2420; NIST 2001]), even a potential intruder! Clearly, if everyone knows the method for encoding data, then there must be some secret information that prevents an intruder from decrypting the transmitted data. This is where keys come in.

In Figure 8.2, Alice provides a **key**, K_A, a string of numbers or characters, as input to the encryption algorithm. The encryption algorithm takes the key and the plaintext message, m, as input and produces ciphertext as output. The notation $K_A(m)$ refers to the ciphertext form (encrypted using the key K_A) of the plaintext message, m. The actual encryption algorithm that uses key K_A will be evident from the context. Similarly, Bob will provide a key, K_B, to the **decryption algorithm** that takes the ciphertext and Bob's key as input and produces the original plaintext as output. That is, if Bob receives an encrypted message $K_A(m)$, he decrypts it by computing $K_B(K_A(m)) = m$. In **symmetric key systems**, Alice's and Bob's keys are identical and are secret. In **public key systems**, a pair of keys is used. One of the keys is known to both Bob and Alice (indeed, it is known to the whole world). The other key is known only by either Bob or Alice (but not both). In the following two subsections, we consider symmetric key and public key systems in more detail.

8.2.1 Symmetric Key Cryptography

All cryptographic algorithms involve substituting one thing for another, for example, taking a piece of plaintext and then computing and substituting the appropriate ciphertext to create the encrypted message. Before studying a modern key-based cryptographic system, let us first get our feet wet by studying a very old, very simple symmetric key algorithm attributed to Julius Caesar, known as the **Caesar cipher** (a cipher is a method for encrypting data).

For English text, the Caesar cipher would work by taking each letter in the plaintext message and substituting the letter that is k letters later (allowing wraparound; that is, having the letter z followed by the letter a) in the alphabet. For example if $k = 3$, then the letter a in plaintext becomes d in ciphertext; b in plaintext becomes e in ciphertext, and so on. Here, the value of k serves as the key. As an example, the plaintext message "`bob, i love you. alice`" becomes "`ere, l oryh brx. dolfh`" in ciphertext. While the ciphertext does indeed look like gibberish, it wouldn't take long to break the code if you knew that the Caesar cipher was being used, as there are only 25 possible key values.

An improvement on the Caesar cipher is the **monoalphabetic cipher**, which also substitutes one letter of the alphabet with another letter of the alphabet. However, rather than substituting according to a regular pattern (for example, substitution with an offset of k for all letters), any letter can be substituted for any other letter, as long as each letter has a unique substitute letter, and vice versa. The substitution rule in Figure 8.3 shows one possible rule for encoding plaintext.

The plaintext message "`bob, i love you. alice`" becomes "`nkn, s gktc wky. mgsbc`." Thus, as in the case of the Caesar cipher, this looks like

Plaintext letter:	a b c d e f g h i j k l m n o p q r s t u v w x y z
Ciphertext letter:	m n b v c x z a s d f g h j k l p o i u y t r e w q

Figure 8.3 ♦ A monoalphabetic cipher

gibberish. A monoalphabetic cipher would also appear to be better than the Caesar cipher in that there are 26! (on the order of 10^{26}) possible pairings of letters rather than 25 possible pairings. A brute-force approach of trying all 10^{26} possible pairings would require far too much work to be a feasible way of breaking the encryption algorithm and decoding the message. However, by statistical analysis of the plaintext language, for example, knowing that the letters *e* and *t* are the most frequently occurring letters in typical English text (accounting for 13 percent and 9 percent of letter occurrences), and knowing that particular two- and three-letter occurrences of letters appear quite often together (for example, "in," "it," "the," "ion," "ing," and so forth) make it relatively easy to break this code. If the intruder has some knowledge about the possible contents of the message, then it is even easier to break the code. For example, if Trudy the intruder is Bob's wife and suspects Bob of having an affair with Alice, then she might suspect that the names "bob" and "alice" appear in the text. If Trudy knew for certain that those two names appeared in the ciphertext and had a copy of the example ciphertext message above, then she could immediately determine seven of the 26 letter pairings, requiring 10^9 fewer possibilities to be checked by a brute-force method. Indeed, if Trudy suspected Bob of having an affair, she might well expect to find some other choice words in the message as well.

When considering how easy it might be for Trudy to break Bob and Alice's encryption scheme, one can distinguish three different scenarios, depending on what information the intruder has.

- *Ciphertext-only attack.* In some cases, the intruder may have access only to the intercepted ciphertext, with no certain information about the contents of the plaintext message. We have seen how statistical analysis can help in a **ciphertext-only attack** on an encryption scheme.
- *Known-plaintext attack.* We saw above that if Trudy somehow knew for sure that "bob" and "alice" appeared in the ciphertext message, then she could have determined the (plaintext, ciphertext) pairings for the letters *a, l, i, c, e, b,* and *o*. Trudy might also have been fortunate enough to have recorded all of the ciphertext transmissions and then found Bob's own decrypted version of one of the transmissions scribbled on a piece of paper. When an intruder knows some of the (plaintext, ciphertext) pairings, we refer to this as a **known-plaintext attack** on the encryption scheme.

