Chapter 7 The Data Encryption Standard

7.1 Introduction

In 1973, the National Bureau of Standards (NBS), later to become the National Institute of Standards and Technology (NIST), issued a public request seeking a cryptographic algorithm to become a national standard. IBM submitted an algorithm called LUCIFER in 1974. The NBS forwarded it to the National Security Agency, which reviewed it and, after some modifications, returned a version that was essentially the Data Encryption Standard (DES) algorithm. In 1975, NBS released DES, as well as a free license for its use, and in 1977 NBS made it the official data encryption standard.

DES was used extensively in electronic commerce, for example in the banking industry. If two banks wanted to exchange data, they first used a public key method such as RSA to transmit a key for DES, then they used DES for transmitting the data. It had the advantage of being very fast and reasonably secure.

From 1975 on, there was controversy surrounding DES. Some regarded the key size as too small. Many were worried about NSA’s involvement. For example, had they arranged for it to have a “trapdoor" – in other words, a secret weakness that would allow only them to break the system? It was also suggested that NSA modified the design to avoid the possibility that IBM had inserted a trapdoor in LUCIFER. In any case, the design decisions remained a mystery for many years.

In 1990, Eli Biham and Adi Shamir showed how their method of differential cryptanalysis could be used to attack DES, and soon thereafter they showed how these methods could succeed faster than brute force. This indicated that perhaps the designers of DES had been aware of this type of attack. A few years later, IBM released some details of the design criteria, which showed that indeed they had constructed the system to be resistant to differential cryptanalysis. This cleared up at least some of the mystery surrounding the algorithm.

DES lasted for a long time, but became outdated. Brute force searches (see Section 7.5), though expensive, can now break the system. Therefore, NIST replaced it with the system AES (see Chapter 8) in the year 2000. However, it is worth studying DES since it represents a popular class of algorithms and it was one of the most frequently used cryptographic algorithms in history.

DES is a block cipher; namely, it breaks the plaintext into blocks of 64 bits, and encrypts each block separately. The actual mechanics of how this is done is often called a **Feistel system**, after Horst Feistel, who was part of the IBM team that developed LUCIFER. In the next section, we give a simple algorithm that has many of the characteristics of this type of system, but is small enough to use as an example. In Section 7.3, we show how differential cryptanalysis can be used to attack this simple system. We give the DES algorithm in Section 7.4, Finally, in Section 7.5, we describe some methods used to break DES.

7.2 A Simplified DES-Type Algorithm

The DES algorithm is rather unwieldy to use for examples, so in the present section we present an algorithm that has many of the same features, but is much smaller. Like DES, the present algorithm is a block cipher. Since the blocks are encrypted separately, we assume throughout the present discussion that the full message consists of only one block.

The message has 12 bits and is written in the form L_0R_0, where L_0 consists of the first six bits and R_0 consists of the last six bits. The key K has nine bits. The ith round of the algorithm transforms an input $L_{i-1}R_{i-1}$ to the output L_iR_i using an eight-bit key K_i derived from K.

The main part of the encryption process is a function $f(R_{i-1}, K_i)$ that takes a six-bit input R_{i-1} and an eight-bit input K_i and produces a six-bit output. This will be described later.

The output for the ith round is defined as follows:

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

where $\oplus$ denotes XOR, namely bit-by-bit addition mod 2. This is depicted in Figure 7.1.

Figure 7.1 One Round of a Feistel System

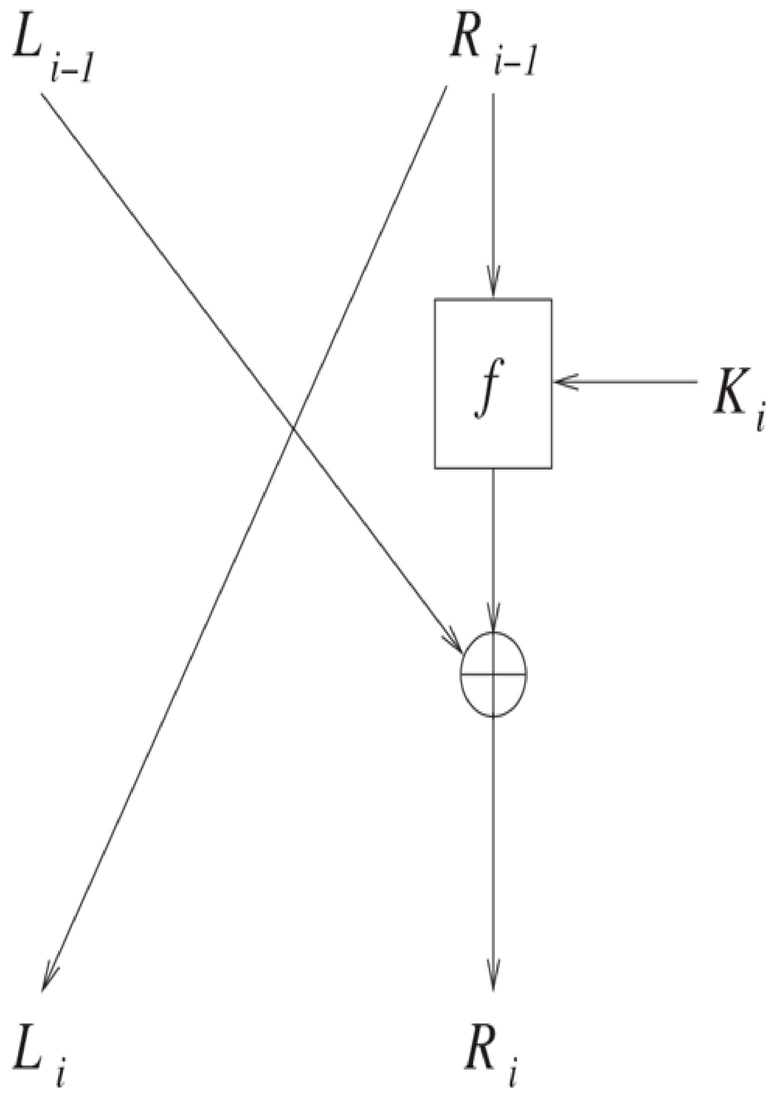


Figure 7.1 Full Alternative Text

This operation is performed for a certain number of rounds, say n, and produces the ciphertext L_nR_n.

How do we decrypt? Start with L_nR_n and switch left and right to obtain R_nL_n. (Note: *This switch is built into the DES encryption algorithm, so it is not needed when decrypting DES.*) Now use the same procedure as before, but with the keys K_i used in reverse order $K_n, \ \ldots, K_1$. Let's see how this works. The first step takes R_nL_n and gives the output

$$[L_n] \quad [R_n \oplus f(L_n, K_n)].$$

We know from the encryption procedure that $L_n = R_{n-1}$ and $R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$. Therefore,

$$\begin{aligned}[L_n] \quad [R_n \oplus f(L_n, K_n)] &= [R_{n-1}] \quad [L_{n-1} \oplus f(R_{n-1}, K_n) \oplus f(L_n, K_n)] \\ &= [R_{n-1}] \quad [L_{n-1}].\end{aligned}$$

The last equality again uses $L_n = R_{n-1}$, so that $f(R_{n-1}, K_n) \oplus f(L_n, K_n)$ is 0. Similarly, the second step of decryption sends $R_{n-1}L_{n-1}$ to $R_{n-2}L_{n-2}$. Continuing, we see that the decryption process leads us back to R_0L_0. Switching the left and right halves, we obtain the original plaintext L_0R_0, as desired.

Note that the decryption process is essentially the same as the encryption process. We simply need to switch left and right and use the keys K_i in reverse order. Therefore, both the sender and receiver use a common key and they can use identical machines (though the receiver needs to reverse left and right inputs).

So far, we have said nothing about the function f. In fact, any f would work in the above procedures. But some choices of f yield much better security than others. The type of f used in DES is similar to that which we describe next. It is built up from a few components.

The first function is an expander. It takes an input of six bits and outputs eight bits. The one we use is given in Figure 7.2.

Figure 7.2 The Expander Function

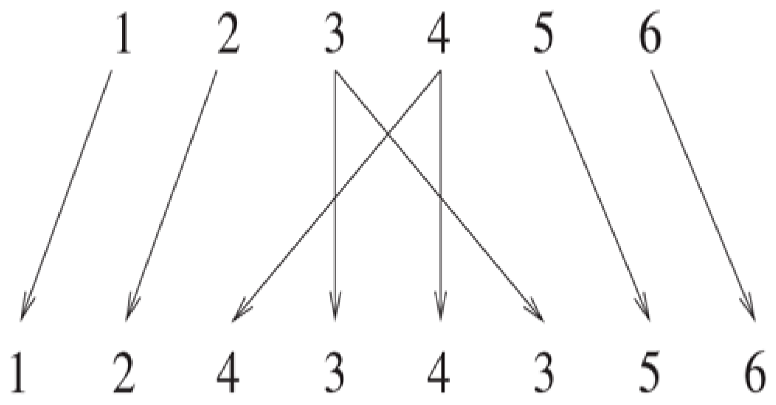


Figure 7.2 Full Alternative Text

This means that the first input bit yields the first output bit, the third input bit yields both the fourth and the sixth output bits, etc. For example, 011001 is expanded to 01010101.

