook (ECB) mode, in which plaintext is handled b bits at a time and each block of plaintext is encrypted using the same key (see Figure 2.3a). The term *codebook* is used because, for a given key, there is a unique ciphertext for every b -bit block of plaintext. Therefore, one can imagine a gigantic codebook in which there is an entry for every possible b -bit plaintext pattern showing its corresponding ciphertext.

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

To overcome the security deficiencies of ECB, we would like a technique in which the same plaintext block, if repeated, produces different ciphertext blocks.

Cipher Block Chaining Mode

In the cipher block chaining (CBC) mode (see Figure 20.7), the input to the encryption algorithm is the XOR of the current plaintext block and the preceding ciphertext

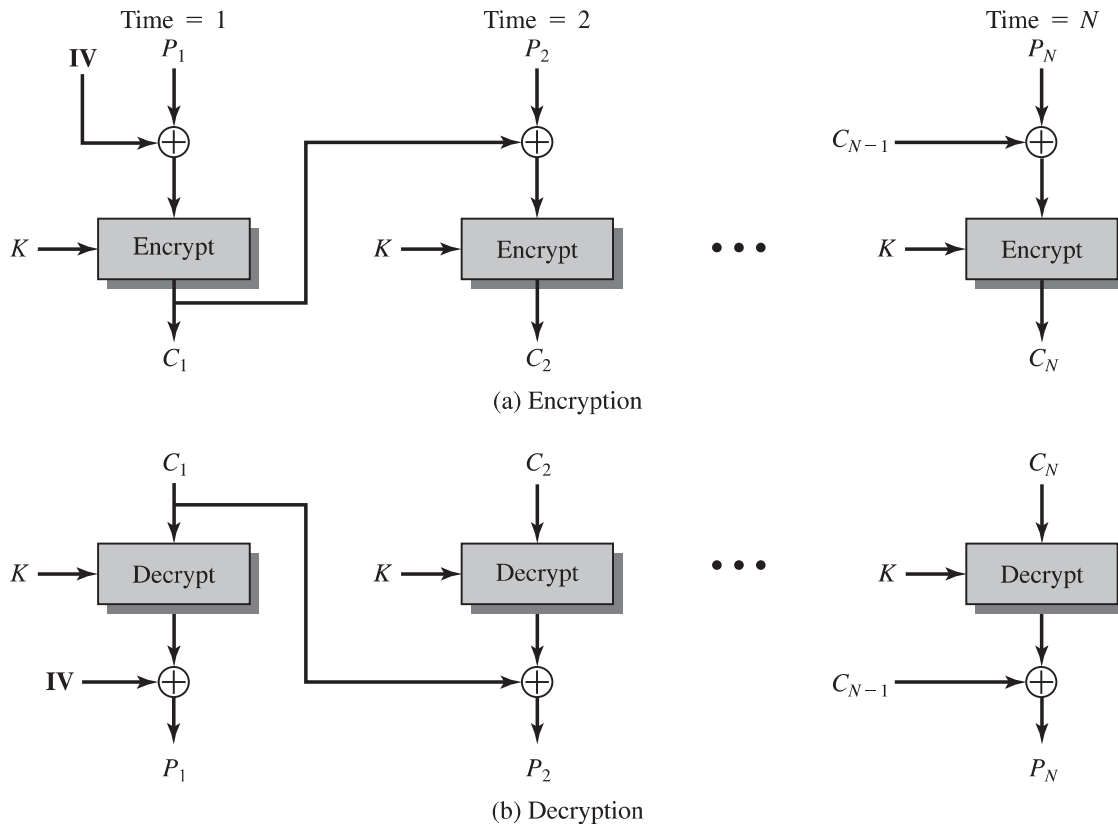


Figure 20.7 Cipher Block Chaining (CBC) Mode

block; the same key is used for each block. In effect, we have chained together the processing of the sequence of plaintext blocks. The input to the encryption function for each plaintext block bears no fixed relationship to the plaintext block. Therefore, repeating patterns of b -bits are not exposed.

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

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

where $E[K, X]$ is the encryption of plaintext X using key K , and \oplus is the exclusive-OR operation. Then

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

which verifies Figure 20.7b.

To produce the first block of ciphertext, an initialization vector (IV) is XORed with the first block of plaintext. On decryption, the IV is XORed with the output of the decryption algorithm to recover the first block of plaintext.

The IV must be known to both the sender and receiver. For maximum security, the IV should be protected as well as the key. This could be done by sending the IV

using ECB encryption. One reason for protecting the IV is as follows: If an opponent is able to fool the receiver into using a different value for IV, then the opponent is able to invert selected bits in the first block of plaintext. To see this, consider the following:

$$\begin{aligned} C_1 &= E(K, [IV \oplus P_1]) \\ P_1 &= IV \oplus D(K, C_1) \end{aligned}$$

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

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

Then, using the properties of XOR, we can state

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

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

Cipher Feedback Mode

It is possible to convert any block cipher into a stream cipher by using the cipher feedback (CFB) mode. A stream cipher eliminates the need to pad a message to be an integral number of blocks. It also can operate in real time. Thus, if a character stream is being transmitted, each character can be encrypted and transmitted immediately using a character-oriented stream cipher.

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

Figure 20.8 depicts the CFB scheme. In the figure, it is assumed that the unit of transmission is s bits; a common value is $s = 8$. As with CBC, the units of plaintext are chained together, so the ciphertext of any plaintext unit is a function of all the preceding plaintext.

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

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

$$C_1 = P_1 \oplus S_s[E(K, IV)]$$

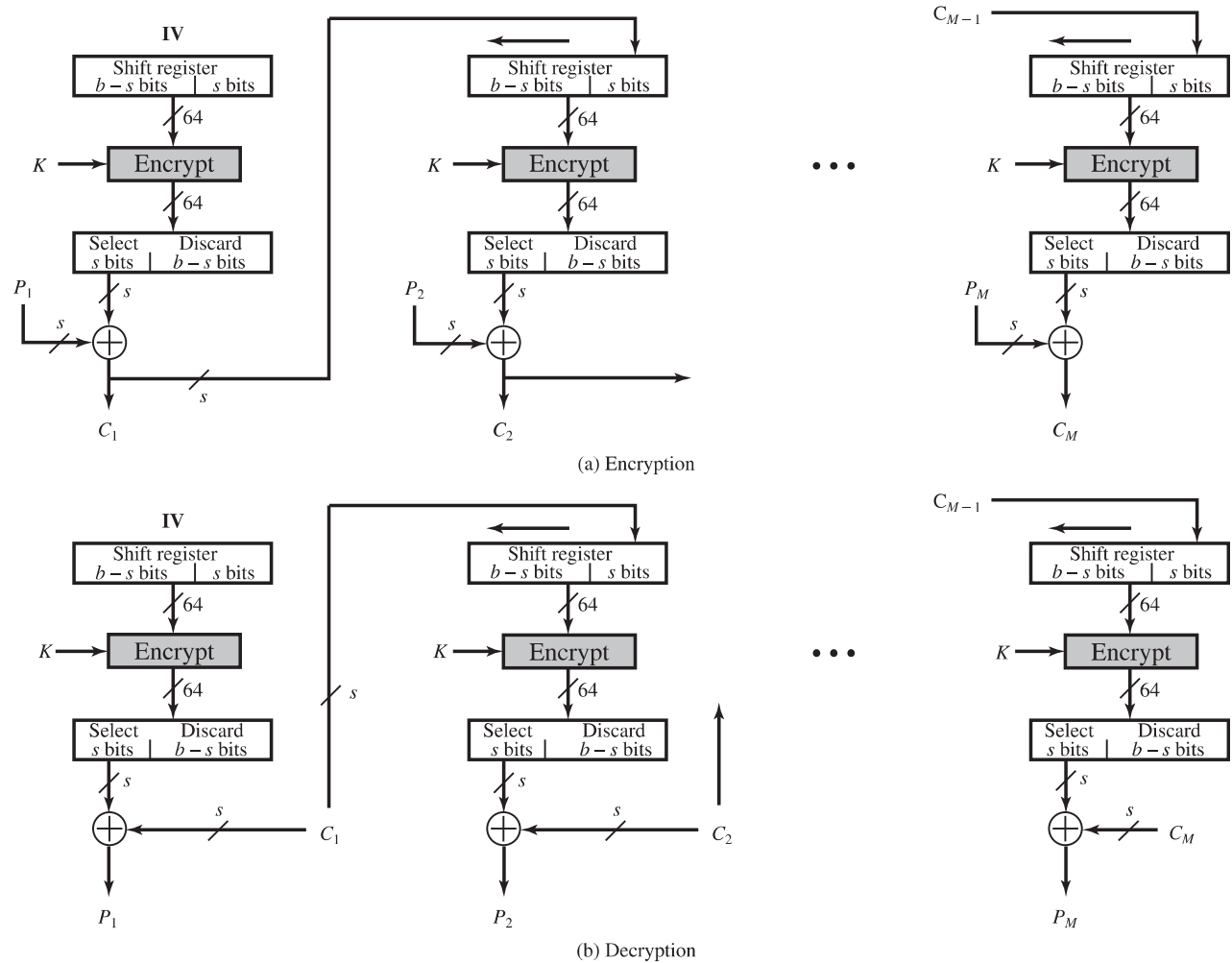


Figure 20.8 *s*-bit Cipher Feedback (CFB) Mode

Therefore,

$$P_1 = C_1 \oplus S_s[E(K, IV)]$$

The same reasoning holds for subsequent steps in the process.

Counter Mode

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

Figure 20.9 depicts the CTR mode. A counter equal to the plaintext block size is used. The only requirement stated in SP 800-38A is that the counter value must be different for each plaintext block that is encrypted. Typically, the counter is initialized to some value and then incremented by 1 for each subsequent block (modulo 2^b , where b is the block size). For encryption, the counter is encrypted then XORed with the plaintext block to produce the ciphertext block; there is no chaining. For decryption, the same sequence of counter values is used, with each encrypted counter XORed with a ciphertext block to recover the corresponding plaintext block.

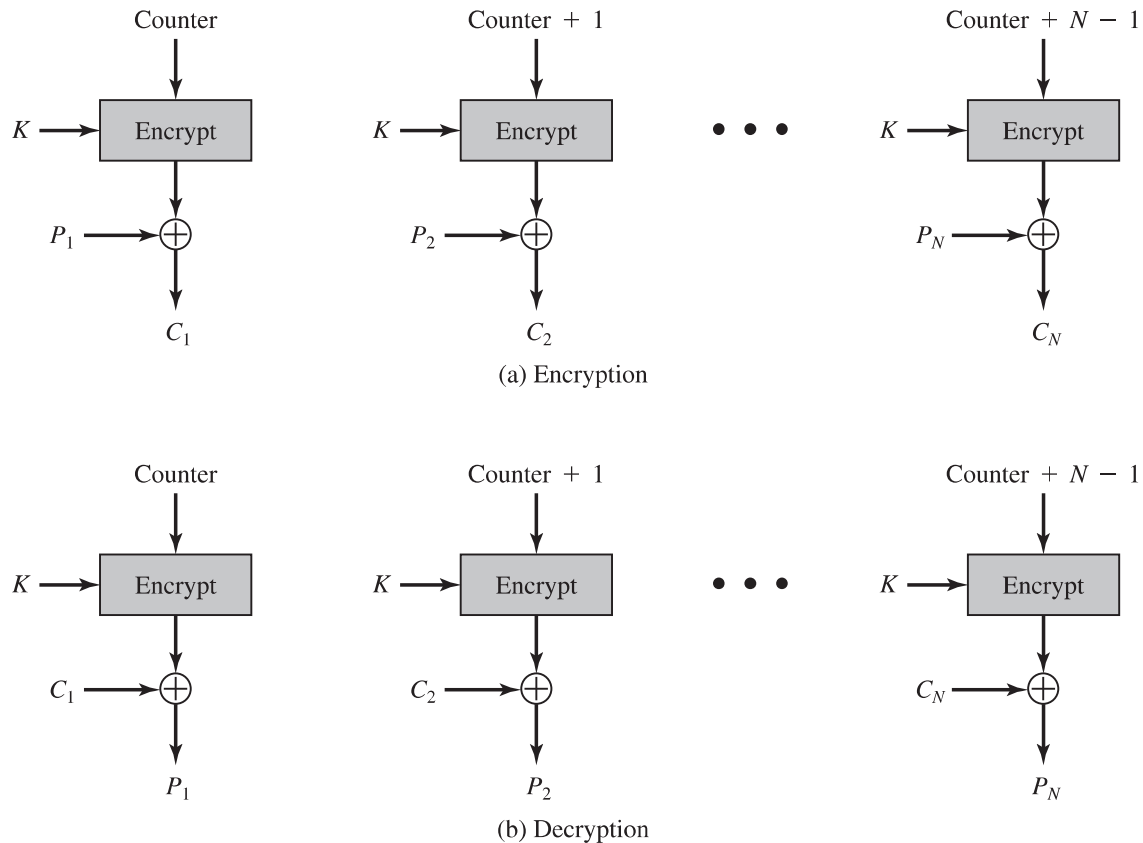


Figure 20.9 Counter (CTR) Mode

[LIPM00] lists the following advantages of CTR mode:

- **Hardware efficiency:** Unlike the three chaining modes, encryption (or decryption) in CTR mode can be done in parallel on multiple blocks of plaintext or ciphertext. For the chaining modes, the algorithm must complete the computation on one block before beginning on the next block. This limits the maximum throughput of the algorithm to the reciprocal of the time for one execution of block encryption or decryption. In CTR mode, the throughput is only limited by the amount of parallelism that is achieved.
- **Software efficiency:** Similarly, because of the opportunities for parallel execution in CTR mode, processors that support parallel features, such as aggressive pipelining, multiple instruction dispatch per clock cycle, a large number of registers, and SIMD instructions, can be effectively utilized.
- **Preprocessing:** The execution of the underlying encryption algorithm does not depend on input of the plaintext or ciphertext. Therefore, if sufficient memory is available and security is maintained, preprocessing can be used to prepare the output of the encryption boxes that feed into the XOR functions in Figure 20.9. When the plaintext or ciphertext input is presented, then the only computation is a series of XORs. Such a strategy greatly enhances throughput.
- **Random access:** The i th block of plaintext or ciphertext can be processed in random access fashion. With the chaining modes, block C_i cannot be computed until the $i - 1$ prior block are computed. There may be applications in

which a ciphertext is stored and it is desired to decrypt just one block; for such applications, the random access feature is attractive.

- **Provable security:** It can be shown that CTR is at least as secure as the other modes discussed in this section.
- **Simplicity:** Unlike ECB and CBC modes, CTR mode requires only the implementation of the encryption algorithm and not the decryption algorithm. This matters most when the decryption algorithm differs substantially from the encryption algorithm, as it does for AES. In addition, the decryption key scheduling need not be implemented.

20.6 KEY DISTRIBUTION

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

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

Options 1 and 2 call for manual delivery of a key. For **link encryption** between two directly-connected devices, this is a reasonable requirement, because each link encryption device is only going to be exchanging data with its partner on the other end of the link. However, for **end-to-end encryption** over a network, manual delivery is awkward. In a distributed system, any given host or terminal may need to engage in exchanges with many other hosts and terminals over time. Thus, each device needs a number of keys, supplied dynamically. The problem is especially difficult in a wide area distributed system.

Option 3 is a possibility for either link encryption or end-to-end encryption, but if an attacker ever succeeds in gaining access to one key, then all subsequent keys are revealed. Even if frequent changes are made to the link encryption keys, these should be done manually. To provide keys for end-to-end encryption, option 4 is preferable.

Figure 20.10 illustrates an implementation that satisfies option 4 for end-to-end encryption. In the figure, link encryption is ignored. This can be added, or not, as required. For this scheme, two kinds of keys are identified:

- **Session key:** When two end systems (hosts, terminals, etc.) wish to communicate, they establish a logical connection (e.g., virtual circuit). For the duration of that logical connection, all user data are encrypted with a one-time session key. At the conclusion of the session, or connection, the session key is destroyed.
- **Permanent key:** A permanent key is a key used between entities for the purpose of distributing session keys.

The configuration consists of the following elements:

- **Key distribution center (KDC):** The key distribution center (KDC) determines which systems are allowed to communicate with each other. When permission is granted for two systems to establish a connection, the KDC provides a one-time session key for that connection.
- **Security service module (SSM):** This module, which may consist of functionality at one protocol layer, performs end-to-end encryption and obtains session keys on behalf of users.

The steps involved in establishing a connection are shown in Figure 20.10. When one host wishes to set up a connection to another host, it transmits a connection-request packet (step 1). The SSM saves that packet and applies to the KDC for permission to establish the connection (step 2). The communication between the SSM and the KDC is encrypted using a master key shared only by this SSM and the KDC. If the KDC approves the connection request, it generates the session key and delivers it to the two appropriate SSMs, using a unique permanent key for each SSM (step 3). The requesting SSM can now release the connection request packet, and a connection is set up between the two end systems (step 4). All user data exchanged between the two end systems are encrypted by their respective SSMs using the one-time session key.

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

Another approach to key distribution uses public-key encryption, which will be discussed in Chapter 21.

