Data Encryption Standard (DES)

Implementation in C

Cryptographic and Security Implementations

T. K. Garai

Indian Statistical Institute

Data Encryption Standard (DES)

Implementation in C

by

Student Name	Roll Number
Tamas Kanti Garai	CrS2116

Course:	M.Tech
Depertment:	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 imput $(110101)_2$ 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 $F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ 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 Standars 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.

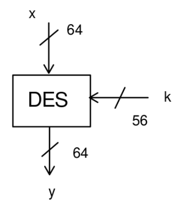

Figure 3.1: DES block cipher

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 iss 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 XORed (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:

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

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

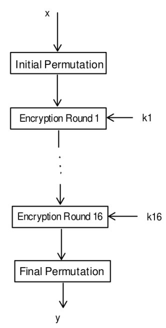

Figure 3.2: Iterative structure of DES

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.

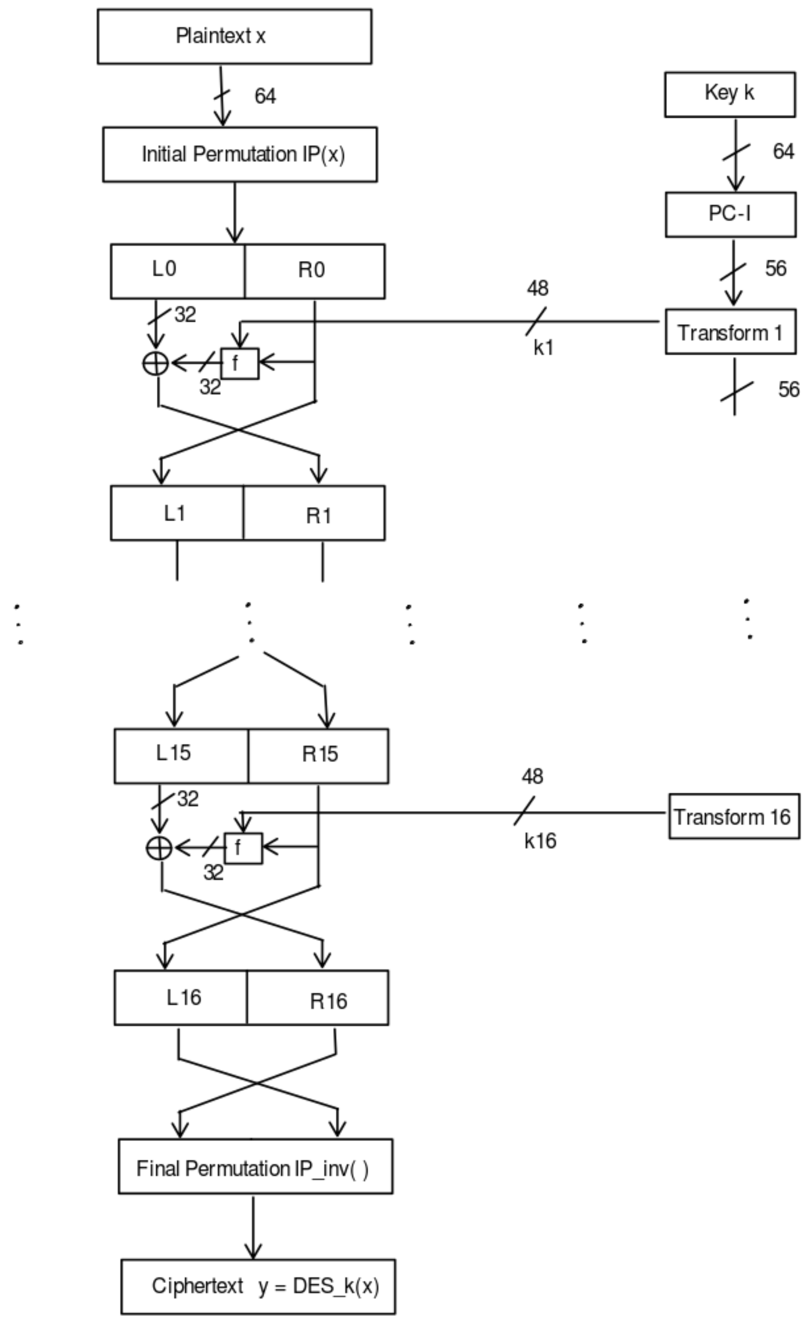

Figure 3.3: The Feistel structure of DES

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							
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

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

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

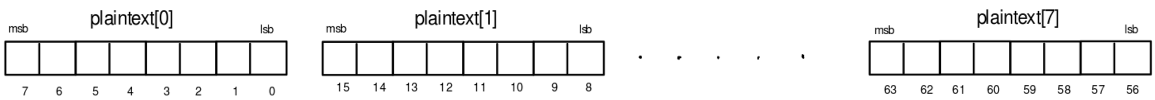

Figure 4.2: Bit map convention

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:

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

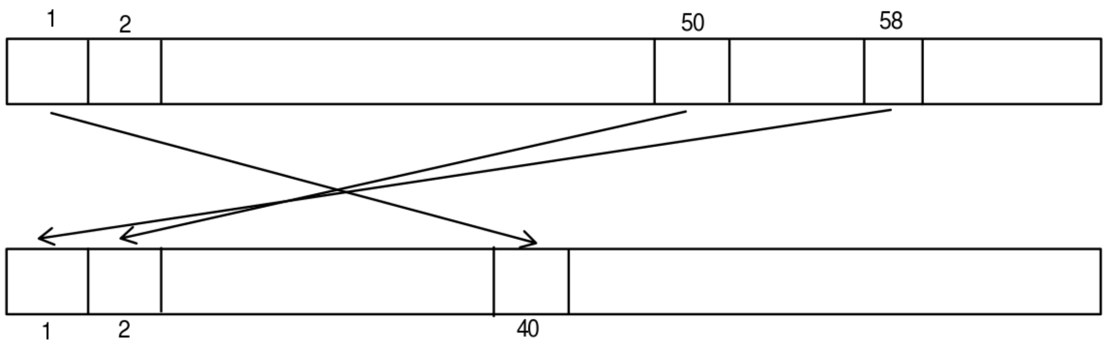

Figure 4.3: How IP works!

