

Data Encryption Standard (DES)

Implementation in C

Cryptographic and Security Implementations

T. K. Garai



Data Encryption Standard (DES)

Implementation in C

by

Student Name	Roll Number
Tamas Kanti Garai	CrS2116

Course: M.Tech
Department: Cryptology and Security
Instructor: **Dr. Sabyasachi Karati**
Project Duration: August-September, 2022



Contents

1 Overview	1
2 Introduction to DES	2
3 Overview of DES	3
4 Internal Structures and Their Implementations	5
4.1 Initial and Final Permutation	5
4.2 The f -Function	6
4.2.1 Expansion Function	6
4.2.2 S-Boxes	7
4.2.3 Permutation P	9
4.3 Key Schedule	9
4.4 Decryption	12
4.4.1 Reversed Key Schelude	12
5 C-Code for DES	16
6 Modes of Encryption	25
6.1 Output Feedback (OFB) Mode	25
7 C-code for OFB using DES	27
References	37

List of Figures

3.1	DES block cipher	3
3.2	Iterative structure of DES	3
3.3	The Feistel structure of DES	4
4.1	Initial and Final Permutation	5
4.2	Bit map convention	5
4.3	How IP works!	6
4.4	Expansion permutation E	7
4.5	Example of decoding of the input $(110101)_2$ by S-box	7
4.6	S-Box S_1	8
4.7	S-Box S_2	8
4.8	S-Box S_3	8
4.9	S-Box S_4	8
4.10	S-Box S_5	9
4.11	S-Box S_6	9
4.12	S-Box S_7	9
4.13	S-Box S_8	9
4.14	P Permutation	9
4.15	f-Function	10
4.16	Bit-connections of $PC - 1$ and $PC - 2$	10
4.17	Key Schedule	11
4.18	Reversed Key Schedule	14
4.19	DES Decryption	15
6.1	Block Diagram of OFB	25

1

Overview

A *Block Cipher* is simply another name for a (strong) pseudorandom permutation. That is, a block cipher $F : \{0, 1\}^n \times \{0, 1\}^l \rightarrow \{0, 1\}^l$ is a keyed function such that, for all k , the function F_k defined by $F_k(x) = F(k, x)$ is a bijection (i.e., a permutation). Here n is the key length of F , and l is its block length. A block cipher is much more than just an encryption algorithm. It can be used as a versatile building block of so many cryptographic scheme. For instance, we can use them for building different types of block-based encryption schemes, and we can even use block ciphers for realizing stream ciphers. Block ciphers can also be used for constructing hash functions, message authentication codes which are also knowns as MACs, or key establishment protocols.

In this project our main focus would be to describe a very famous Block Cipher named *Data Encryption Standard(DES)* and implement that using c as a programming language. Then we will discuss about different ways of encryption are called *modes of operation* and implement one of the mode with DES as the building block.

2

Introduction to DES

In the year 1972 US National Bureau of Standards currently known as National Institute of Standards and Technology (NIST) initiated a request for proposals for a standardized cipher in the USA. The main motive was to set a standard secure cryptographic scheme which can be used for a variety of applications. In the year 1974 a team of cryptographers working at IBM submit a design and it became the most promising submission for encryption standard. The submitted algorithm was based on Lucifer. Lucifer is a Feistel cipher which encrypts blocks of 64 bits using a key size of 128 bits. It is known later that the National Security Agency(NSA) investigate its security in details and they suggested some changes in the design. Because of NSA's involvement some people worried that there must be some trapdoor in the algorithm which NSA is hiding from the world. Despite these type of criticism in 1977 the NBS released all specifications of the modifies IBM cipher as the *Data Encryption Standard(DES)* to the public.

Due to rapid increase of personal computers from early 1980's, DES is underwent serious scrutiny. However no serious flaw in the design found until the year 1990. Originally DES used as a standard for encryption for 10 years until 1987. After that some serious attack on DES came to visit and finally it was replaced by *Advanced Encryption Standard(AES)*. [3]

3

Overview of DES

Message block length is 64 bit and key length is 56 bit in DES. DES is a symmetric or private key cipher.

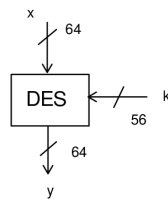


Figure 3.1: DES block cipher

Figure 3.2 shows the round structure of DES. From the main key k , 16 round keys, k_i 's are generated to use in 16 different but identical rounds. A detailed view on the internals of DES is shown in Fig. 3.3. Each round of DES is basically a Feistel Structure. It can lead to very strong ciphers if carefully designed. One advantage of Feistel networks is that encryption and decryption are the same operation only with a reversed key schedule. We discuss the Feistel network in the following: After the initial bitwise permutation IP of a 64-bit plaintext x , the plaintext is split into two halves L_0 and R_0 . These two 32-bit halves are the input to the Feistel network, which consists of 16 rounds. The right half R_i is fed into the function f . The output of the f function is XOR ed (denoted by the symbol \oplus) with the left 32-bit half L_i . Finally, the right and left half are swapped. This process repeats in the next round and can be expressed as:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, k)$$

where $i = 1, \dots, 16$.

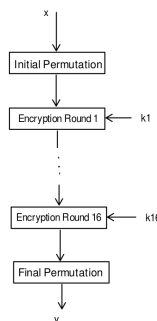


Figure 3.2: Iterative structure of DES

After round 16, the 32-bit halves L_{16} and R_{16} are swapped again, and the final permutation IP^{-1} is the last operation of DES. The final permutation IP^{-1} is the inverse of the initial permutation IP . In

each round, a round key k_i is derived from the main 56-bit key by key scheduling. It is crucial to note that the Feistel structure really only encrypts (decrypts) half of the input bits per each round, namely the left half of the input. The right half is copied to the next round unchanged. In particular, the right half is not encrypted with the f function.

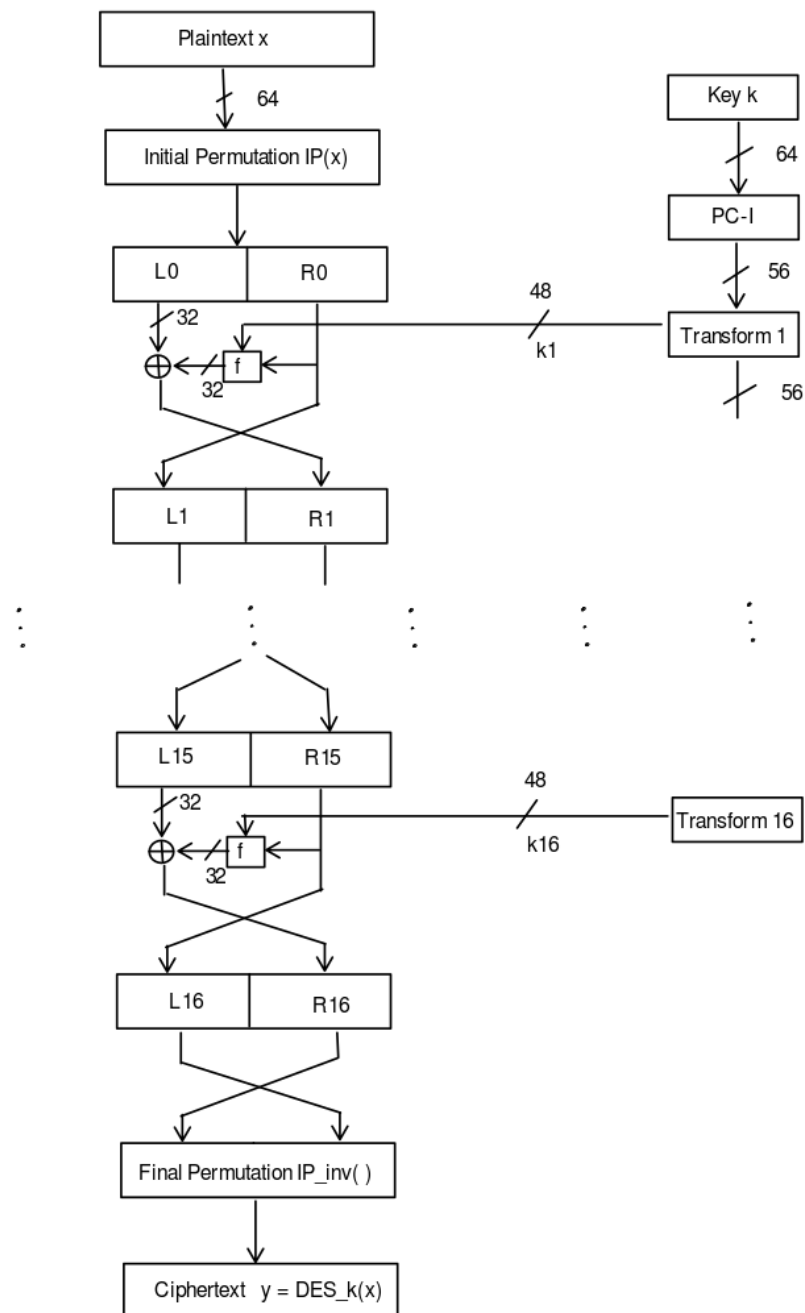


Figure 3.3: The Feistel structure of DES

4

Internal Structures and Their Implementations

The building blocks of Feistel Structures are the initial and final permutation, the actual DES rounds with its core, the f -function and the key scheduling.[1]

4.1. Initial and Final Permutation

The initial and final permutation in DES are bit wise permutations. It can be viewed as a simple cross-wiring. The two permutations are given by:

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

Figure 4.1: Initial and Final Permutation

These permutations do not contribute in the security of DES. The exact reason behind the existence of these two permutations is not known still. The permutation IP just take 64 bit message and swap its bit positions according to the given table in figure 4.1.

Figure 4.3 demonstrate how IP will manipulate the bit pattern of the 64 bit message.

My Implementation

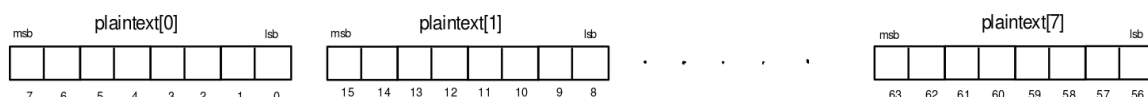


Figure 4.2: Bit map convention

Before I start explaining my implementation it is very important to understand what is my ordering of bits in the 8 byte of plain text. The ordering I've used throughout the program is shown in the figure 4.2.

I've invoked the permutation using the following c function:

```
1 void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size)
2 {
3     for (int i = 0; i < perm_size; i++)
4     {
5         int byte_number_a = (perm[i] - 1) >> 3;
6         int bit_number_a = ((perm[i] - 1) & 7);
7         int byte_number_b = i >> 3;
8         int bit_number_b = i & 7;
9         b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
10    }
11 }
```

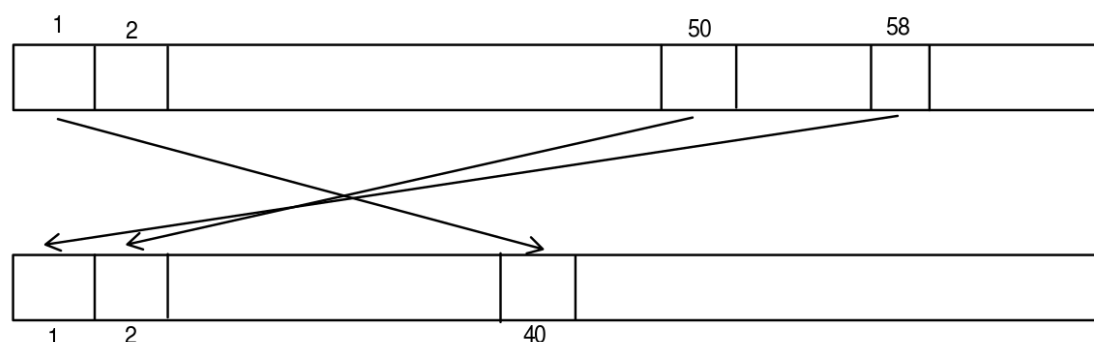


Figure 4.3: How IP works!