- *Chosen-plaintext attack.* In a **chosen-plaintext attack**, the intruder is able to choose the plaintext message and obtain its corresponding ciphertext form. For the simple encryption algorithms we've seen so far, if Trudy could get Alice to send the message, "`The quick brown fox jumps over the lazy dog`," she could completely break the encryption scheme. We'll see shortly that for more sophisticated encryption techniques, a chosen-plaintext attack does not necessarily mean that the encryption technique can be broken.

Five hundred years ago, techniques improving on monoalphabetic encryption, known as **polyalphabetic encryption**, were invented. The idea behind polyalphabetic encryption is to use multiple monoalphabetic ciphers, with a specific monoalphabetic cipher to encode a letter in a specific position in the plaintext message. Thus, the same letter, appearing in different positions in the plaintext message, might be encoded differently. An example of a polyalphabetic encryption scheme is shown in Figure 8.4. It has two Caesar ciphers (with $k = 5$ and $k = 19$), shown as rows. We might choose to use these two Caesar ciphers, C_1 and C_2, in the repeating pattern C_1, C_2, C_2, C_1, C_2. That is, the first letter of plaintext is to be encoded using C_1, the second and third using C_2, the fourth using C_1, and the fifth using C_2. The pattern then repeats, with the sixth letter being encoded using C_1, the seventh with C_2, and so on. The plaintext message "`bob, i love you.`" is thus encrypted "`ghu, n etox dhz.`" Note that the first *b* in the plaintext message is encrypted using C_1, while the second *b* is encrypted using C_2. In this example, the encryption and decryption "key" is the knowledge of the two Caesar keys ($k = 5$, $k = 19$) and the pattern C_1, C_2, C_2, C_1, C_2.

Block Ciphers

Let us now move forward to modern times and examine how symmetric key encryption is done today. There are two broad classes of symmetric encryption techniques: **stream ciphers** and **block ciphers**. We'll briefly examine stream ciphers in Section 8.7 when we investigate security for wireless LANs. In this section, we focus on block ciphers, which are used in many secure Internet protocols, including PGP (for secure e-mail), SSL (for securing TCP connections), and IPsec (for securing the network-layer transport).

Plaintext letter:	a b c d e f g h i j k l m n o p q r s t u v w x y z
$C_1(k = 5)$:	f g h i j k l m n o p q r s t u v w x y z a b c d e
$C_2(k = 19)$:	t u v w x y z a b c d e f g h i j k l m n o p q r s

Figure 8.4 ♦ A polyalphabetic cipher using two Caesar ciphers

input	output	input	output
000	110	100	011
001	111	101	010
010	101	110	000
011	100	111	001

Table 8.1 ♦ A specific 3-bit block cipher

In a block cipher, the message to be encrypted is processed in blocks of k bits. For example, if $k = 64$, then the message is broken into 64-bit blocks, and each block is encrypted independently. To encode a block, the cipher uses a one-to-one mapping to map the k-bit block of cleartext to a k-bit block of ciphertext. Let's look at an example. Suppose that $k = 3$, so that the block cipher maps 3-bit inputs (cleartext) to 3-bit outputs (ciphertext). One possible mapping is given in Table 8.1. Notice that this is a one-to-one mapping; that is, there is a different output for each input. This block cipher breaks the message up into 3-bit blocks and encrypts each block according to the above mapping. You should verify that the message 010110001111 gets encrypted into 101000111001.

Continuing with this 3-bit block example, note that the mapping in Table 8.1 is just one mapping of many possible mappings. How many possible mappings are there? To answer this question, observe that a mapping is nothing more than a permutation of all the possible inputs. There are 2^3 (= 8) possible inputs (listed under the input columns). These eight inputs can be permuted in 8! = 40,320 different ways. Since each of these permutations specifies a mapping, there are 40,320 possible mappings. We can view each of these mappings as a key—if Alice and Bob both know the mapping (the key), they can encrypt and decrypt the messages sent between them.

The brute-force attack for this cipher is to try to decrypt ciphtertext by using all mappings. With only 40,320 mappings (when $k = 3$), this can quickly be accomplished on a desktop PC. To thwart brute-force attacks, block ciphers typically use much larger blocks, consisting of $k = 64$ bits or even larger. Note that the number of possible mappings for a general k-block cipher is $2^k!$, which is astronomical for even moderate values of k (such as $k = 64$).