The c funtion 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 funtion. 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 output 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)

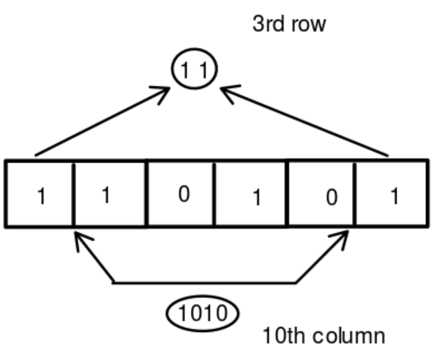

Figure 4.5: Example of decoding of the imput $(110101)_2$ by S-box

My Implementation

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

```c
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 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

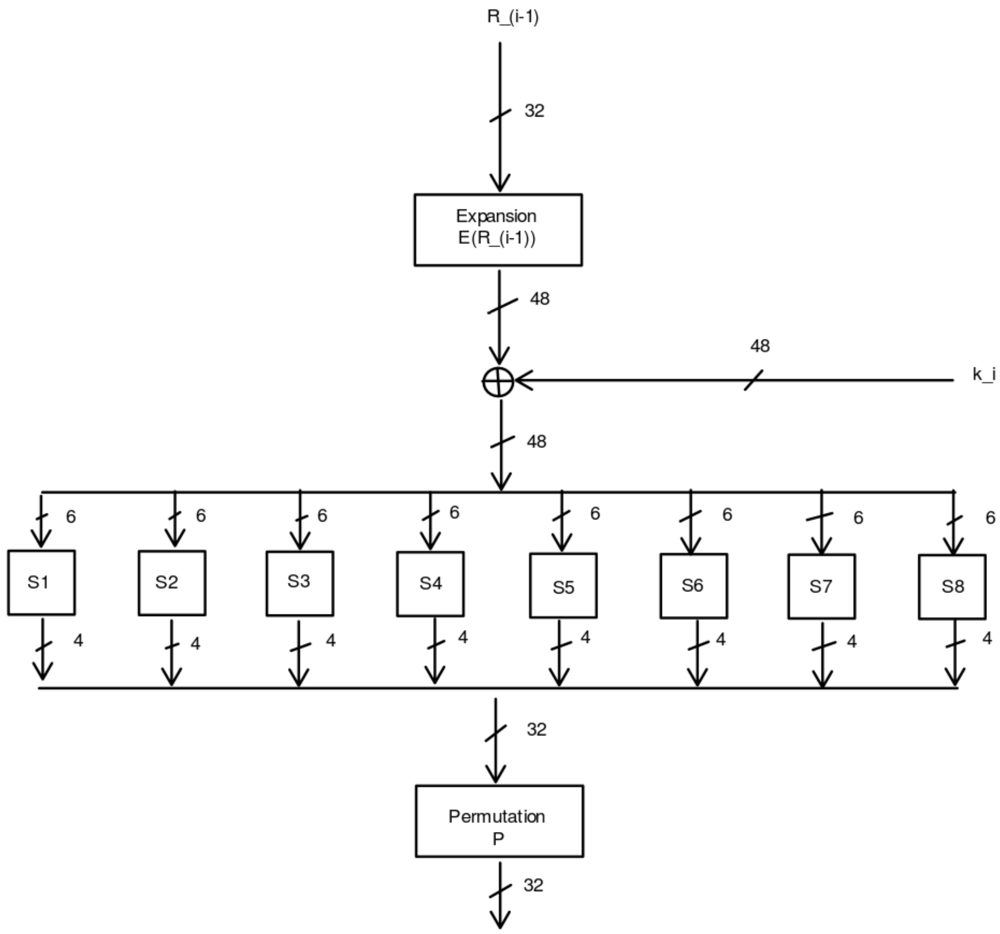

Figure 4.15: f-Function

$PC-1$							
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 shiftedf, 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 the figure 4.16

My Implementation

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

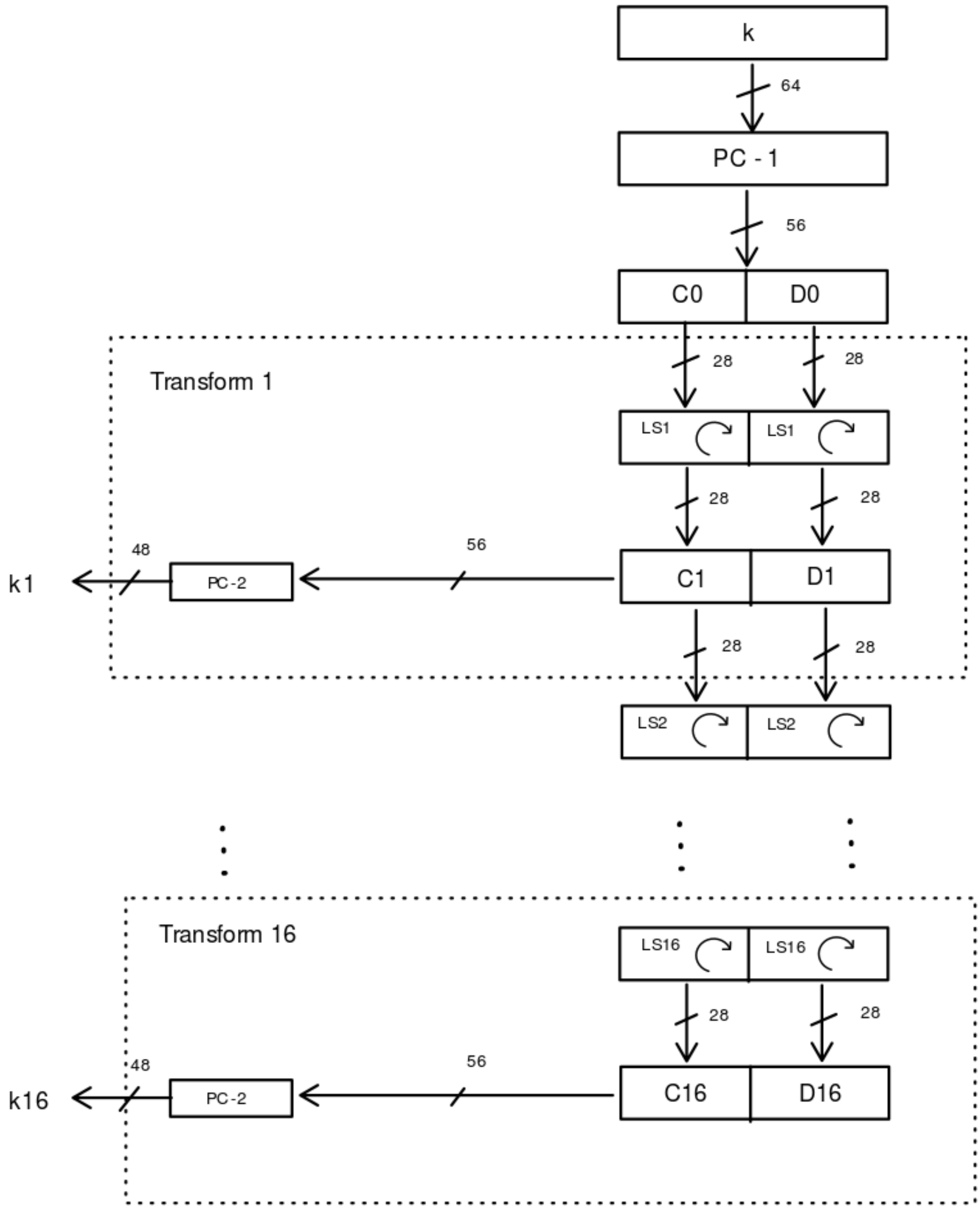

Figure 4.17: Key Schedule