The main components are called S-boxes. We use two:

$$S_1 \quad \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

$$S_2 \quad \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}.$$

The input for an S-box has four bits. The first bit specifies which row will be used: 0 for the first row, 1 for the second. The other three bits represent a binary number that specifies the column: 000 for the first column, 001 for the second, ..., 111 for the last column. The output for the S-box consists of the three bits in the specified location. For example, an input of 1010 for S_1 means we look at the second row, third column, which yields the output 110.

The key K consists of nine bits. The key K_i for the ith round of encryption is obtained by using eight bits of K, starting with the ith bit. For example, if $K = 010011001$, then $K_4 = 01100101$ (after five bits, we reached the end of K, so the last two bits were obtained from the beginning of K).

We can now describe $f(R_{i-1}, K_i)$. The input R_{i-1} consists of six bits. The expander function is used to expand it to eight bits. The result is XORed with K_i to produce another eight-bit number. The first four bits are sent to S_1, and the last four bits are sent to S_2. Each S-box outputs three bits, which are concatenated to form a six-bit number. This is $f(R_{i-1}, K_i)$. We present this in Figure 7.3.

Figure 7.3 The Function $f(R_{i-1}, K_i)$

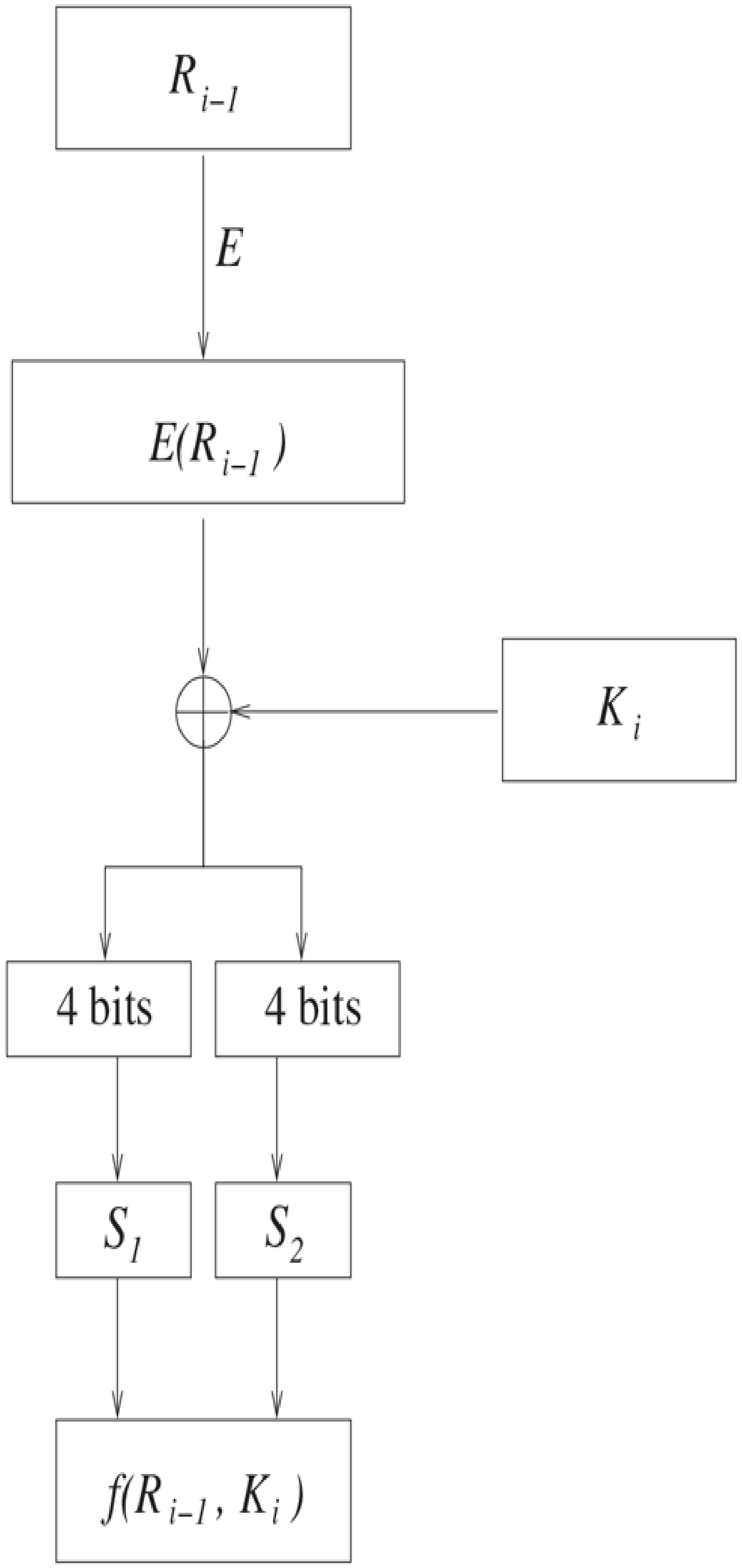


Figure 7.3 Full Alternative Text

For example, suppose $R_{i-1} = 100110$ and $K_i = 01100101$. We have

$$E(100110) \oplus K_i = 10101010 \oplus 01100101 = 11001111.$$

The first four bits are sent to S_1 and the last four bits are sent to S_2. The second row, fifth column of S_1 contains 000. The second row, last column of S_2 contains 100. Putting these outputs one after the other yields $f(R_{i-1}, K_i) = 000100$.

We can now describe what happens in one round. Suppose the input is

$$L_{i-1}R_{i-1} = 011100100110$$

and $K_i = 01100101$, as previously. This means that $R_{i-1} = 100110$, as in the example just discussed. Therefore, $f(R_{i-1}, K_i) = 000100$. This is XORed with $L_{i-1} = 011100$ to yield $R_i = 011000$. Since $L_i = R_{i-1}$, we obtain

$$L_iR_i = 100110011000.$$

The output becomes the input for the next round.

For work on this and another simplified DES algorithm and how they behave under multiple encryption, see [Konikoff-Toplosky].

7.3 Differential Cryptanalysis

This section is rather technical and can be skipped on a first reading.

Differential cryptanalysis was introduced by Biham and Shamir around 1990, though it was probably known much earlier to the designers of DES at IBM and NSA. The idea is to compare the differences in the ciphertexts for suitably chosen pairs of plaintexts and thereby deduce information about the key. Note that the difference of two strings of bits can be found by XORing them. Because the key is introduced by XORing with $E(R_{i-1})$, looking at the XOR of the inputs removes the effect of the key at this stage and hence removes some of the randomness introduced by the key. We'll see that this allows us to deduce information as to what the key could be.

7.3.1 Differential Cryptanalysis for Three Rounds

We eventually want to describe how to attack the above system when it uses four rounds, but we need to start by analyzing three rounds. Therefore, we temporarily start with L_1R_1 instead of L_0R_0.

The situation is now as follows. We have obtained access to a three-round encryption device that uses the preceding procedure. We know all the inner workings of the encryption algorithm such as the S-boxes, but we do not know the key. We want to find the key by a chosen plaintext attack. We use various inputs L_1R_1 and obtain outputs L_4R_4.

We have

$$R_2 = L_1 \oplus f(R_1, K_2)$$
$$L_3 = R_2 = L_1 \oplus f(R_1, K_2)$$
$$R_4 = L_3 \oplus f(R_3, K_4) = L_1 \oplus f(R_1, K_2) \oplus f(R_3, K_4).$$

Suppose we have another message $L_1^*R_1^*$ with $R_1 = R_1^*$. For each i, let $R_i{}' = R_i \oplus R_i^*$ and $L_i{}' = L_i \oplus L_i^*$. Then $L_i{}'R_i{}'$ is the "difference" (or sum; we are working mod 2) of L_iR_i and $L_i^*R_i^*$. The preceding calculation applied to $L_1^*R_1^*$ yields a formula for R_4^*. Since we have assumed that $R_1 = R_1^*$, we have $f(R_1, K_2) = f(R_1^*, K_2)$. Therefore, $f(R_1, K_2) \oplus f(R_1^*, K_2) = 0$ and

$$R_4^{'} = R_4 \oplus R_4^* = L_1^{'} \oplus f(R_3, K_4) \oplus f(R_3^*, K_4).$$

This may be rearranged to

$$R_4^{'} \oplus L_1^{'} = f(R_3, K_4) \oplus f(R_3^*, K_4).$$

Finally, since $R_3 = L_4$ and $R_3^* = L_4^*$, we obtain

$$R_4^{'} \oplus L_1^{'} = f(L_4, K_4) \oplus f(L_4^*, K_4).$$

Note that if we know the input XOR, namely $L_1{}'R_1{}'$, and if we know the outputs L_4R_4 and $L_4^*R_4^*$, then we know everything in this last equation except K_4.

Now let's analyze the inputs to the S-boxes used to calculate $f(L_4, K_4)$ and $f(L_4^*, K_4)$. If we start with L_4, we first expand and then XOR with K_4 to obtain $E(L_4) \oplus K_4$, which are the bits sent to S_1 and S_2. Similarly, L_4^* yields $E(L_4^*) \oplus K_4$. The XOR of these is

$$E(L_4) \oplus E(L_4^*) = E(L_4 \oplus L_4^*) = E(L_4^{'})$$

(the first equality follows easily from the bit-by-bit description of the expansion function). Therefore, we know

1. the XORs of the inputs to the two S-boxes (namely, the first four and the last four bits of $E(L_4^{'})$);
2. the XORs of the two outputs (namely, the first three and the last three bits of $R_4^{'} \oplus L_1^{'}$).