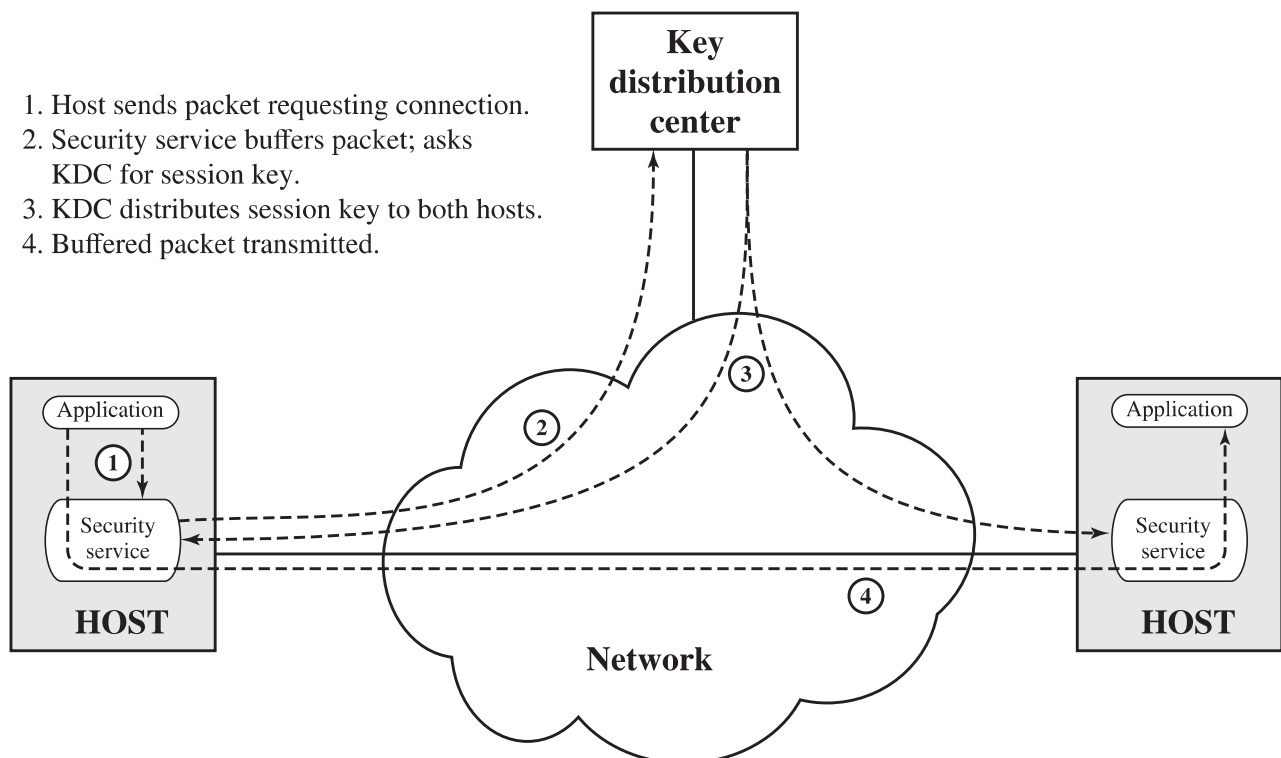


Figure 20.10 Automatic Key Distribution for Connection-Oriented Protocol

20.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Advanced Encryption Standard (AES) block cipher brute-force attack computationally secure cipher block chaining (CBC) mode cipher feedback (CFB) mode ciphertext counter mode	cryptanalysis cryptography Data Encryption Standard (DES) decryption electronic codebook (ECB) mode encryption end-to-end encryption Feistel cipher key distribution	keystream link encryption modes of operation plaintext RC4 session key stream cipher subkey symmetric encryption triple DES (3DES)
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------

Review Questions

- 20.1 List the dimensions across which cryptographic systems are classified.
- 20.2 What are different types of cryptanalysis attacks?
- 20.3 When is an encryption scheme considered to be computationally secure?
- 20.4 What is the difference between a block cipher and a stream cipher?
- 20.5 List the different block cipher modes of operation.
- 20.6 Why do some block cipher modes of operation only use encryption while others use both encryption and decryption?
- 20.7 What is triple encryption?
- 20.8 Why is the middle portion of 3DES a decryption rather than an encryption?
- 20.9 State the advantages of CTR mode.
- 20.10 List ways in which secret keys can be distributed to two communicating parties.
- 20.11 What is the difference between a session key and a master key?
- 20.12 What is a security service module?

Problems

- 20.1 Let the encryption of a message M with a secret key K using DES be denoted by $Y = \text{DES}_K(M)$, and the bit-complement of Z be denoted by $c(Z)$. Prove that the key-complementation property of DES: $c(Y) = \text{DES}_{c(K)}(c(M))$.
- 20.2 Consider a Feistel cipher composed of 16 rounds with block length 128 bits and key length 128 bits. Suppose for a given k , the key scheduling algorithm determines values for the first 8 round keys, k_1, k_2, \dots, k_8 , then sets

$$k_9 = k_8, k_{10} = k_7, k_{11} = k_6, \dots, k_{16} = k_1$$

Suppose you have a ciphertext c . Explain how, with access to an encryption oracle, you can decrypt c and determine m using just a single oracle query. This shows that such a cipher is vulnerable to a chosen plaintext attack. (An encryption oracle can be thought of as a device that, when given a plaintext, returns the corresponding ciphertext. The internal details of the device are not known to you and you cannot break open the device. You can only gain information from the oracle by making queries to it and observing its responses.)

- 20.3 For any block cipher, the fact that it is a nonlinear function is crucial to its security. To see this, suppose we have a linear block cipher EL that encrypts 128-bit blocks of plaintext into 128-bit blocks of ciphertext. Let $EL(k, m)$ denote the encryption of a 128-bit message m under a key k (the actual bit length of k is irrelevant). Thus

$$EL(k, [m_1 \oplus m_2]) = EL(k, m_1) \oplus EL(k, m_2) \text{ for all 128-bit patterns } m_1, m_2$$

Describe how, with 128 chosen ciphertexts, an adversary can decrypt any ciphertext without knowledge of the secret key k . (A “chosen ciphertext” means that an adversary has the ability to choose a ciphertext and then obtain its decryption. Here, you have 128 plaintext/ciphertext pairs with which to work and you have the ability to choose the value of the ciphertexts.)

- 20.4 Suppose that your organization wants you to ensure the security of its data while the data is in transit. Which one out of stream cipher and block cipher would you select and why?
- 20.5 RC4 has a secret internal state which is a permutation of all the possible values of the vector \mathbf{S} and the two indices i and j .
- Using a straightforward scheme to store the internal state, how many bits are used?
 - Suppose we think of it from the point of view of how much information is represented by the state. In that case, we need to determine how many different states there are, then take the log to the base 2 to find out how many bits of information this represents. Using this approach, how many bits would be needed to represent the state?
- 20.6 With the ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error in the transmitted C_k (Figure 20.6) obviously corrupts P_k and P_{k+1} :
- Are any blocks beyond P_{k+1} affected?
 - Suppose that there is a bit error in the source version of P_k . Through how many ciphertext blocks is this error propagated? What is the effect at the receiver?
- 20.7 Can we perform encryption operations in parallel on multiple blocks of plaintext in any of the five modes? How about decryption?
- 20.8 You want to build a hardware device to do block encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Figure 20.11 shows two possibilities, both of which follow from the definition of CBC. Which of the two would you choose:
- For security?
 - For performance?
- 20.9 Can you suggest a security improvement to either option in Figure 20.11, using only three DES chips and some number of XOR functions? Assume you are still limited to two keys.
- 20.10 Fill in the remainder of this table:

Mode	Encrypt	Decrypt
ECB	$C_j = E(K, P_j) \quad j = 1, \dots, N$	$P_j = D(K, C_j) \quad j = 1, \dots, N$
CBC	$C_1 = E(K, [P_1 \oplus IV])$ $C_j = E(K, [P_j \oplus C_{j-1}]) \quad j = 2, \dots, N$	$P_1 = D(K, C_1) \oplus IV$ $P_j = D(K, C_j) \oplus C_{j-1} \quad j = 2, \dots, N$
CFB		
CTR		

- 20.11 CBC-Pad is a block cipher mode of operation used in the RC5 block cipher, but it could be used in any block cipher. CBC-Pad handles plaintext of any length. The ciphertext is longer than the plaintext by at most the size of a single block. Padding is used to assure that the plaintext input is a multiple of the block length. It is assumed that the

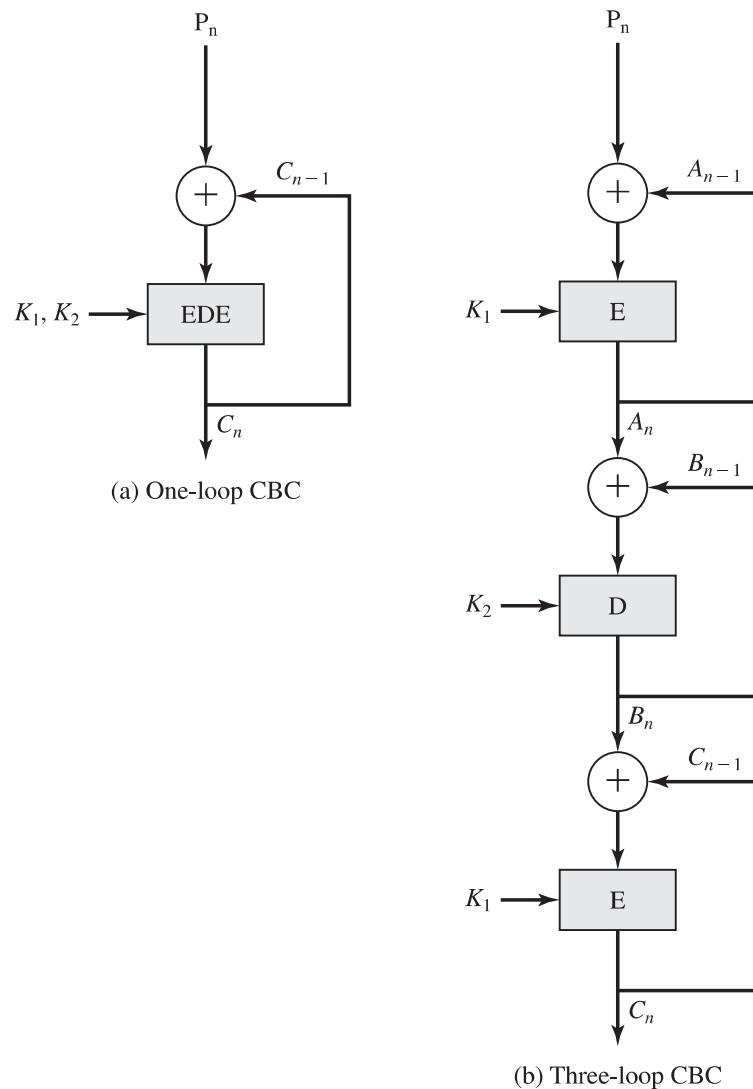


Figure 20.11 Use of Triple DES in CBC Mode

original plaintext is an integer number of bytes. This plaintext is padded at the end by from 1 to bb bytes, where bb equals the block size in bytes. The pad bytes are all the same and set to a byte that represents the number of bytes of padding. For example, if there are 8 bytes of padding, each byte has the bit pattern 00001000. Why not allow zero bytes of padding? That is, if the original plaintext is an integer multiple of the block size, why not refrain from padding?

- 20.12** Padding may not always be appropriate. For example, one might wish to store the encrypted data in the same memory buffer that originally contained the plaintext. In that case, the ciphertext must be the same length as the original plaintext. A mode for that purpose is the ciphertext stealing (CTS) mode. Figure 20.12a shows an implementation of this mode.
- Explain how it works.
 - Describe how to decrypt C_{n-1} and C_n .
- 20.13** Figure 20.12b shows an alternative to CTS for producing ciphertext of equal length to the plaintext when the plaintext is not an integer multiple of the block size.
- Explain the algorithm.
 - Explain why CTS is preferable to this approach illustrated in Figure 20.12b.
- 20.14** If a bit error occurs in the transmission of a ciphertext character in b -bit CFB mode, how far does the error propagate?

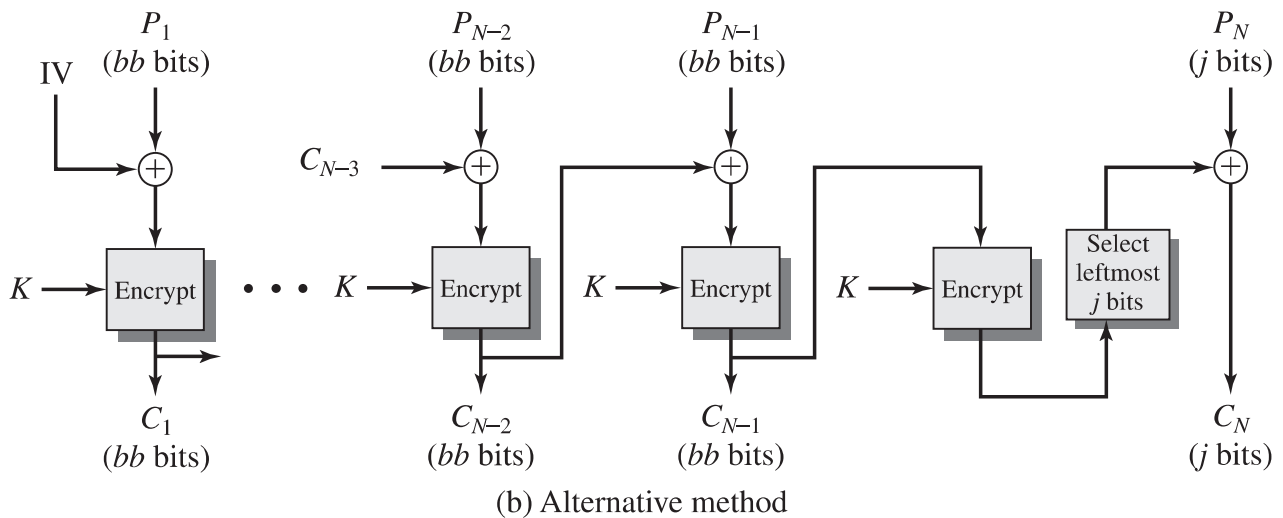
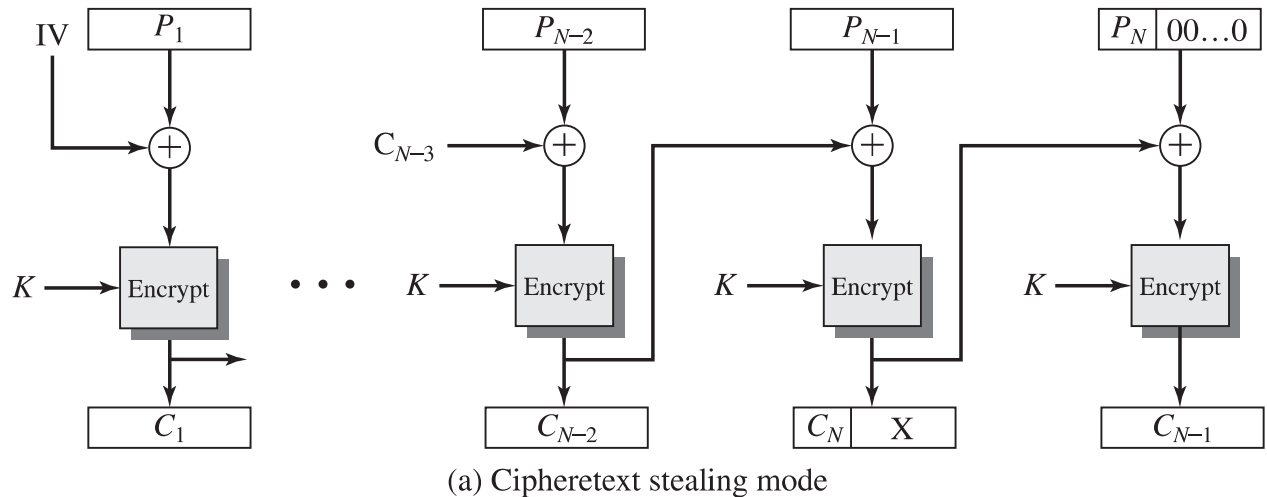


Figure 20.12 Block Cipher Modes for Plaintext Not a Multiple of Block Size

- 20.15** One of the most widely used message authentication codes (MACs), referred to as the Data Authentication Algorithm, is based on DES. The algorithm is both a FIPS publication (FIPS PUB 113) and an ANSI standard (X9.17). The algorithm can be defined as using the cipher block chaining (CBC) mode of operation of DES with an initialization vector of zero (see Figure 20.7). The data (e.g., message, record, file, or program) to be authenticated are grouped into contiguous 64-bit blocks: P_1, P_2, \dots, P_N . If necessary, the final block is padded on the right with 0s to form a full 64-bit block. The MAC consists of either the entire ciphertext block C_N or the leftmost M bits of the block, with $16 \leq M \leq 64$. Show the same result can be produced using the cipher feedback mode.
- 20.16** As discussed in Section Section 20.7, two parties can exchange the new keys by encrypting them with recently used old keys. Discuss the security implications of such an approach.
- 20.17** Suppose someone suggests the following way to confirm that the two of you are both in possession of the same secret key. You create a random bit string the length of the key, XOR it with the key, and send the result over the channel. Your partner XORs the incoming block with the key (which should be the same as your key) and sends it back. You check, and if what you receive is your original random string, you have verified that your partner has the same secret key, yet neither of you has ever transmitted the key. Is there a flaw in this scheme?

PUBLIC-KEY CRYPTOGRAPHY AND MESSAGE AUTHENTICATION

21.1 Secure Hash Functions

- Simple Hash Functions
- The SHA Secure Hash Function
- SHA-3

21.2 HMAC

- HMAC Design Objectives
- HMAC Algorithm
- Security of HMAC

21.3 Authenticated Encryption

21.4 The RSA Public-Key Encryption Algorithm

- Description of the Algorithm
- The Security of RSA

21.5 Diffie-Hellman and Other Asymmetric Algorithms

- Diffie-Hellman Key Exchange
- Other Public-Key Cryptography Algorithms

21.6 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Understand the operation of SHA-1 and SHA-2.
- ◆ Present an overview of the use of HMAC for message authentication.
- ◆ Describe the RSA algorithm.
- ◆ Describe the Diffie-Hellman algorithm.

This chapter provides technical detail on the topics introduced in Sections 2.2 through 2.4.

21.1 SECURE HASH FUNCTIONS

The one-way hash function, or secure hash function, is important not only in message authentication but also in digital signatures. The requirements for, and security of, secure hash functions are discussed in Section 2.2. Here, we look at several hash functions, concentrating on perhaps the most widely used family of hash functions: Secure Hash Algorithm (SHA).

Simple Hash Functions

All hash functions operate using the following general principles. The input (message, file, etc.) is viewed as a sequence of n -bit blocks. The input is processed one block at a time in an iterative fashion to produce an n -bit hash function.

One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block. This can be expressed as follows:

$$C_i = b_{i1} \oplus b_{i2} \oplus \dots \oplus b_{im}$$

where

- C_i = i th bit of the hash code, $1 \leq i \leq n$,
- m = number of n -bit blocks in the input,
- b_{ij} = i th bit in j th block, and
- \oplus = XOR operation.

Figure 21.1 illustrates this operation; it produces a simple parity for each bit position and is known as a longitudinal redundancy check. It is reasonably effective for random data as a data integrity check. Each n -bit hash value is equally likely. Thus, the probability that a data error will result in an unchanged hash value is 2^{-n} . With more predictably formatted data, the function is less effective. For example, in most normal text files, the high-order bit of each octet is always zero. So if a 128-bit hash value is used, instead of an effectiveness of 2^{-128} , the hash function on this type of data has an effectiveness of 2^{-112} .

	Bit 1	Bit 2	• • •	Bit n
Block 1	b_{11}	b_{21}		b_{n1}
Block 2	b_{12}	b_{22}		b_{n2}
	•	•	•	•
	•	•	•	•
	•	•	•	•
Block m	b_{1m}	b_{2m}		b_{nm}
Hash code	C_1	C_2		C_n

Figure 21.1 Simple Hash Function Using Bitwise XOR

A simple way to improve matters is to perform a 1-bit circular shift, or rotation, on the hash value after each block is processed. The procedure can be summarized as follows:

1. Initially set the n -bit hash value to zero.
2. Process each successive n -bit block of data as follows:
 - a. Rotate the current hash value to the left by 1 bit.
 - b. XOR the block into the hash value.