The c function taking two array of unsigned char variable a and b , the permutation according to which it will manipulate the bit string and the size of the permutation. It will manipulate the bit string a and keep the manipulated bit string in the array b .

I have taken the permutation as a one dimensional array. Then from the permutation I calculate the byte position of the input block $a[]$ in the variable $byte_number_a$ and then its bit position in the variable bit_number_a , then right shift the $a[byte_number_a]$ byte bit_number_a so that the desired bit become the *lsb* of the resultant, then 'and' it with 1 so that I get the desired bit. After that I left shift the obtained bit by bit_number_b and or it with $b[byte_number_b]$ so that after the completion of the loop, $b[]$ become the desired manipulated bit stream.

The same c function is used throughout the whole implementation whenever we invoke some permutation to manipulate bit string.

4.2. The f-Function

The most vital part of the DES is the f function. It plays the most crucial role in the security of DES. In i -th round it takes the right half R_{i-1} of the output of the previous round and the current round key k_i as input. The output of the f-function is used as an XOR-mask for encrypting the left half input bits L_{i-1} . The structure of the f-function is shown in the figure 4.15

4.2.1. Expansion Function

First 32-bit input is expanded to 48 bit by partitioning the input into eight 4 bit blocks and by expanding each block to 6 bits. This happen in the E-box, which is a special type of permutation, shown in figure 4.4

I've just used the same i_perm c function for this also.

My Implementation

The c function for expanding is as follows:

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

Figure 4.4: Expansion permutation E

```

1 void expand(unsigned char *a, unsigned char *c)
2 {
3     unsigned char b[6] = {0};
4     int i, j;
5     i_perm(a, b, E, 48);
6     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3)
7     {
8         c[i] = b[j] & 0x3f;
9         c[i + 1] = b[j] >> 6;
10        c[i + 1] |= (b[j + 1] & 0xf) << 2;
11        c[i + 2] = b[j + 1] >> 4;
12        c[i + 2] |= (b[j + 2] & 0x3) << 4;
13        c[i + 3] = b[j + 2] >> 2;
14    }
15 }

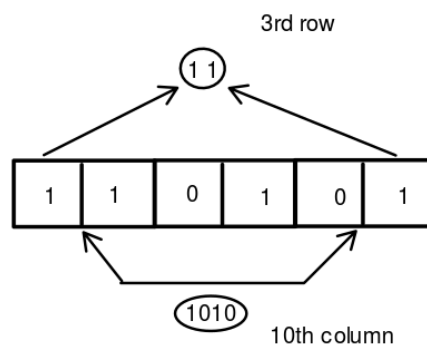
```

Here I have taken the 32 bit input as the array $a[]$ and at first invoke the permutation E using the c-function i_perm , which will give the 6 8-byte unsigned char inside $b[]$. Then to get the output as 8 6-byte unsigned char inside the array $c[]$, I've done the following operations inside the for loop in line 8 of the code snippet.

4.2.2. S-Boxes

Next, the 48 bit result of the expansion is XORed with the round key k_i and the eight 6-bit blocks are fed into eight different S-boxes (Substitution Boxes). Each S-box outputs a 4-bit block. So the output of all the S-boxes becomes a 32-bit result.

Here we demonstrate with an example (figure 4.5), how S-box gives the 4-bit output by taking a 6-bit input. Each S-box is a 4×16 matrix. Now if the input to the S-box is $b = (110101)_2$ then it will give output the element in the $(11)_2$ -th row and $(1010)_2$ -th column of the matrix of the S-box. (Here matrix row and column starting from 0)

Figure 4.5: Example of decoding of the input $(110101)_2$ by S-box

My Implementation

The c function for invoking S-boxes is as follows:

```

1 void sbox(unsigned char *c, unsigned char *e)
2 {
3     unsigned char d[8] = { 0 };
4     int i;
5     for (i = 0; i < 8; i++)
6     {
7         int row = ((c[i] >> 5) << 1) + (c[i] & 1);
8         int column = (c[i] >> 1) & 0xf;
9         d[i] = s[i][row][column];
10    }
11    for (i = 0; i < 4; i++)
12        e[i] = d[i << 1] | (d[(i << 1) + 1] << 4);
13 }

```

In the program the S-boxes are taken as 3-dimensional arrays, where first co-ordinate denote the S-box number, 2nd co-ordinate denote the row number and 3-rd co-ordinate denotes the column number. Here I have calculated the row and column as described in the example of figure 4.5 inside the for loop of the code snippet in line 5 and the output of the i-th S-box is stored inside $d[i]$. After that we merge couple of consecutive $d[i]$'s inside the for loop in line 11 of the code snippet to get the output inside 4 8-byte unsigned char array $e[]$.

The eight S-boxes are given in figure 4.6 to 4.13.

S_1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	01	10	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Figure 4.6: S-Box S_1

S_2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	01	08	14	06	11	03	04	09	07	02	13	12	00	05	10
1	03	13	04	07	15	02	08	14	12	00	01	10	06	09	11	05
2	00	14	07	11	10	04	13	01	05	08	12	06	09	03	02	15
3	13	08	10	01	03	15	04	02	11	06	07	12	00	05	14	09

Figure 4.7: S-Box S_2

S_3	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	00	09	14	06	03	15	05	01	13	12	07	11	04	02	08
1	13	07	00	09	03	04	06	10	02	08	05	14	12	11	15	01
2	13	06	04	09	08	15	03	00	11	01	02	12	05	10	14	07
3	01	10	13	00	06	09	08	07	04	15	14	03	11	05	02	12

Figure 4.8: S-Box S_3

S_4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	07	13	14	03	00	06	09	10	01	02	08	05	11	12	04	15
1	13	08	11	05	06	15	00	03	04	07	02	12	01	10	14	09
2	10	06	09	00	12	11	07	13	15	01	03	14	05	02	08	04
3	03	15	00	06	10	01	13	08	09	04	05	11	12	07	02	14

Figure 4.9: S-Box S_4

S_5	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	02	12	04	01	07	10	11	06	08	05	03	15	13	00	14	09
1	14	11	02	12	04	07	13	01	05	00	15	10	03	09	08	06
2	04	02	01	11	10	13	07	08	15	09	12	05	06	03	00	14
3	11	08	12	07	01	14	02	13	06	15	00	09	10	04	05	03

Figure 4.10: S-Box S_5

S_6	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	01	10	15	09	02	06	08	00	13	03	04	14	07	05	11
1	10	15	04	02	07	12	09	05	06	01	13	14	00	11	03	08
2	09	14	15	05	02	08	12	03	07	00	04	10	01	13	11	06
3	04	03	02	12	09	05	15	10	11	14	01	07	06	00	08	13

Figure 4.11: S-Box S_6

S_7	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	04	11	02	14	15	00	08	13	03	12	09	07	05	10	06	01
1	13	00	11	07	04	09	01	10	14	03	05	12	02	15	08	06
2	01	04	11	13	12	03	07	14	10	15	06	08	00	05	09	02
3	06	11	13	08	01	04	10	07	09	05	00	15	14	02	03	12

Figure 4.12: S-Box S_7

S_8	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	02	08	04	06	15	11	01	10	09	03	14	05	00	12	07
1	01	15	13	08	10	03	07	04	12	05	06	11	00	14	09	02
2	07	11	04	01	09	12	14	02	00	06	10	13	15	03	05	08
3	02	01	14	07	04	10	08	13	15	12	09	00	03	05	06	11

Figure 4.13: S-Box S_8

4.2.3. Permutation P

The 32-bit output of the S-boxes is again manipulated with another permutation P given in figure 4.14.

My Implementation

I have used the `i_perm` function with input permutation P here again to invoke the permutation P .

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

Figure 4.14: P Permutation

4.3. Key Schedule

The *Key Schedule* in DES gives 16 round keys(subkeys) k_i each consists of 48 bits from the original 56 bit key. Although some authors state that the input key of DES is of 64-bit, where every 8th bit of the 8bit blocks is the odd parity bit over the preceding seven bits.

From the 64 bit superkey we will get 56 bit key by ignoring the last bit of every 8 bit block. We will do this by simply invoking a permutation $PC - 1$ (given in 4.16) on superkey. Then the resulting 56-bit key is split into two halves C_0 and D_0 , and the actual key schedule starts as shown in figure 4.17

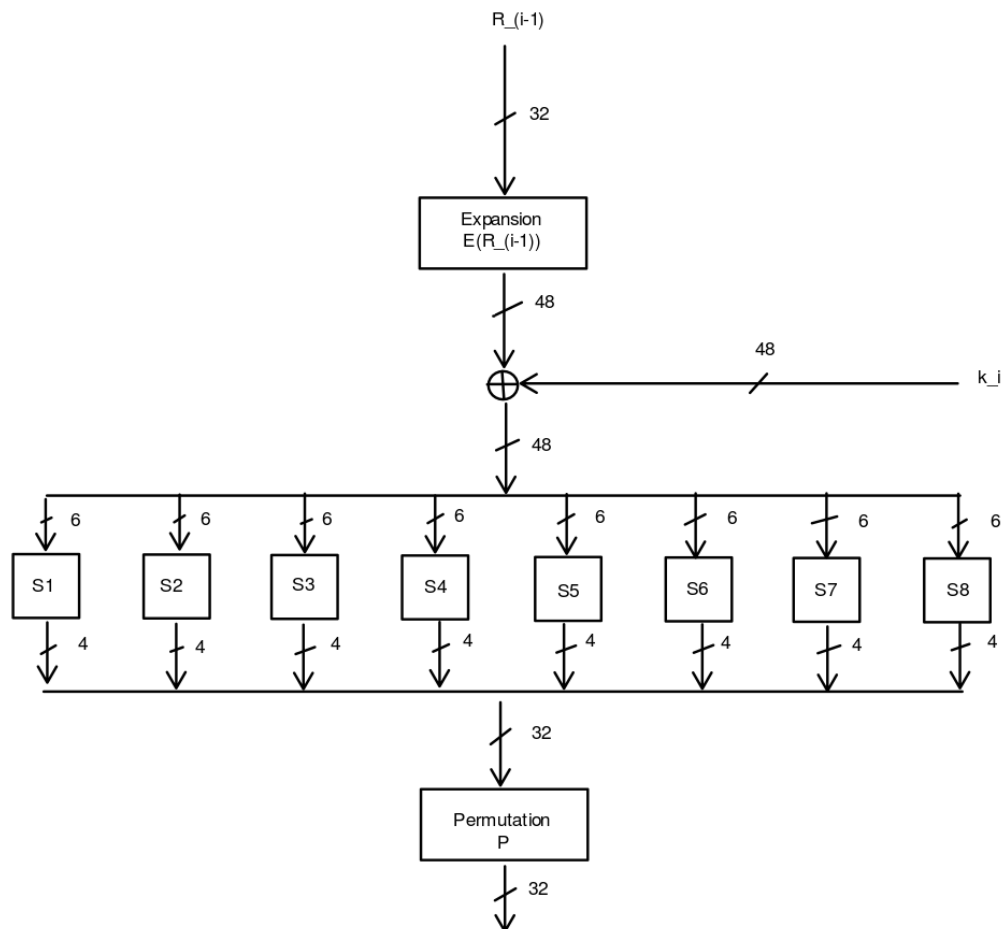


Figure 4.15: f-Function

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

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

Figure 4.16: Bit-connections of $PC - 1$ and $PC - 2$

The two 28-bit halves are cyclically shifted, i.e., rotated, left by one or two bit positions depending on the round i according to the following rules:

- In round $i = 1, 2, 9, 16$, the two halves are rotated left by one bit.
- In all the other rounds, the two halves are rotated left by two bits.

