

Implementation of Output Feedback Mode using DES in C

Sourav Ghosh

Guide :Asst. Prof. Sabyasachi Karati

Indian Statistical Institute

September 2022

Abstract

In this report we will focus on the implementation of most studied historical symmetric-key block cipher Data Encryption Standard (DES) in C language. Then using DES we will encrypt a text file with the help of one of the modes of operation called Output Feedback mode (OFB) .

Contents

1	Introduction	4
2	Permutation Function	5
3	The DES Round Function	8
4	Key Scheduling	11
5	Encryption & Decryption Function	13
6	Output Feedback Mode using DES	14

1 Introduction

DES was developed in the 1970s by IBM with help from the National Security Agency and adopted by the US in 1977 as a Federal Information Processing Standard. The idea was to find a single secure cryptographic algorithm which could be used for a variety of cryptographic applications in the domain of symmetric key cryptography. After many years of research till now the best attack on DES in practice is an exhaustive search over all 2^{56} possible keys.

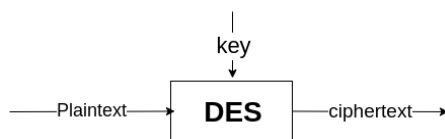


Figure 1: DES Overview

The DES block cipher is a 16-round Feistel network with a block length of 64 bits and an effective key length of 56 bit. The round function takes a 48 bit sub-keys k_1, k_2, \dots, k_{16} from the 56 bit effective key with the help of key scheduling. To construct the round function we also need 8 s-boxes that are crucial element of DES construction and were very carefully designed. Each s-box maps a 6-bit input to a 4 bit output. Studies of DES have shown that if the S-boxes were slightly modified, DES would have been much more vulnerable to the attack.

Here I will write a c code to to construct a DES and then use that to encrypt a file using one of Modes of Encryption process called Output Feedback mode (OFB).

In construction of DES encryption function, I will build each of the part (permutation, s-box, key-scheduling) separately and then merge them to construct the whole DES encryption function. Then similarly I did decryption function with reverse key scheduling.

2 Permutation Function

To add a non linearity(confusion+diffusion) factor in DES we do a initial permutation before 16 round encryptions and apply inverse of this initial permutation after 16th round of encryption process.

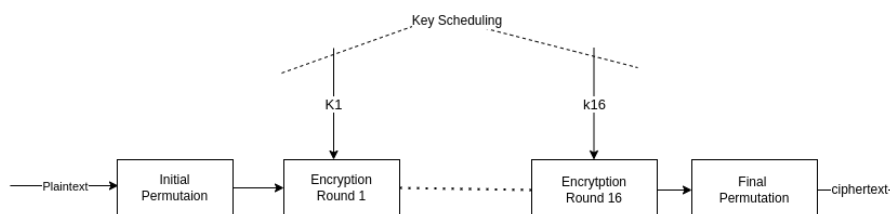


Figure 2: DES Outline

Implementation Idea: We are given with the two permutations. So I permuted the bits of our plain text according to the given permutation matrix. Similarly after 16th round of encryption I applied the inverse permutation function on the 16th round cipher text.

```
//initial_permutation
int initial_perm[64] = { 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
                        };

//final_permutation
int final_perm[64] = {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 3: initial and final permutation matrix

Here I defined a permutation function, which takes character array c as a input character array, b as a output character array after applying the permutation, perm is the permutation matrix required for the corresponding operations and lastly size is the number of elements in the matrix(we can derive this also but for the simplicity I took this as an argument. This function

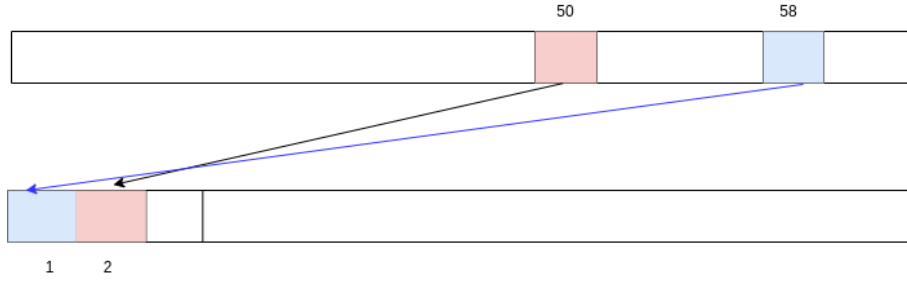


Figure 4: Interpretation of initial permutation matrix

we can use for expansion also). So here we are applying the permutation perm on array c and we are getting the output of this operation in the array b.