```
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    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 lalf. 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:

```c
void Reversed_key_scheduling(unsigned char *key, unsigned char *subkey, int k)
{
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    lkey[6] = key[3] & 0xf;
    rkey[0] = key[3] >> 4;
    rkey[1] = key[4] & 0xf;
    rkey[2] = key[4] >> 4;
    rkey[3] = key[5] & 0xf;
    rkey[4] = key[5] >> 4;
    rkey[5] = key[6] & 0xf;
    rkey[6] = key[6] >> 4;
    if (k == 0)
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = lkey[i];
            new_rkey[i] = rkey[i];
        }
    }
    else if (k == 1 || k == 8 | k == 15 | k == 16)
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = ((lkey[i] << 1) | ((lkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
            new_rkey[i] = ((rkey[i] << 1) | ((rkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
        }
    }
    else
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = ((lkey[i] << 2) | ((lkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
            new_rkey[i] = ((rkey[i] << 2) | ((rkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
        }
    }
    key[0] = new_lkey[0] | (new_lkey[1] << 4);
    key[1] = new_lkey[2] | (new_lkey[3] << 4);
    key[2] = new_lkey[4] | (new_lkey[5] << 4);
    key[3] = new_lkey[6] | (new_rkey[0] << 4);
    key[4] = new_rkey[1] | (new_rkey[2] << 4);
    key[5] = new_rkey[3] | (new_rkey[4] << 4);
    key[6] = new_rkey[5] | (new_rkey[6] << 4);
    i_perm(key, subkey_8bit, PC2, 48);
    for (i = 0, j = 0; i <= 4, j <= 3; i += 4, j += 3)
    {
        subkey[i] = subkey_8bit[j] & 0x3f;
        subkey[i + 1] = subkey_8bit[j] >> 6;
        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
    }
}
```