To get the 48-bit round keys k_i , the two halves are permuted bitwise again with $PC - 2$, which is given in the figure 4.16

My Implementation

```
1 void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
2 {
```

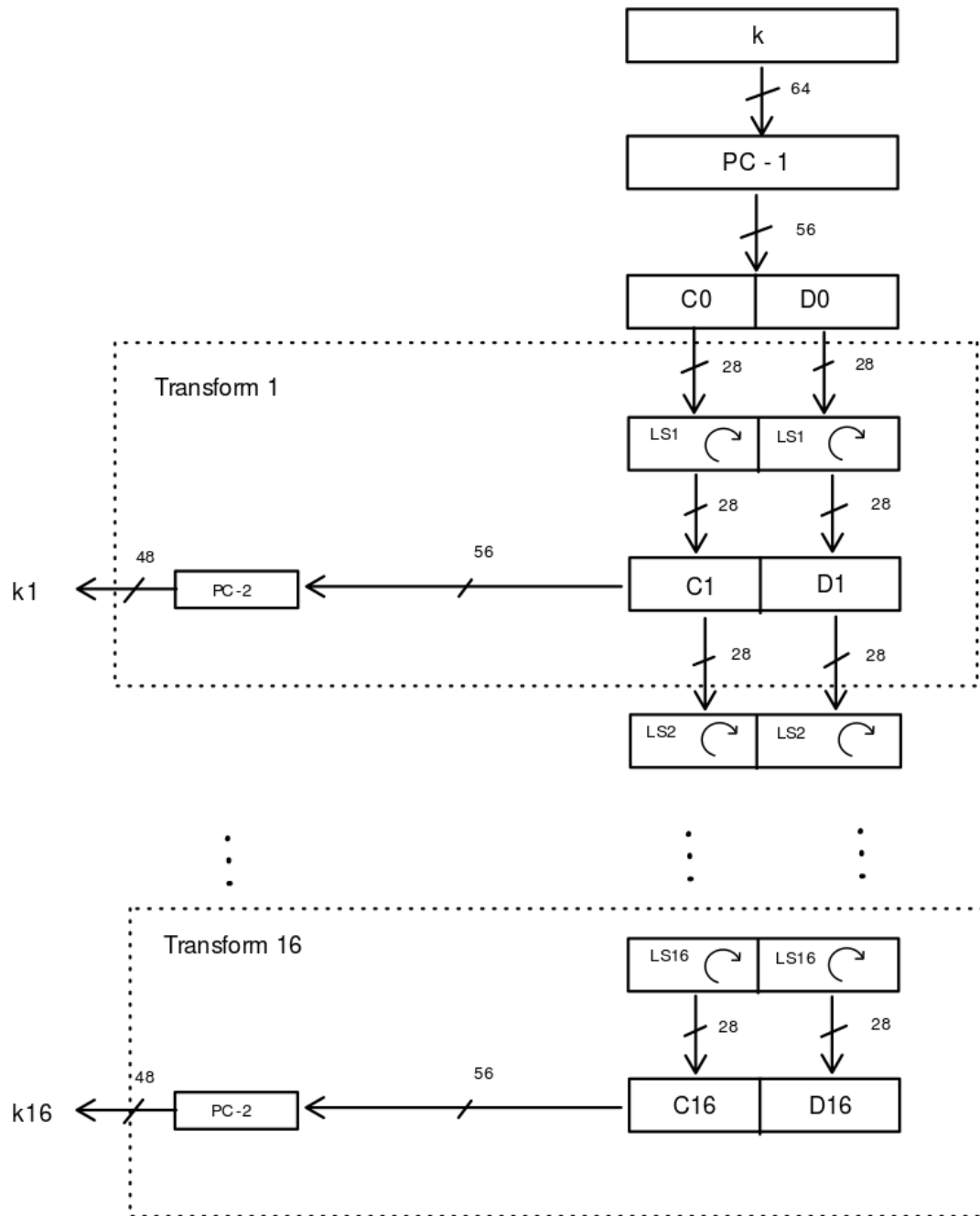


Figure 4.17: Key Schedule

```

3  unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
   new_lkey[7] = {0};
4  int i, j;
5  lkey[0] = key[0] & 0xf;
6  lkey[1] = key[0] >> 4;
7  lkey[2] = key[1] & 0xf;
8  lkey[3] = key[1] >> 4;
9  lkey[4] = key[2] & 0xf;
10 lkey[5] = key[2] >> 4;
11 lkey[6] = key[3] & 0xf;

```

```

12  rkey[0] = key[3] >> 4;
13  rkey[1] = key[4] & 0xf;
14  rkey[2] = key[4] >> 4;
15  rkey[3] = key[5] & 0xf;
16  rkey[4] = key[5] >> 4;
17  rkey[5] = key[6] & 0xf;
18  rkey[6] = key[6] >> 4;
19  if (k == 0 || k == 1 || k == 8 || k == 15)
20  {
21      for (i = 0; i < 7; i++)
22      {
23          new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
24          new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
25      }
26  }
27  else
28  {
29      for (i = 0; i < 7; i++)
30      {
31          new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
32          new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
33      }
34  }
35  key[0] = new_lkey[0] | (new_lkey[1] << 4);
36  key[1] = new_lkey[2] | (new_lkey[3] << 4);
37  key[2] = new_lkey[4] | (new_lkey[5] << 4);
38  key[3] = new_lkey[6] | (new_rkey[0] << 4);
39  key[4] = new_rkey[1] | (new_rkey[2] << 4);
40  key[5] = new_rkey[3] | (new_rkey[4] << 4);
41  key[6] = new_rkey[5] | (new_rkey[6] << 4);
42  i_perm(key, subkey_8bit, PC2, 48);
43  for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3)
44  {
45      subkey[i] = subkey_8bit[j] & 0x3f;
46      subkey[i + 1] = subkey_8bit[j] >> 6;
47      subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
48      subkey[i + 2] = subkey_8bit[j + 1] >> 4;
49      subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
50      subkey[i + 3] = subkey_8bit[j + 2] >> 2;
51  }
52 }

```

In the c-function *key_scheduling* I have taking input the 56-bit key, the all zero initialized subkey(in which I will store the subkey) and the round number *k*. I am dividing the 56-bit key into two 28-bit halves. the left half is an array *lkey*[7] each of *lkey*[*i*] contains 4-bits. Similar is the case for right half. Then in line 19 and 27 of the code snippet, I have done the left rotation according to the round number. After that I am merging every two consecutive 4-bit left half array and right half array to get the new array of 8-bits, *key*[7]. Then the permutation *PC* – 2 is applied on *key*[] to get the 48-bit subkey inside the array *subkey_8bit*[6]. Each *subkey_8bit*[*i*] contains 8 bits. But we have to xor the key with the expanded right half of the previous round output, in which each array contains 6-bit. Thats why for easyness in XORing we make the 6 8-bit subkey array to 8 6-bit subkey inside the for loop in line 43 of the code snippet.

4.4. Decryption

One of the main advantage of DES is that its decryption algorithm is exactly identical with its encryption algorithm. The only difference is that we the reverse key scheduling while decrypting.

4.4.1. Reversed Key Schelude

It is almost exactly same as key schedule, only difference is that here the two 28-bit halves are cyclically shifted, i.e., rotated right by one or two bit positions depending on the round *i* according to the following rules:

- In round 1, the key is not rotated.
- In round *i* = 2, 9, 16, the two halves are rotated right by one bit.
- In all the other rounds, the two halves are rotated right by two bits.

My Implementation

The c-function for reversed key schedule is as follows:

```

1 void Reversed_key_scheduling(unsigned char *key, unsigned char *subkey, int k)
2 {
3     unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
4     new_lkey[7] = {0};
5     int i, j;
6     lkey[0] = key[0] & 0xf;
7     lkey[1] = key[0] >> 4;
8     lkey[2] = key[1] & 0xf;
9     lkey[3] = key[1] >> 4;
10    lkey[4] = key[2] & 0xf;
11    lkey[5] = key[2] >> 4;
12    lkey[6] = key[3] & 0xf;
13    rkey[0] = key[3] >> 4;
14    rkey[1] = key[4] & 0xf;
15    rkey[2] = key[4] >> 4;
16    rkey[3] = key[5] & 0xf;
17    rkey[4] = key[5] >> 4;
18    rkey[5] = key[6] & 0xf;
19    rkey[6] = key[6] >> 4;
20    if (k == 0)
21    {
22        for (i = 0; i < 7; i++)
23        {
24            new_lkey[i] = lkey[i];
25            new_rkey[i] = rkey[i];
26        }
27    }
28    else if (k == 1 || k == 8 || k == 15 || k == 16)
29    {
30        for (i = 0; i < 7; i++)
31        {
32            new_lkey[i] = ((lkey[i] << 1) | ((lkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
33            new_rkey[i] = ((rkey[i] << 1) | ((rkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
34        }
35    }
36    else
37    {
38        for (i = 0; i < 7; i++)
39        {
40            new_lkey[i] = ((lkey[i] << 2) | ((lkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
41            new_rkey[i] = ((rkey[i] << 2) | ((rkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
42        }
43    }
44    key[0] = new_lkey[0] | (new_lkey[1] << 4);
45    key[1] = new_lkey[2] | (new_lkey[3] << 4);
46    key[2] = new_lkey[4] | (new_lkey[5] << 4);
47    key[3] = new_lkey[6] | (new_rkey[0] << 4);
48    key[4] = new_rkey[1] | (new_rkey[2] << 4);
49    key[5] = new_rkey[3] | (new_rkey[4] << 4);
50    key[6] = new_rkey[5] | (new_rkey[6] << 4);
51    i_perm(key, subkey_8bit, PC2, 48);
52    for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3)
53    {
54        subkey[i] = subkey_8bit[j] & 0x3f;
55        subkey[i + 1] = subkey_8bit[j] >> 6;
56        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
57        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
58        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
59        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
60    }
61 }

```

The explanation of the code logic is similar to the key scheduling code.