```

void permu (unsigned char *c, unsigned char b[], int *perm, int size)
{
    int i, x, s, y, z;
    for (i = 0; i < size; i++)
    {
        x = i >> 3;                                //division by 8
        s = i & 7;
        y = (perm[i] - 1) & 7;    // remainder after dividing by 8
        z = (perm[i] - 1) >> 3;
        b[x] += ((c[z] >> (7 - y)) & 1) << (7 - s);
    }
}

```

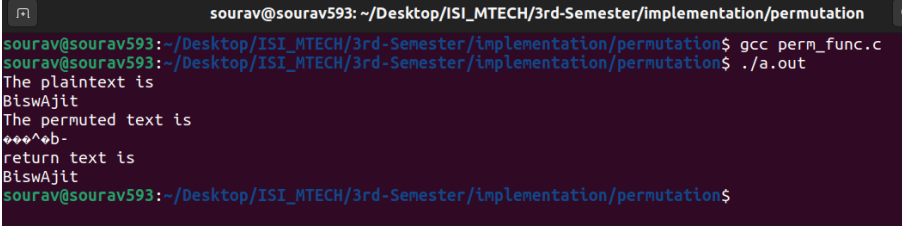
Here I defined character arrays so each position of character array is taking 8 bit size. For the character array c[8], the machine will allocate the size of 64 bit. In permutation function c and b will occupy same size. We have to identify which position of c have to be mapped in which position of b. Observed that to find this position we have to identify the source and the target bits are lying in which c[z] and b[x] respectively. Lets discuss how we can find that x and z. Since I have taken a for loop from i=0 and by the construction of the matrix perm which have values from 1 to 64 we can set

$$x = i \gg 3 \text{ \& } z = (perm[i] - 1) \gg 3;$$

Now after getting x and z we have to find the exact position of source and target bit position. Here if I take the remainder after (perm[i]-1) divided by eight, I will get the desired target bit position on c[z].

$$y = (perm[i] - 1) \& 7;$$

Now how we know the position but how to access those bit position? To get the value of the source bit position we have to calculate the $(c[z] \gg (7 - y)) \& 1$. Similarly to put the source bit value to the correct target position we have to do left shift by $(7 - s)$ where $s = i \& 7$. (note $b[8] = \{0\}$).



```
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation/permutation
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/permutation$ gcc perm_func.c
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/permutation$ ./a.out
The plaintext is
BiswAjit
The permuted text is
BiswAjit
return text is
BiswAjit
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/permutation$
```

Figure 5: applying initial and final permutation of plain text

Now for the verification purpose we can apply inverse permutation on the permuted text and check that we can get the initial text or not. Giving $c[8] = \text{'BiswAjit'}$ as plain text I verified my algorithm.

3 The DES Round Function

The DES round function sometimes called the DES mangler function is a crucial part for construction of DES. At each round of the 16th round it takes the right R_{i-1} of the output of the previous round and the current round key k_i as input(from key scheduling algorithm). The output of the f-function is used as an XOR-mask for encrypting the left half input bits.

$$L_i = R_{i-1} \quad \& \quad R_i = L_{i-1} \oplus f_{i-1}(R_{i-1})$$

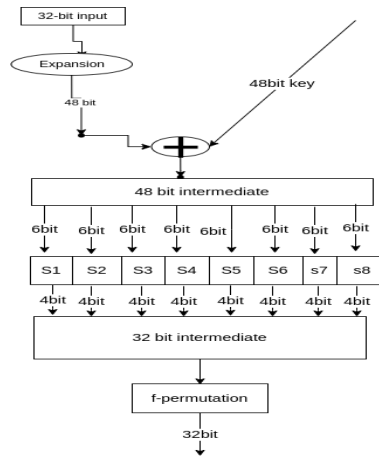


Figure 6: f-function

The way I created the permutation function perm I can use that function here for expansion and give 48 bit output. After that xoring with 48bit round key I have to put the consecutive 6 bits in each s-boxes to get 4bit output from each s box. So in total we are getting 32 bit output from the s-boxes. After that using the permutation function and permutation matrix of f permutation we are getting the output.

Implementation: If I do the right shift by 2 on expanded[0] then I will have the first 6bits of the expanded array then I stored them in a character array compr. In the next step I required the last 2bits of expanded[0] and first 4bits from the expanded[1]. Likewise I proceed to the whole array.