Let's restrict our attention to S_1. The analysis for S_2 will be similar. It is fairly fast to run through all pairs of four-bit inputs with a given XOR (there are only 16 of them) and see which ones give a desired output XOR. These can be computed once for all and stored in a table.

For example, suppose we have input XOR equal to 1011 and we are looking for output XOR equal to 100. We can run through the input pairs (1011, 0000), (1010, 0001), (1001, 0010), ..., each of which has XOR equal to 1011, and look at the output XORs. We find that the pairs (1010, 0001) and (0001, 1010) both produce output XORs 100. For example, 1010 means we look at the second row, third column of S_1, which is 110. Moreover, 0001 means we look at the first row, second column, which is 010. The output XOR is therefore $110 \oplus 010 = 100$.

We know L_4 and L_4^*. For example, suppose $L_4 = 101110$ and $L_4^* = 000010$. Therefore, $E(L_4) = 10111110$ and $E(L_4^*) = 00000010$, so the inputs to S_1 are $1011 \oplus K_4^L$ and $0000 \oplus K_4^L$, where K_4^L denotes the left four bits of K_4. If we know that the output XOR for S_1 is 100, then $(1011 \oplus K_4^L,\ 0000 \oplus K_4^L)$ must be one of the pairs on the list we just calculated, namely (1010, 0001) and (0001, 1010). This means that $K_4^L = 0001$ or 1010.

If we repeat this procedure a few more times, we should be able to eliminate one of the two choices for K_4 and hence determine four bits of K. Similarly, using S_2, we find four more bits of K. We therefore know eight of the nine bits of K. The last bit can be found by trying both possibilities and seeing which one produces the same encryptions as the machine we are attacking.

Here is a summary of the procedure (for notational convenience, we describe it with both S-boxes used simultaneously, though in the examples we work with the S-boxes separately):

1. Look at the list of pairs with input $\text{XOR} = E(L_4^{'})$ and output $\text{XOR} = R_4^{'} \oplus L_1^{'}$.
2. The pair $\left(E(L_4) \oplus K_4,\ E(L_4^{*}) \oplus K_4\right)$ is on this list.
3. Deduce the possibilities for K_4.
4. Repeat until only one possibility for K_4 remains.

Example

We start with

$$L_1 R_1 = 000111011011$$

and the machine encrypts in three rounds using the key $K = 001001101$, though we do not yet know K. We obtain (note that since we are starting with $L_1 R_1$, we start with the shifted key $K_2 = 01001101$)

$$L_4 R_4 = 000011100101.$$

If we start with

$$L_1^{*} R_1^{*} = 101110011011$$

(note that $R_1 = R_1^{*}$), then

$$L_4^{*} R_4^{*} = 100100011000.$$

We have $E(L_4) = 00000011$ and $E(L_4^{*}) = 10101000$. The inputs to S_1 have XOR equal to 1010 and the inputs to S_2 have XOR equal to 1011. The S-boxes have output XOR $R_4^{'} \oplus L_{1}^{'}{}_1 = 111101 \oplus 101001 = 010100$, so the output XOR from S_1 is 010 and that from S_2 is 100.

For the pairs $(1001, 0011)$, $(0011, 1001)$, S_1 produces output XOR equal to 010. Since the first member of one of these pairs should be the left four bits of $E(L_4) \oplus K_4 = 0000 \oplus K_4$, the first four bits of K_4 are in $\{1001, 0011\}$. For the pairs $(1100, 0111)$, $(0111, 1100)$, S_2 produces output XOR equal to 100. Since the first member of one of these pairs should be the right four bits of $E(L_4) \oplus K_4 = 0011 \oplus K_4$, the last four bits of K_4 are in $\{1111, 0100\}$.

Now repeat (with the same machine and the same key K) and with

$$L_1R_1 = 010111011011 \text{ and } L_1^*R_1^* = 101110011011.$$

A similar analysis shows that the first four bits of K_4 are in $\{0011, 1000\}$ and the last four bits are in $\{0100, 1011\}$. Combining this with the previous information, we see that the first four bits of K_4 are 0011 and the last four bits are 0100. Therefore, $K = 00 * 001101$ (recall that K_4 starts with the fourth bit of K.

It remains to find the third bit of K. If we use $K = 000001101$, it encrypts L_1R_1 to 001011101010, which is not L_4R_4, while $K = 001001101$ yields the correct encryption. Therefore, the key is $K = 001001101$.

7.3.2 Differential Cryptanalysis for Four Rounds

Suppose now that we have obtained access to a four-round device. Again, we know all the inner workings of the algorithm except the key, and we want to determine the key. The analysis we used for three rounds still applies, but to extend it to four rounds we need to use more probabilistic techniques.

There is a weakness in the box S_1. If we look at the 16 input pairs with XOR equal to 0011, we discover that 12 of them have output XOR equal to 011. Of course, we expect on the average that two pairs should yield a given output XOR, so the present case is rather extreme. A little variation is to be expected; we'll see that this large variation makes it easy to find the key.

There is a similar weakness in S_2, though not quite as extreme. Among the 16 input pairs with XOR equal to 1100, there are eight with output XOR equal to 010.

Suppose now that we start with randomly chosen R_0 and R_0^* such that $R_0' = R_0 \oplus R_0^* = 001100$. This is expanded to $E(001100) = 00111100$. Therefore the input XOR for S_1 is 0011 and the input XOR for S_2 is 1100. With probability 12/16 the output XOR for S_1 will be 011, and with probability 8/16 the output XOR for S_2 will be 010. If we assume the outputs of the two S-boxes are independent, we see that the combined output XOR will be 011010 with probability (12/16)(8/16) = 3/8. Because the expansion function sends bits 3 and 4 to both S_1 and S_2, the two boxes cannot be assumed to have independent outputs, but 3/8 should still be a reasonable estimate for what happens.

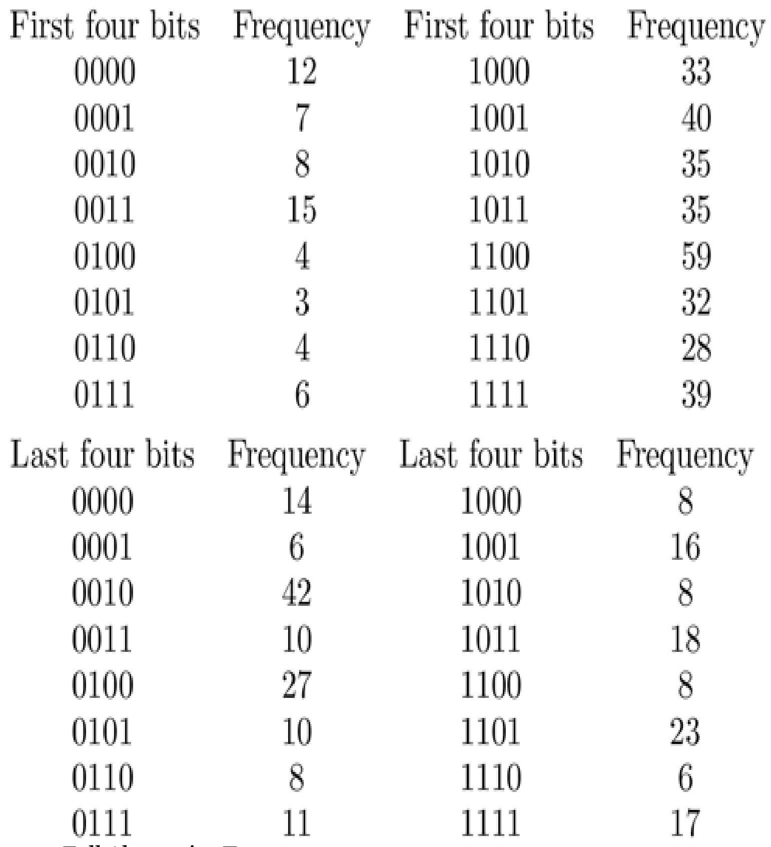
Now suppose we choose L_0 and L_0^* so that $L_0{}' = L_0 \oplus L_0^* = 011010$. Recall that in the encryption algorithm the output of the S-boxes is XORed with L_0 to obtain R_1. Suppose the output XOR of the S-boxes is 011010. Then $R_1{}' = 011010 \oplus L_0{}' = 000000$. Since $R_1{}' = R_1 \oplus R_1^*$, it follows that $R_1 = R_1^*$.

Putting everything together, we see that if we start with two randomly chosen messages with XOR equal to

$L_0^{'}R_0^{'} = 011010001100$, then there is a probability of around 3/8 that $L_1{}'R_1{}' = 001100000000$.

Here's the strategy for finding the key. Try several randomly chosen pairs of inputs with XOR equal to 011010001100. Look at the outputs L_4R_4 and $L_4^*R_4^*$. Assume that $L_1^{'}R_1^{'} = 001100000000$. Then use three-round differential cryptanalysis with $L_1^{'} = 001100$ and the known outputs to deduce a set of possible keys K_4. When $L_1^{'}R_1^{'} = 001100000000$, which should happen around 3/8 of the time, this list of keys will contain K_4, along with some other random keys. The remaining 5/8 of the time, the list should contain random keys. Since there seems to be no reason that any incorrect key should appear frequently, the correct key K_4 will probably appear in the lists of keys more often than the other keys.

Here is an example. Suppose we are attacking a four-round device. We try one hundred random pairs of inputs L_0R_0 and $L_0^*R_0^* = L_0R_0 \oplus 011010001100$. The frequencies of possible keys we obtain are in the following table. We find it easier to look at the first four bits and the last four bits of K_4 separately.