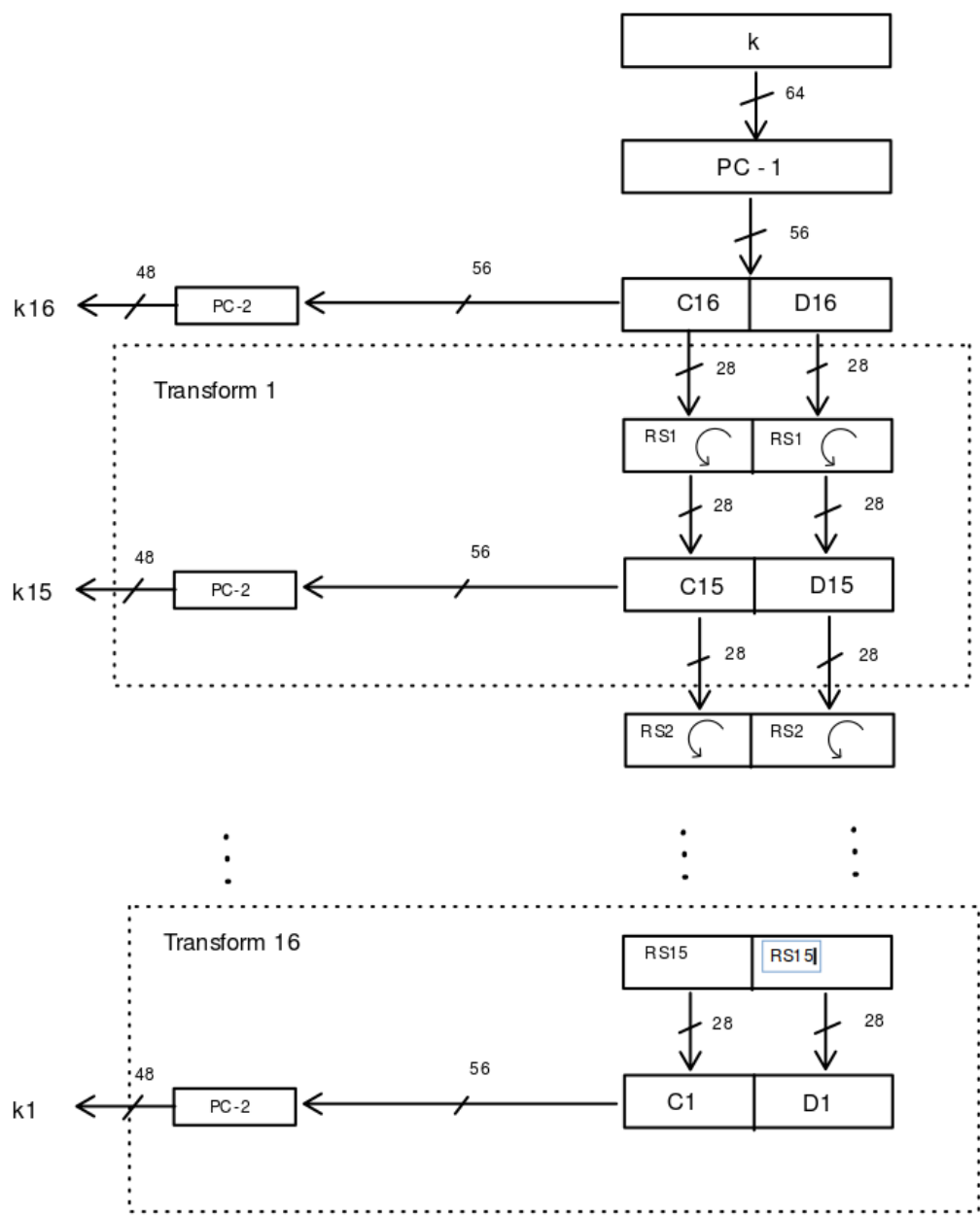


Figure 4.18: Reversed Key Schedule

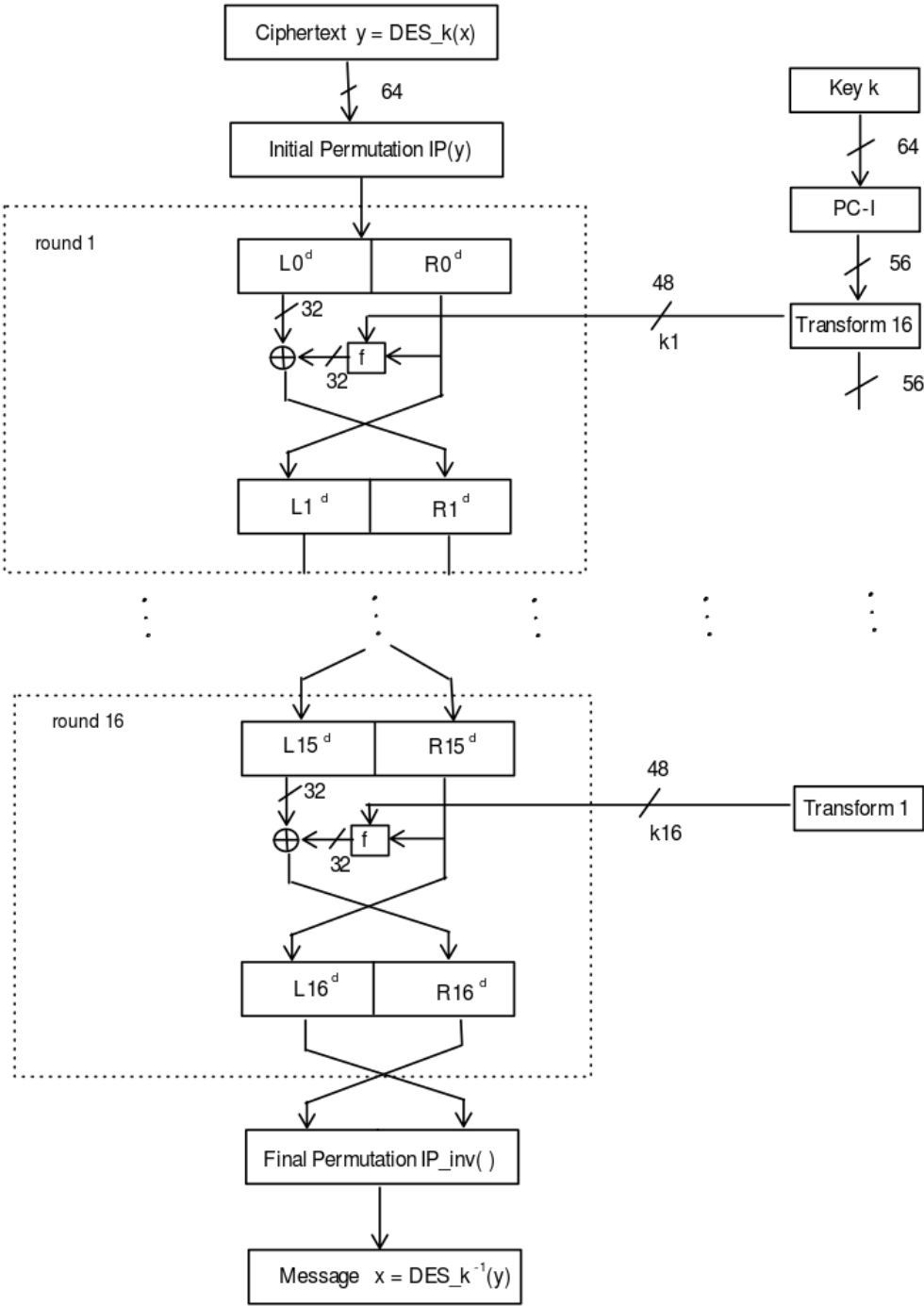


Figure 4.19: DES Decryption

5

C-Code for DES

DES Encryption:

```
1 #include<stdio.h>
2
3 void DES_enc(unsigned char *, unsigned char *, unsigned char *);
4 void swap(unsigned char *, unsigned char *);
5 void i_perm(unsigned char *, unsigned char *, int *, int);
6 void feistel_structure(unsigned char *, int , unsigned char *);
7 void expand(unsigned char *, unsigned char *);
8 void sbox(unsigned char *, unsigned char *);
9 void key_scheduling(unsigned char *, unsigned char *, int );
10
11 int IP[] = {
12     58, 50, 42, 34, 26, 18, 10, 2,
13     60, 52, 44, 36, 28, 20, 12, 4,
14     62, 54, 46, 38, 30, 22, 14, 6,
15     64, 56, 48, 40, 32, 24, 16, 8,
16     57, 49, 41, 33, 25, 17,  9, 1,
17     59, 51, 43, 35, 27, 19, 11, 3,
18     61, 53, 45, 37, 29, 21, 13, 5,
19     63, 55, 47, 39, 31, 23, 15, 7
20 },
21 IP_inv[] = {
22     40, 8, 48, 16, 56, 24, 64, 32,
23     39, 7, 47, 15, 55, 23, 63, 31,
24     38, 6, 46, 14, 54, 22, 62, 30,
25     37, 5, 45, 13, 53, 21, 61, 29,
26     36, 4, 44, 12, 52, 20, 60, 28,
27     35, 3, 43, 11, 51, 19, 59, 27,
28     34, 2, 42, 10, 50, 18, 58, 26,
29     33, 1, 41,  9, 49, 17, 57, 25
30 },
31 E[] = {
32     32,  1,  2,  3,  4,  5,
33     4,  5,  6,  7,  8,  9,
34     8,  9, 10, 11, 12, 13,
35     12, 13, 14, 15, 16, 17,
36     16, 17, 18, 19, 20, 21,
37     20, 21, 22, 23, 24, 25,
38     24, 25, 26, 27, 28, 29,
39     28, 29, 30, 31, 32,  1
40 },
41 s[8][4][16] = {
42     {
43         14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12,
44         5, 9, 0, 7,
45         0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
46         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
47         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
48     },
49     {
50         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
51         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
52         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
53     },
54     {
55         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
56         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
57         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
58     },
59     {
60         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
61         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
62         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
63     },
64     {
65         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
66         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
67         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
68     },
69     {
70         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
71         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
72         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
73     },
74     {
75         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
76         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
77         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
78     },
79     {
80         1, 14, 4, 13, 10, 6, 12, 11, 9, 5, 3, 8,
81         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
82         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
83     }
84 }
```

```

48         15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12,
49             0, 5, 10,
3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
50 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
51 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9
52     },
53     {
54         10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7,
11, 4, 2, 8,
55 13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
56 13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
57 1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12
58     },
59     {
60         7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
12, 4, 15,
61 13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
62 10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
63 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
64     },
65     {
66         2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
0, 14, 9,
67 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
68 4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
69 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
70     },
71     {
72         12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
7, 5, 11,
73 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
74 9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
75 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
76     },
77     {
78         4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
10, 6, 1,
79 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
80 1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
81 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
82     },
83     {
84         13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
0, 12, 7,
85 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
86 7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
87 2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
88     }
89 },
90 f_perm[32] = { 16, 7, 20, 21,
29, 12, 28, 17,
91 1, 15, 23, 26,
92 5, 18, 31, 10,
93 2, 8, 24, 14,
94 32, 27, 3, 9,
95 19, 13, 30, 6,
96 22, 11, 4, 25 },
97 PC1[56] = { 57, 49, 41, 33, 25, 17, 9,
1, 58, 50, 42, 34, 26, 18,
99 10, 2, 59, 51, 43, 35, 27,
100 19, 11, 3, 60, 52, 44, 36,
101 63, 55, 47, 39, 31, 23, 15,
102 7, 62, 54, 46, 38, 30, 22,
103 14, 6, 61, 53, 45, 37, 29,
104 21, 13, 5, 28, 20, 12, 4 },
105 PC2[48] = { 14, 17, 11, 24, 1, 5,
3, 28, 15, 6, 21, 10,
107 23, 19, 12, 4, 26, 8,
108 16, 7, 27, 20, 13, 2,
109 41, 52, 31, 37, 47, 55,
110 30, 40, 51, 45, 33, 48,
111