This has the effect of “randomizing” the input more completely and overcoming any regularities that appear in the input.

Although the second procedure provides a good measure of data integrity, it is virtually useless for data security when an encrypted hash code is used with a plaintext message, as in Figures 2.5a and b. Given a message, it is an easy matter to produce a new message that yields that hash code: Simply prepare the desired alternate message, then append an n -bit block that forces the new message plus block to yield the desired hash code.

Although a simple XOR or rotated XOR (RXOR) is insufficient if only the hash code is encrypted, you may still feel that such a simple function could be useful when the message as well as the hash code is encrypted. But one must be careful. A technique originally proposed by the National Bureau of Standards used the simple XOR applied to 64-bit blocks of the message and then an encryption of the entire message that used the cipher block chaining (CBC) mode. We can define the scheme as follows: Given a message consisting of a sequence of 64-bit blocks X_1, X_2, \dots, X_N , define the hash code C as the block-by-block XOR of all blocks and append the hash code as the final block:

$$C = X_{N+1} = X_1 \oplus X_2 \oplus \dots \oplus X_N$$

Next, encrypt the entire message plus hash code, using CBC mode to produce the encrypted message Y_1, Y_2, \dots, Y_{N+1} . [JUEN85] points out several ways in which the ciphertext of this message can be manipulated in such a way that it is not detectable by the hash code. For example, by the definition of CBC (see Figure 20.7), we have:

$$\begin{aligned} X_1 &= IV \oplus D(K, Y_1) \\ X_i &= Y_{i-1} \oplus D(K, Y_i) \\ X_{N+1} &= Y_N \oplus D(K, Y_{N+1}) \end{aligned}$$

But X_{N+1} is the hash code:

$$\begin{aligned} X_{N+1} &= X_1 \oplus X_2 \oplus \dots \oplus X_N \\ &= [IV \oplus D(K, Y_1)] \oplus [Y_1 \oplus D(K, Y_2)] \oplus \dots \oplus [Y_{N-1} \oplus D(K, Y_N)] \end{aligned}$$

Because the terms in the preceding equation can be XORed in any order, it follows that the hash code would not change if the ciphertext blocks were permuted.

The SHA Secure Hash Function

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). Indeed, because virtually every other widely used hash function had been found to have substantial cryptanalytic weaknesses, SHA was more or less the last remaining standardized hash algorithm by 2005. SHA was developed by the National Institute of Standards and Technology (NIST) and published as FIPS 180 in 1993. When weaknesses were discovered in SHA (now known as SHA-0), a revised version was issued as FIPS 180-1 in 1995 and is referred to as **SHA-1**. The actual standards document is entitled “Secure Hash Standard. SHA-1 is also specified in RFC 3174 (*US Secure Hash Algorithm 1 (SHA1)*, 2001), which essentially duplicates the material in FIPS 180-1 but adds a C code implementation.

SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively (see Table 21.1). Collectively, these hash algorithms are known as **SHA-2**. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIPS 180-3 in 2008, which added a 224-bit version of SHA-256, whose hash value is obtained by truncating the 256-bit hash value of SHA-256. SHA-1 and SHA-2 are also specified in RFC 6234 (*US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, 2011), which essentially duplicates the material in FIPS 180-3 but adds a C code implementation. The most recent version is FIPS 180-4 [*Secure Hash Standard (SHS)*, August 2015] which added two variants of SHA-512 with 224-bit and 256-bit hash sizes, as SHA-512 is more efficient than SHA-256 on many 64-bit systems.

In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on SHA-2 by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using 2^{69} operations, far fewer than the 2^{80} operations previously thought needed to find a collision with an SHA-1 hash [WANG05]. This result has hastened the transition to SHA-2.

In this section, we provide a description of SHA-512. The other versions are quite similar. The algorithm takes as input a message with a maximum length of less

Table 21.1 Comparison of SHA Parameters

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512	SHA-512/224	SHA-512/256
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$
Word size	32	32	32	64	64	64	64
Block size	512	512	512	1024	1024	1024	1024
Message digest size	160	224	256	384	512	224	256
Number of steps	80	64	64	80	80	80	80
Security	80	112	128	192	256	112	128

Notes: 1. All sizes are measured in bits.

2. Security refers to the fact that a birthday attack on a message digest of size n produces a collision with a work factor of approximately $2^{n/2}$.

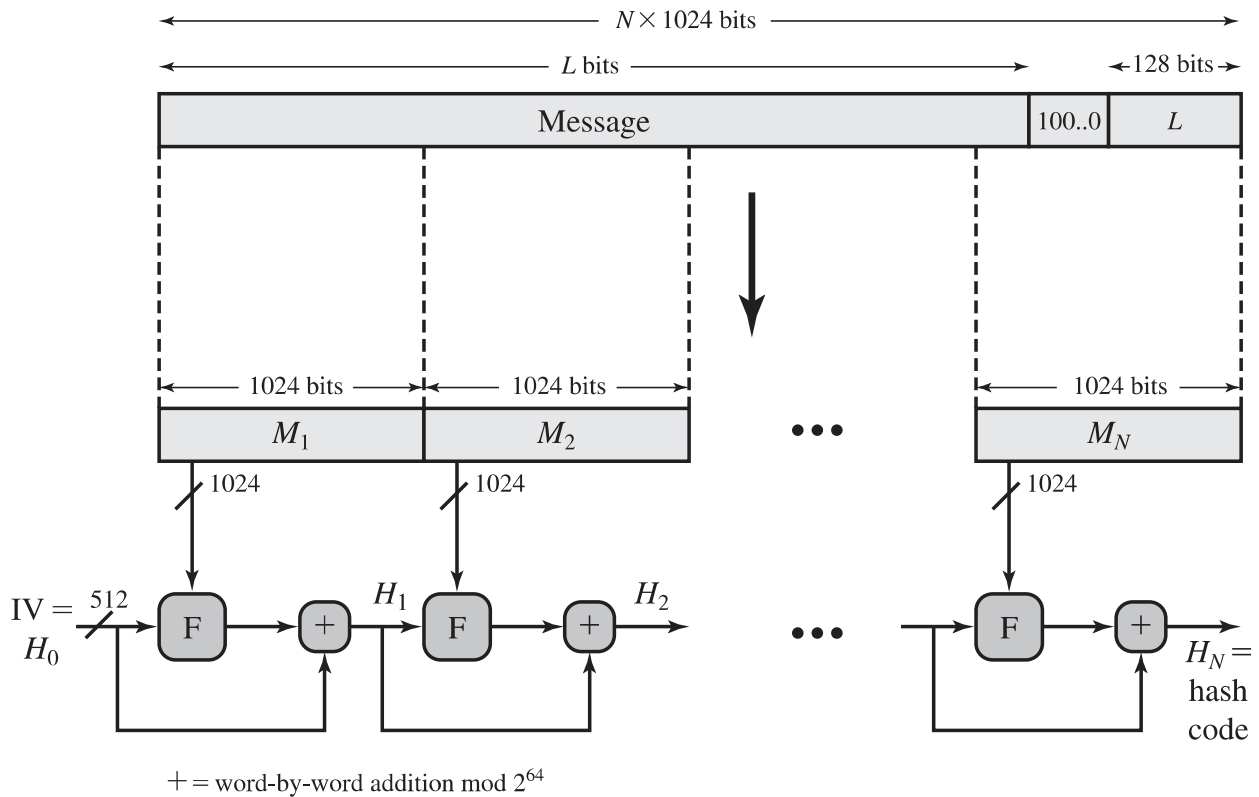


Figure 21.2 Message Digest Generation Using SHA-512

than 2^{128} bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks. Figure 21.2 depicts the overall processing of a message to produce a digest. The processing consists of the following steps:

- **Step 1: Append padding bits.** The message is padded so its length is congruent to 896 modulo 1024 [length $\equiv 896 \pmod{1024}$]. Padding is always added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1-bit followed by the necessary number of 0-bits.
- **Step 2: Append length.** A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding).

The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. In Figure 21.2, the expanded message is represented as the sequence of 1024-bit blocks M_1, M_2, \dots, M_N , so the total length of the expanded message is $N \times 1024$ bits.

- **Step 3: Initialize hash buffer.** A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following 64-bit integers (hexadecimal values):

$$a = 6A09E667F3BCC908$$

$$b = BB67AE8584CAA73B$$

$$c = 3C6EF372FE94F82B$$

$$d = A54FF53A5F1D36F1$$

$$e = 510E527FADE682D1$$

$$f = 9B05688C2B3E6C1F$$

$$g = 1F83D9ABFB41BD6B$$

$$h = 5BE0CD19137E2179$$

These values are stored in big-endian format, which is the most significant byte of a word in the low-address (leftmost) byte position. These words were obtained by taking the first 64 bits of the fractional parts of the square roots of the first eight prime numbers.

- **Step 4: Process message in 1024-bit (128-word) blocks.** The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 21.2. The logic is illustrated in Figure 21.3.

Each round takes as input the 512-bit buffer value $abcdefgh$ and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, H_{i-1} . Each round t makes use of a 64-bit value W_t , derived from the current 1024-bit block being processed (M_i). Each round also makes use of an additive constant K_t , where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. The constants provide a “randomized” set of 64-bit patterns, which should eliminate any regularities in the input data. The operations performed during a round consist of circular shifts, and primitive Boolean functions based on AND, OR, NOT, and XOR.

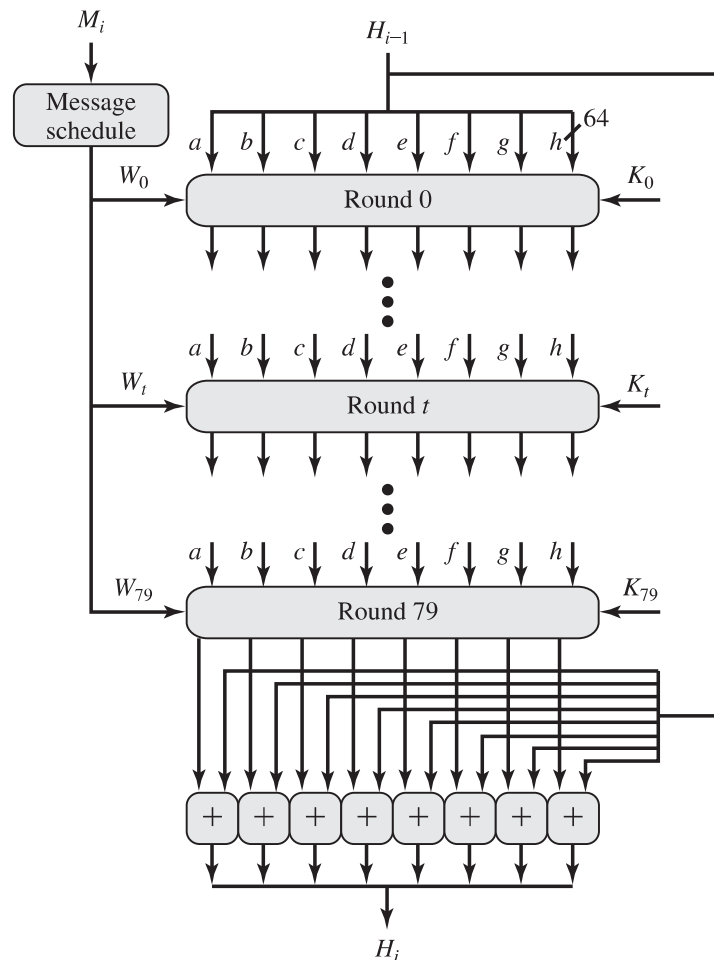


Figure 21.3 SHA-512 Processing of a Single 1024-Bit Block

The output of the eightieth round is added to the input to the first round (H_{i-1}) to produce H_i . The addition is done independently for each of the eight words in the buffer, with each of the corresponding words in H_{i-1} using addition modulo 2^{64} .

- **Step 5: Output.** After all N 1024-bit blocks have been processed, the output from the N th stage is the 512-bit message digest.

The SHA-512 algorithm has the property that every bit of the hash code is a function of every bit of the input. The complex repetition of the basic function F produces results that are well mixed; that is, it is unlikely that two messages chosen at random, even if they exhibit similar regularities, will have the same hash code. Unless there is some hidden weakness in SHA-512, which has not so far been published, the difficulty of coming up with two messages having the same message digest is on the order of 2^{256} operations, while the difficulty of finding a message with a given digest is on the order of 2^{512} operations.

SHA-3

SHA-2, particularly the 512-bit version, would appear to provide unassailable security. However, SHA-2 shares the same structure and mathematical operations as its predecessors, and this is a cause for concern. Because it would take years to find a suitable replacement for SHA-2, should it become vulnerable, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. The basic requirements that needed to be satisfied by any candidate for SHA-3 are the following:

1. It must be possible to replace SHA-2 with SHA-3 in any application by a simple drop-in substitution. Therefore, SHA-3 must support hash value lengths of 224, 256, 384, and 512 bits.
2. SHA-3 must preserve the online nature of SHA-2. That is, the algorithm must process comparatively small blocks (512 or 1024 bits) at a time instead of requiring that the entire message be buffered in memory before processing it.

After an extensive consultation and vetting process, NIST selected a winning submission and formally published SHA-3 as FIPS 202 (*SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015).

The structure and functions used for SHA-3 are substantially different from those shared by SHA-2 and SHA-1. Thus, if weaknesses are discovered in either SHA-2 or SHA-3, users have the option to switch to the other standard. SHA-2 has held up well and NIST considers it secure for general use. So for now, SHA-3 is a complement to SHA-2 rather than a replacement. The relatively compact nature of SHA-3 may make it useful for so-called “embedded” or smart devices that connect to electronic networks but are not themselves full-fledged computers. Examples include sensors in a building-wide security system and home appliances that can be controlled remotely. A detailed presentation of SHA-3 is provided in Appendix K.

21.2 HMAC

In this section, we look at the hash code approach to message authentication. Appendix E looks at message authentication based on block ciphers. In recent years, there has been increased interest in developing a MAC derived from a cryptographic hash code, such as SHA-1. The motivations for this interest are as follows:

- Cryptographic hash functions generally execute faster in software than conventional encryption algorithms such as DES.
- Library code for cryptographic hash functions is widely available.

A hash function such as SHA-1 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 [BELL96]. HMAC has been issued as RFC 2104 (*HMAC: Keyed-Hashing for Message Authentication*, 1997), has been chosen as the mandatory-to-implement MAC for IP Security, and is used in other Internet protocols, such as Transport Layer Security (TLS, soon to replace Secure Sockets Layer) and Secure Electronic Transaction (SET).

HMAC Design Objectives

RFC 2104 lists the following design objectives for HMAC:

- To use, without modifications, available hash functions—in particular, hash functions that perform well in software, and for which code is freely and widely available.
- To allow for easy replaceability 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 on 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 in an 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.

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

HMAC Algorithm

Figure 21.4 illustrates the overall operation of HMAC. Let us define the following terms:

- H = embedded hash function (e.g., SHA)
- M = message input to HMAC (including the padding specified in the embedded hash function)
- Y_i = i th block of M , $0 \leq i \leq (L - 1)$
- 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; if key length is greater than b , the key is input to the hash function to produce an n -bit key; recommended length is $\geq n$
- 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

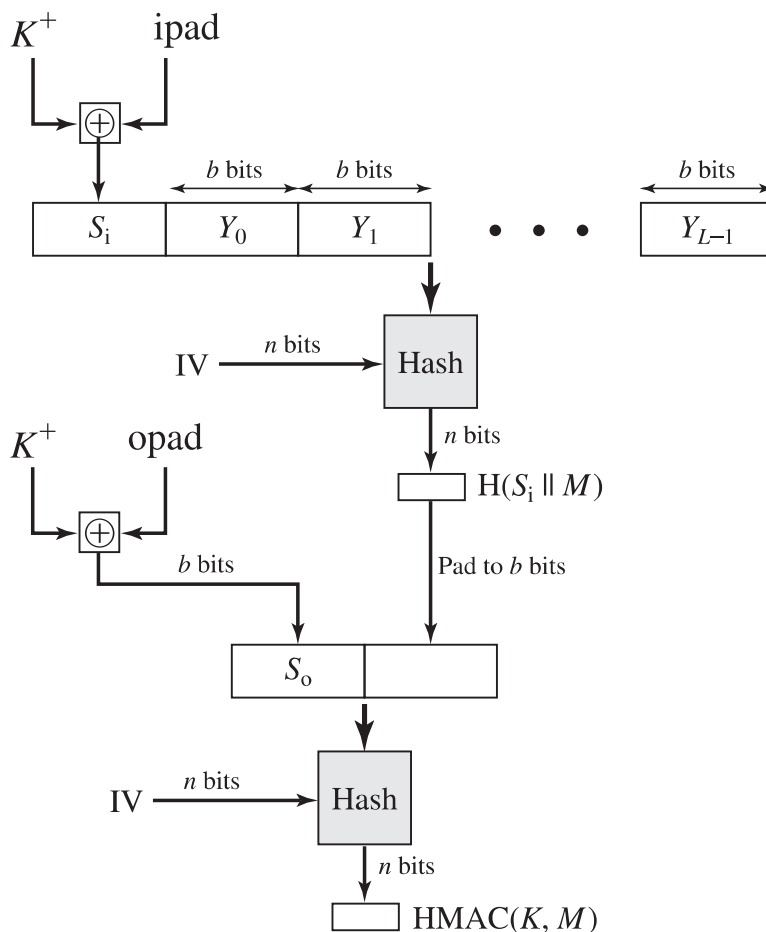


Figure 21.4 HMAC Structure

Then HMAC can be expressed as follows:

$$\text{HMAC}(K, M) = \text{H}[(K^+ \oplus \text{opad}) \parallel \text{H}[K^+ \oplus \text{ipad}] \parallel M]$$

In words,

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 zero bytes 0x00).
2. XOR (bitwise exclusive-OR) K^+ with ipad to produce the b -bit block S_i .
3. Append M to S_i .
4. Apply H to the stream generated in step 3.
5. XOR K^+ with opad to produce the b -bit block S_o .
6. Append the hash result from step 4 to S_o .
7. Apply H to the stream generated in step 6 and output the result.

Note the XOR with ipad results in flipping one-half of the bits of K . Similarly, the XOR with opad results in flipping one-half of the bits of K , but a different set of bits. In effect, by passing S_i and S_o through the hash algorithm, we have pseudorandomly generated two keys from K .

HMAC should execute in approximately the same time as the embedded hash function for long messages. HMAC adds three executions of the basic hash function (for S_i , S_o , and the block produced from the inner hash).

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.