First four bits	Frequency	First four bits	Frequency
0000	12	1000	33
0001	7	1001	40
0010	8	1010	35
0011	15	1011	35
0100	4	1100	59
0101	3	1101	32
0110	4	1110	28
0111	6	1111	39

Last four bits	Frequency	Last four bits	Frequency
0000	14	1000	8
0001	6	1001	16
0010	42	1010	8
0011	10	1011	18
0100	27	1100	8
0101	10	1101	23
0110	8	1110	6
0111	11	1111	17

7.3-1 Full Alternative Text

It is therefore likely that $K_4 = 11000010$. Therefore, the key K is 10*110000.

To determine the remaining bit, we proceed as before. We can compute that 000000000000 is encrypted to 100011001011 using $K = 101110000$ and is encrypted to 001011011010 using $K = 100110000$. If the machine we are attacking encrypts 000000000000 to 100011001011, we conclude that the second key cannot be correct, so the correct key is probably $K = 101110000$.

The preceding attack can be extended to more rounds by extensions of these methods. It might be noticed that we could have obtained the key at least as quickly by simply running through all possibilities for the key. That is certainly true in this simple model. However, in more elaborate systems such as DES, differential cryptanalytic techniques are much more efficient than exhaustive searching through all keys, at least until the number of rounds becomes fairly large. In particular, the reason that DES uses 16 rounds appears to be because differential cryptanalysis is more efficient than exhaustive search until 16 rounds are used.

There is another attack on DES, called **linear cryptanalysis**, that was developed by Mitsuru Matsui [Matsui]. The main ingredient is an approximation of DES by a linear function of the input bits. It is theoretically faster than an exhaustive search for the key and requires around 2^{43} plaintext–ciphertext pairs to find the key. It seems that the designers of DES had not anticipated linear cryptanalysis. For details of the method, see [Matsui].

7.4 DES

A block of plaintext consists of 64 bits. The key has 56 bits, but is expressed as a 64-bit string. The 8th, 16th, 24th, ..., bits are parity bits, arranged so that each block of eight bits has an odd number of 1s. This is for error detection purposes. The output of the encryption is a 64-bit ciphertext.

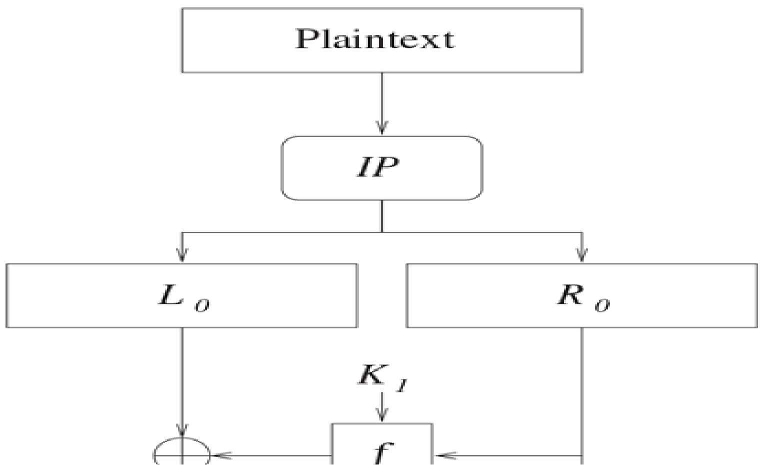
The DES algorithm, depicted in Figure 7.4, starts with a plaintext m of 64 bits, and consists of three stages:

1. The bits of m are permuted by a fixed initial permutation to obtain $m_0 = IP(m)$. Write $m_0 = L_0R_0$, where L_0 is the first 32 bits of m_0 and R_0 is the last 32 bits.
2. For $1 \le i \le 16$, perform the following:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i), \end{aligned}$$

   where K_i is a string of 48 bits obtained from the key K and f is a function to be described later.
3. Switch left and right to obtain $R_{16}L_{16}$, then apply the inverse of the initial permutation to get the ciphertext $c = IP^{-1}(R_{16}L_{16})$.

Figure 7.4 The DES Algorithm



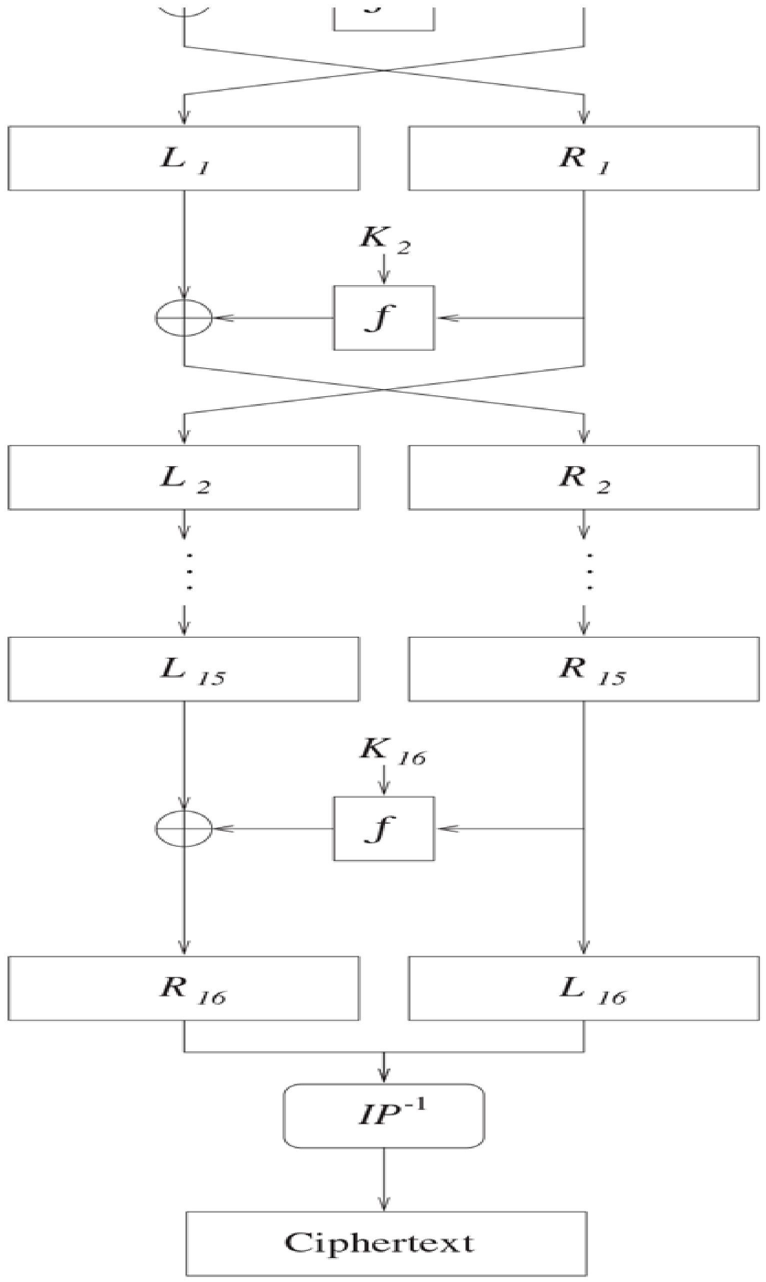
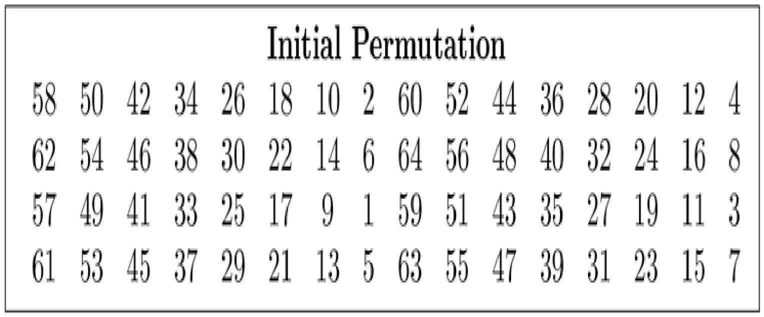


Figure 7.4 Full Alternative Text

Decryption is performed by exactly the same procedure, except that the keys $K_1, \ \dots \ , K_{16}$ are used in reverse order. The reason this works is the same as for the simplified system described in Section 7.2. Note that the left–right switch in step 3 of the DES algorithm means that we do not have to do the left–right switch that was needed for decryption in Section 7.2.

We now describe the steps in more detail.

The initial permutation, which seems to have no cryptographic significance, but which was perhaps designed to make the algorithm load more efficiently into chips that were available in 1970s, can be described by the Initial Permutation table. This means that the 58th bit of m becomes the first bit of m_0, the 50th bit of m becomes the second bit of m_0, etc.



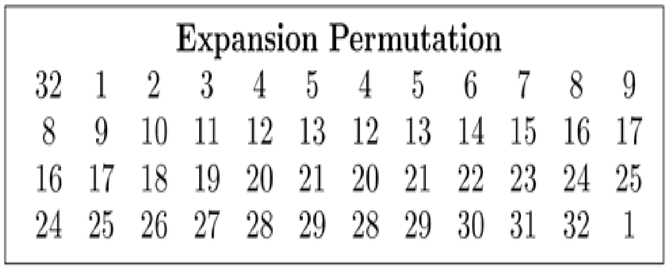
Initial Permutation

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

7.4-2 Full Alternative Text

The function $f(R, K_i)$, depicted in Figure 7.5, is described in several steps.