Although full-table block ciphers, as just described, with moderate values of k can produce robust symmetric key encryption schemes, they are unfortunately difficult to implement. For $k = 64$ and for a given mapping, Alice and Bob would need to maintain a table with 2^{64} input values, which is an infeasible task. Moreover, if Alice and Bob were to change keys, they would have to each regenerate

the table. Thus, a full-table block cipher, providing predetermined mappings between all inputs and outputs (as in the example above), is simply out of the question.

Instead, block ciphers typically use functions that simulate randomly permuted tables. An example (adapted from [Kaufman 1995]) of such a function for $k = 64$ bits is shown in Figure 8.5. The function first breaks a 64-bit block into 8 chunks, with each chunk consisting of 8 bits. Each 8-bit chunk is processed by an 8-bit to 8-bit table, which is of manageable size. For example, the first chunk is processed by the table denoted by T_1. Next, the 8 output chunks are reassembled into a 64-bit block. The positions of the 64 bits in the block are then scrambled (permuted) to produce a 64-bit output. This output is fed back to the 64-bit input, where another cycle begins. After n such cycles, the function provides a 64-bit block of ciphertext. The purpose of the rounds is to make each input bit affect most (if not all) of the final output bits. (If only one round were used, a given input bit would affect only 8 of the 64 output bits.) The key for this block cipher algorithm would be the eight permutation tables (assuming the scramble function is publicly known).

Today there are a number of popular block ciphers, including DES (standing for Data Encryption Standard), 3DES, and AES (standing for Advanced Encryption Standard). Each of these standards uses functions, rather than predetermined tables, along the lines of Figure 8.5 (albeit more complicated and specific to each cipher). Each of these algorithms also uses a string of bits for a key. For example, DES uses 64-bit blocks with a 56-bit key. AES uses 128-bit blocks and can operate with keys that are 128, 192, and 256 bits long. An algorithm's key determines the specific

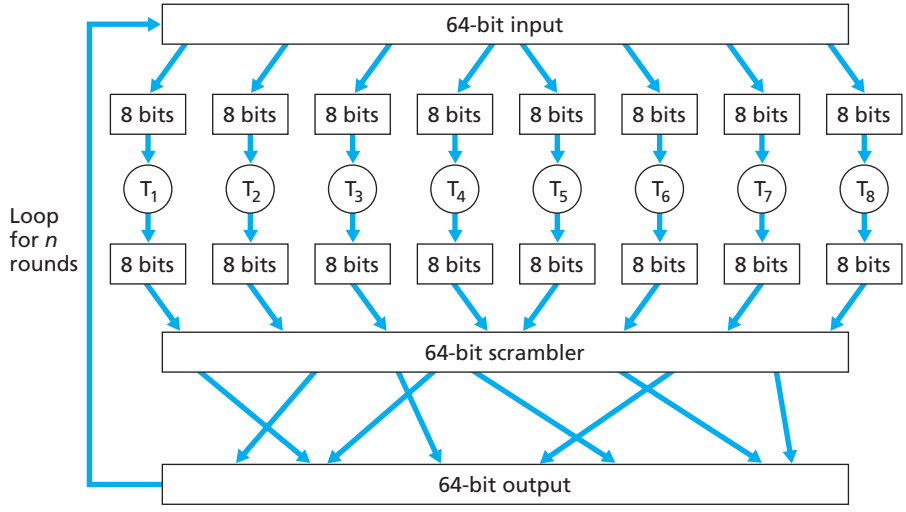

Figure 8.5 ♦ An example of a block cipher

"mini-table" mappings and permutations within the algorithm's internals. The brute-force attack for each of these ciphers is to cycle through all the keys, applying the decryption algorithm with each key. Observe that with a key length of n, there are 2^n possible keys. NIST [NIST 2001] estimates that a machine that could crack 56-bit DES in one second (that is, try all 2^{56} keys in one second) would take approximately 149 trillion years to crack a 128-bit AES key.

Cipher-Block Chaining

In computer networking applications, we typically need to encrypt long messages (or long streams of data). If we apply a block cipher as described by simply chopping up the message into k-bit blocks and independently encrypting each block, a subtle but important problem occurs. To see this, observe that two or more of the cleartext blocks can be identical. For example, the cleartext in two or more blocks could be "HTTP/1.1". For these identical blocks, a block cipher would, of course, produce the same ciphertext. An attacker could potentially guess the cleartext when it sees identical ciphertext blocks and may even be able to decrypt the entire message by identifying identical ciphtertext blocks and using knowledge about the underlying protocol structure [Kaufman 1995].