```

```

112         44, 49, 39, 56, 34, 53,
113         46, 42, 50, 36, 29, 32 };
114
115 void main()
116 {
117     unsigned char plain_text[8] = "SohanDas", permuted_text[8] = {0}, cipher_text[8] = {0},
118     super_key[8] = "Samapuki", key[7] = {0};
119     int i, j, k;
120     printf("plain text: ");
121     for (i = 0; i < 8; i++)
122         printf("%c", plain_text[i]);
123     DES_enc(plain_text, cipher_text, super_key);
124     printf("\ncipher text: ");
125     for (i = 0; i < 8; i++)
126         printf("%hhx, ", cipher_text[i]);
127     printf("\n");
128 }
129 void DES_enc(unsigned char *plain_text, unsigned char *cipher_text, unsigned char *super_key)
130 //taking plain_text array and storing the encrypted text in th cipher_text array
131 {
132     unsigned char permuted_text[8] = {0}, key[8] = {0};
133     int i, j, k;
134     i_perm(plain_text, permuted_text, IP, 64); //invoking the initial permutation
135     i_perm(super_key, key, PC1, 56); //invoking PC-1 permutation of key scheduling
136     for (k = 0; k < 16; k++)
137         feistel_structure(permuted_text, k, key); //applying Feistel structure for 16 times
138     for (i = 0; i < 4; i++)
139         swap(&permuted_text[i], &permuted_text[i + 4]); //after all the Feistel round is done,
140         //final swap
141     i_perm(permuted_text, cipher_text, IP_inv, 64); //invoking final permutation, the inverse
142     //of the initial permutation
143 }
144 void swap(unsigned char *p, unsigned char *q) //basic swap function
145 {
146     unsigned char x = *p;
147     *p = *q;
148     *q = x;
149 }
150 void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size) //invoke any
151 //permutation
152 {
153     for (int i = 0; i < perm_size; i++)
154     {
155         int byte_number_a = (perm[i] - 1) >> 3; //byte number of the array a[]
156         int bit_number_a = ((perm[i] - 1) & 7); //bit position of the element a[byte_number_a]
157         int byte_number_b = i >> 3; //byte number of the array b[]
158         int bit_number_b = i & 7; //bit position of the element b[byte_number_b]
159         b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
160     }
161 }
162 void feistel_structure(unsigned char *permuted_text, int k, unsigned char *key)
163 {
164     unsigned char left_permuted_text[4] = {permuted_text[0], permuted_text[1], permuted_text[2], permuted_text[3]},
165     right_permuted_text[4] = {permuted_text[4], permuted_text[5], permuted_text[6], permuted_text[7]},
166     expanded_right_permuted_text[8] = {0},
167     sbbox_output[4] = {0}, fies_perm_output[4] = {0}, subkey[8] = {0};
168     int i;
169     expand(right_permuted_text, expanded_right_permuted_text); //expanding the right half of
170     //the permuted text
171     key_scheduling(key, subkey, k); //will put the roundkey in the array named subkey
172     for (i = 0; i < 8; i++)
173         expanded_right_permuted_text[i] ^= subkey[i]; //xoring the right half of the permuted
174         //text with round key
175     sbbox(expanded_right_permuted_text, sbbox_output); //invoking sbbox
176     i_perm(sbbox_output, fies_perm_output, f_perm, 32); //invoking the last permutation inside
177     //f function
178     for (i = 0; i < 4; i++)

```

```

172 {
173     permuted_text[i] ^= fies_perm_output[i]; //xoring the Feistel output with left half of
        the previous round output
174     swap(&permuted_text[i], &permuted_text[i + 4]); //swapping the left half and right
        half
175 }
176 }
177
178 void expand(unsigned char *a, unsigned char *c)
179 {
180     unsigned char b[6] = {0};
181     int i, j;
182     i_perm(a, b, E, 48); //invoking expansion permutation
183     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
        from 6 8-bit block
184     {
185         c[i] = b[j] & 0x3f;
186         c[i + 1] = b[j] >> 6;
187         c[i + 1] |= (b[j + 1] & 0xf) << 2;
188         c[i + 2] = b[j + 1] >> 4;
189         c[i + 2] |= (b[j + 2] & 0x3) << 4;
190         c[i + 3] = b[j + 2] >> 2;
191     }
192 }
193
194 void sbbox(unsigned char *c, unsigned char *e)
195 {
196     unsigned char d[8] = { 0 };
197     int i;
198     for (i = 0; i < 8; i++)
199     {
200         int row = ((c[i] >> 5) << 1) + (c[i] & 1); //calculating row and column of the sbbox
201         int column = (c[i] >> 1) & 0xf;
202         d[i] = s[i][row][column];
203     }
204     for (i = 0; i < 4; i++)
205         e[i] = d[i << 1] | (d[(i << 1) + 1] << 4); //merging two consecutive 4-bit block to
        get a 8 bit block
206 }
207
208 void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
209 {
210     unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
211     int i, j;
212     lkey[0] = key[0] & 0xf;
213     lkey[1] = key[0] >> 4;
214     lkey[2] = key[1] & 0xf;
215     lkey[3] = key[1] >> 4;
216     lkey[4] = key[2] & 0xf;
217     lkey[5] = key[2] >> 4;
218     lkey[6] = key[3] & 0xf;
219     rkey[0] = key[3] >> 4;
220     rkey[1] = key[4] & 0xf;
221     rkey[2] = key[4] >> 4;
222     rkey[3] = key[5] & 0xf;
223     rkey[4] = key[5] >> 4;
224     rkey[5] = key[6] & 0xf;
225     rkey[6] = key[6] >> 4; //dividing the key in left half and right half
226     if (k == 0 || k == 1 | k == 8 | k == 15) //rotate left by one bit while round number is 0
        or 1 or 8 or 15
227     {
228         for (i = 0; i < 7; i++)
229         {
230             new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
231             new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
232         }
233     }
234     else //rotate left by two bit while round number is different
235     {
236         for (i = 0; i < 7; i++)

```

```

237     {
238         new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
239         new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
240     }
241 }
242 key[0] = new_lkey[0] | (new_lkey[1] << 4); //merging two consecutive 4-bit block to get 8-
    bit block
243 key[1] = new_lkey[2] | (new_lkey[3] << 4);
244 key[2] = new_lkey[4] | (new_lkey[5] << 4);
245 key[3] = new_lkey[6] | (new_rkey[0] << 4);
246 key[4] = new_rkey[1] | (new_rkey[2] << 4);
247 key[5] = new_rkey[3] | (new_rkey[4] << 4);
248 key[6] = new_rkey[5] | (new_rkey[6] << 4);
249 i_perm(key, subkey_8bit, PC2, 48);
250 for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
    from 6 8-bit block
251 {
252     subkey[i] = subkey_8bit[j] & 0x3f;
253     subkey[i + 1] = subkey_8bit[j] >> 6;
254     subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
255     subkey[i + 2] = subkey_8bit[j + 1] >> 4;
256     subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
257     subkey[i + 3] = subkey_8bit[j + 2] >> 2;
258 }
259 }

```

DES Decryption

```

1 #include<stdio.h>
2
3 void DES_dec(unsigned char *, unsigned char *, unsigned char *);
4 void swap(unsigned char *, unsigned char *);
5 void i_perm(unsigned char *, unsigned char *, int *, int);
6 void feistel_structure(unsigned char *, int , unsigned char *);
7 void expand(unsigned char *, unsigned char *);
8 void sbox(unsigned char *, unsigned char *);
9 void key_scheduling(unsigned char *, unsigned char *, int );
10
11 int IP[] = {
12     58, 50, 42, 34, 26, 18, 10, 2,
13     60, 52, 44, 36, 28, 20, 12, 4,
14     62, 54, 46, 38, 30, 22, 14, 6,
15     64, 56, 48, 40, 32, 24, 16, 8,
16     57, 49, 41, 33, 25, 17, 9, 1,
17     59, 51, 43, 35, 27, 19, 11, 3,
18     61, 53, 45, 37, 29, 21, 13, 5,
19     63, 55, 47, 39, 31, 23, 15, 7
20 },
21 IP_inv[] = {
22     40, 8, 48, 16, 56, 24, 64, 32,
23     39, 7, 47, 15, 55, 23, 63, 31,
24     38, 6, 46, 14, 54, 22, 62, 30,
25     37, 5, 45, 13, 53, 21, 61, 29,
26     36, 4, 44, 12, 52, 20, 60, 28,
27     35, 3, 43, 11, 51, 19, 59, 27,
28     34, 2, 42, 10, 50, 18, 58, 26,
29     33, 1, 41, 9, 49, 17, 57, 25
30 },
31 E[] = {
32     32, 1, 2, 3, 4, 5,
33     4, 5, 6, 7, 8, 9,
34     8, 9, 10, 11, 12, 13,
35     12, 13, 14, 15, 16, 17,
36     16, 17, 18, 19, 20, 21,
37     20, 21, 22, 23, 24, 25,
38     24, 25, 26, 27, 28, 29,
39     28, 29, 30, 31, 32, 1
40 },
41 s[8][4][16] = {
42     {
43         14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12,
44         5, 9, 0, 7,
45         0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,

```



```

44         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
45         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
46     },
47     {
48         15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12,
49         0, 5, 10,
50         3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
51         0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
52         13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9
53     },
54     {
55         10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7,
56         11, 4, 2, 8,
57         13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
58         13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
59         1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12
60     },
61     {
62         7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
63         12, 4, 15,
64         13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
65         10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
66         3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
67     },
68     {
69         2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
70         0, 14, 9,
71         14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
72         4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
73         11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
74     },
75     {
76         12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
77         7, 5, 11,
78         10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
79         9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
80         4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
81     },
82     {
83         4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
84         10, 6, 1,
85         13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
86         1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
87         6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
88     },
89     {
90         13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
91         0, 12, 7,
92         1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
93         7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
94         2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
95     }
96 },
97 f_perm[32] = { 16, 7, 20, 21,
98               29, 12, 28, 17,
99               1, 15, 23, 26,
100              5, 18, 31, 10,
101              2, 8, 24, 14,
102              32, 27, 3, 9,
103              19, 13, 30, 6,
104              22, 11, 4, 25 },
105 PC1[56] = { 57, 49, 41, 33, 25, 17, 9,
106             1, 58, 50, 42, 34, 26, 18,
107             10, 2, 59, 51, 43, 35, 27,
108             19, 11, 3, 60, 52, 44, 36,
109             63, 55, 47, 39, 31, 23, 15,
110             7, 62, 54, 46, 38, 30, 22,
111             14, 6, 61, 53, 45, 37, 29,
112             21, 13, 5, 28, 20, 12, 4 },
113 PC2[48] = { 14, 17, 11, 24, 1, 5,
114             3, 28, 15, 6, 21, 10,

```

```

108         23, 19, 12, 4, 26, 8,
109         16, 7, 27, 20, 13, 2,
110         41, 52, 31, 37, 47, 55,
111         30, 40, 51, 45, 33, 48,
112         44, 49, 39, 56, 34, 53,
113         46, 42, 50, 36, 29, 32 };
114
115 void main()
116 {
117     unsigned char cipher_text[8] = {87, 193, 235, 190, 25, 243, 26, 171}, permuted_text[8] =
118         {0}, plain_text[8] = {0}, super_key[8] = "Samapuki", key[7] = {0};
119     int i, j, k;
120     printf("plain text: ");
121     for (i = 0; i < 8; i++)
122         printf("%hhhu, ", cipher_text[i]);
123     DES_dec(cipher_text, plain_text, super_key);
124     printf("\ncipher text: ");
125     for (i = 0; i < 8; i++)
126         printf("%c", plain_text[i]);
127     printf("\n");
128 }
129 void DES_dec(unsigned char *cipher_text, unsigned char *plain_text, unsigned char *super_key)
130     //taking cipher_text array and storing the decrypted text in th plain_text array
131 {
132     unsigned char permuted_text[8] = {0}, key[8] = {0};
133     int i, j, k;
134     i_perm(cipher_text, permuted_text, IP, 64); //invoking the initial permutation
135     i_perm(super_key, key, PC1, 56); //invoking PC-1 permutation of key scheduling
136     for (k = 0; k < 16; k++)
137         feistel_structure(permuted_text, k, key); //applying Feistel structure for 16 times
138     for (i = 0; i < 4; i++)
139         swap(&permuted_text[i], &permuted_text[i + 4]); //after all the Feistel round is done,
140         //final swap
141     i_perm(permuted_text, plain_text, IP_inv, 64); //invoking final permutation, the inverse
142     //of the initial permutation
143 }
144 void swap(unsigned char *p, unsigned char *q) //basic swap function
145 {
146     unsigned char x = *p;
147     *p = *q;
148     *q = x;
149 }
150 void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size) //invoke any
151     permutation
152 {
153     for (int i = 0; i < perm_size; i++)
154     {
155         int byte_number_a = (perm[i] - 1) >> 3; //byte number of the array a[]
156         int bit_number_a = ((perm[i] - 1) & 7); //bit position of the element a[byte_number_a]
157         int byte_number_b = i >> 3; //byte number of the array b[]
158         int bit_number_b = i & 7; //bit position of the element b[byte_number_b]
159         b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
160     }
161 }
162 void feistel_structure(unsigned char *permuted_text, int k, unsigned char *key)
163 {
164     unsigned char left_permuted_text[4] = {permuted_text[0], permuted_text[1], permuted_text
165         [2], permuted_text[3]}, right_permuted_text[4] = {permuted_text[4], permuted_text[5],
166         permuted_text[6], permuted_text[7]}, expanded_right_permuted_text[8] = {0},
167         sbox_output[4] = {0}, fies_perm_output[4] = {0}, subkey[8] = {0};
168     int i;
169     expand(right_permuted_text, expanded_right_permuted_text); //expanding the right half of
170     //the permuted text
171     key_scheduling(key, subkey, k); //will put the roundkey in the array named subkey
172     for (i = 0; i < 8; i++)
173         expanded_right_permuted_text[i] ^= subkey[i]; //xoring the right half of the permuted
174         //text with round key