1. First, R is expanded to $E(R)$ by the following table.



Expansion Permutation

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

7.4-3 Full Alternative Text

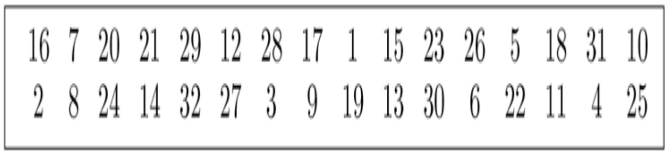
This means that the first bit of $E(R)$ is the 32nd bit of R, etc. Note that $E(R)$ has 48 bits.

2. Compute $E(R) \oplus K_i$, which has 48 bits, and write it as $B_1 B_2 \cdots B_8$, where each B_j has six bits.

3. There are eight S-boxes $S_1, \dots, S_8$, given on page 150. B_j is the input for S_j. Write $B_j = b_1 b_2 \cdots b_6$. The row of the S-box is specified by $b_1 b_6$ while $b_2 b_3 b_4 b_5$ determines the column. For example, if $B_3 = 001001$, we look at the row 01, which is the second row (00 gives the first row) and column 0100, which is the 5th column (0100 represents 4 in binary; the first column is numbered 0, so the fifth is labeled 4). The entry in S_3 in this

location is 3, which is 3 in binary. Therefore, the output of S_3 is 0011 in this case. In this way, we obtain eight four-bit outputs $C_1,\, C_2,\, \ldots C_8$.

4. The string $C_1 C_2 \cdots C_8$ is permuted according to the following table.

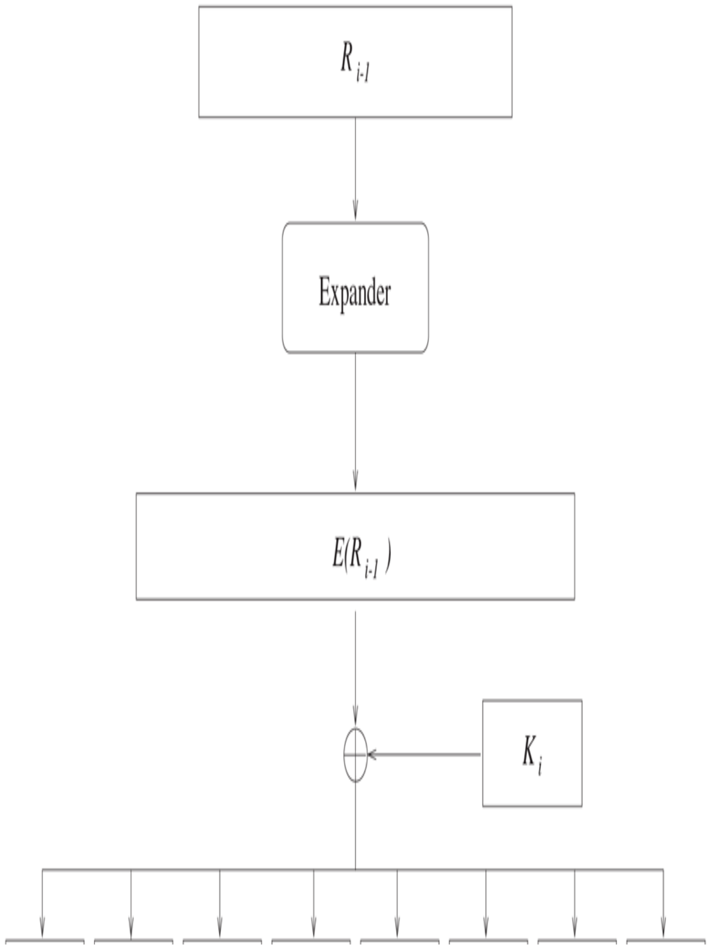


16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

7.4-4 Full Alternative Text

The resulting 32-bit string is $f(R,\, K_j)$.

Figure 7.5 The DES Function $f(R_{i-1},\, K_i)$



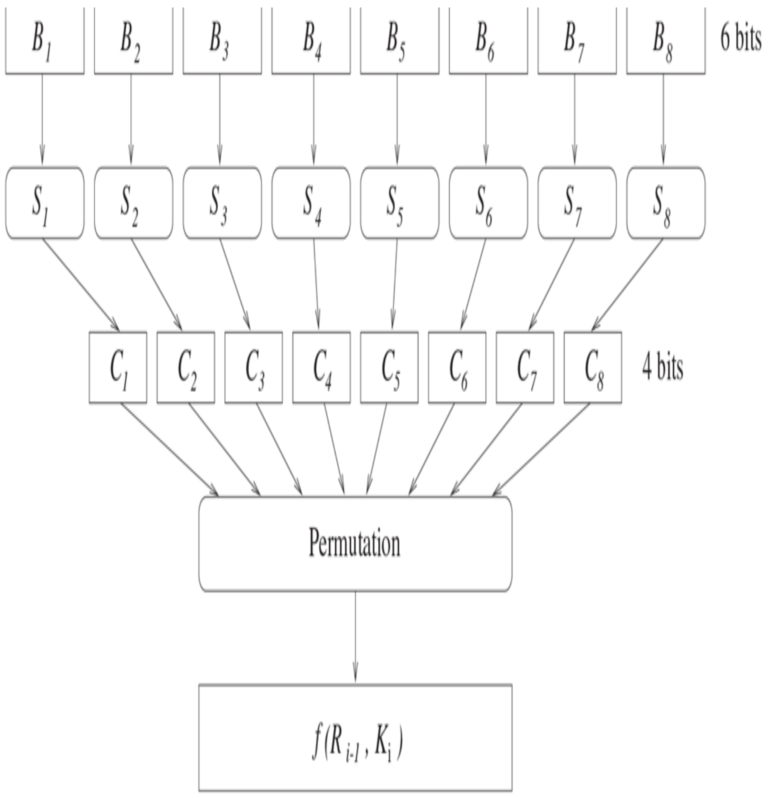
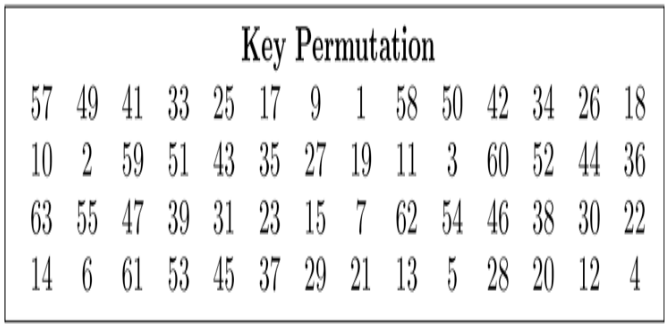


Figure 7.5 Full Alternative Text

Finally, we describe how to obtain $K_1, \ldots K_{16}$. Recall that we start with a 64-bit K.

1. The parity bits are discarded and the remaining bits are permuted by the following table.



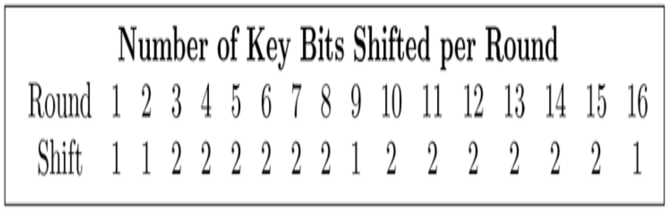
Key Permutation

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

7.4-5 Full Alternative Text

Write the result as C_0D_0, where C_0 and D_0 have 28 bits.

2. For $1 \leq i \leq 16$, let $C_i = LS_i(C_{i-1})$ and $D_i = LS_i(D_{i-1})$. Here LS_i means shift the input one or two places to the left, according to the following table.

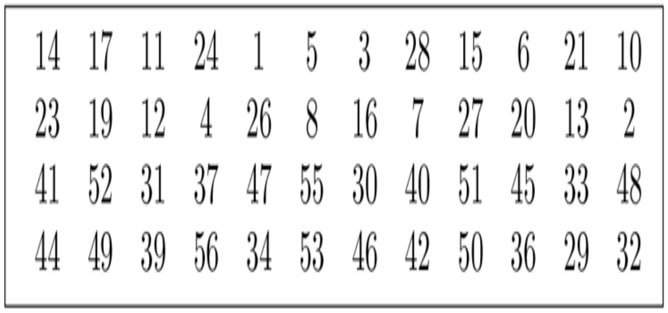


Number of Key Bits Shifted per Round

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

7.4-6 Full Alternative Text

3. 48 bits are chosen from the 56-bit string $C_i D_i$ according to the following table. The output is K_i.



14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

7.4-7 Full Alternative Text

It turns out that each bit of the key is used in approximately 14 of the 16 rounds.

A few remarks are in order. In a good cipher system, each bit of the ciphertext should depend on all bits of the plaintext. The expansion $E(R)$ is designed so that this will happen in only a few rounds. The purpose of the initial permutation is not completely clear. It has no cryptographic purpose. The S-boxes are the heart of the algorithm and provide the security. Their design was somewhat of a mystery until IBM published the following criteria in the early 1990s (for details, see [Coppersmith1]).

1. Each S-box has six input bits and four output bits. This was the largest that could be put on one chip in 1974.
2. The outputs of the S-boxes should not be close to being linear functions of the inputs (linearity would have made the system much easier to analyze).
3. Each row of an S-box contains all numbers from 0 to 15.
4. If two inputs to an S-box differ by one bit, the outputs must differ by at least two bits.