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-MAC pairs created with the same key. In essence, it is proved in [BELL96] that for a given level of effort (time, message-MAC 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 hash: $\text{H}(M) = \text{H}(M')$. This is the birthday attack mentioned previously.

We have stated that this requires a level of effort of $2^{n/2}$ for a hash length of n . On this basis, the security of the earlier MD5 hash function was called into question, because a level of effort of 2^{64} 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 offline 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 offline 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 2^{64} observed blocks (2^{72} 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 acceptable to use MD5 rather than SHA as the embedded hash function for HMAC, although use of MD5 is now uncommon.

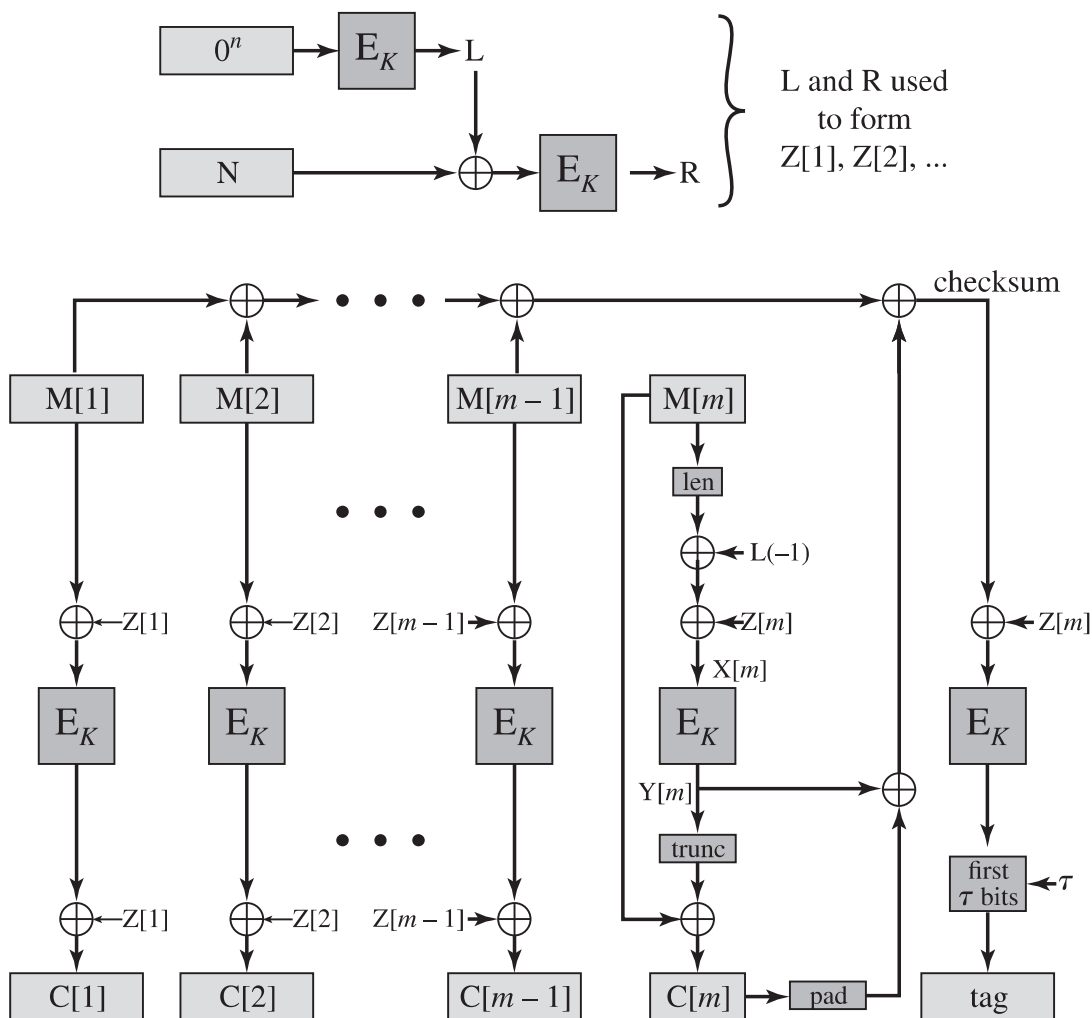
21.3 AUTHENTICATED ENCRYPTION

Authenticated encryption (AE) is a term used to describe encryption systems that simultaneously protect confidentiality and authenticity (integrity) of communications; that is, AE provides both message encryption and message authentication. Many applications and protocols require both forms of security, but until recently the two services have been designed separately. AE is implemented using a block cipher mode structure. One example that is used in a number of applications is CCM, described in Appendix E. In this section, we examine Offset Codebook (OCB) [ROGA03]. OCB is an NIST proposed block cipher mode of operation [ROGA01], and is a proposed Internet Standard defined in RFC 7253 (*The OCB Authenticated-Encryption Algorithm*, 2014). OCB is also approved as an authenticated encryption technique in the IEEE 802.11 wireless LAN standard. And, as mentioned in Chapter 13, OCB is included in MiniSec, the open-source IoT security module.

A key objective for OCB is efficiency. This is achieved by minimizing the number of encryptions required per message and by allowing for parallel operation on the blocks of a message.

Figure 21.5 shows the overall structure for OCB encryption and authentication. Typically, AES is used as the encryption algorithm. The message M to be encrypted and authenticated is divided into n -bit blocks, with the exception of the last block, which may be less than n bits. Typically, $n = 128$. Only a single pass through the message is required to generate both the ciphertext and the authentication code. The total number of blocks is $m = \lceil \text{len}(M)/n \rceil$.

Note the encryption structure for OCB is similar to that of electronic codebook (ECB) mode. Each block is encrypted independently of the other blocks, so that it is possible to perform all m encryptions simultaneously. As was mentioned in Chapter 20, with ECB, if the same b -bit block of plaintext appears more than once in the message, it always produces the same ciphertext. Because of this, for lengthy messages, the ECB mode may not be secure. OCB eliminates this property by using an offset $Z[i]$ for each



n = block length in bits

N = nonce

$\text{len}(M[m])$ = length of $M[m]$ represented as an n -bit integer

$\text{trunc}(Y[m])$ = deletes least significant bits so that result is same length as $M[m]$

pad = pad with least significant 0 bits to length n

τ = length of authentication tag

Figure 21.5 OCB Encryption and Authentication

block $M[i]$, such that each $Z[i]$ is unique; the offset is XORed with the plaintext and XORed again with the encrypted output. Thus, with encryption key K we have

$$C[i] = E_K(M[i] \oplus Z[i]) \oplus Z[i]$$

where $E_K(X)$ is the encryption of plaintext X using key K , and \oplus is the exclusive-OR operation. Because of the use of the offset, two blocks in the same message that are identical will produce two different ciphertexts.

The upper part of Figure 21.5 indicates how the $Z[i]$ s are generated. An arbitrary n -bit value N called the nonce is chosen; the only requirement is that if multiple messages are encrypted with the same key, a different nonce must be used each time such

that each nonce is only used once. Each different value of N will produce a different set of $Z[i]$. Thus, if two different messages have identical blocks in the same position in the message, they will produce different ciphertexts because the $Z[i]$ will be different.

The calculation of the $Z[i]$ is somewhat complex and is summarized in the following equations:

$$\begin{aligned} L(0) &= L = E_K(0^n) && \text{where } 0^n \text{ consists of } n \text{ zero bits.} \\ R &= E_K(N \oplus L) \\ L(i) &= 2 \cdot L(i-1) && 1 \leq i \leq m \\ Z[1] &= L \oplus R \\ Z[i] &= Z(i-1) \oplus L(\text{ntz}(i)) && 1 \leq i \leq m \end{aligned}$$

The operator \cdot refers to multiplication over the finite field $\text{GF}(2^n)$; a discussion of finite fields is beyond our scope and is covered in [STAL17]. The operator $\text{ntz}(i)$ denotes the number of trailing (least significant) zeros in i . The resulting $Z[i]$ values are a maximal Hamming distance apart [WALK05].

Thus, the values $Z[i]$ are a function of both the nonce and the encryption key. The nonce does not need to be kept secret and is communicated to the recipient in a manner outside the scope of the specification.

Because the length of M may not be an integer multiple of n , the final block is treated differently, as shown in Figure 21.5. The length of $M[m]$, represented as an n -bit integer, is used to calculate $X[m] = \text{len}(M[m]) \oplus L(-1) \oplus Z[m]$. $L(-1)$ is defined as $L/2$ over the finite field or, equivalently, $L \cdot 2^{-1}$. Next, $Y[m] = E_K(X[m])$. Then, $Y[m]$ is truncated to $\text{len}(M[m])$ bits (by deleting the necessary number of least significant bits) and XORed with $M[m]$. Thus, the final ciphertext C is the same length as the original plaintext M .

A checksum is produced from the message M as follows:

$$\text{checksum} = M[1] \oplus M[2] \oplus \dots \oplus Y[m] \oplus C[m]0^*$$

Where $C[m]0^*$ consists of $C[m]$ padded with least significant bits to the length n . Finally, an authentication tag of length τ is generated, using the same key as is used for encryption:

$$\text{tag} = \text{first } \tau \text{ bits of } E_K(\text{checksum} \oplus Z[m])$$

The bit length τ of the tag varies according to the application. The size of the tag controls the level of authentication. To verify the authentication tag, the decryptor can recompute the checksum, then recompute the tag, and finally check that is equal to the one that was sent. If the ciphertext passes the test, then OCB produces the plaintext normally.

Figure 21.6 summarizes the OCB algorithms for encryption and decryption. It is easy to see that decryption is the inverse of encryption. We have

$$\begin{aligned} E_K(M[i] \oplus Z[i]) \oplus Z[i] &= C[i] \\ E_K(M[i] \oplus Z[i]) &= C[i] \oplus Z[i] \\ D_K(E_K(M[i] \oplus Z[i])) &= D_K(C[i] \oplus Z[i]) \\ M[i] \oplus Z[i] &= D_K(C[i] \oplus Z[i]) \\ M[i] &= D_K(C[i] \oplus Z[i]) \oplus Z[i] \end{aligned}$$

<p>algorithm OCB-Encrypt$_K(N, M)$ Partition M into $M[1] \dots M[m]$ $L \leftarrow L(0) \leftarrow E_K(0^n)$ $R \leftarrow E_K(N \oplus L)$ for $i \leftarrow 1$ to m do $L(i) \leftarrow 2 \cdot L(i - 1)$ $L(-1) = L \cdot 2^{-1}$ $Z[1] \leftarrow L \oplus R$ for $i \leftarrow 2$ to m do $Z[i] \leftarrow Z[i - 1] \oplus L(\text{ntz}(i))$ for $i \leftarrow 1$ to $m - 1$ do $C[i] \leftarrow E_K(M[i] \oplus Z[i]) \oplus Z[i]$ $X[m] \leftarrow \text{len}(M[m]) \oplus L(-1) \oplus Z[m]$ $Y[m] \leftarrow E_K(X[m])$ $C[m] \leftarrow M[m] \oplus (\text{first } \text{len}(M[m]) \text{ bits of } Y[m])$ Checksum \leftarrow $M[1] \oplus \dots \oplus M[m - 1] \oplus C[m]0^* \oplus Y[m]$ Tag $\leftarrow E_K(\text{Checksum} \oplus Z[m])$ [first τ bits]</p>	<p>algorithm OCB-Decrypt$_K(N, M)$ Partition M into $M[1] \dots M[m]$ $L \leftarrow L(0) \leftarrow E_K(0^n)$ $R \leftarrow E_K(N \oplus L)$ for $i \leftarrow 1$ to m do $L(i) \leftarrow 2 \cdot L(i - 1)$ $L(-1) = L \cdot 2^{-1}$ $Z[1] \leftarrow L \oplus R$ for $i \leftarrow 2$ to m do $Z[i] \leftarrow Z[i - 1] \oplus L(\text{ntz}(i))$ for $i \leftarrow 1$ to $m - 1$ do $M[i] \leftarrow D_K(C[i] \oplus Z[i]) \oplus Z[i]$ $X[m] \leftarrow \text{len}(M[m]) \oplus L(-1) \oplus Z[m]$ $Y[m] \leftarrow E_K(X[m])$ $M[m] \leftarrow (\text{first } \text{len}(C[m]) \text{ bits of } Y[m]) \oplus C[m]$ Checksum \leftarrow $M[1] \oplus \dots \oplus M[m - 1] \oplus C[m]0^* \oplus Y[m]$ Tag' $\leftarrow E_K(\text{Checksum} \oplus Z[m])$ [first τ bits]</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 21.6 OCB Algorithms

21.4 THE RSA PUBLIC-KEY ENCRYPTION ALGORITHM

Perhaps the most widely used public-key algorithms are RSA and Diffie-Hellman. We examine RSA plus some security considerations in this section.¹ Diffie-Hellman is covered in Section 21.5.

Description of the Algorithm

One of the first public-key schemes was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78]. The RSA scheme has since that time reigned supreme as the most widely accepted and implemented approach to public-key encryption. RSA is a block cipher in which the plaintext and ciphertext are integers between 0 and $n - 1$ for some n .

Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C :

$$C = M^e \bmod n$$

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Both sender and receiver must know the values of n and e , and only the receiver knows the value of d . 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 requirements must be met:

1. It is possible to find values of e, d, n such that $M^{ed} \bmod n = M$ for all $M < n$.

¹This section uses some elementary concepts from number theory. For a review, see Appendix B.

2. It is relatively easy to calculate M^e and C^d for all values of $M < n$.
3. It is infeasible to determine d given e and n .

The first two requirements are easily met. The third requirement can be met for large values of e and n .

More should be said about the first requirement. We need to find a relationship of the form

$$M^{ed} \bmod n = M$$

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 Appendix B that for p, q prime, $\phi(pq) = (p - 1)(q - 1)$. $\phi(n)$, referred to as the Euler totient of n , is the number of positive integers less than n and relatively prime to n . The relationship between e and d can be expressed as

$$ed \bmod \phi(n) = 1$$

This is equivalent to saying

$$\begin{aligned} ed \bmod \phi(n) &= 1 \\ d \bmod \phi(n) &= e^{-1} \end{aligned}$$

That is, e and d are multiplicative inverses mod $\phi(n)$. According to the rules of modular arithmetic, this is true only if d (and therefore e) is relatively prime to $\phi(n)$. Equivalently, $\gcd(\phi(n), d) = 1$; that is, the greatest common divisor of $\phi(n)$ and d is 1.

Figure 21.7 summarizes the RSA algorithm. Begin by selecting two prime numbers, p and q , and calculating their product n , which is the modulus for encryption and

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p - 1)(q - 1)$	
Select integer e	$\gcd(\phi(n), e) = 1$; $1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

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

Decryption	
Ciphertext:	C
Plaintext:	$M = C^d \pmod n$

Figure 21.7 The RSA Algorithm

decryption. Next, we need the quantity $\phi(n)$. Then select an integer e that is relatively prime to $\phi(n)$ [i.e., the greatest common divisor of e and $\phi(n)$ is 1]. Finally, calculate d as the multiplicative inverse of e , modulo $\phi(n)$. It can be shown that d and e have the desired properties.

Suppose user A has published its public key and user B wishes to send the message M to A. Then B calculates $C = M^e \pmod{n}$ and transmits C . On receipt of this ciphertext, user A decrypts by calculating $M = C^d \pmod{n}$.

An example, from [SING99], is shown in Figure 21.8. For this example, the keys were generated as follows:

1. Select two prime numbers, $p = 17$ and $q = 11$.
2. Calculate $n = pq = 17 \times 11 = 187$.
3. Calculate $\phi(n) = (p - 1)(q - 1) = 16 \times 10 = 160$.
4. Select e such that e is relatively prime to $\phi(n) = 160$ and less than $\phi(n)$; we choose $e = 7$.
5. Determine d such that $de \pmod{160} = 1$ and $d < 160$. The correct value is $d = 23$, because $23 \times 7 = 161 = (1 \times 160) + 1$.

The resulting keys are public key $PU = \{7, 187\}$ and private key $PR = \{23, 187\}$. The example shows the use of these keys for a plaintext input of $M = 88$. For encryption, we need to calculate $C = 88^7 \pmod{187}$. Exploiting the properties of modular arithmetic, we can do this as follows:

$$88^7 \pmod{187} = [(88^4 \pmod{187}) \times (88^2 \pmod{187}) \times (88^1 \pmod{187})] \pmod{187}$$

$$88^1 \pmod{187} = 88$$

$$88^2 \pmod{187} = 7744 \pmod{187} = 77$$

$$88^4 \pmod{187} = 59,969,536 \pmod{187} = 132$$

$$88^7 \pmod{187} = (88 \times 77 \times 132) \pmod{187} = 894,432 \pmod{187} = 11$$

For decryption, we calculate $M = 11^{23} \pmod{187}$:

$$11^{23} \pmod{187} = [(11^1 \pmod{187}) \times (11^2 \pmod{187}) \times (11^4 \pmod{187}) \times (11^8 \pmod{187}) \times (11^8 \pmod{187})] \pmod{187}$$

$$11^1 \pmod{187} = 11$$

$$11^2 \pmod{187} = 121$$

$$11^4 \pmod{187} = 14,641 \pmod{187} = 55$$

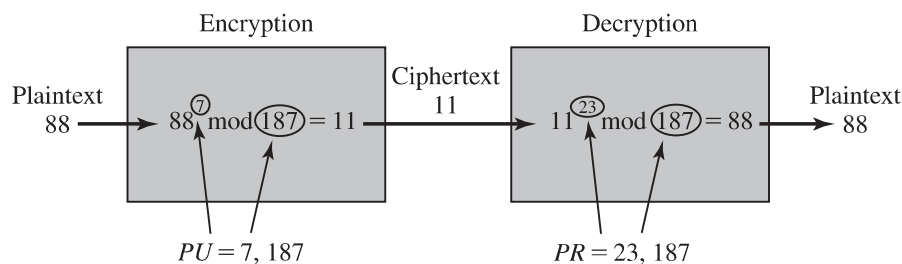


Figure 21.8 Example of RSA Algorithm

$$11^8 \bmod 187 = 214,358,881 \bmod 187 = 33$$

$$11^{23} \bmod 187 = (11 \times 121 \times 55 \times 33 \times 33) \bmod 187 = 79,720,245$$

$$\bmod 187 = 88$$

The Security of RSA

Four possible approaches to attacking the RSA algorithm are as follows:

- **Brute force:** This involves trying all possible private keys.
- **Mathematical attacks:** There are several approaches, all equivalent in effort to factoring the product of two primes.
- **Timing attacks:** These depend on the running time of the decryption algorithm.
- **Chosen ciphertext attacks:** This type of attack exploits properties of the RSA algorithm. A discussion of this attack is beyond the scope of this book.

The defense against the brute force approach is the same for RSA as for other cryptosystems; namely, use a large key space. Thus, the larger the number of bits in d , the better. However, because the calculations involved, both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run.