```

```

169     sbox(expanded_right_permuted_text, sbox_output); //invoking sbox
170     i_perm(sbox_output, fies_perm_output, f_perm, 32); //invoking the last permutation inside
        f function
171     for (i = 0; i < 4; i++)
172     {
173         permuted_text[i] ^= fies_perm_output[i]; //xoring the Feistel output with left half of
            the previous round output
174         swap(&permuted_text[i], &permuted_text[i + 4]); //swapping the left half and right
            half
175     }
176 }
177
178 void expand(unsigned char *a, unsigned char *c)
179 {
180     unsigned char b[6] = {0};
181     int i, j;
182     i_perm(a, b, E, 48); //invoking expansion permutation
183     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
        from 6 8-bit block
184     {
185         c[i] = b[j] & 0x3f;
186         c[i + 1] = b[j] >> 6;
187         c[i + 1] |= (b[j + 1] & 0xf) << 2;
188         c[i + 2] = b[j + 1] >> 4;
189         c[i + 2] |= (b[j + 2] & 0x3) << 4;
190         c[i + 3] = b[j + 2] >> 2;
191     }
192 }
193
194 void sbox(unsigned char *c, unsigned char *e)
195 {
196     unsigned char d[8] = { 0 };
197     int i;
198     for (i = 0; i < 8; i++)
199     {
200         int row = ((c[i] >> 5) << 1) + (c[i] & 1); //calculating row and column of the sbox
201         int column = (c[i] >> 1) & 0xf;
202         d[i] = s[i][row][column];
203     }
204     for (i = 0; i < 4; i++)
205         e[i] = d[i << 1] | (d[(i << 1) + 1] << 4); //merging two consecutive 4-bit block to
            get a 8 bit block
206 }
207
208 void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
209 {
210     unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
211     int i, j;
212     lkey[0] = key[0] & 0xf;
213     lkey[1] = key[0] >> 4;
214     lkey[2] = key[1] & 0xf;
215     lkey[3] = key[1] >> 4;
216     lkey[4] = key[2] & 0xf;
217     lkey[5] = key[2] >> 4;
218     lkey[6] = key[3] & 0xf;
219     rkey[0] = key[3] >> 4;
220     rkey[1] = key[4] & 0xf;
221     rkey[2] = key[4] >> 4;
222     rkey[3] = key[5] & 0xf;
223     rkey[4] = key[5] >> 4;
224     rkey[5] = key[6] & 0xf;
225     rkey[6] = key[6] >> 4; //dividing the key in left half and right half
226     if (k == 0) //no rotation in the first round
227     {
228         for (i = 0; i < 7; i++)
229         {
230             new_lkey[i] = lkey[i];
231             new_rkey[i] = rkey[i];
232         }
233     }

```

```

234     else if (k == 1 || k == 8 || k == 15) //rotate left by one bit while round number 1 or 8
        or 15
235     {
236         for (i = 0; i < 7; i++)
237         {
238             new_lkey[i] = ((lkey[i] << 1) | ((lkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
239             new_rkey[i] = ((rkey[i] << 1) | ((rkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
240         }
241     }
242     else //rotate left by two bit while round number is different
243     {
244         for (i = 0; i < 7; i++)
245         {
246             new_lkey[i] = ((lkey[i] << 2) | ((lkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
247             new_rkey[i] = ((rkey[i] << 2) | ((rkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
248         }
249     }
250     key[0] = new_lkey[0] | (new_lkey[1] << 4); //merging two consecutive 4-bit block to get 8-
        bit block
251     key[1] = new_lkey[2] | (new_lkey[3] << 4);
252     key[2] = new_lkey[4] | (new_lkey[5] << 4);
253     key[3] = new_lkey[6] | (new_rkey[0] << 4);
254     key[4] = new_rkey[1] | (new_rkey[2] << 4);
255     key[5] = new_rkey[3] | (new_rkey[4] << 4);
256     key[6] = new_rkey[5] | (new_rkey[6] << 4);
257     i_perm(key, subkey_8bit, PC2, 48);
258     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
        from 6 8-bit block
259     {
260         subkey[i] = subkey_8bit[j] & 0x3f;
261         subkey[i + 1] = subkey_8bit[j] >> 6;
262         subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
263         subkey[i + 2] = subkey_8bit[j + 1] >> 4;
264         subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
265         subkey[i + 3] = subkey_8bit[j + 2] >> 2;
266     }
267 }

```

6

Modes of Encryption

In the previous sections we have discussed about DES and implement it in c programming. Now we will discuss about one modes of encryption and implement that using DES as its building block. There are 5 very famous modes of encryption:

- Electronic Code Book(ECB) mode
- Cipher Block Chaing(CBC) mode
- Cipher Feedback(CFB) mode
- Output Feedback(OFB) mode
- Counter(CTR) mode

The last three modes use the block cipher as a building block for a stream cipher. Now let's discuss the OFB mode in detail:

6.1. Output Feedback (OFB) Mode

OFB is actually a stream cipher which is using DES to generate random string to pad that with the plain text. To generate the initial key stream, we use one IV that fed into the DES function. That will generate a random looking 64 bit key stream. We will xor that with 64 bits of plaintext to generate 64 bits of ciphertext. For encrypting the next 64 bits of plaintext, we again have to generate a new 64 bit key stream. For that we will fed the DES with the output of $DES_k(IV)$. That will again make a new random looking 64 bit key stream. We will XOR it with the next 64 bit plaintext. Figure 6.1.

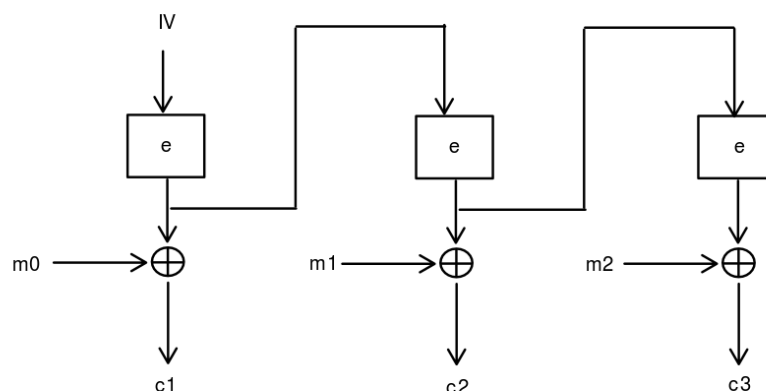


Figure 6.1: Block Diagram of OFB

Definition: Let $e()$ be a block cipher of block size b ; let x_i , y_i and s_i be bit strings of length b ; and IV be a nonce of length b .

Encryption(first block): $s_1 = e_k(IV)$ and $y_1 = s_1 \oplus x_1$

Encryption(general block): $s_i = e_k(s_{i-1})$ and $y_i = s_i \oplus x_i, i \geq 2$

Decryption(first block): $s_1 = e_k(IV)$ and $x_1 = s_1 \oplus y_1$

Decryption(general block): $s_i = e_k(s_{i-1})$ and $x_i = s_i \oplus y_i, i \geq 2$

My Implementation

```

1 void main()
2 {
3     unsigned char plain_text[8] = {0}, cipher_text[8] = {0}, super_key[8] = "Samapuki", key
4     [7] = {0}, IV[8] = "Tamasuki", padding_key[8] = {0};
5     int i, j, k, m = 0;
6     int fd = open("encrypted", O_RDONLY), fd2 = open("decrypted", O_WRONLY | O_CREAT |
7     O_TRUNC);
8     do
9     {
10        for(i = 0; i < 8; i++)
11        {
12            plain_text[i] = 0;
13            cipher_text[i] = 0;
14            key[i] = 0;
15            padding_key[i] = 0;
16        }
17        read(fd, cipher_text, 8 * sizeof(unsigned char));
18        printf("\ncipher text: \n");
19        for (i = 0; i < 8; i++)
20            printf("%c", cipher_text[i]);
21        DES_enc(IV, padding_key, super_key);
22        printf("\nplain text: \n");
23        for (i = 0; i < 8; i++)
24        {
25            plain_text[i] = cipher_text[i] ^ padding_key[i];
26            printf("%c", plain_text[i]);
27            IV[i] = padding_key[i];
28        }
29        write(fd2, plain_text, 8 * sizeof(unsigned char));
30        printf("\n");
31        m++;
32    }
33    // while(m <= 30);
34    while(plain_text[7] != 0);
35    close(fd);
36    close(fd2);
37 }

```

Here in the program the variable *padding_key* is the 64 bit key stream bit which has been XORed with the 64 bit of plaintext.