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

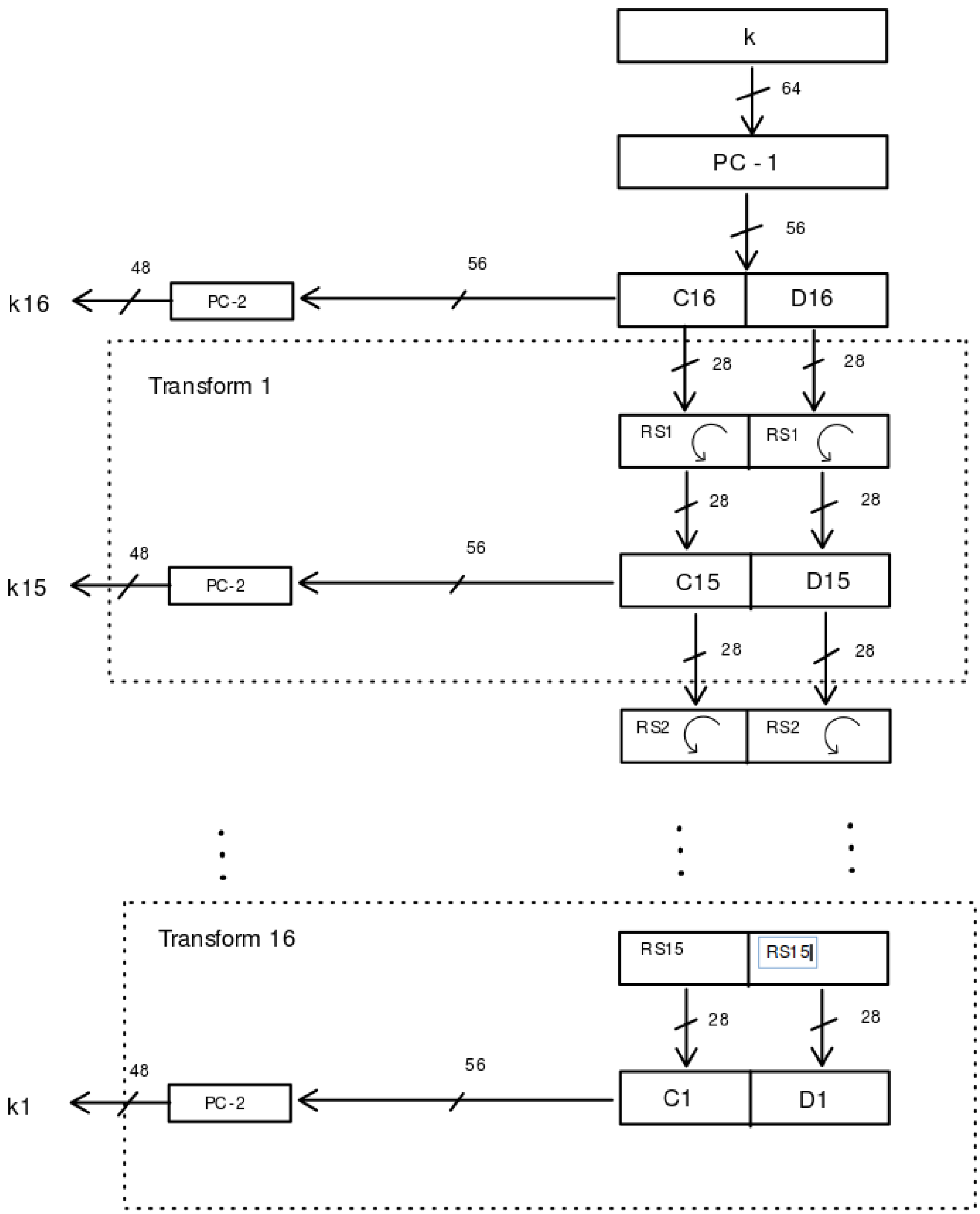

Figure 4.18: Reversed Key Schedule

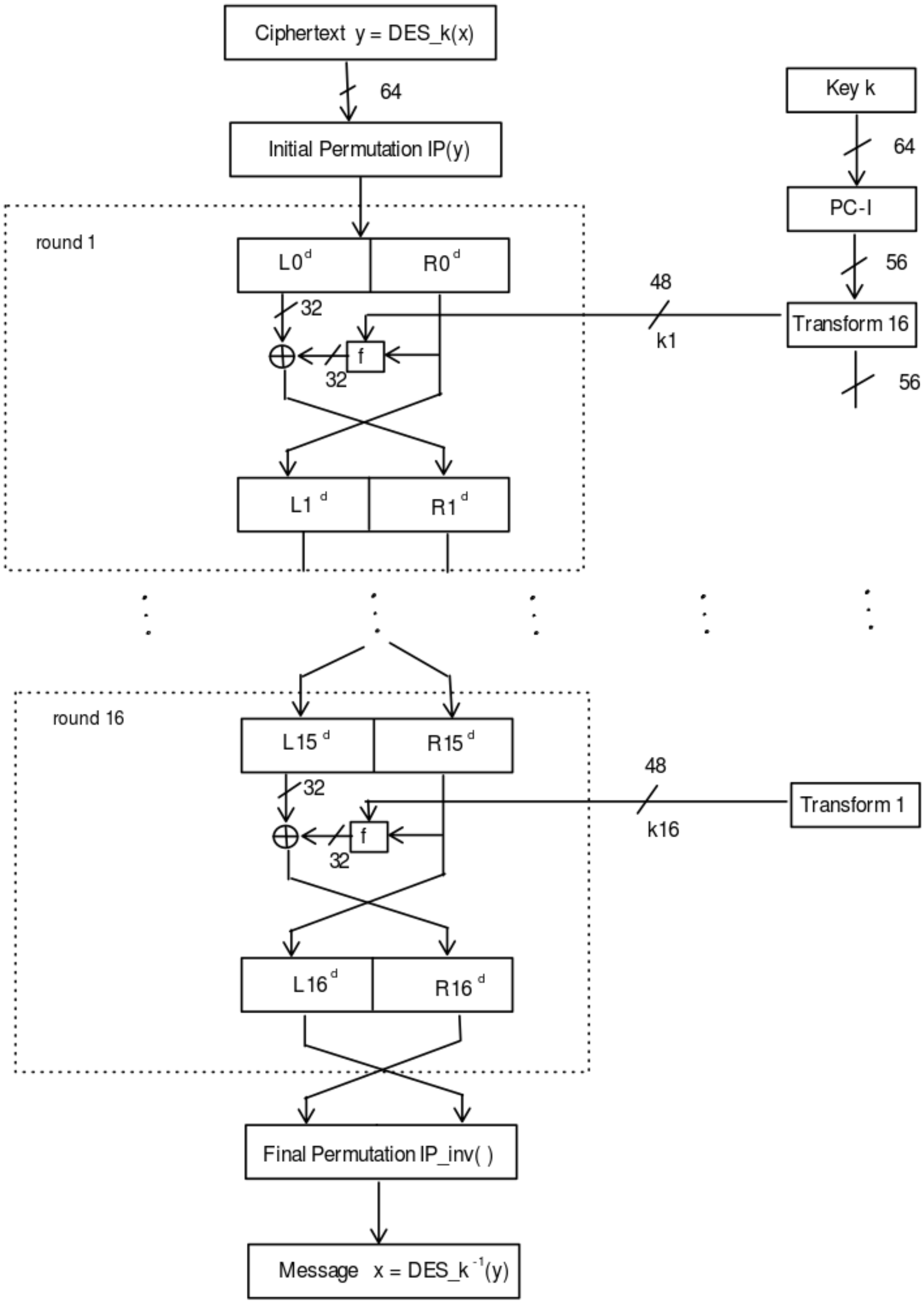

Figure 4.19: DES Decryption

5

C-Code for DES

DES Encryption:

```
#include<stdio.h>

void DES_enc(unsigned char *, unsigned char *, unsigned char *);
void swap(unsigned char *, unsigned char *);
void i_perm(unsigned char *, unsigned char *, int *, int);
void feistel_structure(unsigned char *, int , unsigned char *);
void expand(unsigned char *, unsigned char *);
void sbox(unsigned char *, unsigned char *);
void key_scheduling(unsigned char *, unsigned char *, int );

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

```
                                                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
                                              },
                      {
                                                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
                                              },
                      {
                                                7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
                                                    12, 4, 15,
                        13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
                        10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
                        3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
                                              },
                      {
                                                2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
                                                     0, 14, 9,
                        14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
                        4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
                        11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
                                              },
                      {
                                                12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
                                                     7, 5, 11,
                        10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
                        9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
                        4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
                                              },
                      {
                                                4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
                                                     10, 6, 1,
                        13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
                        1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
                        6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
                                              },
                      {
                                                13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
                                                     0, 12, 7,
                        1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
                        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
                                              }
                                      },
        f_perm[32] = {     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 },
    PC1[56] = { 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 },
    PC2[48] = { 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,
```

```c
                         44, 49, 39, 56, 34, 53,
                         46, 42, 50, 36, 29, 32 };

void main()
{
    unsigned char plain_text[8] = "SohanDas", permuted_text[8] = {0}, cipher_text[8] = {0}, super_key[8] = "Samapuki", key[7] = {0};
    int i, j, k;
    printf("plain text: ");
    for (i = 0; i < 8; i++)
        printf("%c", plain_text[i]);
    DES_enc(plain_text, cipher_text, super_key);
    printf("\ncipher text: ");
    for (i = 0; i < 8; i++)
        printf("%hhu, ", cipher_text[i]);
    printf("\n");
}

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
{
    unsigned char permuted_text[8] = {0}, key[8] = {0};
    int i, j, k;
    i_perm(plain_text, permuted_text, IP, 64);//invoking the initial permutation
    i_perm(super_key, key, PC1, 56);//invoking PC-1 permutation of key scheduling
    for (k = 0; k < 16; k++)
        feistel_structure(permuted_text, k, key);//applying Feistel structure for 16 times
    for (i = 0; i < 4; i++)
        swap(&permuted_text[i], &permuted_text[i + 4]);//after all the Feistel round is done, final swap
    i_perm(permuted_text, cipher_text, IP_inv, 64);//invoking final permutation, the inverse of the initial permutation
}

void swap(unsigned char *p, unsigned char *q)//basic swap function
{
    unsigned char x = *p;
    *p = *q;
    *q = x;
}

void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size)//invoke any permutation
{
    for (int i = 0; i < perm_size; i++)
    {
        int byte_number_a = (perm[i] - 1) >> 3;//byte number of the array a[]
        int bit_number_a = ((perm[i] - 1) & 7);//bit position of the element a[byte_number_a]
        int byte_number_b = i >> 3;//byte number of the array b[]
        int bit_number_b = i & 7;//bit position of the element b[byte_number_b]
        b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
    }
}

void feistel_structure(unsigned char *permuted_text, int k, unsigned char * key)
{
    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};
    int i;
    expand(right_permuted_text, expanded_right_permuted_text);//expanding the right half of the permuted text
    key_scheduling(key, subkey, k);//will put the roundkey in the array named subkey
    for(i = 0; i < 8; i++)
        expanded_right_permuted_text[i] ^= subkey[i];//xoring the right half of the permuted text with round key
    sbox(expanded_right_permuted_text, sbox_output);//invoking sbox
    i_perm(sbox_output, fies_perm_output, f_perm, 32);//invoking the last permutation inside f function
    for (i = 0; i < 4; i++)
```

```
    {
        permuted_text[i] ^= fies_perm_output[i];//xoring the Feistel output with left half of
             the previous round output
        swap(&permuted_text[i], &permuted_text[i + 4]);//swapping the left half and right
            half
    }
}

void expand(unsigned char *a, unsigned char *c)
{
    unsigned char b[6] = {0};
    int i, j;
    i_perm(a, b, E, 48);//invoking expansion permutation
    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
    {
        c[i] = b[j] & 0x3f;
        c[i + 1] = b[j] >> 6;
        c[i + 1] |= (b[j + 1] & 0xf) << 2;
        c[i + 2] = b[j + 1] >> 4;
        c[i + 2] |= (b[j + 2] & 0x3) << 4;
        c[i + 3] = b[j + 2] >> 2;
    }
}

void sbox(unsigned char *c, unsigned char *e)
{
    unsigned char d[8] = { 0 };
    int i;
    for (i = 0; i < 8; i++)
    {
        int row = ((c[i] >> 5) << 1) + (c[i] & 1);//calculating row and column of the sbox
        int column = (c[i] >> 1) & 0xf;
        d[i] = s[i][row][column];
    }
    for (i = 0; i < 4; i++)
        e[i] = d[i << 1] | (d[(i << 1) + 1] << 4);//merging two consecutive 4-bit block to
            get a 8 bit block
}

void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
{
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    lkey[6] = key[3] & 0xf;
    rkey[0] = key[3] >> 4;
    rkey[1] = key[4] & 0xf;
    rkey[2] = key[4] >> 4;
    rkey[3] = key[5] & 0xf;
    rkey[4] = key[5] >> 4;
    rkey[5] = key[6] & 0xf;
    rkey[6] = key[6] >> 4;//dividing the key in left half and right half
    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
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
            new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
        }
    }
    else//rotate left by two bit while round number is different
    {
        for (i = 0; i < 7; i++)
```