In this subsection, we provide an overview of mathematical and timing attacks.

THE FACTORING PROBLEM We can identify three approaches to attacking RSA mathematically:

- Factor n into its two prime factors. This enables calculation of $\phi(n) = (p - 1) \times (q - 1)$, which, in turn, enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
- Determine $\phi(n)$ directly, without first determining p and q . Again, this enables determination of $d \equiv e^{-1} \pmod{\phi(n)}$.
- Determine d directly, without first determining $\phi(n)$.

Most discussions of the cryptanalysis of RSA have focused on the task of factoring n into its two prime factors. Determining $\phi(n)$ given n is equivalent to factoring n [RIBE96]. With presently known algorithms, determining d given e and n appears to be at least as time consuming as the factoring problem. Hence, we can use factoring performance as a benchmark against which to evaluate the security of RSA.

For a large n with large prime factors, factoring is a hard problem, but not as hard as it used to be. Just as it had done for DES, RSA Laboratories issued challenges for the RSA cipher with key sizes of 100, 110, 120, and so on, digits. The latest challenge to be met is the RSA-768 challenge with a key length of 232 decimal digits, or 768 bits. Table 21.2 shows the results to date.

A striking fact about Table 21.2 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

Table 21.2 Progress in Factorization

Number of Decimal Digits	Number of Bits	Date Achieved
100	332	April 1991
110	365	April 1992
120	398	June 1993
129	428	April 1994
130	431	April 1996
140	465	February 1999
155	512	August 1999
160	530	April 2003
174	576	December 2003
200	663	May 2005
193	640	November 2005
232	768	December 2009

further refinements in the GNFS, and the use of an even better algorithm is also a possibility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve. It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better. Thus, we need to be careful in choosing a key size for RSA. For the near future, a key size in the range of 1024 to 2048 bits seems secure.

In addition to specifying the size of n , a number of other constraints have been suggested by researchers. To avoid values of n that may be factored more easily, the algorithm's inventors suggest the following constraints on p and q :

1. p and q should differ in length by only a few digits. Thus, for a 1024-bit key (309 decimal digits), both p and q should be on the order of magnitude of 10^{75} to 10^{100} .
2. Both $(p - 1)$ and $(q - 1)$ should contain a large prime factor.
3. $\text{gcd}(p - 1, q - 1)$ should be small.

In addition, it has been demonstrated that if $e < n$ and $d < n^{1/4}$, then d can be easily determined [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]. Timing attacks are applicable not just to RSA, but also 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. The attack exploits the common use of a modular exponentiation algorithm in RSA encryption and decryption, but the attack can be adapted to work with any implementation that does not run in fixed time. In the modular exponentiation algorithm, exponentiation is accomplished bit by bit, with one modular multiplication performed at each iteration and an additional modular multiplication performed for each 1 bit.

As Kocher points out in his paper, the attack is simplest to understand in an extreme case. Suppose the target system uses a modular multiplication function that is very fast in almost all cases but in a few cases takes much more time than an entire average modular exponentiation. The attack proceeds bit-by-bit starting with the leftmost bit, b_k . Suppose 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. The operation of the subsequent step depends on the unknown exponent bit. If the bit is set, $d \leftarrow (d \times a) \bmod n$ will be executed. For a few values of a and d , the modular multiplication will be extremely slow, and the attacker knows which these are. Therefore, if the observed time to execute the decryption algorithm is always slow when this particular iteration is slow with a 1 bit, then this bit is assumed to be 1. If a number of observed execution times for the entire algorithm are fast, then this bit is assumed to be 0.

In practice, modular exponentiation implementations do not have such extreme timing variations, in which the execution time of a single iteration can exceed the mean execution time of the entire algorithm. Nevertheless, there is enough variation to make this attack practical. For details, see [KOCH96].

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 do not add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.
- **Blinding:** Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.

RSA Data Security incorporates a blinding feature into some of its products. The private-key operation $M = C^d \bmod n$ is implemented as follows:

1. Generate a secret random number r between 0 and $n - 1$.
2. Compute $C' = C(r^e) \bmod n$, where e is the public exponent.
3. Compute $M' = (C')^d \bmod n$ with the ordinary RSA implementation.
4. Compute $M = M'r^{-1} \bmod n$. In this equation, r^{-1} is the multiplicative inverse of $r \bmod n$. It can be demonstrated that this is the correct result by observing that $r^{ed} \bmod n = r \bmod n$.

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

21.5 DIFFIE-HELLMAN AND OTHER ASYMMETRIC ALGORITHMS

Diffie-Hellman Key Exchange

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography [DIFF76] and is generally referred to as the Diffie-Hellman key exchange. A number of commercial products employ this key exchange technique.

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

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

$$a \bmod p, a^2 \bmod p, \dots, a^{p-1} \bmod p$$

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

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

$$b = a^i \bmod p \quad \text{where } 0 \leq i \leq (p - 1)$$

The exponent i is referred to as the discrete logarithm, or index, of b for the base a , mod p . We denote this value as $\text{dlog}_{a,p}(b)$.²

THE ALGORITHM With this background, we can define the Diffie-Hellman key exchange, which is summarized in Figure 21.9. For this scheme, there are two publicly known numbers: a prime number q , and an integer α that is a primitive root of q . Suppose the users A and B wish to exchange a key. User A selects a random integer $X_A < q$ and computes $Y_A = \alpha^{X_A} \bmod q$. Similarly, user B independently selects a random integer $X_B < q$ and computes $Y_B = \alpha^{X_B} \bmod q$. Each side keeps the X value private and makes the Y value available publicly to the other side. User A computes the key as $K = (Y_B)^{X_A}$ and user B computes the key as $K = (Y_A)^{X_B} \bmod q$. These two calculations produce identical results:

$$\begin{aligned} K &= (Y_B)^{X_A} \bmod q \\ &= (\alpha^{X_B} \bmod q)^{X_A} \bmod q \\ &= (\alpha^{X_B})^{X_A} \bmod q \\ &= \alpha^{X_B X_A} \bmod q \\ &= (\alpha^{X_A})^{X_B} \bmod q \\ &= (\alpha^{X_A} \bmod q)^{X_B} \bmod q \\ &= (Y_A)^{X_B} \bmod q \end{aligned}$$

²Many texts refer to the discrete logarithm as the *index*. There is no generally agreed notation for this concept, much less an agreed name.

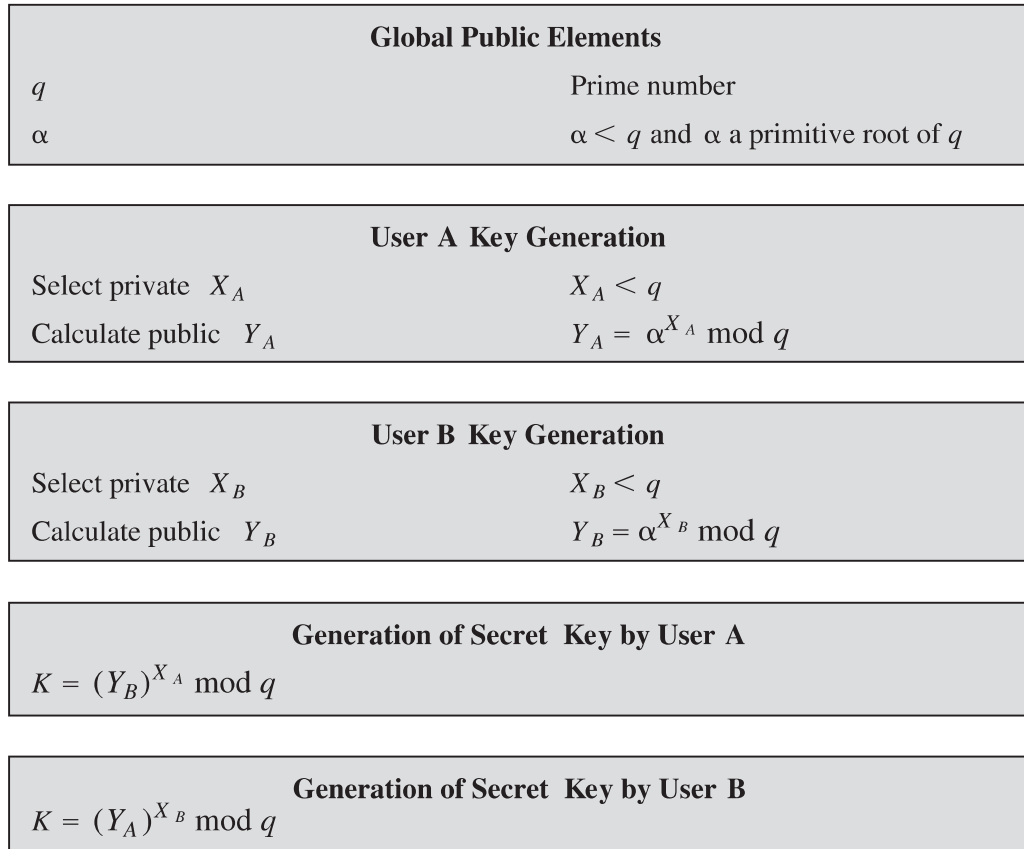


Figure 21.9 The Diffie-Hellman Key Exchange Algorithm

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

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

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

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

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

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

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

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

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

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

We assume an attacker would have available the following information:

$$q = 353; \alpha = 3; Y_A = 40; Y_B = 248$$

In this simple example, it would be possible by brute force to determine the secret key 160. In particular, an attacker E can determine the common key by discovering a solution to the equation $3^a \text{ mod } 353 = 40$ or the equation $3^b \text{ mod } 353 = 248$. The brute force approach is to calculate powers of 3 modulo 353, stopping when the result equals either 40 or 248. The desired answer is reached with the exponent value of 97, which provides $3^{97} \text{ mod } 353 = 40$.

With larger numbers, the problem becomes impractical.

KEY EXCHANGE PROTOCOLS Figure 21.10 shows a simple protocol that makes use of the Diffie-Hellman calculation. Suppose user A wishes to set up a connection with user B, and use a secret key to encrypt messages on that connection. User A can generate a one-time private key X_A , calculate Y_A , and send that to user B. User B responds by generating a private value X_B , calculating Y_B , and sending Y_B to user A. Both users can now calculate the key. The necessary public values q and α would need to be known ahead of time. Alternatively, user A could pick values for q and α and include those in the first message.

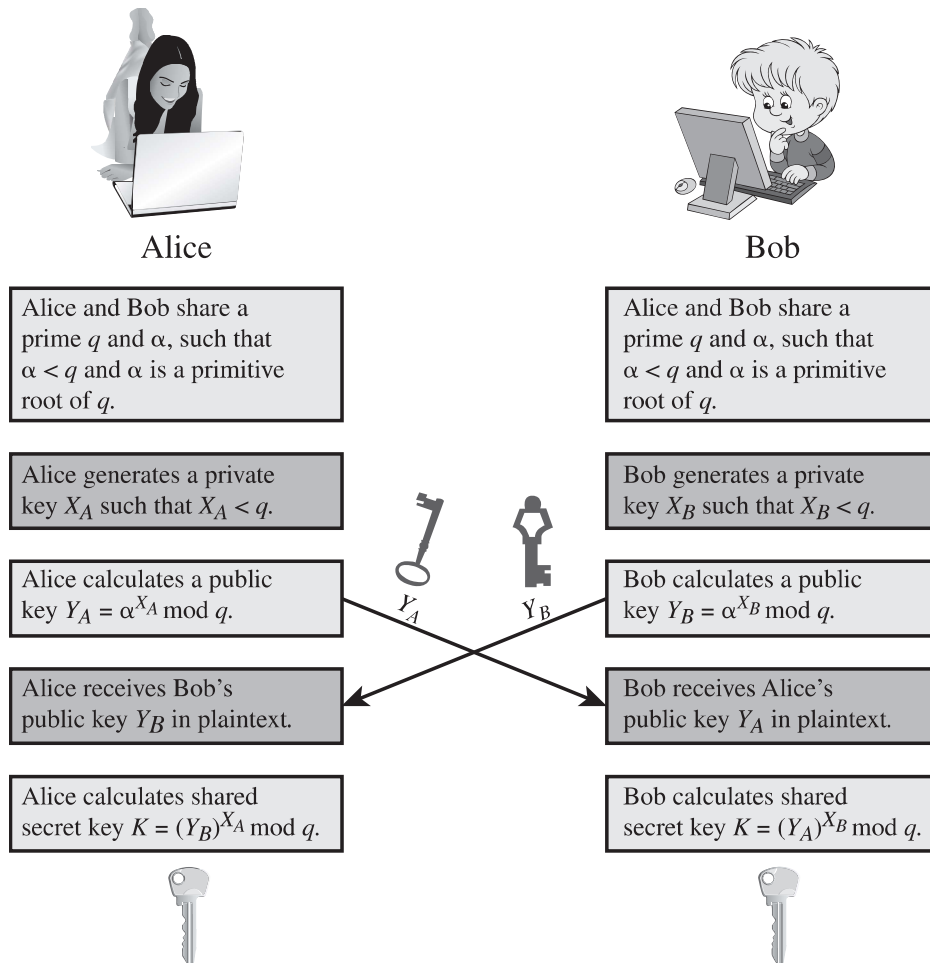


Figure 21.10 Diffie-Hellman Key Exchange

As an example of another use of the Diffie-Hellman algorithm, suppose in a group of users (e.g., all users on a LAN), each generates a long-lasting private key and calculates a public key. These public values, together with global public values for q and α , are stored in some central directory. At any time, user B can access user A's public value, calculate a secret key, and use that to send an encrypted message to user A. If the central directory is trusted, then this form of communication provides both confidentiality and a degree of authentication. Because only A and B can determine the key, no other user can read the message (confidentiality). User A knows that only user B could have created a message using this key (authentication). However, the technique does not protect against replay attacks.

MAN-IN-THE-MIDDLE ATTACK The protocol depicted in Figure 21.10 is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is the adversary. The attack proceeds as follows:

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

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

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

The key exchange protocol is vulnerable to such an attack because it does not authenticate the participants. This vulnerability can be overcome with the use of digital signatures and public-key certificates; these topics are explored later in this chapter, and in Chapter 2.

Other Public-Key Cryptography Algorithms

Two other public-key algorithms have found commercial acceptance: DSS, and elliptic-curve cryptography.

DIGITAL SIGNATURE STANDARD The National Institute of Standards and Technology (NIST) has published this as Federal Information Processing Standard FIPS 186-4 [*Digital Signature Standard (DSS)*, July 2013]. The DSS makes use of the SHA-1 and

presents a new digital signature technique, the Digital Signature Algorithm (DSA). The DSS was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There were further minor revisions in 1996 and 2013. 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.

ELLIPTIC-CURVE CRYPTOGRAPHY The vast majority of the products and standards that use public-key cryptography for encryption and digital signatures use RSA. The bit length for secure RSA use has increased over recent years, and this has put a heavier processing load on applications using RSA. This burden has ramifications, especially for electronic commerce sites that conduct large numbers of secure transactions. Recently, a competing system has begun to challenge RSA: elliptic curve cryptography (ECC). Already, ECC is showing up in standardization efforts, including the IEEE P1363 Standard for Public-Key Cryptography. A version of ECC used for digital signature is included as an option in FIPS 186-4.

The principal attraction of ECC compared to RSA is that it appears to offer equal security for a far smaller bit 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. Thus, 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. The technique is based on the use of a mathematical construct known as the elliptic curve.

21.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Diffie-Hellman key exchange digital signature Digital Signature Standard (DSS) elliptic-curve cryptography (ECC) HMAC key exchange MD5	message authentication message authentication code (MAC) message digest one-way hash function private key public key public-key certificate public-key encryption	RSA secret key Secure Hash Algorithm (SHA) secure hash function SHA-1 SHA-2 SHA-3 strong collision resistance weak collision resistance
----------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------

Review Questions

- 21.1 In the context of a hash function, what is a compression function?
- 21.2 Why is a message always padded before hashing with SHA-1 even when the message is already a multiple of the block length?
- 21.3 What are the minimum requirements for HMAC to be provably secure?

- 21.4 What is a one-way function?
 21.5 Briefly explain Diffie-Hellman key exchange.

Problems

- 21.1 Consider a 32-bit hash function defined as the concatenation of two 16-bit functions: XOR and RXOR, defined in Section 21.2 as “two simple hash functions.”
- Will this checksum detect all errors caused by an odd number of error bits? Explain.
 - Will this checksum detect all errors caused by an even number of error bits? If not, characterize the error patterns that will cause the checksum to fail.
 - Comment on the effectiveness of this function for use as a hash function for authentication.
- 21.2
- Consider the following hash function. Messages are in the form of a sequence of decimal numbers, $M = (a_1, a_2, \dots, a_t)$. The hash value h is calculated as $H(M) = \prod_{i=1}^t a_i \bmod n$, for some predefined value n . Does this hash function satisfy the requirements for a hash function H listed in Section 2.2? Explain your answer.
 - Repeat part (a) for the hash function $H(M) = \prod_{i=1}^t (a_i)^2 \bmod n$.
 - Calculate the hash function of part (b) for $M = (18, 63, 90, 72, 34)$ and $n = 99$.
- 21.3 It is possible to use a hash function to construct a block cipher with a structure similar to DES. Because a hash function is one way and a block cipher must be reversible (to decrypt), how is it possible?
- 21.4 Now consider the opposite problem: using an encryption algorithm to construct a one-way hash function. Consider using RSA with a known key. Then process a message consisting of a sequence of blocks as follows: Encrypt the first block, XOR the result with the second block and encrypt again, and so on. Show that this scheme is not secure by solving the following problem. Given a two-block message B1, B2, and its hash

$$\text{RSAH}(B1, B2) = \text{RSA}(\text{RSA}(B1) \oplus B2)$$

and given an arbitrary block C1, choose C2 so that $\text{RSAH}(C1, C2) = \text{RSAH}(B1, B2)$. Thus, the hash function does not satisfy weak collision resistance.

- 21.5 Figure 21.11 shows an alternative means of implementing HMAC.
- Describe the operation of this implementation.
 - What potential benefit does this implementation have over that shown in Figure 21.4?
- 21.6 Perform encryption and decryption using the RSA algorithm, as in Figure 21.8, for the following:
- $p = 13; q = 31, e = 19; M = 2$
 - $p = 11; q = 31, e = 7; M = 4$
 - $p = 3; q = 17, e = 5; M = 5$
 - $p = 5; q = 17, e = 7; M = 6$
 - $p = 7; q = 17, e = 29; M = 3$

Hint: Decryption is not as hard as you think; use some finesse.