7

C-code for OFB using DES

OFB Encryption[2]

```
1 #include<stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <time.h>
6 void DES_enc(unsigned char *, unsigned char *, unsigned char *);
7 void swap(unsigned char *, unsigned char *);
8 void i_perm(unsigned char *, unsigned char *, int *, int);
9 void feistel_structure(unsigned char *, int , unsigned char *);
10 void expand(unsigned char *, unsigned char *);
11 void sbox(unsigned char *, unsigned char *);
12 void key_scheduling(unsigned char *, unsigned char *, int );
13
14 int IP[] = {
15     58, 50, 42, 34, 26, 18, 10, 2,
16     60, 52, 44, 36, 28, 20, 12, 4,
17     62, 54, 46, 38, 30, 22, 14, 6,
18     64, 56, 48, 40, 32, 24, 16, 8,
19     57, 49, 41, 33, 25, 17, 9, 1,
20     59, 51, 43, 35, 27, 19, 11, 3,
21     61, 53, 45, 37, 29, 21, 13, 5,
22     63, 55, 47, 39, 31, 23, 15, 7
23 },
24 IP_inv[] = {
25     40, 8, 48, 16, 56, 24, 64, 32,
26     39, 7, 47, 15, 55, 23, 63, 31,
27     38, 6, 46, 14, 54, 22, 62, 30,
28     37, 5, 45, 13, 53, 21, 61, 29,
29     36, 4, 44, 12, 52, 20, 60, 28,
30     35, 3, 43, 11, 51, 19, 59, 27,
31     34, 2, 42, 10, 50, 18, 58, 26,
32     33, 1, 41, 9, 49, 17, 57, 25
33 },
34 E[] = {
35     32, 1, 2, 3, 4, 5,
36     4, 5, 6, 7, 8, 9,
37     8, 9, 10, 11, 12, 13,
38     12, 13, 14, 15, 16, 17,
39     16, 17, 18, 19, 20, 21,
40     20, 21, 22, 23, 24, 25,
41     24, 25, 26, 27, 28, 29,
42     28, 29, 30, 31, 32, 1
43 },
44 s[8][4][16] = {
45     {
46         14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12,
47         5, 9, 0, 7,
48         0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
49         4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
50         15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
51     }
52 }
```

```

49         },
50     {
51         15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12,
           0, 5, 10,
52         3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
53         0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
54         13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9
55     },
56     {
57         10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7,
           11, 4, 2, 8,
58         13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
59         13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
60         1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12
61     },
62     {
63         7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
           12, 4, 15,
64         13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
65         10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
66         3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
67     },
68     {
69         2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
           0, 14, 9,
70         14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
71         4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
72         11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
73     },
74     {
75         12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
           7, 5, 11,
76         10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
77         9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
78         4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
79     },
80     {
81         4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
           10, 6, 1,
82         13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
83         1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
84         6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
85     },
86     {
87         13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
           0, 12, 7,
88         1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
89         7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
90         2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
91     }
92 },
93 f_perm[32] = { 16, 7, 20, 21,
94               29, 12, 28, 17,
95               1, 15, 23, 26,
96               5, 18, 31, 10,
97               2, 8, 24, 14,
98               32, 27, 3, 9,
99               19, 13, 30, 6,
100              22, 11, 4, 25 },
101 PC1[56] = { 57, 49, 41, 33, 25, 17, 9,
102             1, 58, 50, 42, 34, 26, 18,
103             10, 2, 59, 51, 43, 35, 27,
104             19, 11, 3, 60, 52, 44, 36,
105             63, 55, 47, 39, 31, 23, 15,
106             7, 62, 54, 46, 38, 30, 22,
107             14, 6, 61, 53, 45, 37, 29,
108             21, 13, 5, 28, 20, 12, 4 },
109 PC2[48] = { 14, 17, 11, 24, 1, 5,
110             3, 28, 15, 6, 21, 10,
111             23, 19, 12, 4, 26, 8,
112             16, 7, 27, 20, 13, 2,

```



```

113         41, 52, 31, 37, 47, 55,
114         30, 40, 51, 45, 33, 48,
115         44, 49, 39, 56, 34, 53,
116         46, 42, 50, 36, 29, 32 };
117
118 void main()
119 {
120     unsigned char plain_text[8] = {0}, permuted_text[8] = {0}, cipher_text[8] = {0},
121         super_key[8] = "lucifer1", key[7] = {0}, IV[8] = {0}, padding_key[8] = {0};
122     int i, j, k;
123     srand(time(NULL));
124     for(i = 0; i < 8; i++)
125         IV[i] = rand() & 255; //taking IV randomly
126     int fd = open("tk.txt", O_RDONLY, 0666), fd2 = open("encrypted", O_WRONLY | O_CREAT |
127         O_TRUNC, 0666);
128     write(fd2, IV, 8 * sizeof(unsigned char)); //printing the IV at the beginning of the
129     cipher_text file
130     do
131     {
132         for(i = 0; i < 8; i++) //initializing all the variable by zero so that no junk value
133             become there in any intermediate state
134         {
135             plain_text[i] = 0;
136             cipher_text[i] = 0;
137             padding_key[i] = 0;
138         }
139         read(fd, plain_text, 8 * sizeof(unsigned char)); //reading 8-byte from the tk.txt file
140         and putting the i-th byte in the array plain_text[]
141         DES_enc(IV, padding_key, super_key); //encrypting the output of the DES output of
142         previous round
143         for (i = 0; i < 8; i++)
144         {
145             cipher_text[i] = plain_text[i] ^ padding_key[i]; //padding the plain text with the
146             output of DES
147             IV[i] = padding_key[i]; //setting the feedback of the next round
148         }
149         write(fd2, cipher_text, 8 * sizeof(unsigned char)); //writting the cipher text in the
150         file named encrypted
151     }
152     while(plain_text[7] != 0); //continuing the loop until the last byte of the plain text
153     become NULL character
154     printf("\nciphertext is contained in the file named \"encrypted\".\n\n");
155     close(fd);
156     close(fd2);
157 }
158
159 void DES_enc(unsigned char *plain_text, unsigned char *cipher_text, unsigned char *super_key)
160 //taking plain_text array and storing the encrypted text in th cipher_text array
161 {
162     unsigned char permuted_text[8] = {0}, key[8] = {0};
163     int i, j, k;
164     i_perm(plain_text, permuted_text, IP, 64); //invoking the initial permutation
165     i_perm(super_key, key, PC1, 56); //invoking PC-1 permutation of key scheduling
166     for (k = 0; k < 16; k++)
167         feistel_structure(permuted_text, k, key); //applying Feistel structure for 16 times
168     for (i = 0; i < 4; i++)
169         swap(&permuted_text[i], &permuted_text[i + 4]); //after all the Feistel round is done,
170         final swap
171     i_perm(permuted_text, cipher_text, IP_inv, 64); //invoking final permutation, the inverse
172     of the initial permutation
173 }
174
175 void swap(unsigned char *p, unsigned char *q) //basic swap function
176 {
177     unsigned char x = *p;
178     *p = *q;
179     *q = x;
180 }
181
182 void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size) //invoke any
183 permutation

```

```

171 {
172     for (int i = 0; i < perm_size; i++)
173     {
174         int byte_number_a = (perm[i] - 1) >> 3; //byte number of the array a[]
175         int bit_number_a = ((perm[i] - 1) & 7); //bit position of the element a[byte_number_a]
176         int byte_number_b = i >> 3; //byte number of the array b[]
177         int bit_number_b = i & 7; //bit position of the element b[byte_number_b]
178         b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
179     }
180 }
181
182 void feistel_structure(unsigned char *permuted_text, int k, unsigned char * key)
183 {
184     unsigned char left_permuted_text[4] = {permuted_text[0], permuted_text[1], permuted_text
185         [2], permuted_text[3]}, right_permuted_text[4] = {permuted_text[4], permuted_text[5],
186         permuted_text[6], permuted_text[7]}, expanded_right_permuted_text[8] = {0},
187     sbbox_output[4] = {0}, fies_perm_output[4] = {0}, subkey[8] = {0};
188
189     int i;
190     expand(right_permuted_text, expanded_right_permuted_text); //expanding the right half of
191     the permuted text
192     key_scheduling(key, subkey, k); //will put the roundkey in the array named subkey
193     for(i = 0; i < 8; i++)
194         expanded_right_permuted_text[i] ^= subkey[i]; //xoring the right half of the permuted
195         text with round key
196     sbbox(expanded_right_permuted_text, sbbox_output); //invoking sbbox
197     i_perm(sbbox_output, fies_perm_output, f_perm, 32); //invoking the last permutation inside
198     f function
199     for (i = 0; i < 4; i++)
200     {
201         permuted_text[i] ^= fies_perm_output[i]; //xoring the Feistel output with left half of
202         the previous round output
203         swap(&permuted_text[i], &permuted_text[i + 4]); //swapping the left half and right
204         half
205     }
206 }
207
208 void expand(unsigned char *a, unsigned char *c)
209 {
210     unsigned char b[6] = {0};
211     int i, j;
212     i_perm(a, b, E, 48); //invoking expansion permutation
213     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
214         from 6 8-bit block
215     {
216         c[i] = b[j] & 0x3f;
217         c[i + 1] = b[j] >> 6;
218         c[i + 1] |= (b[j + 1] & 0xf) << 2;
219         c[i + 2] = b[j + 1] >> 4;
220         c[i + 2] |= (b[j + 2] & 0x3) << 4;
221         c[i + 3] = b[j + 2] >> 2;
222     }
223 }
224
225 void sbbox(unsigned char *c, unsigned char *e)
226 {
227     unsigned char d[8] = { 0 };
228     int i;
229     for (i = 0; i < 8; i++)
230     {
231         int row = ((c[i] >> 5) << 1) + (c[i] & 1); //calculating row and column of the sbbox
232         int column = (c[i] >> 1) & 0xf;
233         d[i] = s[i][row][column];
234     }
235     for (i = 0; i < 4; i++)
236         e[i] = d[i << 1] | (d[(i << 1) + 1] << 4); //merging two consecutive 4-bit block to
237         get a 8 bit block
238 }
239
240 void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
241 {

```

```

231 unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
    new_lkey[7] = {0};
232 int i, j;
233 lkey[0] = key[0] & 0xf;
234 lkey[1] = key[0] >> 4;
235 lkey[2] = key[1] & 0xf;
236 lkey[3] = key[1] >> 4;
237 lkey[4] = key[2] & 0xf;
238 lkey[5] = key[2] >> 4;
239 lkey[6] = key[3] & 0xf;
240 rkey[0] = key[3] >> 4;
241 rkey[1] = key[4] & 0xf;
242 rkey[2] = key[4] >> 4;
243 rkey[3] = key[5] & 0xf;
244 rkey[4] = key[5] >> 4;
245 rkey[5] = key[6] & 0xf;
246 rkey[6] = key[6] >> 4; //dividing the key in left half and right half
247 if (k == 0 || k == 1 || k == 8 || k == 15) //rotate left by one bit while round number is 0
    or 1 or 8 or 15
248 {
249     for (i = 0; i < 7; i++)
250     {
251         new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
252         new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
253     }
254 }
255 else //rotate left by two bit while round number is different
256 {
257     for (i = 0; i < 7; i++)
258     {
259         new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
260         new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
261     }
262 }
263 key[0] = new_lkey[0] | (new_lkey[1] << 4); //merging two consecutive 4-bit block to get 8-
    bit block
264 key[1] = new_lkey[2] | (new_lkey[3] << 4);
265 key[2] = new_lkey[4] | (new_lkey[5] << 4);
266 key[3] = new_lkey[6] | (new_rkey[0] << 4);
267 key[4] = new_rkey[1] | (new_rkey[2] << 4);
268 key[5] = new_rkey[3] | (new_rkey[4] << 4);
269 key[6] = new_rkey[5] | (new_rkey[6] << 4);
270 i_perm(key, subkey_8bit, PC2, 48);
271 for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
    from 6 8-bit block
272 {
273     subkey[i] = subkey_8bit[j] & 0x3f;
274     subkey[i + 1] = subkey_8bit[j] >> 6;
275     subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
276     subkey[i + 2] = subkey_8bit[j + 1] >> 4;
277     subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
278     subkey[i + 3] = subkey_8bit[j + 2] >> 2;
279 }
280 }

```

OFB Decryption

```

1 #include<stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <time.h>
6
7 void DES_enc(unsigned char *, unsigned char *, unsigned char *);
8 void swap(unsigned char *, unsigned char *);
9 void i_perm(unsigned char *, unsigned char *, int *, int);
10 void feistel_structure(unsigned char *, int, unsigned char *);
11 void expand(unsigned char *, unsigned char *);
12 void sbox(unsigned char *, unsigned char *);
13 void key_scheduling(unsigned char *, unsigned char *, int );
14
15 int IP[] = {