```
        {
            new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
            new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
        }
    }
    key[0] = new_lkey[0] | (new_lkey[1] << 4);//merging two consecutive 4-bit block to get 8-bit block
    key[1] = new_lkey[2] | (new_lkey[3] << 4);
    key[2] = new_lkey[4] | (new_lkey[5] << 4);
    key[3] = new_lkey[6] | (new_rkey[0] << 4);
    key[4] = new_rkey[1] | (new_rkey[2] << 4);
    key[5] = new_rkey[3] | (new_rkey[4] << 4);
    key[6] = new_rkey[5] | (new_rkey[6] << 4);
    i_perm(key, subkey_8bit, PC2, 48);
    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
    {
        subkey[i] = subkey_8bit[j] & 0x3f;
        subkey[i + 1] = subkey_8bit[j] >> 6;
        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
    }
}
```

DES Decryption

```
#include<stdio.h>

void DES_dec(unsigned char *, unsigned char *, unsigned char *);
void swap(unsigned char *, unsigned char *);
void i_perm(unsigned char *, unsigned char *, int *, int);
void feistel_structure(unsigned char *, int , unsigned char *);
void expand(unsigned char *, unsigned char *);
void sbox(unsigned char *, unsigned char *);
void key_scheduling(unsigned char *, unsigned char *, int );

int IP[] = {
                    58, 50, 42, 34, 26, 18, 10, 2,
                    60, 52, 44, 36, 28, 20, 12, 4,
                    62, 54, 46, 38, 30, 22, 14, 6,
                    64, 56, 48, 40, 32, 24, 16, 8,
                    57, 49, 41, 33, 25, 17,  9, 1,
                    59, 51, 43, 35, 27, 19, 11, 3,
                    61, 53, 45, 37, 29, 21, 13, 5,
                    63, 55, 47, 39, 31, 23, 15, 7
                },
    IP_inv[] = {
                    40, 8, 48, 16, 56, 24, 64, 32,
                    39, 7, 47, 15, 55, 23, 63, 31,
                    38, 6, 46, 14, 54, 22, 62, 30,
                    37, 5, 45, 13, 53, 21, 61, 29,
                    36, 4, 44, 12, 52, 20, 60, 28,
                    35, 3, 43, 11, 51, 19, 59, 27,
                    34, 2, 42, 10, 50, 18, 58, 26,
                    33, 1, 41,  9, 49, 17, 57, 25
                },
        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
                        },
        s[8][4][16] =  {        {
                                                  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
                                                },
                        {
                                                  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
                                                },
                        {
                                                  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
                                                },
                        {
                                                  7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
                          13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
                          10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
                          3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
                                                },
                        {
                                                  2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
                          14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
                          4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
                          11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
                                                },
                        {
                                                  12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
                          10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
                          9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
                          4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
                                                },
                        {
                                                  4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
                          13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
                          1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
                          6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
                                                },
                        {
                                                  13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
                          1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
                          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
                                                }
                                        },
        f_perm[32] = {     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 },
    PC1[56] = { 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 },
    PC2[48] = { 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 };

void main()
{
    unsigned char cipher_text[8] = {87, 193, 235, 190, 25, 243, 26, 171}, permuted_text[8] = {0}, plain_text[8] = {0}, super_key[8] = "Samapuki", key[7] = {0};
    int i, j, k;
    printf("plain text: ");
    for (i = 0; i < 8; i++)
        printf("%hhu, ", cipher_text[i]);
    DES_dec(cipher_text, plain_text, super_key);
    printf("\ncipher text: ");
    for (i = 0; i < 8; i++)
        printf("%c", plain_text[i]);
    printf("\n");
}

void DES_dec(unsigned char *cipher_text, unsigned char *plain_text, unsigned char *super_key) //taking cipher_text array and storing the decrypted text in th plain_text array
{
    unsigned char permuted_text[8] = {0}, key[8] = {0};
    int i, j, k;
    i_perm(cipher_text, permuted_text, IP, 64);//invoking the initial permutation
    i_perm(super_key, key, PC1, 56);//invoking PC-1 permutation of key scheduling
    for (k = 0; k < 16; k++)
        feistel_structure(permuted_text, k, key);//applying Feistel structure for 16 times
    for (i = 0; i < 4; i++)
        swap(&permuted_text[i], &permuted_text[i + 4]);//after all the Feistel round is done, final swap
    i_perm(permuted_text, plain_text, IP_inv, 64);//invoking final permutation, the inverse of the initial permutation
}

void swap(unsigned char *p, unsigned char *q)//basic swap function
{
    unsigned char x = *p;
    *p = *q;
    *q = x;
}

void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size)//invoke any permutation
{
    for (int i = 0; i < perm_size; i++)
    {
        int byte_number_a = (perm[i] - 1) >> 3;//byte number of the array a[]
        int bit_number_a = ((perm[i] - 1) & 7);//bit position of the element a[byte_number_a]
        int byte_number_b = i >> 3;//byte number of the array b[]
        int bit_number_b = i & 7;//bit position of the element b[byte_number_b]
        b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
    }
}

void feistel_structure(unsigned char *permuted_text, int k, unsigned char * key)
{
    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};
    int i;
    expand(right_permuted_text, expanded_right_permuted_text);//expanding the right half of the permuted text
    key_scheduling(key, subkey, k);//will put the roundkey in the array named subkey
    for(i = 0; i < 8; i++)
        expanded_right_permuted_text[i] ^= subkey[i];//xoring the right half of the permuted text with round key
```

```
    sbox(expanded_right_permuted_text, sbox_output);//invoking sbox
    i_perm(sbox_output, fies_perm_output, f_perm, 32);//invoking the last permutation inside f function
    for (i = 0; i < 4; i++)
    {
        permuted_text[i] ^= fies_perm_output[i];//xoring the Feistel output with left half of the previous round output
        swap(&permuted_text[i], &permuted_text[i + 4]);//swapping the left half and right half
    }
}

void expand(unsigned char *a, unsigned char *c)
{
    unsigned char b[6] = {0};
    int i, j;
    i_perm(a, b, E, 48);//invoking expansion permutation
    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
    {
        c[i] = b[j] & 0x3f;
        c[i + 1] = b[j] >> 6;
        c[i + 1] |= (b[j + 1] & 0xf) << 2;
        c[i + 2] = b[j + 1] >> 4;
        c[i + 2] |= (b[j + 2] & 0x3) << 4;
        c[i + 3] = b[j + 2] >> 2;
    }
}

void sbox(unsigned char *c, unsigned char *e)
{
    unsigned char d[8] = { 0 };
    int i;
    for (i = 0; i < 8; i++)
    {
        int row = ((c[i] >> 5) << 1) + (c[i] & 1);//calculating row and column of the sbox
        int column = (c[i] >> 1) & 0xf;
        d[i] = s[i][row][column];
    }
    for (i = 0; i < 4; i++)
        e[i] = d[i << 1] | (d[(i << 1) + 1] << 4);//merging two consecutive 4-bit block to get a 8 bit block
}

void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
{
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0}, new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    lkey[6] = key[3] & 0xf;
    rkey[0] = key[3] >> 4;
    rkey[1] = key[4] & 0xf;
    rkey[2] = key[4] >> 4;
    rkey[3] = key[5] & 0xf;
    rkey[4] = key[5] >> 4;
    rkey[5] = key[6] & 0xf;
    rkey[6] = key[6] >> 4;//dividing the key in left half and right half
    if (k == 0)//no rotation in the first round
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = lkey[i];
            new_rkey[i] = rkey[i];
        }
    }
```