- 21.7 In a public-key system using RSA, you intercept the ciphertext $C = 61$ sent to a user whose public key is $e = 11, n = 91$. What is the plaintext M ?
- 21.8 In an RSA system, the public key of Harry is as follows: $e = 47, n = 4757$. What is the private key of Harry?
- 21.9 In the RSA cryptosystem, it is obvious that if one can factor RSA modulus $n = pq$, then one can compute $\phi(n) = (p - 1)(q - 1)$. As a result, one can also compute the secret key d since $ed \equiv 1 \pmod{\phi(n)}$. Prove the converse that if both n and $\phi(n)$ are known, then one can find p and q without factoring n .

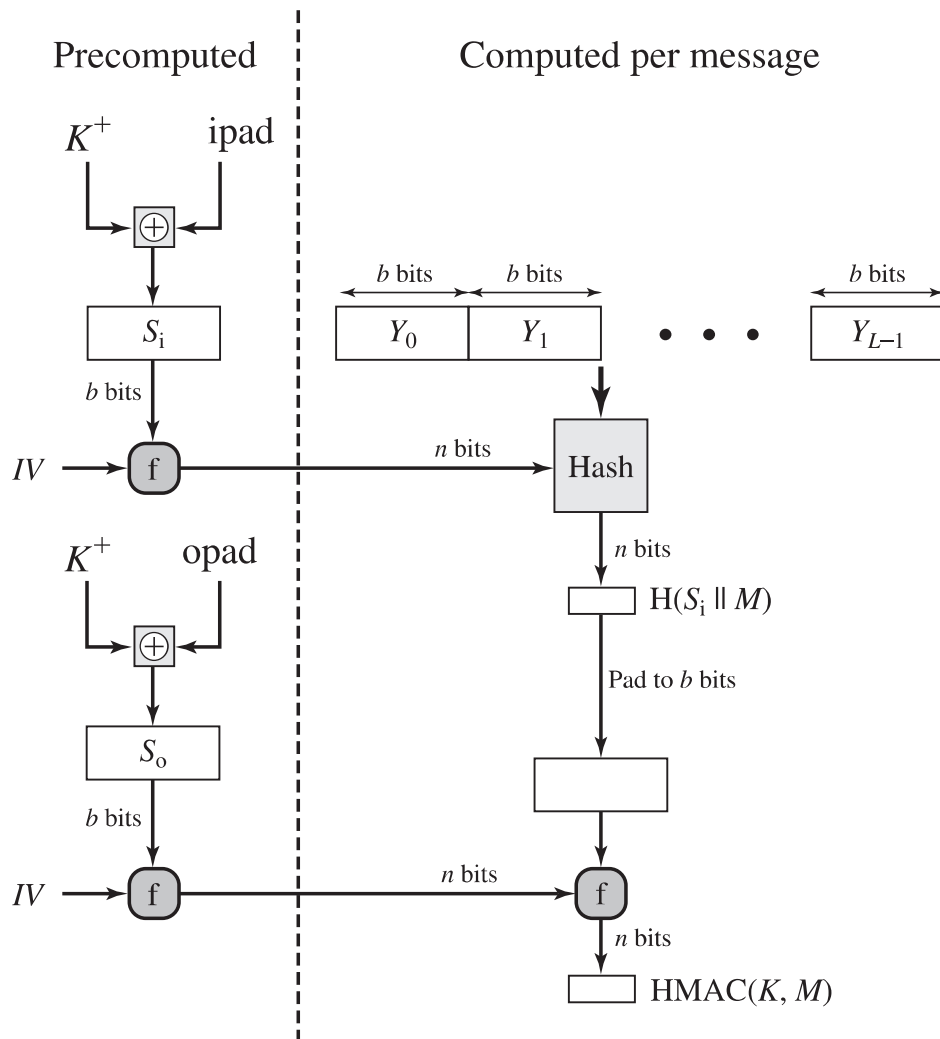


Figure 21.11 Alternative Implementation of HMAC

21.10 Consider the following scheme:

1. Pick an odd number, E .
2. Pick two prime numbers, P and Q , where $(P - 1)(Q - 1) - 1$ is evenly divisible by E .
3. Multiply P and Q to get N .
4. Calculate $D = \frac{(P - 1)(Q - 1)(E - 1) + 1}{E}$.

Is this scheme equivalent to RSA? Show why or why not.

- 21.11** Suppose Bob uses the RSA cryptosystem with a very large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends a message to Bob by representing each alphabetic character as an integer between 0 and 25 ($A \rightarrow 0, \dots, Z \rightarrow 25$), and then encrypting each number separately using RSA with large e and large n . Is this method secure? If not, describe the most efficient attack against this encryption method.
- 21.12** Consider a Diffie-Hellman scheme with a common prime $q = 23$ and a primitive root $\alpha = 5$.
- a. Alice has public key $Y_A = 10$, what is Alice's private key X_A ?
 - b. Bob has public key $Y_B = 8$, what is the shared secret key K ?

PART FIVE: Network Security

CHAPTER

22

INTERNET SECURITY PROTOCOLS AND STANDARDS

22.1 Secure E-mail and S/MIME

- MIME
- S/MIME

22.2 DomainKeys Identified Mail

- Internet Mail Architecture
- DKIM Strategy

22.3 Secure Sockets Layer (SSL) and Transport Layer Security (TLS)

- TLS Architecture
- TLS Protocols
- TLS Attacks
- SSL/TLS Attacks

22.4 HTTPS

- Connection Initiation
- Connection Closure

22.5 IPv4 and IPv6 Security

- IP Security Overview
- The Scope of IPsec
- Security Associations
- Encapsulating Security Payload
- Transport and Tunnel Modes

22.6 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Provide an overview of MIME.
- ◆ Understand the functionality of S/MIME and the security threats it addresses.
- ◆ Explain the key components of SSL.
- ◆ Discuss the use of HTTPS.
- ◆ Provide an overview of IPsec.
- ◆ Discuss the format and functionality of the Encapsulating Security Payload.

This chapter looks at some of the most widely used and important Internet security protocols and standards.

22.1 SECURE E-MAIL AND S/MIME

S/MIME (Secure/Multipurpose Internet Mail Extension) is a security enhancement to the MIME Internet e-mail format standard.

MIME

MIME is an extension to the old RFC 822 (*Standard For The Format Of ARPA Internet Text Messages*, 1982): specification of an Internet mail format. RFC 822 defines a simple header with To, From, Subject, and other fields that can be used to route an e-mail message through the Internet and that provides basic information about the e-mail content. RFC 822 assumes a simple ASCII text format for the content.

MIME provides a number of new header fields that define information about the body of the message, including the format of the body and any encoding that is done to facilitate transfer. Most important, MIME defines a number of content formats, which standardize representations for the support of multimedia e-mail. Examples include text, image, audio, and video.

S/MIME

S/MIME is a complex capability that is defined in a number of documents. The most important documents relevant to S/MIME include the following:

- **RFC 5750** (*S/MIME Version 3.2 Certificate Handling*, 2010): Specifies conventions for X.509 certificate usage by (S/MIME) v3.2.
- **RFC 5751** (*S/MIME Version 3.2 Message Specification*, 2010): The principal defining document for S/MIME message creation and processing.
- **RFC 4134** (*Examples of S/MIME Messages*, 2005): Gives examples of message bodies formatted using S/MIME.
- **RFC 2634** (*Enhanced Security Services for S/MIME*, 1999): Describes four optional security service extensions for S/MIME.

- **RFC 5652** (*Cryptographic Message Syntax (CMS)*, 2009): The Cryptographic Message Syntax is used to digitally sign, digest, authenticate, or encrypt arbitrary message content.
- **RFC 3370** (*CMS Algorithms*, 2002): Describes the conventions for using several cryptographic algorithms with the CMS.
- **RFC 5752** (*Multiple Signatures in CMS*, 2010): Describes the use of multiple, parallel signatures for a message.
- **RFC 1847** (*Security Multiparts for MIME—Multipart/Signed and Multipart/Encrypted*, 1995): Defines a framework within which security services may be applied to MIME body parts. The use of a digital signature is relevant to S/MIME, as explained subsequently.

S/MIME functionality is built into the majority of modern e-mail software and interoperates between them. S/MIME is defined as a set of additional MIME content types (see Table 22.1) and provides the ability to sign and/or encrypt e-mail messages. In essence, these content types support four new functions:

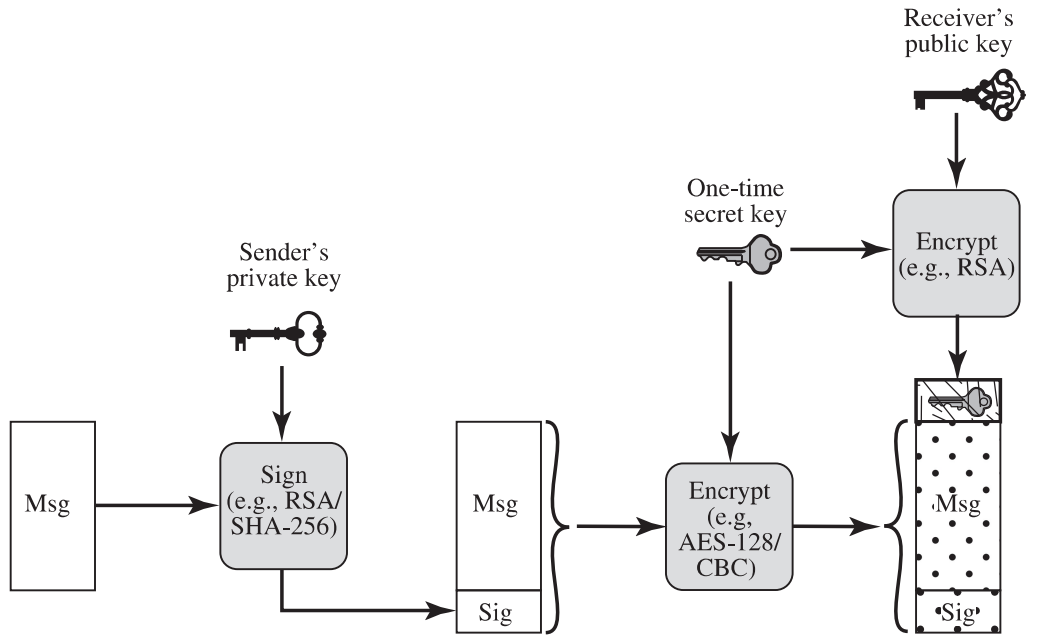
- **Enveloped data:** Consists of encrypted content of any type and encrypted-content encryption keys for one or more recipients.
- **Signed data:** A digital signature is formed by taking the message digest of the content to be signed, then encrypting that with the private key of the signer. The content 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 message content, although they cannot verify the signature.
- **Signed and enveloped data:** Signed-only and encrypted-only entities may be nested, so encrypted data may be signed, and signed data or clear-signed data may be encrypted.

Table 22.1 S/MIME Content Types

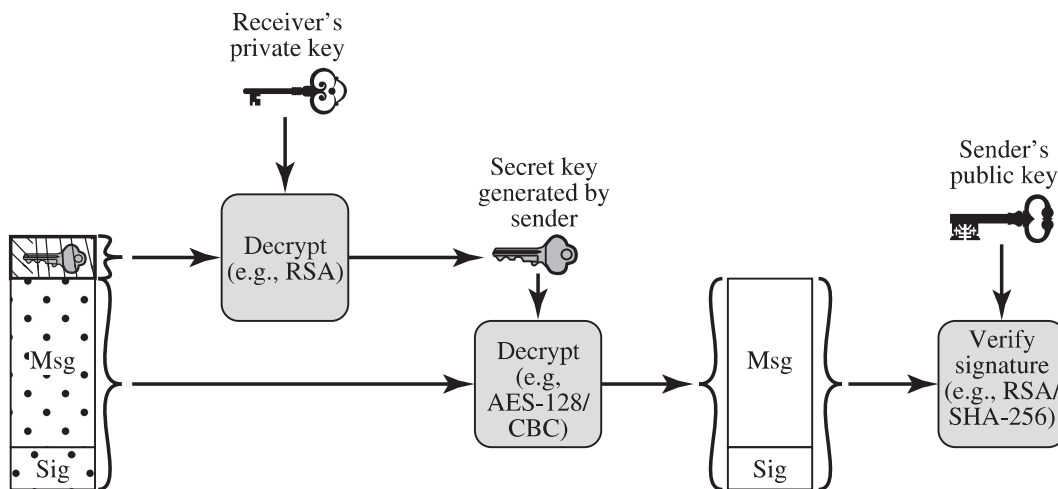
Type	Subtype	S/MIME Parameter	Description
Multipart	Signed		A clear-signed message in two parts: one is the message and the other is the signature.
Application	pkcs7-mime	signedData	A signed S/MIME entity
	pkcs7-mime	envelopedData	An encrypted S/MIME entity
	pkcs7-mime	degenerate signedData	An entity 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

Figure 22.1 provides a general overview of S/MIME functional flow.

SIGNED AND CLEAR-SIGNED DATA The preferred algorithms used for signing S/MIME messages use either an RSA or a Digital Signature Algorithm (DSA) signature of an SHA-256 message hash. The process works as follows. Take the message you want to send and map it into a fixed-length code of 256 bits, using SHA-256. The 256-bit message digest is, for all practical purposes, unique for this message. It would be virtually impossible for someone to alter this message or substitute



(a) Sender signs, and then encrypts message



(b) Receiver decrypts message, and then verifies sender's signature

Figure 22.1 Simplified S/MIME Functional Flow

another message and still come up with the same digest. Then, S/MIME encrypts the digest using RSA and the sender's private RSA key. The result is the digital signature, which is attached to the message, as we discuss in Chapter 2. Now, anyone who gets this message can recompute the message digest then decrypt the signature using RSA and the sender's public RSA key. If the message digest in the signature matches the message digest that was calculated, then the signature is valid. Since this operation only involves encrypting and decrypting a 256-bit block, it takes up little time. The DSA can be used instead of RSA as the signature algorithm.

The signature is a binary string, and sending it in that form through the Internet e-mail system could result in unintended alteration of the contents, because some e-mail software will attempt to interpret the message content looking for control characters such as line feeds. To protect the data, either the signature alone or the signature plus the message are mapped into printable ASCII characters using a scheme known as radix-64 or base64 mapping. Radix-64 maps each input group of three octets of binary data into four ASCII characters (see Appendix G).

ENVELOPED DATA The default algorithms used for encrypting S/MIME messages are AES and RSA. To begin, S/MIME generates a pseudorandom secret key; this is used to encrypt the message using AES or some other conventional encryption scheme, such as 3DES. In any conventional encryption application, the problem of key distribution must be addressed. In S/MIME, each conventional key is used only once. That is, a new pseudorandom key is generated for each new message encryption. This session key is bound to the message and transmitted with it. The secret key is used as input to the public-key encryption algorithm, RSA, which encrypts the key with the recipient's public RSA key. On the receiving end, S/MIME uses the receiver's private RSA key to recover the secret key, then uses the secret key and AES to recover the plaintext message.

If encryption is used alone, radix-64 is used to convert the ciphertext to ASCII format.

PUBLIC-KEY CERTIFICATES As can be seen from the discussion so far, S/MIME contains a clever, efficient, interlocking set of functions and formats to provide an effective encryption and signature service. To complete the system, one final area needs to be addressed, that of public-key management.

The basic tool that permits widespread use of S/MIME is the public-key certificate. S/MIME uses certificates that conform to the international standard X.509v3 that we discuss in Chapter 23.

22.2 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 specified in Internet Standard RFC 4871 (*DomainKeys Identified Mail (DKIM) Signatures*, 2007). DKIM has been

widely adopted by a range of e-mail providers, including corporations, government agencies, gmail, yahoo, and many Internet service providers (ISPs).

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 RFC 5598 (*Internet Mail Architecture*, 2009). This subsection provides 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 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 22.2 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 between the 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).

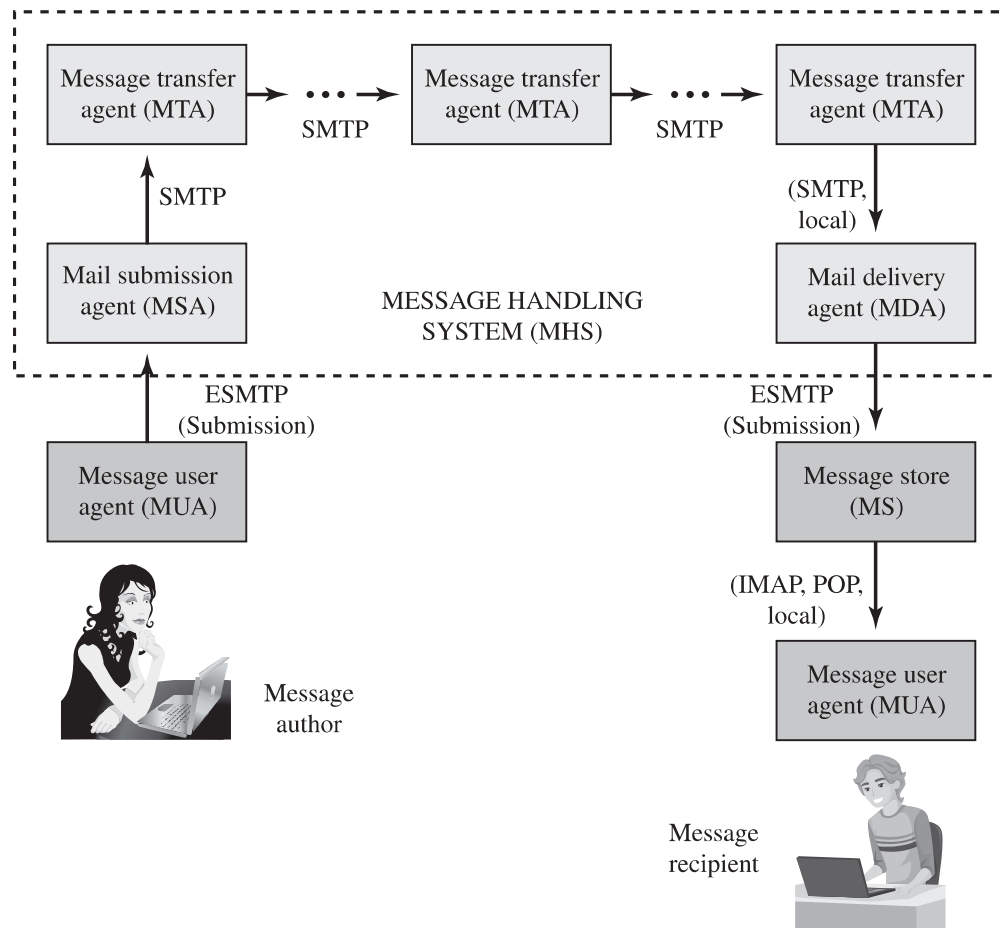


Figure 22.2 Function Modules and Standardized Protocols Used Between Them in the Internet Mail Architecture

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.