5. If two inputs to an S-box differ in exactly the middle two bits, then the outputs differ in at least two bits.

6. If two inputs to an S-box differ in their first two bits but have the same last two bits, the outputs must be unequal.

7. There are 32 pairs of inputs having a given XOR. For each of these pairs, compute the XOR of the outputs. No more than eight of these output XORs should be the same. This is clearly to avoid an attack via differential cryptanalysis.

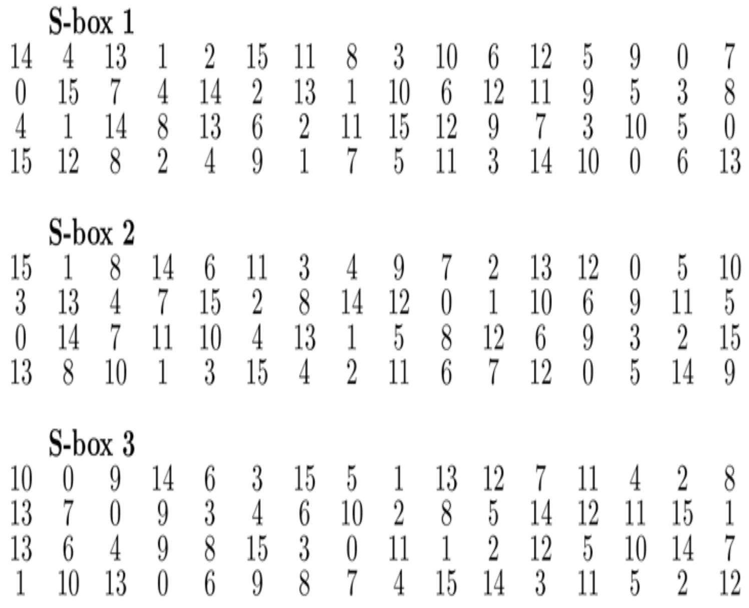
8. A criterion similar to (7), but involving three S-boxes.

In the early 1970s, it took several months of searching for a computer to find appropriate S-boxes. Now, such a search could be completed in a very short time.

7.4.1 DES Is Not a Group

One possible way of effectively increasing the key size of DES is to double encrypt. Choose keys K_1 and K_2 and encrypt a plaintext P by $E_{K_2}(E_{K_1}(P))$. Does this increase the security?

S-Boxes



S-box 1

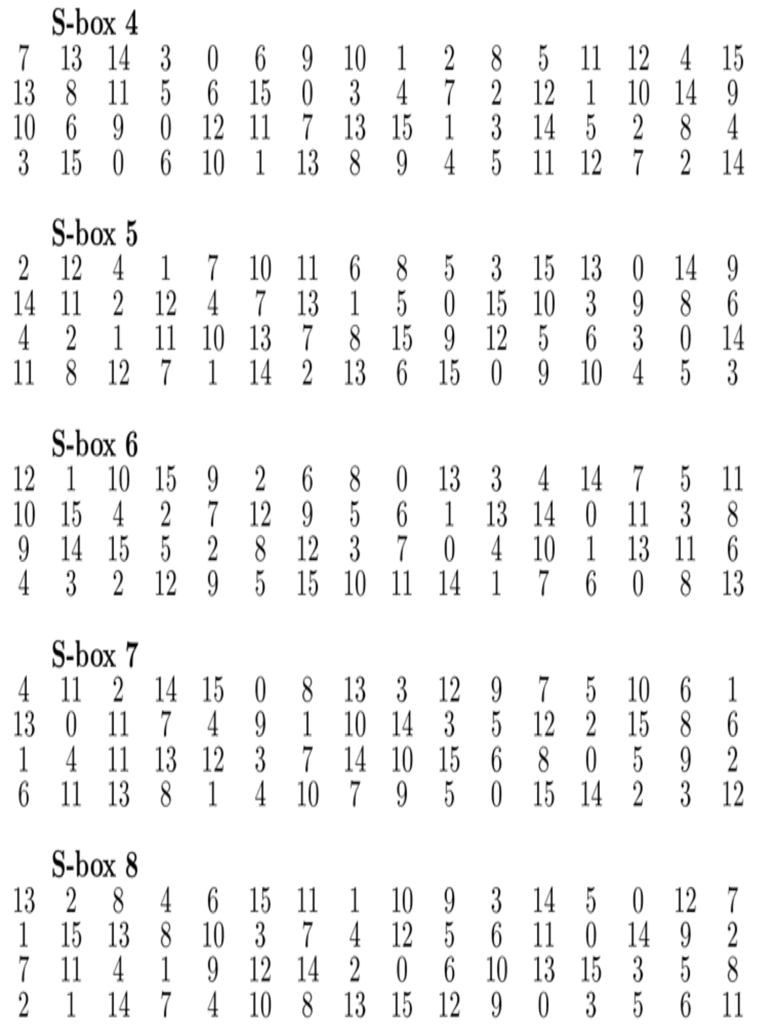
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-box 2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-box 3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12



S-box 4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-box 5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-box 6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-box 7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-box 8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Full Alternative Text

Meet-in-the-middle attacks on cryptosystems are discussed in Section 6.5. It is pointed out that, if an attacker has sufficient memory, double encryption provides little extra protection. Moreover, if a cryptosystem is such that double encryption is equivalent to a single encryption, then there is no additional security obtained by double encryption.

In addition, if double encryption is equivalent to single encryption, then the (single encryption) cryptosystem is much less secure than one might guess initially (see Exercise 3 in Chapter 12). If this were true for DES, for example, then an exhaustive search through all 2^{56} keys could be replaced by a search of length around 2^{28}, which would be quite easy to do.

For affine ciphers (Section 2.2) and for RSA (Chapter 9), double encrypting with two keys K_1 and K_2 is equivalent to encrypting with a third key K_3. Is the same true for DES? Namely, is there a key K_3 such that $E_{K_3} = E_{K_2} E_{K_1}$? This question is often rephrased in the equivalent form “Is DES a group?” (The reader who is unfamiliar with group theory can ask “Is DES closed under composition?”)

Fortunately, it turns out that DES is not a group. We sketch the proof. For more details, see [Campbell-Wiener]. Let E_0 represent encryption with the key consisting entirely of 0s and let E_1 represent encryption with the key consisting entirely of 1s. These keys are weak for cryptographic purposes (see Exercise 5). Moreover, D. Coppersmith found that applying $E_1 \circ E_0$ repeatedly to certain plaintexts yielded the original plaintext after around 2^{32} iterations. A sequence of encryptions (for some plaintext P)

$$E_1E_0(P),\ E_1E_0(E_1E_0(P)),\ E_1E_0(E_1E_0(E_1E_0(P))),\ \ldots,\ (E_1E_0)^n(P) = P,$$

where n is the smallest positive integer such that $(E_1E_0)^n(P) = P$, is called a cycle of length n.

Lemma

If m is the smallest positive integer such that $(E_1E_0)^m(P) = P$ for all P, and n is the length of a cycle (so $(E_1E_0)^n(P_0) = P_0$ for a particular P_0), then n divides m.

Proof. Divide n into m, with remainder r. This means that $m = nq + r$ for some integer q, and $0 \le r < n$. Since $(E_1E_0)^n(P_0) = P_0$, encrypting q times with $(E_1E_0)^n$ leaves P_0 unchanged. Therefore,

$$P_0 = (E_1E_0)^m(P_0) = (E_1E_0)^r(E_1E_0)^{nq}(P_0) = (E_1E_0)^r(P_0).$$

Since n is the smallest positive integer such that $(E_1E_0)^n(P_0) = P_0$, and $0 \le r < n$, we must have $r = 0$. This means that $m = nq$, so n divides m.

Suppose now that DES is closed under composition. Then $E_1E_0 = E_K$ for some key K. Moreover, $E_K^2, E_K^3, \ldots$ are also represented by DES keys. Since there are only 2^{56} possible keys, we must have $E_K^j = E_K^i$ for some integers i, j with $0 \le i < j \le 2^{56}$ (otherwise we would have $2^{56} + 1$ distinct encryption keys). Decrypt i times: $E_K^{j-i} = D_K^i E_K^j = D_K^i E_K^i$, which is the identity map. Since $0 < j - i \le 2^{56}$, the smallest positive integer m such that E_K^m is the identity map also satisfies $m \le 2^{56}$.

Coppersmith found the lengths of the cycles for 33 plaintexts P_0. By the lemma, m is a multiple of these cycle lengths. Therefore, m is greater than or equal to the least common multiple of these cycle lengths, which turned out to be around 10^{277}. But if DES is closed under composition, we showed that $m \le 2^{56}$. Therefore, DES is not closed under composition.

7.5 Breaking DES

DES was the standard cryptographic system for the last 20 years of the twentieth century, but, in the latter half of this period, DES was showing signs of age. In this section we discuss the breaking of DES.

From 1975 onward, there were questions regarding the strength of DES. Many in the academic community complained about the size of the DES keys, claiming that a 56-bit key was insufficient for security. In fact, a few months after the NBS release of DES, Whitfield Diffie and Martin Hellman published a paper titled "Exhaustive cryptanalysis of the NBS Data Encryption Standard" [Diffie-Hellman2] in which they estimated that a machine could be built for $20 million (in 1977 dollars) that would crack DES in roughly a day. This machine's purpose was specifically to attack DES, which is a point that we will come back to later.