To address this problem, we can mix some randomness into the ciphertext so that identical plaintext blocks produce different ciphertext blocks. To explain this idea, let $m(i)$ denote the ith plaintext block, $c(i)$ denote the ith ciphertext block, and $a \oplus b$ denote the exclusive-or (XOR) of two bit strings, a and b. (Recall that the $0 \oplus 0 = 1 \oplus 1 = 0$ and $0 \oplus 1 = 1 \oplus 0 = 1$, and the XOR of two bit strings is done on a bit-by-bit basis. So, for example, $10101010 \oplus 11110000 = 01011010$.) Also, denote the block-cipher encryption algorithm with key S as K_S. The basic idea is as follows. The sender creates a random k-bit number $r(i)$ for the ith block and calculates $c(i) = K_S(m(i) \oplus r(i))$. Note that a new k-bit random number is chosen for each block. The sender then sends $c(1)$, $r(1)$, $c(2)$, $r(2)$, $c(3)$, $r(3)$, and so on. Since the receiver receives $c(i)$ and $r(i)$, it can recover each block of the plaintext by computing $m(i) = K_S(c(i)) \oplus r(i)$. It is important to note that, although $r(i)$ is sent in the clear and thus can be sniffed by Trudy, she cannot obtain the plaintext $m(i)$, since she does not know the key K_S. Also note that if two plaintext blocks $m(i)$ and $m(j)$ are the same, the corresponding ciphertext blocks $c(i)$ and $c(j)$ will be different (as long as the random numbers $r(i)$ and $r(j)$ are different, which occurs with very high probability).

As an example, consider the 3-bit block cipher in Table 8.1. Suppose the plaintext is 010010010. If Alice encrypts this directly, without including the randomness, the resulting ciphertext becomes 101101101. If Trudy sniffs this ciphertext, because each of the three cipher blocks is the same, she can correctly surmise that each of the three plaintext blocks are the same. Now suppose instead Alice generates the random blocks $r(1) = 001$, $r(2) = 111$, and $r(3) = 100$ and uses the above technique to generate the ciphertext $c(1) = 100$, $c(2) = 010$, and $c(3) = 000$. Note that the three

ciphertext blocks are different even though the plaintext blocks are the same. Alice then sends $c(1)$, $r(1)$, $c(2)$, and $r(2)$. You should verify that Bob can obtain the original plaintext using the shared key K_S.

The astute reader will note that introducing randomness solves one problem but creates another: namely, Alice must transmit twice as many bits as before. Indeed, for each cipher bit, she must now also send a random bit, doubling the required bandwidth. In order to have our cake and eat it too, block ciphers typically use a technique called **Cipher Block Chaining (CBC)**. The basic idea is to send only *one random value along with the very first message, and then have the sender and receiver use the computed coded blocks in place of the subsequent random number.* Specifically, CBC operates as follows:

1. Before encrypting the message (or the stream of data), the sender generates a random k-bit string, called the **Initialization Vector (IV)**. Denote this initialization vector by $c(0)$. The sender sends the IV to the receiver *in cleartext*.
2. For the first block, the sender calculates $m(1) \oplus c(0)$, that is, calculates the exclusive-or of the first block of cleartext with the IV. It then runs the result through the block-cipher algorithm to get the corresponding ciphertext block; that is, $c(1) = K_S(m(1) \oplus c(0))$. The sender sends the encrypted block $c(1)$ to the receiver.
3. For the ith block, the sender generates the ith ciphertext block from $c(i) = K_S(m(i) \oplus c(i - 1))$.

Let's now examine some of the consequences of this approach. First, the receiver will still be able to recover the original message. Indeed, when the receiver receives $c(i)$, it decrypts it with K_S to obtain $s(i) = m(i) \oplus c(i - 1)$; since the receiver also knows $c(i - 1)$, it then obtains the cleartext block from $m(i) = s(i) \oplus c(i - 1)$. Second, even if two cleartext blocks are identical, the corresponding ciphtertexts (almost always) will be different. Third, although the sender sends the IV in the clear, an intruder will still not be able to decrypt the ciphertext blocks, since the intruder does not know the secret key, S. Finally, the sender only sends one overhead block (the IV), thereby negligibly increasing the bandwidth usage for long messages (consisting of hundreds of blocks).

As an example, let's now determine the ciphertext for the 3-bit block cipher in Table 8.1 with plaintext 010010010 and IV = $c(0) = 001$. The sender first uses the IV to calculate $c(1) = K_S(m(1) \oplus c(0)) = 100$. The sender then calculates $c(2) = K_S(m(2) \oplus c(1)) = K_S(010 \oplus 100) = 000$, and $c(3) = K_S(m(3) \oplus c(2)) = K_S(010 \oplus 000) = 101$. The reader should verify that the receiver, knowing the IV and K_S can recover the original plaintext.

CBC has an important consequence when designing secure network protocols: we'll need to provide a mechanism within the protocol to distribute the IV from sender to receiver. We'll see how this is done for several protocols later in this chapter.