```

```

16      58, 50, 42, 34, 26, 18, 10, 2,
17      60, 52, 44, 36, 28, 20, 12, 4,
18      62, 54, 46, 38, 30, 22, 14, 6,
19      64, 56, 48, 40, 32, 24, 16, 8,
20      57, 49, 41, 33, 25, 17, 9, 1,
21      59, 51, 43, 35, 27, 19, 11, 3,
22      61, 53, 45, 37, 29, 21, 13, 5,
23      63, 55, 47, 39, 31, 23, 15, 7
24  },
25  IP_inv[] = {
26      40, 8, 48, 16, 56, 24, 64, 32,
27      39, 7, 47, 15, 55, 23, 63, 31,
28      38, 6, 46, 14, 54, 22, 62, 30,
29      37, 5, 45, 13, 53, 21, 61, 29,
30      36, 4, 44, 12, 52, 20, 60, 28,
31      35, 3, 43, 11, 51, 19, 59, 27,
32      34, 2, 42, 10, 50, 18, 58, 26,
33      33, 1, 41, 9, 49, 17, 57, 25
34  },
35  E[] = {
36      32, 1, 2, 3, 4, 5,
37      4, 5, 6, 7, 8, 9,
38      8, 9, 10, 11, 12, 13,
39      12, 13, 14, 15, 16, 17,
40      16, 17, 18, 19, 20, 21,
41      20, 21, 22, 23, 24, 25,
42      24, 25, 26, 27, 28, 29,
43      28, 29, 30, 31, 32, 1
44  },
45  s[8][4][16] = {
46      {
47          14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12,
48          5, 9, 0, 7,
49          0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
50          4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
51          15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13
52      },
53      {
54          15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12,
55          0, 5, 10,
56          3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
57          0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
58          13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9
59      },
60      {
61          10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7,
62          11, 4, 2, 8,
63          13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
64          13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
65          1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12
66      },
67      {
68          7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
69          12, 4, 15,
70          13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
71          10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
72          3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
73      },
74      {
75          2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
76          0, 14, 9,
77          14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
78          4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
79          11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
80      },
81      {
82          12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
83          7, 5, 11,
84          10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
85          9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
86          4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
87      }
88  },

```

```

81         {
82             4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
10, 6, 1,
83             13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
84             1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
85             6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
86             },
87         {
88             13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
0, 12, 7,
89             1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
90             7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
91             2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
92             },
93         },
94     f_perm[32] = { 16, 7, 20, 21,
29, 12, 28, 17,
95     1, 15, 23, 26,
96     5, 18, 31, 10,
97     2, 8, 24, 14,
98     32, 27, 3, 9,
99     19, 13, 30, 6,
100    22, 11, 4, 25 },
101    PC1[56] = { 57, 49, 41, 33, 25, 17, 9,
102    1, 58, 50, 42, 34, 26, 18,
103    10, 2, 59, 51, 43, 35, 27,
104    19, 11, 3, 60, 52, 44, 36,
105    63, 55, 47, 39, 31, 23, 15,
106    7, 62, 54, 46, 38, 30, 22,
107    14, 6, 61, 53, 45, 37, 29,
108    21, 13, 5, 28, 20, 12, 4 },
109    PC2[48] = { 14, 17, 11, 24, 1, 5,
110    3, 28, 15, 6, 21, 10,
111    23, 19, 12, 4, 26, 8,
112    16, 7, 27, 20, 13, 2,
113    41, 52, 31, 37, 47, 55,
114    30, 40, 51, 45, 33, 48,
115    44, 49, 39, 56, 34, 53,
116    46, 42, 50, 36, 29, 32 };
117
118 void main()
119 {
120     unsigned char plain_text[8] = {0}, cipher_text[8] = {0}, super_key[8] = "lucifer1", key
121     [7] = {0}, IV[8] = {0}, padding_key[8] = {0};
122     int i, j, k;
123     int fd = open("encrypted", O_RDONLY, 0666), fd2 = open("decrypted", O_WRONLY | O_CREAT |
O_TRUNC, 0666);
124     read(fd, IV, 8 * sizeof(unsigned char)); //reading the IV from the cipher text
125     do
126     {
127         for(i = 0; i < 8; i++) //initializing all the variable by zero so that no junk value
become there in any intermediate state
128         {
129             plain_text[i] = 0;
130             cipher_text[i] = 0;
131             padding_key[i] = 0;
132         }
133         read(fd, cipher_text, 8 * sizeof(unsigned char)); //reading 8-byte from the encrypted
file and putting the i-th byte in the array cipher_text[]
134         DES_enc(IV, padding_key, super_key); //encrypting the output of the DES output of
previous round
135         for (i = 0; i < 8; i++)
136         {
137             plain_text[i] = cipher_text[i] ^ padding_key[i]; //padding the cipher text with
the output of DES
138             IV[i] = padding_key[i]; //setting the feedback of the next round
139         }
140         write(fd2, plain_text, 8 * sizeof(unsigned char)); //writting the plain text in the
file named encrypted
141     }
142     while(plain_text[7] != 0); //continuing the loop until the last byte of the plain text

```

```

    become NULL character
143 printf("\nplaintext is contained in the file named \"decrypted\".\n\n");
144
145 close(fd);
146 close(fd2);
147 }
148
149 void DES_enc(unsigned char *plain_text, unsigned char *cipher_text, unsigned char *super_key)
    //taking plain_text array and storing the encrypted text in th cipher_text array
150 {
151     unsigned char permuted_text[8] = {0}, key[8] = {0};
152     int i, j, k;
153     i_perm(plain_text, permuted_text, IP, 64); //invoking the initial permutation
154     i_perm(super_key, key, PC1, 56); //invoking PC-1 permutation of key scheduling
155     for (k = 0; k < 16; k++)
156         feistel_structure(permuted_text, k, key); //applying Feistel structure for 16 times
157     for (i = 0; i < 4; i++)
158         swap(&permuted_text[i], &permuted_text[i + 4]); //after all the Feistel round is done,
        final swap
159     i_perm(permuted_text, cipher_text, IP_inv, 64); //invoking final permutation, the inverse
        of the initial permutation
160 }
161
162 void swap(unsigned char *p, unsigned char *q) //basic swap function
163 {
164     unsigned char x = *p;
165     *p = *q;
166     *q = x;
167 }
168
169 void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size) //invoke any
    permutation
170 {
171     for (int i = 0; i < perm_size; i++)
172     {
173         int byte_number_a = (perm[i] - 1) >> 3; //byte number of the array a[]
174         int bit_number_a = ((perm[i] - 1) & 7); //bit position of the element a[byte_number_a]
175         int byte_number_b = i >> 3; //byte number of the array b[]
176         int bit_number_b = i & 7; //bit position of the element b[byte_number_b]
177         b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
178     }
179 }
180
181 void feistel_structure(unsigned char *permuted_text, int k, unsigned char *key)
182 {
183     unsigned char left_permuted_text[4] = {permuted_text[0], permuted_text[1], permuted_text
        [2], permuted_text[3]}, right_permuted_text[4] = {permuted_text[4], permuted_text[5],
        permuted_text[6], permuted_text[7]}, expanded_right_permuted_text[8] = {0},
        sbox_output[4] = {0}, fies_perm_output[4] = {0}, subkey[8] = {0};
184     int i;
185     expand(right_permuted_text, expanded_right_permuted_text); //expanding the right half of
        the permuted text
186     key_scheduling(key, subkey, k); //will put the roundkey in the array named subkey
187     for (i = 0; i < 8; i++)
188         expanded_right_permuted_text[i] ^= subkey[i]; //xoring the right half of the permuted
        text with round key
189     sbox(expanded_right_permuted_text, sbox_output); //invoking sbox
190     i_perm(sbox_output, fies_perm_output, f_perm, 32); //invoking the last permutation inside
        f function
191     for (i = 0; i < 4; i++)
192     {
193         permuted_text[i] ^= fies_perm_output[i]; //xoring the Feistel output with left half of
        the previous round output
194         swap(&permuted_text[i], &permuted_text[i + 4]); //swapping the left half and right
        half
195     }
196 }
197
198 void expand(unsigned char *a, unsigned char *c)
199 {
200     unsigned char b[6] = {0};

```

```

201     int i, j;
202     i_perm(a, b, E, 48); //invoking expansion permutation
203     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3) //making the output as 8 6-bit block
        from 6 8-bit block
204     {
205         c[i] = b[j] & 0x3f;
206         c[i + 1] = b[j] >> 6;
207         c[i + 1] |= (b[j + 1] & 0xf) << 2;
208         c[i + 2] = b[j + 1] >> 4;
209         c[i + 2] |= (b[j + 2] & 0x3) << 4;
210         c[i + 3] = b[j + 2] >> 2;
211     }
212 }
213
214 void sbbox(unsigned char *c, unsigned char *e)
215 {
216     unsigned char d[8] = { 0 };
217     int i;
218     for (i = 0; i < 8; i++)
219     {
220         int row = ((c[i] >> 5) << 1) + (c[i] & 1); //calculating row and column of the sbbox
221         int column = (c[i] >> 1) & 0xf;
222         d[i] = s[i][row][column];
223     }
224     for (i = 0; i < 4; i++)
225         e[i] = d[i << 1] | (d[(i << 1) + 1] << 4); //merging two consecutive 4-bit block to
        get a 8 bit block
226 }
227
228 void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
229 {
230     unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
231     int i, j;
232     lkey[0] = key[0] & 0xf;
233     lkey[1] = key[0] >> 4;
234     lkey[2] = key[1] & 0xf;
235     lkey[3] = key[1] >> 4;
236     lkey[4] = key[2] & 0xf;
237     lkey[5] = key[2] >> 4;
238     lkey[6] = key[3] & 0xf;
239     rkey[0] = key[3] >> 4;
240     rkey[1] = key[4] & 0xf;
241     rkey[2] = key[4] >> 4;
242     rkey[3] = key[5] & 0xf;
243     rkey[4] = key[5] >> 4;
244     rkey[5] = key[6] & 0xf;
245     rkey[6] = key[6] >> 4; //dividing the key in left half and right half
246     if (k == 0 || k == 1 | k == 8 | k == 15) //rotate left by one bit while round number is 0
        or 1 or 8 or 15
247     {
248         for (i = 0; i < 7; i++)
249         {
250             new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
251             new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
252         }
253     }
254     else //rotate left by two bit while round number is different
255     {
256         for (i = 0; i < 7; i++)
257         {
258             new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
259             new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
260         }
261     }
262     key[0] = new_lkey[0] | (new_lkey[1] << 4); //merging two consecutive 4-bit block to get 8-
        bit block
263     key[1] = new_lkey[2] | (new_lkey[3] << 4);
264     key[2] = new_lkey[4] | (new_lkey[5] << 4);
265     key[3] = new_lkey[6] | (new_rkey[0] << 4);
266     key[4] = new_rkey[1] | (new_rkey[2] << 4);

```

```
267     key[5] = new_rkey[3] | (new_rkey[4] << 4);
268     key[6] = new_rkey[5] | (new_rkey[6] << 4);
269     i_perm(key, subkey_8bit, PC2, 48);
270     for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3)//making the output as 8 6-bit block
        from 6 8-bit block
271     {
272         subkey[i] = subkey_8bit[j] & 0x3f;
273         subkey[i + 1] = subkey_8bit[j] >> 6;
274         subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
275         subkey[i + 2] = subkey_8bit[j + 1] >> 4;
276         subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
277         subkey[i + 3] = subkey_8bit[j + 2] >> 2;
278     }
279 }
```


References

- [1] Jan Pelzl (auth.) Christof Paar. *Understanding cryptography: a textbook for students and practitioners*. 1st ed. Springer, 2010. ISBN: 3642041000; 9783642041006; 3642446493; 9783642446498; 3642041019; 9783642041013. URL: libgen.li/file.php?md5=9884ee9fe8edf0bfc9d4e900044e6e57.
- [2] GeeksForGeeks. *GeeksForGeeks*. 2022. URL: <https://geeksforgeeks.com> (visited on 06/02/2022).
- [3] Yehuda Lindell Jonathan Katz. *Introduction to Modern Cryptography*. 3rd ed. Chapman & Hall / CRC Cryptography and Network Security Series. CRC Press, 2021. ISBN: 9781351133036; 1351133039.