In 1987 DES came under its second five-year review. At this time, NBS asked for suggestions whether to accept the standard for another period, to modify the standard, or to dissolve the standard altogether. The discussions regarding DES saw NSA opposing the recertification of DES. The NBS argued at that time that DES was beginning to show signs of weakness, given the current of level of computing power, and proposed doing away with DES entirely and replacing it with a set of NSA-designed algorithms whose inner workings would be known only to NSA and be well protected from reverse engineering techniques. This proposal was turned down, partially due to the fact that several key American industries would be left unprotected while replacement algorithms were put in place. In the end, DES was reapproved as a standard, yet in the process it was acknowledged that DES was showing signs of weakness.

Five years later, after NBS had been renamed NIST, the next five-year review came around. Despite the weaknesses mentioned in 1987 and the technology advances that had taken place in five years, NIST recertified the DES algorithm in 1992.

In 1993, Michael Wiener, a researcher at Bell-Northern Research, proposed and designed a device that would attack DES more efficiently than ever before. The idea was to use the already well-developed switching technology available to the telephone industry.

The year 1996 saw the formulation of three basic approaches for attacking symmetric ciphers such as DES. The first method was to do distributive computation across a vast collection of machines. This had the advantage that it was relatively cheap, and the cost that was involved could be easily distributed over many people. Another approach was to design custom architecture (such as Michael Wiener's idea) for attacking DES. This promised to be more effective, yet also more expensive, and could be considered as the high-end approach. The middle-of-the-line approach involved programmable logic arrays and has received the least attention to date.

In all three of these cases, the most popular approach to attacking DES was to perform an exhaustive search of the keyspace. For DES this seemed to be reasonable since, as mentioned earlier, more complicated cryptanalytic techniques had failed to show significant improvement over exhaustive search.

The distributive computing approach to breaking DES became very popular, especially with the growing popularity of the Internet. In 1997 the RSA Data Security company issued a challenge to find the key and crack a

DES encrypted message. Whoever cracked the message would win a $10,000 prize. Only five months after the announcement of the 1997 DES Challenge, Rocke Verser submitted the winning DES key. What is important about this is that it represents an example where the distributive computing approach had successfully attacked DES. Rocke Verser had implemented a program where thousands of computers spread over the Internet had managed to crack the DES cipher. People volunteered time on their personal (and corporate) machines, running Verser's program under the agreement that Verser would split the winnings 60% to 40% with the owner of the computer that actually found the key. The key was finally found by Michael Sanders. Roughly 25% of the DES keyspace had been searched by that time. The DES Challenge phrase decrypted to "Strong cryptography makes the world a safer place."

In the following year, RSA Data Security issued DES Challenge II. This time the correct key was found by Distributed Computing Technologies, and the message decrypted to "Many hands make light work." The key was found after searching roughly 85% of the possible keys and was done in 39 days. The fact that the winner of the second challenge searched more of the keyspace and performed the task quicker than the first task shows the dramatic effect that a year of advancement in technology can have on cryptanalysis.

In the summer of 1998 the Electronic Frontier Foundation (EFF) developed a project called DES Cracker whose purpose was to reveal the vulnerability of the DES algorithm when confronted with a specialized architecture. The DES Cracker project was founded on a simple principle: The average computer is ill suited for the task of cracking DES. This is a reasonable statement since ordinary computers, by their very nature, are multipurpose machines that are designed to handle generic tasks such as running an operating system or even playing a computer game or two. What the EFF team proposed to do was build specialized hardware that would take advantage of the parallelizable nature of the exhaustive search. The team had a budget of $200,000.

We now describe briefly the architecture that the EFF team's research produced. For more information regarding the EFF Cracker as well as the other tasks their cracker was designed to handle, see [Gilmore].

The EFF DES Cracker consisted of basically three main parts: a personal computer, software, and a large collection of specialized chips. The computer was connected to the array of chips and the software oversaw the tasking of each chip. For the most part, the software didn't interact much with the hardware; it just gave the chips the necessary information to start processing and waited until the chips returned candidate keys. In this sense, the hardware efficiently eliminated a large number of invalid keys and only returned keys that were potentially promising. The software then processed each of the promising candidate keys on its own, checking to see if one of the promising keys was in fact the actual key.

The DES Cracker took a 128-bit (16-byte) sample of ciphertext and broke it into two 64-bit (8-byte) blocks of text. Each chip in the EFF DES Cracker consisted of 24 search units. A search unit was a subset of a chip whose task was to take a key and two 64-bit blocks of ciphertext and attempt to decrypt the first 64-bit block using the key. If the "decrypted" ciphertext looked interesting, then the search unit decrypted the second block and checked to see if that "decrypted" ciphertext was also interesting. If both decrypted texts were interesting then the search unit told the software that the key it checked was promising. If, when the first 64-bit block of ciphertext was decrypted, the decrypted text did not seem interesting enough, then the search unit incremented its key by 1 to form a new key. It then tried this new key, again checking to see if the result was interesting, and proceeded this way as it searched through its allotted region of keyspace.

How did the EFF team define an “interesting” decrypted text? First they assumed that the plaintext satisfied some basic assumption, for example that it was written using letters, numbers, and punctuation. Since the data they were decrypting was text, they knew each byte corresponded to an eight-bit character. Of the 256 possible values that an eight-bit character type represented, only 69 characters were interesting (the uppercase and lowercase alphabet, the numbers, the space, and a few punctuation marks). For a byte to be considered interesting, it had to contain one of these 69 characters, and hence had a $69/256$ chance of being interesting. Approximating this ratio to $1/4$, and assuming that the decrypted bytes are in fact independent, we see that the chance that an 8-byte block of decrypted text was interesting is $1/4^8 = 1/65536$. Thus only $1/65536$ of the keys it examined were considered promising.

This was not enough of a reduction. The software would still spend too much time searching false candidates. In order to narrow down the field of promising key candidates even further, it was necessary to use the second 8-byte block of text. This block was decrypted to see if the result was interesting. Assuming independence between the blocks, we get that only $1/4^{16} = 1/65536^2$ of the keys could be considered promising. This significantly reduced the amount of keyspace that the software had to examine.

Each chip consisted of 24 search units, and each search unit was given its own region of the keyspace that it was responsible for searching. A single 40-MHz chip would have taken roughly 38 years to search the entire keyspace. To reduce further the amount of time needed to process the keys, the EFF team used 64 chips on a single circuit board, then 12 boards to each chassis, and finally two chassis were connected to the personal computer that oversaw the communication with the software.

The end result was that the DES Cracker consisted of about 1500 chips and could crack DES in roughly 4.5 days on average. The DES Cracker was by no means an optimum model for cracking DES. In particular, each of the chips that it used ran at 40 MHz, which is slow by modern standards. Newer models could certainly be produced in the future that employ chips running at much faster clock cycles.

This development strongly indicated the need to replace DES. There were two main approaches to achieving increased security. The first used DES multiple times and led to the popular method called Triple DES or 3DES. Multiple encryption for block ciphers is discussed in Section 6.4.

The second approach was to find a new system that employs a larger key size than 56 bits. This led to AES, which is discussed in Chapter 8.

7.6 Password Security

When you log in to a computer and enter your password, the computer checks that your password belongs to you and then grants access. However, it would be quite dangerous to store the passwords in a file in the computer. Someone who obtains that file would then be able to open anyone's account. Making the file available only to the computer administrator might be one solution; but what happens if the administrator makes a copy of the file shortly before changing jobs? The solution is to encrypt the passwords before storing them.

Let $f(x)$ be a **one-way function**. This means that it is easy to compute $f(x)$, but it is very difficult to solve $y = f(x)$ for x. A password x can then be stored as $f(x)$, along with the user's name. When the user logs in, and enters the password x, the computer calculates $f(x)$ and checks that it matches the value of $f(x)$ corresponding to that user. An intruder who obtains the password file will have only the value of $f(x)$ for each user. To log in to the account, the intruder needs to know x, which is hard to compute since $f(x)$ is a one-way function.

In many systems, the encrypted passwords are stored in a public file. Therefore, anyone with access to the system can obtain this file. Assume the function $f(x)$ is known. Then all the words in a dictionary, and various modifications of these words (writing them backward, for example) can be fed into $f(x)$. Comparing the results with the password file will often yield the passwords of several users.

This **dictionary attack** can be partially prevented by making the password file not publicly available, but there is still the problem of the departing (or fired) computer administrator. Therefore, other ways of making the information more secure are also needed.

Here is another interesting problem. It might seem desirable that $f(x)$ can be computed very quickly. However, a slightly slower $f(x)$ can slow down a dictionary attack. But slowing down $f(x)$ too much could also cause problems. If $f(x)$ is designed to run in a tenth of a second on a very fast computer, it could take an unacceptable amount of time to log in on a slower computer. There doesn't seem to be a completely satisfactory way to resolve this.

One way to hinder a dictionary attack is with what is called **salt**. Each password is randomly padded with an additional 12 bits. These 12 bits are then used to modify the function $f(x)$. The result is stored in the password file, along with the user's name and the values of the 12-bit salt. When a user enters a password x, the computer finds the value of the salt for this user in the file, then uses it in the computation of the modified $f(x)$, which is compared with the value stored in the file.