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 (*Internet Message Format*, 2008) message headers. At the receiving end, the MDA can access the corresponding public key via a DNS and verify 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, which uses the originator's private key to sign the content of the message. The motivation for DKIM is based on the following reasoning:

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 22.3 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 user's administrative domain. An e-mail message is generated by an e-mail client

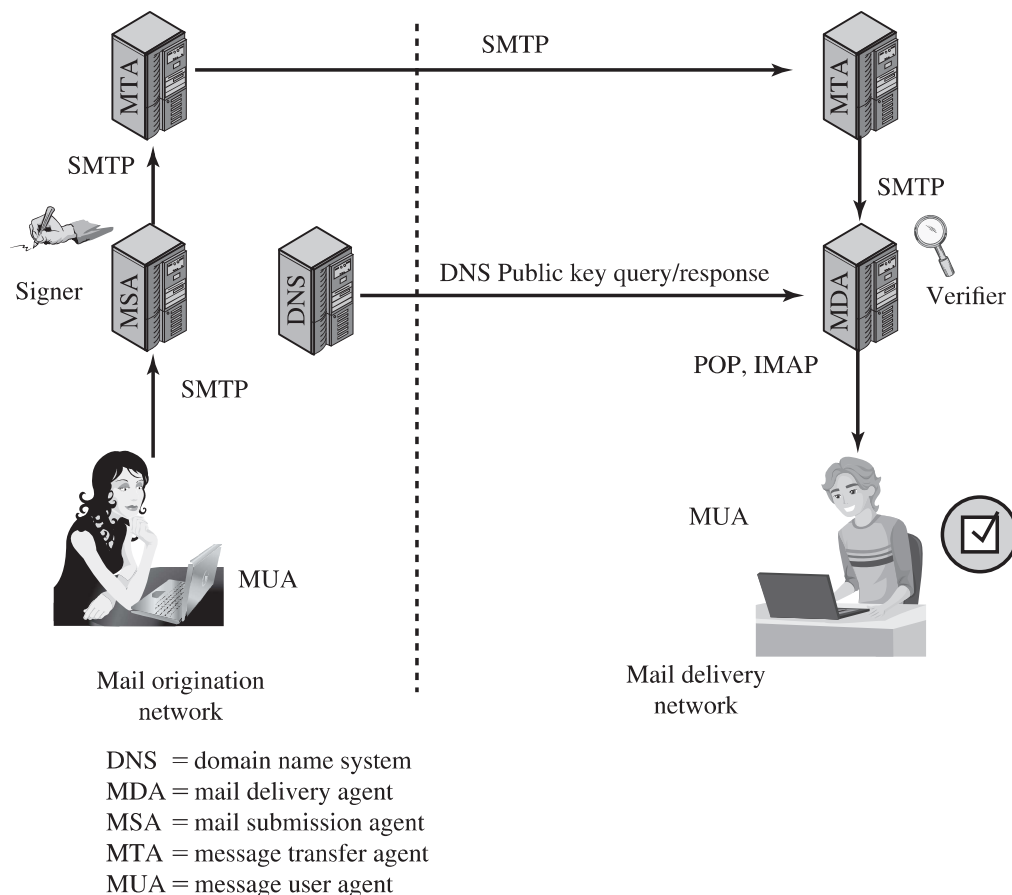


Figure 22.3 Simple Example of DKIM Deployment

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.

22.3 SECURE SOCKETS LAYER (SSL) AND TRANSPORT LAYER SECURITY (TLS)

One of the most widely used security services is the Secure Sockets Layer (SSL) and the follow-on Internet standard RFC 4346 (*The Transport Layer Security (TLS) Protocol Version 1.1*, 2006). TLS has largely supplanted earlier SSL implementations. TLS is a general-purpose service implemented as a set of protocols that rely on TCP. At this level, there are two implementation choices. For full generality, TLS could be provided as part of the underlying protocol suite and therefore be transparent to applications. Alternatively, TLS can be embedded in specific packages. For example, most browsers come equipped with SSL, and most Web servers have implemented the protocol.

TLS Architecture

TLS is designed to make use of TCP to provide a reliable end-to-end secure service. TLS is not a single protocol but rather two layers of protocols, as illustrated in Figure 22.4.

The 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 TLS. Three higher-layer protocols are defined as part of TLS: the Handshake Protocol, the Change Cipher Spec Protocol, and the Alert Protocol. These TLS-specific protocols are used in the management of TLS exchanges, and are examined later in this section.

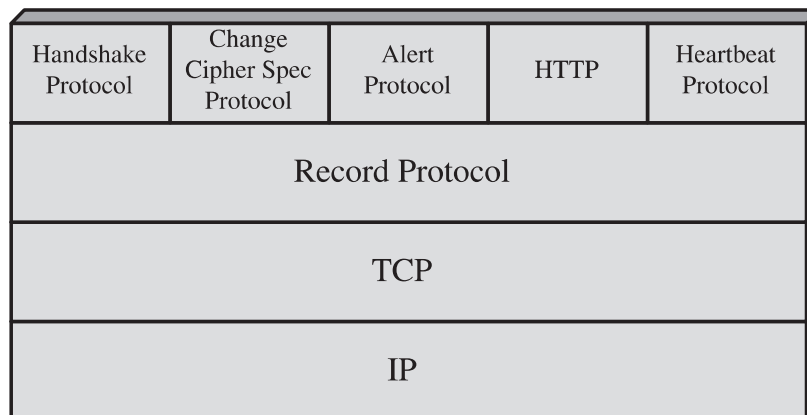


Figure 22.4 SSL/TLS Protocol Stack

Two important TLS concepts are the TLS session and the TLS 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 TLS, such connections are peer-to-peer relationships. The connections are transient. Every connection is associated with one session.
- **Session:** A TLS session is an association between a client and a server. Sessions are created by the Handshake Protocol. Sessions define a set of cryptographic 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.

TLS Protocols

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 symmetric 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 22.5 indicates the overall operation of the SSL Record Protocol. The first step is fragmentation. Each upper-layer message is fragmented into blocks of 2^{14} bytes (16,384 bytes) or less. Next, compression is optionally applied. The next

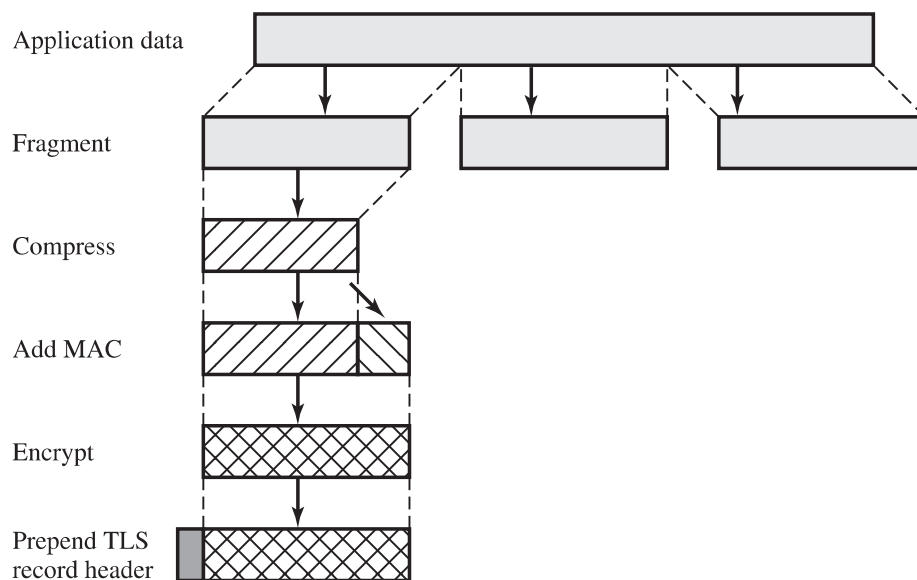


Figure 22.5 TLS Record Protocol Operation

step in processing is to compute a message authentication code over the compressed data. Next, the compressed message plus the MAC are encrypted using symmetric encryption.

The final step of SSL Record Protocol processing is to prepend a header, which includes version and length fields.

The content types that have been defined are `change_cipher_spec`, `alert`, `handshake`, and `application_data`. The first three are the TLS-specific protocols, discussed next. Note that no distinction is made among the various applications (e.g., HTTP) that might use TLS; the content of the data created by such applications is opaque to TLS.

The Record Protocol then transmits the resulting unit in a TCP segment. Received data are decrypted, verified, decompressed, and reassembled, then delivered to higher-level users.

CHANGE CIPHER SPEC PROTOCOL The Change Cipher Spec Protocol is one of the four TLS-specific protocols that use the TLS Record Protocol, and it is the simplest. This protocol consists of a single message, 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.

ALERT PROTOCOL The Alert Protocol is used to convey TLS-related alerts to the peer entity. As with other applications that use TLS, alert messages are compressed and encrypted, as specified by the current state.

Each message in this protocol consists of two bytes. The first byte takes the value `warning(1)` or `fatal(2)` to convey the severity of the message. If the level is fatal, TLS 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. An example of a fatal alert is an incorrect MAC. An example of a nonfatal alert is a `close_notify` message, which notifies the recipient that the sender will not send any more messages on this connection.

HANDSHAKE PROTOCOL The most complex part of TLS 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 TLS record. The Handshake Protocol is used before any application data are transmitted.

The Handshake Protocol consists of a series of messages exchanged by client and server. Figure 22.6 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 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_hello` message with the following parameters:

- **Version:** The highest TLS version understood by the client.
- **Random:** A client-generated random structure, consisting of a 32-bit timestamp and 28 bytes generated by a secure random number generator. These values are used during key exchange to prevent replay attacks.
- **Session ID:** A variable-length session identifier. A nonzero value indicates that the client wishes to update the parameters of an existing connection or create

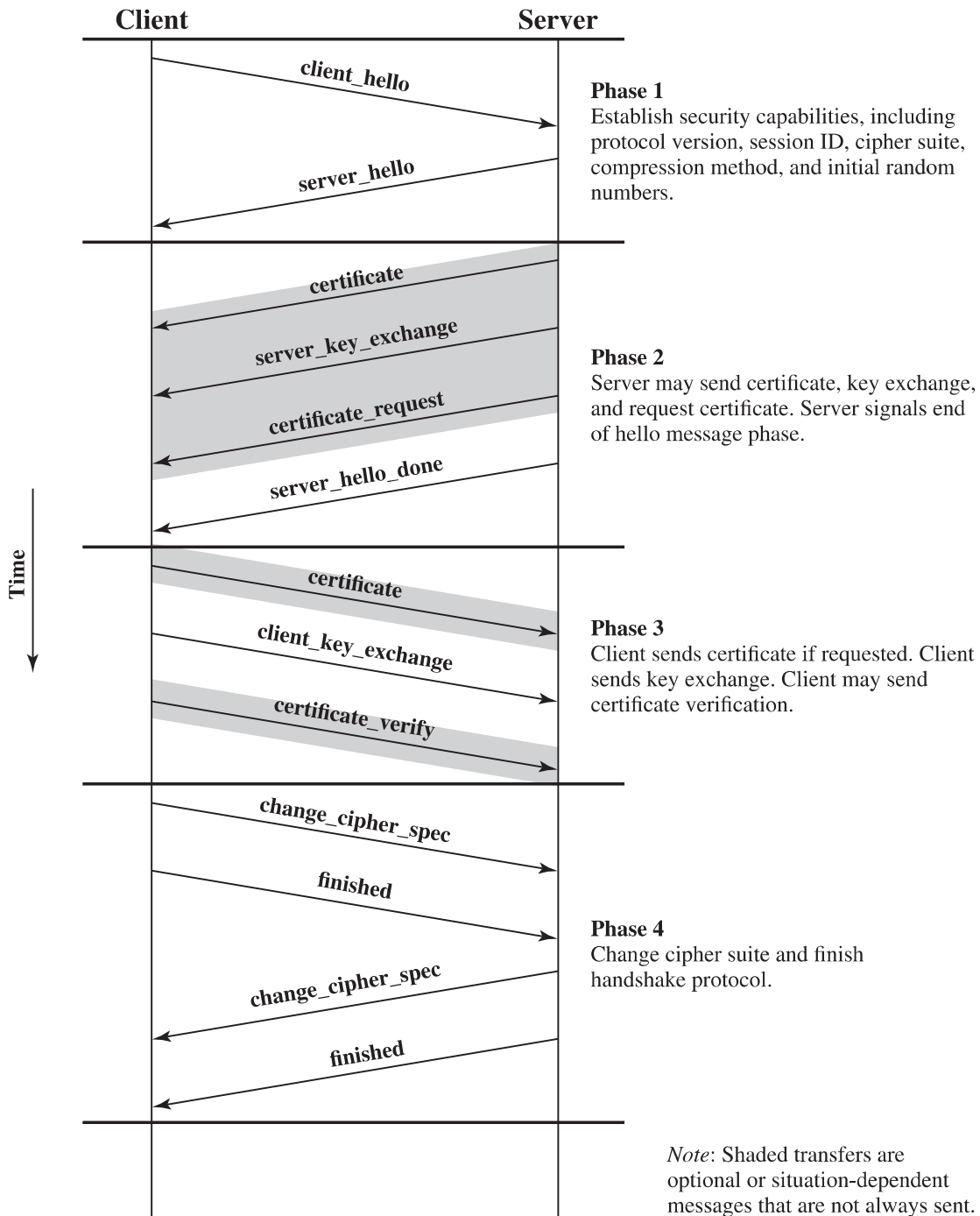


Figure 22.6 Handshake Protocol Action

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.
- **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_hello` message.

The details of **phase 2** depend on the underlying public-key encryption scheme that is used. In some cases, the server passes a certificate to the client, possibly additional key information, and a request for a certificate from the client.

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.

In **phase 3**, upon receipt of the `server_done` message, the client should verify that the server provided a valid certificate if required and check that the `server_hello` parameters are acceptable. If all is satisfactory, the client sends one or more messages back to the server, depending on the underlying public-key scheme.

Phase 4 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 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.

In response to these two messages, the server sends its own `change_cipher_spec` message, 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.

HEARTBEAT PROTOCOL In the context of computer networks, a heartbeat is a periodic signal generated by hardware or software to indicate normal operation or to synchronize other parts of a system. A Heartbeat Protocol is typically used to monitor the availability of a protocol entity. In the specific case of SSL/TLS, a Heartbeat protocol was defined in 2012 in RFC 6250 (*Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*, 2011).

The Heartbeat Protocol runs on the top of the TLS Record Protocol and consists of two message types: `heartbeat_request` and `heartbeat_response`. The use of the Heartbeat Protocol is established during Phase 1 of the Handshake Protocol (see Figure 22.6). Each peer indicates whether it supports heartbeats. If heartbeats are supported, the peer indicates whether it is willing to receive `heartbeat_request` messages and respond with `heartbeat_response` messages or only willing to send `heartbeat_request` messages.

A `heartbeat_request` message can be sent at any time. Whenever a request message is received, it should be answered promptly with a corresponding `heartbeat_response` message. The `heartbeat_request` message includes payload length, payload, and padding fields. The payload is a random content between 16 bytes and 64 Kbytes in length. The corresponding `heartbeat_response` message must include an exact copy of the received payload. The padding is also a random content. The padding enables the sender to perform a path maximum transfer unit (MTU) discovery operation, by sending requests with increasing padding until there is no answer anymore, because one of the hosts on the path cannot handle the message.

The heartbeat serves two purposes. First, it assures the sender that the recipient is still alive, even though there may not have been any activity over the underlying

TCP connection for a while. Second, the heartbeat generates activity across the connection during idle periods, which avoids closure by a firewall that does not tolerate idle connections.

The requirement for the exchange of a payload was designed into the Heartbeat Protocol to support its use in a connectionless version of TLS known as DTLS. Because a connectionless service is subject to packet loss, the payload enables the requestor to match response messages to request messages. For simplicity, the same version of the Heartbeat Protocol is used with both TLS and DTLS. Thus, the payload is required for both TLS and DTLS.

SSL/TLS Attacks

Since the first introduction of SSL in 1994, and the subsequent standardization of TLS, numerous attacks have been devised against these protocols. The appearance of each attack has necessitated changes in the protocol, the encryption tools used, or some aspects of the implementation of SSL and TLS to counter these threats.

ATTACK CATEGORIES We can group the attacks into four general categories:

- **Attacks on the Handshake Protocol:** As early as 1998, an approach to compromising the Handshake Protocol based on exploiting the formatting and implementation of the RSA encryption scheme was presented [BLEI98]. As countermeasures were implemented, the attack was refined and adjusted to not only thwart the countermeasures, but also to speed up the attack [e.g., BARD12].
- **Attacks on the record and application data protocols:** A number of vulnerabilities have been discovered in these protocols, leading to patches to counter the new threats. As a recent example, in 2011, researchers Thai Duong and Juliano Rizzo demonstrated a proof of concept called BEAST (Browser Exploit Against SSL/TLS) that turned what had been considered only a theoretical vulnerability into a practical attack [GOOD11]. BEAST leverages a type of cryptographic attack called a chosen-plaintext attack. The attacker mounts the attack by choosing a guess for the plaintext that is associated with a known ciphertext. The researchers developed a practical algorithm for launching successful attacks. Subsequent patches were able to thwart this attack. The authors of the BEAST attack are also the creators of the 2012 CRIME (Compression Ratio Info-leak Made Easy) attack, which can allow an attacker to recover the content of web cookies when data compression is used along with TLS [GOOD12b]. When used to recover the content of secret authentication cookies, it allows an attacker to perform session hijacking on an authenticated web session.
- **Attacks on the PKI:** Checking the validity of X.509 certificates is an activity subject to a variety of attacks, both in the context of SSL/TLS and elsewhere. For example, [GEOR12] demonstrated that commonly used libraries for SSL/TLS suffer from vulnerable certificate validation implementations. The authors revealed weaknesses in the source code of OpenSSL, GnuTLS, JSSE, ApacheHttpClient, Weberknecht, cURL, PHP, Python, and applications build upon or with these products.
- **Other attacks:** [MEYE13] lists a number of attacks that do not fit into any of the preceding categories. One example is an attack announced in 2011 by the

German hacker group The Hackers Choice, which is a DoS attack [KUMA11]. The attack creates a heavy processing load on a server by overwhelming the target with SSL/TLS handshake requests. Boosting system load is done by establishing new connections or using renegotiation. Assuming that the majority of computation during a handshake is done by the server the attack creates more system load on the server than on the source device, leading to a DoS. The server is forced to continuously recompute random numbers and keys.

The history of attacks and countermeasures for SSL/TLS is representative of that for other Internet-based protocols. A “perfect” protocol and a “perfect” implementation strategy are never achieved. A constant back-and-forth between threats and countermeasures determines the evolution of Internet-based protocols.