```
    else if (k == 1 || k == 8 | k == 15 )//rotate left by one bit while round number 1 or 8
        or 15
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = ((lkey[i] << 1) | ((lkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
            new_rkey[i] = ((rkey[i] << 1) | ((rkey[(i + 6) % 7] >> 3) & 0x1)) & 0xf;
        }
    }
    else//rotate left by two bit while round number is different
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = ((lkey[i] << 2) | ((lkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
            new_rkey[i] = ((rkey[i] << 2) | ((rkey[(i + 6) % 7] >> 2) & 0x3)) & 0xf;
        }
    }
    key[0] = new_lkey[0] | (new_lkey[1] << 4);//merging two consecutive 4-bit block to get 8-
        bit block
    key[1] = new_lkey[2] | (new_lkey[3] << 4);
    key[2] = new_lkey[4] | (new_lkey[5] << 4);
    key[3] = new_lkey[6] | (new_rkey[0] << 4);
    key[4] = new_rkey[1] | (new_rkey[2] << 4);
    key[5] = new_rkey[3] | (new_rkey[4] << 4);
    key[6] = new_rkey[5] | (new_rkey[6] << 4);
    i_perm(key, subkey_8bit, PC2, 48);
    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
    {
        subkey[i] = subkey_8bit[j] & 0x3f;
        subkey[i + 1] = subkey_8bit[j] >> 6;
        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
    }
}
```

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.

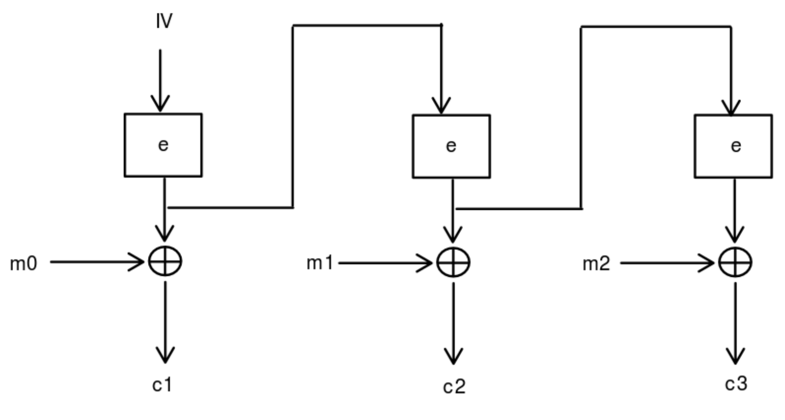

Figure 6.1: Block Diagram of OFB

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

```
void main()
{
    unsigned char plain_text[8] = {0}, cipher_text[8] = {0}, super_key[8] = "Samapuki", key
        [7] = {0}, IV[8] = "Tamasuki", padding_key[8] = {0};
    int i, j, k, m = 0;
    int fd = open("encrypted", O_RDONLY), fd2 = open("decrypted", O_WRONLY | O_CREAT |
        O_TRUNC);
    do
    {
        for(i = 0; i < 8; i++)
        {
            plain_text[i] = 0;
            cipher_text[i] = 0;
            key[i] = 0;
            padding_key[i] = 0;
        }
        read(fd, cipher_text, 8 * sizeof(unsigned char));
        printf("\ncipher text: \n");
        for (i = 0; i < 8; i++)
            printf("%c", cipher_text[i]);
        DES_enc(IV, padding_key, super_key);
        printf("\nplain text: \n");
        for (i = 0; i < 8; i++)
        {
            plain_text[i] = cipher_text[i] ^ padding_key[i];
            printf("%c", plain_text[i]);
            IV[i] = padding_key[i];
        }
        write(fd2, plain_text, 8 * sizeof(unsigned char));
        printf("\n");
        m++;
    }
    // while(m <= 30);
    while(plain_text[7] != 0);
    close(fd);
    close(fd2);
}
```

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]

```c
#include<stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
void DES_enc(unsigned char *, unsigned char *, unsigned char *);
void swap(unsigned char *, unsigned char *);
void i_perm(unsigned char *, unsigned char *, int *, int);
void feistel_structure(unsigned char *, int , unsigned char *);
void expand(unsigned char *, unsigned char *);
void sbox(unsigned char *, unsigned char *);
void key_scheduling(unsigned char *, unsigned char *, int );