When salt is used and the words in the dictionary are fed into $f(x)$, they need to be padded with each of the $2^{12} = 4096$ possible values of the salt. This slows down the computations considerably. Also, suppose an attacker stores the values of $f(x)$ for all the words in the dictionary. This could be done in anticipation of attacking several different password files. With salt, the storage requirements increase dramatically, since each word needs to be stored 4096 times.

The main purpose of salt is to stop attacks that aim at finding a random person's password. In particular, it makes the set of poorly chosen passwords somewhat more secure. Since many people use weak passwords, this is desirable. Salt does not slow down an attack against an individual password (except by preventing use of over-the-counter DES chips; see below). If Eve wants to find Bob’s password and has access to the password file, she finds the value of the salt used for Bob and tries a dictionary attack, for example, using only this value of salt corresponding to Bob. If Bob’s password is not in the dictionary, this will fail, and Eve may have to resort to an exhaustive search of all possible passwords.

In many Unix password schemes, the one-way function was based on DES. The first eight characters of the password are converted to seven-bit ASCII (see Section 4.1). These 56 bits become a DES key. If the password is shorter than eight symbols, it is padded with zeros to obtain the 56 bits. The “plaintext” of all zeros is then encrypted using 25 rounds of DES with this key. The output is stored in the password file. The function

$$\text{password} \quad \rightarrow \quad \text{output}$$

is believed to be one-way. Namely, we know the “ciphertext,” which is the output, and the “plaintext,” which is all zeros. Finding the key, which is the password, amounts to a known plaintext attack on DES, which is generally assumed to be difficult.

In order to increase security, salt is added as follows. A random 12-bit number is generated as the salt. Recall that in DES, the expansion function E takes a 32-bit input R (the right side of the input for the round) and expands it to 48 bits $E(R)$. If the first bit of the salt is 1, the first and 25th bits of $E(R)$ are swapped. If the second bit of the salt is 1, the second and 26th bits of $E(R)$ are swapped. This continues through the 12th bit of the salt. If it is 1, the 12th and 36th bits of $E(R)$ are swapped. When a bit of the salt is 0, it causes no swap. If the salt is all zero, then no swaps occur and we are working with the usual DES. In this way, the salt means that 4096 variations of DES are possible.

One advantage of using salt to modify DES is that someone cannot use high-speed DES chips to compute the one-way function when performing a dictionary attack. Instead, a chip would need to be designed that tries all 4096 modifications of DES caused by the salt; otherwise the attack could be performed with software, which is much slower.

Salt in any password scheme is regarded by many as a temporary measure. As storage space increases and computer speed improves, a factor of 4096 quickly fades, so eventually a new system must be developed.

For more on password protocols, see Section 12.6.

7.7 Exercises

1. Consider the following DES-like encryption method. Start with a message of $2n$ bits. Divide it into two blocks of length n (a left half and a right half): M_0M_1. The key K consists of k bits, for some integer k. There is a function $f(K, M)$ that takes an input of k bits and n bits and gives an output of n bits. One round of encryption starts with a pair M_jM_{j+1}. The output is the pair $M_{j+1}M_{j+2}$, where

   $$M_{j+2} = M_j \oplus f(K, M_{j+1}).$$

   ($\oplus$ means XOR, which is addition mod 2 on each bit). This is done for m rounds, so the ciphertext is M_mM_{m+1}.

   1. If you have a machine that does the m-round encryption just described, how would you use the same machine to decrypt the ciphertext M_mM_{m+1} (using the same key K)? Prove that your decryption method works.

   2. Suppose K has n bits and $f(K, M) = K \oplus M$, and suppose the encryption process consists of $m = 2$ rounds. If you know only a ciphertext, can you deduce the plaintext and the key? If you know a ciphertext and the corresponding plaintext, can you deduce the key? Justify your answers.

   3. Suppose K has n bits and $f(K, M) = K \oplus M$, and suppose the encryption process consists of $m = 3$ rounds. Why is this system not secure?

2. Bud gets a budget 2-round Feistel system. It uses a 32-bit L, a 32-bit R, and a 32-bit key K. The function is $f(R, K) = R \oplus K$, with the same key for each round. Moreover, to avoid transmission errors, he always uses a 32-bit message M and lets $L_0 = R_0 = M$. Eve does not know Bud's key, but she obtains the ciphertext for one of Bud's encryptions. Describe how Eve can obtain the plaintext M and the key K.

3. As described in Section 7.6, a way of storing passwords on a computer is to use DES with the password as the key to encrypt a fixed plaintext (usually 000 . . . 0). The ciphertext is then stored in the file. When you log in, the procedure is repeated and the ciphertexts are compared. Why is this method more secure than the similar-sounding method of using the password as the plaintext and using a fixed key (for example, $000\ldots0$)?

4. Nelson produces budget encryption machines for people who cannot afford a full-scale version of DES. The encryption consists of one round of a Feistel system. The plaintext has 64 bits and is divided into a left half L and a right half R. The encryption uses a function $f(R)$ that takes an input string of 32 bits and outputs a string of 32 bits. (There is no key; anyone naive enough to buy this system should not be trusted to choose a key.) The left half of the ciphertext is $C_0 = R$ and the right half is $C_1 = L \oplus f(R)$. Suppose Alice uses one of these machines to encrypt and send a message to Bob. Bob has an identical machine. How does he use the machine to decrypt the ciphertext he receives? Show that this decryption works (do not quote results about Feistel systems; you are essentially justifying that a special case works).

5.
    1. Let $K = 111\ldots111$ be the DES key consisting of all 1s. Show that if $E_K(P) = C$, then $E_K(C) = P$, so encryption twice with this key returns the plaintext. (Hint: The round keys are sampled from K. Decryption uses these keys in reverse order.)

    2. Find another key with the same property as K in part (a).

6. Alice uses quadruple DES encryption. To save time, she chooses two keys, K_1 and K_2, and encrypts via $c = E_{K_1}(E_{K_1}(E_{K_2}(E_{K_2}(m))))$. One day, Alice chooses K_1 to be the key of all 1s and K_2 to be the key of all 0s. Eve is planning to do a meet-in-the-middle attack, but after examining a few plaintext–ciphertext pairs, she decides that she does not need to carry out this attack. Why? (Hint: Look at Exercise 5.)

7. For a string of bits S, let S denote the complementary string obtained by changing all the 1s to 0s and all the 0s to 1s (equivalently, $S = S \oplus 11111\ldots$). Show that if the DES key K encrypts P to C, then K encrypts P to C. (Hint: This has nothing to do with the structure of the S-boxes. To do the problem, just work through the encryption algorithm and show that the input to the S-boxes is the same for both encryptions. A key point is that the expansion of C is the complementary string for the expansion of C.)

8. Suppose we modify the Feistel setup as follows. Divide the plaintext into three equal blocks: L_0, M_0, R_0. Let the key for the i th round be K_i and let f be some function that produces the appropriate size output. The ith round of encryption is given by

$$L_i = R_{i-1},\ M_i = L_{i-1},\ R_i = f(K_i, R_{i-1}) \oplus M_{i-1}.$$

This continues for n rounds. Consider the decryption algorithm that starts with the ciphertext A_n, B_n, C_n and uses the algorithm

$$A_{i-1} = B_i,\ B_{i-1} = f(K_i, A_i) \oplus C_i,\ C_{i-1} = A_i.$$

This continues for n rounds, down to A_0, B_0, C_0. Show that $A_i = L_i$, $B_i = M_i$, $C_i = R_i$ for all i, so that the decryption algorithm returns the plaintext. (Remark: Note that the decryption algorithm is similar to the encryption algorithm, but cannot be implemented on the same machine as easily as in the case of the Feistel system.)

9. Suppose $E_K(M)$ is the DES encryption of a message M using the key K. We showed in Exercise 7 that DES has the complementation property, namely that if $y = E_K(M)$ then $y = E_{K}(M)$, where M is the bit complement of M. That is, the bitwise complement of the key and the plaintext result in the bitwise complement of the DES ciphertext. Explain how an adversary can use this property in a brute force, chosen plaintext attack to reduce the expected number of keys that would be tried from 2^{55} to 2^{54}. (Hint: Consider a chosen plaintext set of (M_1, C_1) and (M_1, C_2)).

7.8 Computer Problems

1. (For those who are comfortable with programming)

    1. Write a program that performs one round of the simplified DES-type algorithm presented in Section 7.2.

    2. Create a sample input bitstring, and a random key. Calculate the corresponding ciphertext when you perform one round, two rounds, three rounds, and four rounds of the Feistel structure using your implementation. Verify that the decryption procedure works in each case.

    3. Let $E_K(M)$ denote four-round encryption using the key K. By trying all 2^9 keys, show that there are no weak keys for this simplified DES-type algorithm. Recall that a weak key is one such that when we encrypt a plaintext twice we get back the plaintext. That is, a weak key K satisfies $E_K(E_K(M)) = M$ for every possible M. (Note: For each key K, you need to find some M such that $E_K(E_K(M)) \neq M$.)

    4. Suppose you modify the encryption algorithm $E_K(M)$ to create a new encryption algorithm $E'_K(M)$ by swapping the left and right halves after the four Feistel rounds. Are there any weak keys for this algorithm?

2. Using your implementation of $E_K(M)$ from Computer Problem 1(b), implement the CBC mode of operation for this simplified DES-type algorithm.

    1. Create a plaintext message consisting of 48 bits, and show how it encrypts and decrypts using CBC.

    2. Suppose that you have two plaintexts that differ in the 14th bit. Show the effect that this has on the corresponding ciphertexts.