HEARTBLEED A bug discovered in 2014 in the TLS software created one of the potentially most catastrophic TLS vulnerabilities. The bug was in the open-source OpenSSL implementation of the Heartbeat Protocol. It is important to note that this vulnerability is not a design flaw in the TLS specification; rather it is a programming mistake in the OpenSSL library.

To understand the nature of the vulnerability, recall from our previous discussion that the `heartbeat_request` message includes payload length, payload and padding fields. Before the bug was fixed, the OpenSSL version of the Heartbeat Protocol worked as follows: The software reads the incoming request message and allocates a buffer large enough to hold the message header, the payload, and the padding. It then overwrites the current contents of the buffer with the incoming message, changes the first byte to indicate the response message type, then transmits a response message, which includes the payload length field and the payload. However, the software does not check the message length of the incoming message. As a result, an adversary can send a message that indicates the maximum payload length (64 KB) but only includes the minimum payload (16 bytes). This means that almost 64 KB of the buffer is not overwritten and whatever happened to be in memory at the time will be sent to

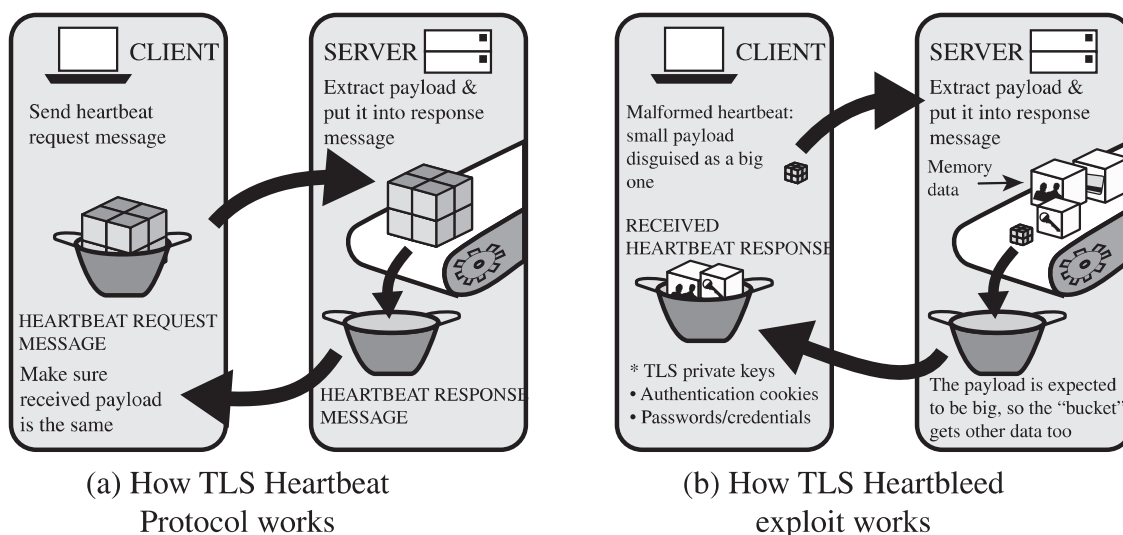


Figure 22.7 The Heartbleed Exploit

Source: “Heartbleed-The Open SSL Heartbeat Exploit” Copyright © 2014 BAE Systems Applied Intelligence. Reprinted with permission.

the requestor. Repeated attacks can result in the exposure of significant amounts of memory on the vulnerable system. Figure 22.7 illustrates the intended behavior and the actual behavior for the Heartbleed exploit.

This is a spectacular flaw. The untouched memory could contain private keys, user identification information, authentication data, passwords, or other sensitive data. The flaw was not discovered for several years. Even though eventually the bug was fixed in all implementations, large amounts of sensitive data were exposed to the Internet. Thus, we have a long exposure period, an easily implemented attack, and an attack that leaves no trace. Full recovery from this bug could take years. Compounding the problem is that OpenSSL is the most widely used TLS implementation. Servers using OpenSSL for TLS include finance, stock trading, personal and corporate email, social networks, banking, online shopping, and government agencies. It has been estimated that over two-thirds of the Internet's Web servers use OpenSSL, giving some idea of the scale of the problem [GOOD14].

22.4 HTTPS

HTTPS (HTTP over SSL) refers to the combination of HTTP and SSL to implement secure communication between a Web browser and a Web server. The HTTPS capability is built into all modern Web browsers. Its use depends on the Web server supporting HTTPS communication.

The principal difference seen by a user of a Web browser is that URL (uniform resource locator) addresses begin with `https://` rather than `http://`. A normal HTTP connection uses port 80. If HTTPS is specified, port 443 is used, which invokes SSL.

When HTTPS is used, the following elements of the communication are encrypted:

- URL of the requested document
- Contents of the document
- Contents of browser forms (filled in by browser user)
- Cookies sent from browser to server and from server to browser
- Contents of HTTP header

HTTPS is documented in RFC 2818 (*HTTP Over TLS*, 2000). There is no fundamental change in using HTTP over either SSL or TLS, and both implementations are referred to as HTTPS.

Connection Initiation

For HTTPS, the agent acting as the HTTP client also acts as the TLS client. The client initiates a connection to the server on the appropriate port then sends the TLS ClientHello to begin the TLS handshake. When the TLS handshake has finished, the client may then initiate the first HTTP request. All HTTP data is to be sent as TLS application data. Normal HTTP behavior, including retained connections, should be followed.

We need to be clear that there are three levels of awareness of a connection in HTTPS. At the HTTP level, an HTTP client requests a connection to an HTTP server by sending a connection request to the next lowest layer. Typically, the next lowest

layer is TCP, but it also may be TLS/SSL. At the level of TLS, a session is established between a TLS client and a TLS server. This session can support one or more connections at any time. As we have seen, a TLS request to establish a connection begins with the establishment of a TCP connection between the TCP entity on the client side and the TCP entity on the server side.

Connection Closure

An HTTP client or server can indicate the closing of a connection by including the following line in an HTTP record: `Connection: close`. This indicates that the connection will be closed after this record is delivered.

The closure of an HTTPS connection requires that TLS close the connection with the peer TLS entity on the remote side, which will involve closing the underlying TCP connection. At the TLS level, the proper way to close a connection is for each side to use the TLS alert protocol to send a `close_notify` alert. TLS implementations must initiate an exchange of closure alerts before closing a connection. A TLS implementation may, after sending a closure alert, close the connection without waiting for the peer to send its closure alert, generating an “incomplete close.” Note an implementation that does this may choose to reuse the session. This should only be done when the application knows (typically through detecting HTTP message boundaries) that it has received all the message data that it cares about.

HTTP clients also must be able to cope with a situation in which the underlying TCP connection is terminated without a prior `close_notify` alert and without a `Connection: close` indicator. Such a situation could be due to a programming error on the server or a communication error that causes the TCP connection to drop. However, the unannounced TCP closure could be evidence of some sort of attack. So the HTTPS client should issue some sort of security warning when this occurs.

22.5 IPv4 AND IPv6 SECURITY

IP Security Overview

The Internet community has developed application-specific security mechanisms in a number of areas, including electronic mail (S/MIME), client/server (Kerberos), Web access (SSL), and others. However, users have some security concerns that cut across protocol layers. For example, an enterprise can run a secure, private TCP/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.

In response to these issues, the Internet Architecture Board (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 do now have some IPsec capability in their products.

IP-level security encompasses three functional areas: authentication, confidentiality, and key management. 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 concerned with the secure exchange of keys. The current version of IPsec, known as IPsecv3, encompasses authentication and confidentiality. Key management is provided by the Internet Key Exchange standard, IKEv2.

We begin this section with an overview of IP security (IPsec) and an introduction to the IPsec architecture. We then look at some of the technical details. Appendix F reviews Internet protocols.

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 the following:

- **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 management 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 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 applications have built-in security protocols, the use of IPsec enhances that security.

The principal feature of IPsec that enables it to support these varied applications is that it can encrypt and/or authenticate *all* traffic at the IP level. Thus, all distributed applications, including remote logon, client/server, e-mail, file transfer, Web access, and so on, can be secured. Figure 9.3 is a typical scenario of IPsec usage.

BENEFITS OF IPSEC The benefits of IPsec include the following:

- 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 company or 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 so is 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 keying 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 off-site 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 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.

The Scope of IPsec

IPsec provides two main functions: a combined authentication/encryption function called Encapsulating Security Payload (ESP) and a key exchange function. For virtual private networks, both authentication and encryption are generally desired, because it is important both to (1) assure that unauthorized users do not penetrate the virtual private network and (2) assure that eavesdroppers on the Internet cannot read messages sent over the virtual private network. There is also an authentication-only function, implemented using an Authentication Header (AH). 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.

The key exchange function allows for manual exchange of keys as well as an automated scheme.

The IPsec specification is quite complex and covers numerous documents. The most important of these are:

- RFC 2401 (*Security Architecture for the Internet Protocol*, 1998)
- RFC 4302 (*IP Authentication Header*, 2005)
- RFC 4303 (*IP Encapsulating Security Payload (ESP)*, 2005)
- RFC 4306 (*Internet Key Exchange (IKEv2) Protocol*, 2005)

In this section, we provide an overview of some of the most important elements of IPsec.

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 relationship 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 ESP.

An SA is uniquely identified by three parameters:

- **Security parameter index (SPI):** A bit string assigned to this SA and having local significance only. The SPI is carried in an ESP header 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.
- **Protocol identifier:** This field in 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).

An IPsec implementation includes a security association database that defines the parameters associated with each SA. An SA is characterized by the following parameters:

- **Sequence number counter:** A 32-bit value used to generate the Sequence Number field in AH or ESP headers.
- **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.
- **Antireplay window:** Used to determine whether an inbound AH or ESP packet is a replay, by defining a sliding window within which the sequence number must fall.
- **AH information:** Authentication algorithm, keys, key lifetimes, and related parameters being used with AH.
- **ESP information:** Encryption and authentication algorithm, keys, initialization values, key lifetimes, and related parameters being used with ESP.
- **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.
- **IPsec protocol mode:** Tunnel, transport, or wildcard (required for all implementations). These modes will be discussed later in this section.
- **Path MTU:** Any observed path maximum transmission unit (maximum size of a packet that can be transmitted without fragmentation) and aging variables (required for all implementations).

The key management mechanism that is used to distribute keys is coupled to the authentication and privacy mechanisms only by way of the security parameters index. Hence, authentication and privacy have been specified independent of any specific key management mechanism.

Encapsulating Security Payload

The Encapsulating Security Payload provides confidentiality services, including confidentiality of message contents and limited traffic flow confidentiality. As an optional feature, ESP can also provide an authentication service.

Figure 22.8 shows the 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.
- **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):** May be required if the encryption algorithm requires the plaintext to be a multiple of some number of octets.
- **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 (e.g., an extension header in IPv6, or an upper-layer protocol such as TCP).
- **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.

Transport and Tunnel Modes

ESP supports two modes of use: transport and tunnel modes. We begin this section with 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. Examples include a TCP or UDP segment, both 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

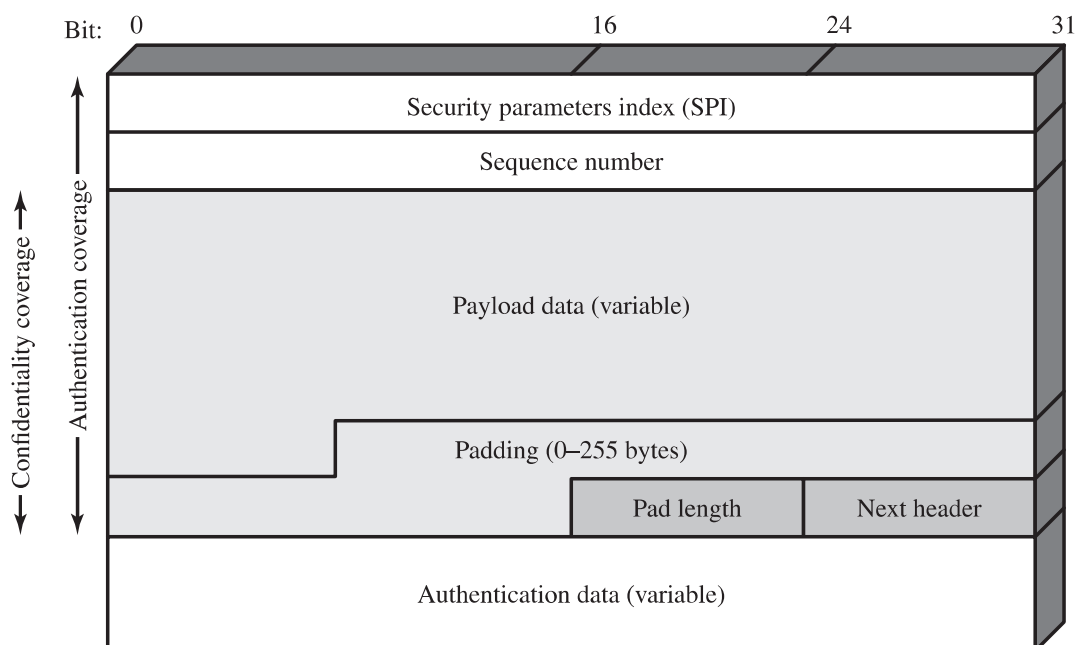


Figure 22.8 IPsec ESP Format

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 extension 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.

TUNNEL MODE Tunnel mode provides protection to the entire IP packet. To achieve this, after the ESP fields are added to the IP packet, the entire packet plus security fields are 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 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 tunneled 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, similar to that shown in Figure 9.3. 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.

22.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

administrative management domain (ADMD)	HTTPS (HTTP over SSL)	Secure Sockets Layer (SSL)
Domain Name System (DNS)	IPsec	S/MIME
DomainKeys Identified Mail (DKIM)	IPv4	Transport Layer Security (TLS)
Encapsulating Security Payload (ESP)	IPv6	
	Multipurpose Internet Mail Extension (MIME)	
	radix-64	

Review Questions

- 22.1 List the default algorithms used for signing S/MIME messages.
- 22.2 What is radix-64 conversion?
- 22.3 Why is radix-64 conversion useful for an e-mail application?
- 22.4 What is DKIM?
- 22.5 During an HTTPS connection, which elements of the communication are encrypted?
- 22.6 What is the difference between an SSL connection and an SSL session?
- 22.7 List the four categories of SSL/TLS attacks.
- 22.8 What is the purpose of HTTPS?
- 22.9 State the three levels of awareness of a connection in HTTPS.
- 22.10 Explain the transport and tunnel modes of ESP.
- 22.11 What are the two ways of providing authentication in IPsec?

Problems

- 22.1 In SSL and TLS, why is there a separate Change Cipher Spec Protocol rather than including a `change_cipher_spec` message in the Handshake Protocol?
- 22.2 Consider the following threats to Web security and describe how each is countered by a particular feature of SSL:
 - a. Man-in-the-middle attack: An attacker interposes during key exchange, acting as the client to the server and as the server to the client.
 - b. Password sniffing: Passwords in HTTP or other application traffic are eavesdropped.
 - c. IP spoofing: Uses forged IP addresses to fool a host into accepting bogus data.
 - d. IP hijacking: An active, authenticated connection between two hosts is disrupted and the attacker takes the place of one of the hosts.
 - e. SYN flooding: An attacker sends TCP SYN messages to request a connection but does not respond to the final message to establish the connection fully. The attacked TCP module typically leaves the “half-open connection” around for a few minutes. Repeated SYN messages can clog the TCP module.
- 22.3 Based on what you have learned in this chapter, there are three levels of awareness of a connection in HTTPS. What are these three levels of awareness? Explain how a TLS request to establish a connection begins.
- 22.4 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 in the IPsec authentication header is designed to thwart such attacks. Because IP is a connectionless, unreliable service, the protocol does not guarantee that packets will be delivered in order 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 (see Figure 22.9). Deduce from the figure how processing proceeds when a packet is received and explain how this counters the replay attack.
- 22.5 IPsec ESP can be used in two different modes of operation. In the **first mode**, ESP is used to encrypt and optionally authenticate the data carried by IP (e.g., a TCP segment). For this mode using IPv4, the ESP header is inserted into the IP packet immediately prior to the transport-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

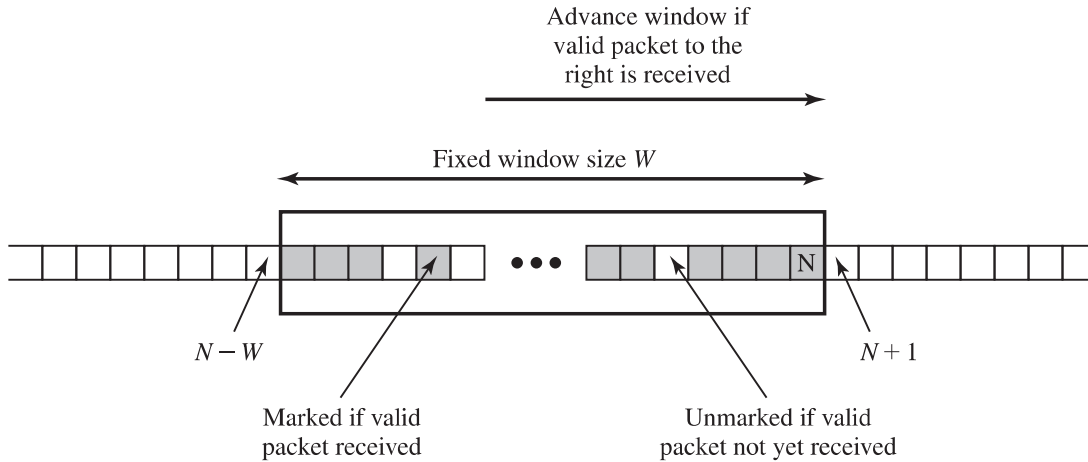


Figure 22.9 Antireplay Mechanism

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 **second mode**, ESP is used to encrypt an entire IP packet. For this mode, the ESP header is prefixed to the packet, and then the packet plus the ESP trailer are 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. Suggest applications for the two modes.

- 22.6** There are many different algorithms for signing S/MIME messages but there are a few algorithms which are otherwise used as default algorithms for achieving this task. Explain the step-by-step procedure of signing the S/MIME messages by using the default algorithms that are specified in the text.
- 22.7** An alternative to the radix-64 conversion in S/MIME is the quoted-printable transfer encoding. The first two encoding rules are as follows:
- 1. General 8-bit representation:** This rule is to be used when none of the other rules apply. Any character is represented by an equal sign followed by a two-digit hexadecimal representation of the octet's value. For example, the ASCII form feed, which has an 8-bit value of decimal 12, is represented by "=0C".
 - 2. Literal representation:** Any character in the range decimal 33 ("!") through decimal 126 ("~"), except decimal 61 ("="), is represented as that ASCII character. The remaining rules deal with spaces and line feeds. Explain the differences between the intended use for the quoted-printable and base 64 encodings.