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

```
                                             },
                        {
                                               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
                                             },
                        {
                                               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
                                             },
                        {
                                               7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
                                                   12, 4, 15,
                          13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
                          10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
                          3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
                                             },
                        {
                                               2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
                                                    0, 14, 9,
                          14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
                          4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
                          11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
                                             },
                        {
                                               12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
                                                    7, 5, 11,
                          10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
                          9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
                          4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
                                             },
                        {
                                               4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
                                                    10, 6, 1,
                          13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
                          1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
                          6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
                                             },
                        {
                                               13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
                                                    0, 12, 7,
                          1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
                          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
                                             }
                                     },
        f_perm[32] = {     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 },
    PC1[56] = { 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 },
    PC2[48] = { 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 };

void main()
{
    unsigned char plain_text[8] = {0}, permuted_text[8] = {0}, cipher_text[8] = {0},
        super_key[8] = "lucifer1", key[7] = {0}, IV[8] = {0}, padding_key[8] = {0};
    int i, j, k;
    srand(time(NULL));
    for(i = 0; i < 8; i++)
        IV[i] = rand() & 255;//taking IV randomly
    int fd = open("tk.txt", O_RDONLY, 0666), fd2 = open("encrypted", O_WRONLY | O_CREAT |
        O_TRUNC, 0666);
    write(fd2, IV, 8 * sizeof(unsigned char));//printing the IV at the begining of the
        ciphertext file
    do
    {
        for(i = 0; i < 8; i++)//initializing all the variable by zero so that no junk value
            become there in any intermediate state
        {
            plain_text[i] = 0;
            cipher_text[i] = 0;
            padding_key[i] = 0;
        }
        read(fd, plain_text, 8 * sizeof(unsigned char));//reading 8-byte from the tk.txt file
             and putting the i-th byte in the array plain_text[]
        DES_enc(IV, padding_key, super_key);//encrypting the output of the DES output of
            previous round
        for (i = 0; i < 8; i++)
        {
            cipher_text[i] = plain_text[i] ^ padding_key[i];//padding the plain text with the
                 output of DES
            IV[i] = padding_key[i];//setting the feedback of the next round
        }
        write(fd2, cipher_text, 8 * sizeof(unsigned char));//writting the cipher text in the
            file named encrypted
    }
    while(plain_text[7] != 0);//continuing the loop until the last byte of the plain text
        become NULL character
    printf("\nciphertext is contained in the file named \"encrypted\".\n\n");
    close(fd);
    close(fd2);
}

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
{
    unsigned char permuted_text[8] = {0}, key[8] = {0};
    int i, j, k;
    i_perm(plain_text, permuted_text, IP, 64);//invoking the initial permutation
    i_perm(super_key, key, PC1, 56);//invoking PC-1 permutation of key scheduling
    for (k = 0; k < 16; k++)
        feistel_structure(permuted_text, k, key);//applying Feistel structure for 16 times
    for (i = 0; i < 4; i++)
        swap(&permuted_text[i], &permuted_text[i + 4]);//after all the Feistel round is done,
             final swap
    i_perm(permuted_text, cipher_text, IP_inv, 64);//invoking final permutation, the inverse
        of the initial permutation
}

void swap(unsigned char *p, unsigned char *q)//basic swap function
{
    unsigned char x = *p;
    *p = *q;
    *q = x;
}

void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size)//invoke any
    permutation
```

```
{
    for (int i = 0; i < perm_size; i++)
    {
        int byte_number_a = (perm[i] - 1) >> 3;//byte number of the array a[]
        int bit_number_a = ((perm[i] - 1) & 7);//bit position of the element a[byte_number_a]
        int byte_number_b = i >> 3;//byte number of the array b[]
        int bit_number_b = i & 7;//bit position of the element b[byte_number_b]
        b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
    }
}

void feistel_structure(unsigned char *permuted_text, int k, unsigned char * key)
{
    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};
    int i;
    expand(right_permuted_text, expanded_right_permuted_text);//expanding the right half of
        the permuted text
    key_scheduling(key, subkey, k);//will put the roundkey in the array named subkey
    for(i = 0; i < 8; i++)
        expanded_right_permuted_text[i] ^= subkey[i];//xoring the right half of the permuted
            text with round key
    sbox(expanded_right_permuted_text, sbox_output);//invoking sbox
    i_perm(sbox_output, fies_perm_output, f_perm, 32);//invoking the last permutation inside
        f function
    for (i = 0; i < 4; i++)
    {
        permuted_text[i] ^= fies_perm_output[i];//xoring the Feistel output with left half of
             the previous round output
        swap(&permuted_text[i], &permuted_text[i + 4]);//swapping the left half and right
            half
    }
}

void expand(unsigned char *a, unsigned char *c)
{
    unsigned char b[6] = {0};
    int i, j;
    i_perm(a, b, E, 48);//invoking expansion permutation
    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
    {
        c[i] = b[j] & 0x3f;
        c[i + 1] = b[j] >> 6;
        c[i + 1] |= (b[j + 1] & 0xf) << 2;
        c[i + 2] = b[j + 1] >> 4;
        c[i + 2] |= (b[j + 2] & 0x3) << 4;
        c[i + 3] = b[j + 2] >> 2;
    }
}

void sbox(unsigned char *c, unsigned char *e)
{
    unsigned char d[8] = { 0 };
    int i;
    for (i = 0; i < 8; i++)
    {
        int row = ((c[i] >> 5) << 1) + (c[i] & 1);//calculating row and column of the sbox
        int column = (c[i] >> 1) & 0xf;
        d[i] = s[i][row][column];
    }
    for (i = 0; i < 4; i++)
        e[i] = d[i << 1] | (d[(i << 1) + 1] << 4);//merging two consecutive 4-bit block to
            get a 8 bit block
}

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

```
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0},
        new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    lkey[6] = key[3] & 0xf;
    rkey[0] = key[3] >> 4;
    rkey[1] = key[4] & 0xf;
    rkey[2] = key[4] >> 4;
    rkey[3] = key[5] & 0xf;
    rkey[4] = key[5] >> 4;
    rkey[5] = key[6] & 0xf;
    rkey[6] = key[6] >> 4;//dividing the key in left half and right half
    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
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
            new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
        }
    }
    else//rotate left by two bit while round number is different
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
            new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
        }
    }
    key[0] = new_lkey[0] | (new_lkey[1] << 4);//merging two consecutive 4-bit block to get 8-
        bit block
    key[1] = new_lkey[2] | (new_lkey[3] << 4);
    key[2] = new_lkey[4] | (new_lkey[5] << 4);
    key[3] = new_lkey[6] | (new_rkey[0] << 4);
    key[4] = new_rkey[1] | (new_rkey[2] << 4);
    key[5] = new_rkey[3] | (new_rkey[4] << 4);
    key[6] = new_rkey[5] | (new_rkey[6] << 4);
    i_perm(key, subkey_8bit, PC2, 48);
    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
    {
        subkey[i] = subkey_8bit[j] & 0x3f;
        subkey[i + 1] = subkey_8bit[j] >> 6;
        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
    }
}
```

OFB Decryption

```
#include<stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

void DES_enc(unsigned char *, unsigned char *, unsigned char *);
void swap(unsigned char *, unsigned char *);
void i_perm(unsigned char *, unsigned char *, int *, int);
void feistel_structure(unsigned char *, int , unsigned char *);
void expand(unsigned char *, unsigned char *);
void sbox(unsigned char *, unsigned char *);
void key_scheduling(unsigned char *, unsigned char *, int );

int IP[] = {
```

```
                    58, 50, 42, 34, 26, 18, 10, 2,
                    60, 52, 44, 36, 28, 20, 12, 4,
                    62, 54, 46, 38, 30, 22, 14, 6,
                    64, 56, 48, 40, 32, 24, 16, 8,
                    57, 49, 41, 33, 25, 17,  9, 1,
                    59, 51, 43, 35, 27, 19, 11, 3,
                    61, 53, 45, 37, 29, 21, 13, 5,
                    63, 55, 47, 39, 31, 23, 15, 7
                },
    IP_inv[] = {
                    40, 8, 48, 16, 56, 24, 64, 32,
                    39, 7, 47, 15, 55, 23, 63, 31,
                    38, 6, 46, 14, 54, 22, 62, 30,
                    37, 5, 45, 13, 53, 21, 61, 29,
                    36, 4, 44, 12, 52, 20, 60, 28,
                    35, 3, 43, 11, 51, 19, 59, 27,
                    34, 2, 42, 10, 50, 18, 58, 26,
                    33, 1, 41,  9, 49, 17, 57, 25
                },
        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
                        },
        s[8][4][16] =  {        {
                                                  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
                                                },
                        {
                                                  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
                                                },
                        {
                                                  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
                                                },
                        {
                                                  7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
                                                      12, 4, 15,
                          13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
                          10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
                          3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14
                                                },
                        {
                                                  2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
                                                       0, 14, 9,
                          14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
                          4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
                          11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3
                                                },
                        {
                                                  12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
                                                       7, 5, 11,
                          10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
                          9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
                          4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13
                                                },
```

```
                        {
                                                  4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
                          13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
                          1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
                          6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12
                                              },
                        {
                                                  13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
                          1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
                          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
                                              }
                                      },
        f_perm[32] = {     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 },
    PC1[56] = { 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 },
    PC2[48] = { 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 };

void main()
{
    unsigned char plain_text[8] = {0}, cipher_text[8] = {0}, super_key[8] = "lucifer1", key[7] = {0}, IV[8] = {0}, padding_key[8] = {0};
    int i, j, k;
    int fd = open("encrypted", O_RDONLY, 0666), fd2 = open("decrypted", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    read(fd, IV, 8 * sizeof(unsigned char));//reading the IV from the cipher text
    do
    {
        for(i = 0; i < 8; i++)//initializing all the variable by zero so that no junk value become there in any intermediate state
        {
            plain_text[i] = 0;
            cipher_text[i] = 0;
            padding_key[i] = 0;
        }
        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[]
        DES_enc(IV, padding_key, super_key);//encrypting the output of the DES output of previous round
        for (i = 0; i < 8; i++)
        {
            plain_text[i] = cipher_text[i] ^ padding_key[i];//padding the cipher text with the output of DES
            IV[i] = padding_key[i];//setting the feedback of the next round
        }
        write(fd2, plain_text, 8 * sizeof(unsigned char));//writting the plain text in the file named encrypted
    }
    while(plain_text[7] != 0);//continuing the loop until the last byte of the plain text
```

```
        become NULL character
    printf("\nplaintext is contained in the file named \"decrypted\".\n\n");

    close(fd);
    close(fd2);
}

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
{
    unsigned char permuted_text[8] = {0}, key[8] = {0};
    int i, j, k;
    i_perm(plain_text, permuted_text, IP, 64);//invoking the initial permutation
    i_perm(super_key, key, PC1, 56);//invoking PC-1 permutation of key scheduling
    for (k = 0; k < 16; k++)
        feistel_structure(permuted_text, k, key);//applying Feistel structure for 16 times
    for (i = 0; i < 4; i++)
        swap(&permuted_text[i], &permuted_text[i + 4]);//after all the Feistel round is done, final swap
    i_perm(permuted_text, cipher_text, IP_inv, 64);//invoking final permutation, the inverse of the initial permutation
}

void swap(unsigned char *p, unsigned char *q)//basic swap function
{
    unsigned char x = *p;
    *p = *q;
    *q = x;
}

void i_perm(unsigned char *a, unsigned char *b, int *perm, int perm_size)//invoke any permutation
{
    for (int i = 0; i < perm_size; i++)
    {
        int byte_number_a = (perm[i] - 1) >> 3;//byte number of the array a[]
        int bit_number_a = ((perm[i] - 1) & 7);//bit position of the element a[byte_number_a]
        int byte_number_b = i >> 3;//byte number of the array b[]
        int bit_number_b = i & 7;//bit position of the element b[byte_number_b]
        b[byte_number_b] |= ((a[byte_number_a] >> bit_number_a) & 1) << bit_number_b;
    }
}

void feistel_structure(unsigned char *permuted_text, int k, unsigned char * key)
{
    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};
    int i;
    expand(right_permuted_text, expanded_right_permuted_text);//expanding the right half of the permuted text
    key_scheduling(key, subkey, k);//will put the roundkey in the array named subkey
    for(i = 0; i < 8; i++)
        expanded_right_permuted_text[i] ^= subkey[i];//xoring the right half of the permuted text with round key
    sbox(expanded_right_permuted_text, sbox_output);//invoking sbox
    i_perm(sbox_output, fies_perm_output, f_perm, 32);//invoking the last permutation inside f function
    for (i = 0; i < 4; i++)
    {
        permuted_text[i] ^= fies_perm_output[i];//xoring the Feistel output with left half of the previous round output
        swap(&permuted_text[i], &permuted_text[i + 4]);//swapping the left half and right half
    }
}

void expand(unsigned char *a, unsigned char *c)
{
    unsigned char b[6] = {0};
```

```c
    int i, j;
    i_perm(a, b, E, 48);//invoking expansion permutation
    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
    {
        c[i] = b[j] & 0x3f;
        c[i + 1] = b[j] >> 6;
        c[i + 1] |= (b[j + 1] & 0xf) << 2;
        c[i + 2] = b[j + 1] >> 4;
        c[i + 2] |= (b[j + 2] & 0x3) << 4;
        c[i + 3] = b[j + 2] >> 2;
    }
}

void sbox(unsigned char *c, unsigned char *e)
{
    unsigned char d[8] = { 0 };
    int i;
    for (i = 0; i < 8; i++)
    {
        int row = ((c[i] >> 5) << 1) + (c[i] & 1);//calculating row and column of the sbox
        int column = (c[i] >> 1) & 0xf;
        d[i] = s[i][row][column];
    }
    for (i = 0; i < 4; i++)
        e[i] = d[i << 1] | (d[(i << 1) + 1] << 4);//merging two consecutive 4-bit block to get a 8 bit block
}

void key_scheduling(unsigned char *key, unsigned char *subkey, int k)
{
    unsigned char lkey[7] = {0}, rkey[7] = {0}, subkey_8bit[6] = {0}, new_rkey[7] = {0}, new_lkey[7] = {0};
    int i, j;
    lkey[0] = key[0] & 0xf;
    lkey[1] = key[0] >> 4;
    lkey[2] = key[1] & 0xf;
    lkey[3] = key[1] >> 4;
    lkey[4] = key[2] & 0xf;
    lkey[5] = key[2] >> 4;
    lkey[6] = key[3] & 0xf;
    rkey[0] = key[3] >> 4;
    rkey[1] = key[4] & 0xf;
    rkey[2] = key[4] >> 4;
    rkey[3] = key[5] & 0xf;
    rkey[4] = key[5] >> 4;
    rkey[5] = key[6] & 0xf;
    rkey[6] = key[6] >> 4;//dividing the key in left half and right half
    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
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = (lkey[i] >> 1) | ((lkey[(i + 1) % 7] & 0x1) << 3);
            new_rkey[i] = (rkey[i] >> 1) | ((rkey[(i + 1) % 7] & 0x1) << 3);
        }
    }
    else//rotate left by two bit while round number is different
    {
        for (i = 0; i < 7; i++)
        {
            new_lkey[i] = (lkey[i] >> 2) | ((lkey[(i + 1) % 7] & 0x3) << 2);
            new_rkey[i] = (rkey[i] >> 2) | ((rkey[(i + 1) % 7] & 0x3) << 2);
        }
    }
    key[0] = new_lkey[0] | (new_lkey[1] << 4);//merging two consecutive 4-bit block to get 8-bit block
    key[1] = new_lkey[2] | (new_lkey[3] << 4);
    key[2] = new_lkey[4] | (new_lkey[5] << 4);
    key[3] = new_lkey[6] | (new_rkey[0] << 4);
    key[4] = new_rkey[1] | (new_rkey[2] << 4);
```

```
    key[5] = new_rkey[3] | (new_rkey[4] << 4);
    key[6] = new_rkey[5] | (new_rkey[6] << 4);
    i_perm(key, subkey_8bit, PC2, 48);
    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
    {
        subkey[i] = subkey_8bit[j] & 0x3f;
        subkey[i + 1] = subkey_8bit[j] >> 6;
        subkey[i + 1] |= (subkey_8bit[j + 1] & 0xf) << 2;
        subkey[i + 2] = subkey_8bit[j + 1] >> 4;
        subkey[i + 2] |= (subkey_8bit[j + 2] & 0x3) << 4;
        subkey[i + 3] = subkey_8bit[j + 2] >> 2;
    }
}
```

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.