```

//making new array of 64 bit to apply 8 s-boxes
compr[0] = expanded[0] >> 2;
compr[1] = (expanded[0] & 0X3) << 4;
compr[1] = compr[1] | (expanded[1] >> 4);
compr[2] = (expanded[1] & 0X0F) << 2;
compr[2] = compr[2] | (expanded[2] >> 6);
  
```



```

compr[3] = expanded[2] & 0X3F;

compr[4] = expanded[3] >> 2;
compr[5] = (expanded[3] & 0X3) << 4;
compr[5] = compr[5] | (expanded[4] >> 4);
compr[6] = (expanded[4] & 0X0F) << 2;
compr[6] = compr[6] | (expanded[5] >> 6);
compr[7] = expanded[5] & 0x3F;

```

S-box S_1

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 7: first s box

We can consider the each s boxes as 4×16 array. From the 6bit input we will calculate the row and column number and then output the corresponding value associated to that row and column number. Observed that in s-boxes each entry is in between 1 to 15 so from s box we always get an output of 4bit. The row number will be decided from the MSB & LSB of given 6bit input and the column number will be decided based on the bits lying between MSB & LSB.

For example if the first S-box input $b = (101101)_2$ indicates the row $(11)_2 = 3$ (i.e., $(3 + 1) = 4$ th row and the column $(0110)_2 = 6$ (i.e., the $(6 + 1) = 7$ th column). If the input b is fed into S-box 1, the output is $1 = (0001)_2$.

Implementation wise this is how we are getting the rows and column number from the compr array. Storing the corresponding values of 8 s-boxes into character array `pre[8]`.

```

//accessing sbox
    for (i = 0; i < 8; i++)
    {
        int row, col;
        row = (compr[i] & 0x1) | ((compr[i] & 0x20) >> 4);
        col = ((compr[i] >> 1) & 0x0F);
        pre[i] = sb[i][row][col];
    }
    //returning to 32 bit array
    for (i = 0; i < 8; i = i + 2)
        final[i >> 1] = ((pre[i] << 4) | pre[i + 1]);

```

Then merging two consecutive byte of pre into a single byte neglecting the zeros. Then we have a character array final of 4byte from pre. Then by our perm function which we already build I did the f-permutation of 32 bits. Thus I am done with DES round function. The next main thing which I have to do is key scheduling.

4 Key Scheduling

The key schedule algorithm gives 16 round keys k_i , each consisting of 48 bits, from the original key of 64 bits we can derive 56 bits effective by deleting the multiple of 8bit positions. Then we divide the 56 bits array into two equal part of 28bits each. The two 28 bits halves are cyclically shifted i.e., rotated by left one or two bit positions depending on the round i according to the following rules:

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

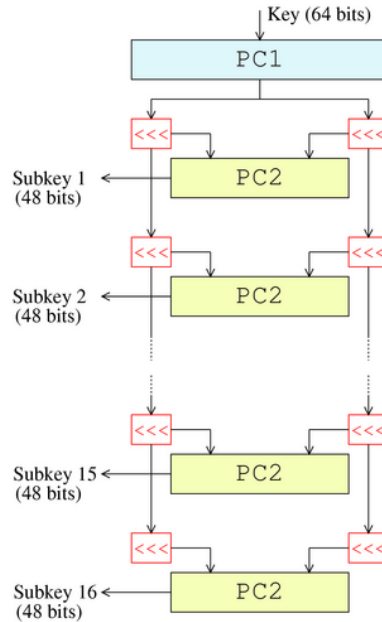


Figure 8: key scheduling algorithm(source Wikipedia)

Note that the rotations only take place within either the left or the right half. The total number of rotation positions is 28. This leads to the interesting property that in the 16th round we get back the initial 56 bit effective key. This is very useful for the decryption key schedule where the subkeys have to be generated in reversed order.

Implementation Here we have two given matrix pc-1 & pc-2. The first one helps me to get 56 bits effective key from the 64 bits key. pc-2 will

```
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation...
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation/final$ gcc key.c
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation/final$ ./a.out
key is:66,73,115,119,97,106,105,116,
keyfifty_six is:0,255,252,130,216,134,44,
left+right at round1 :1,255,249,5,177,12,88,
left+right at round2 :3,255,242,11,98,24,176,
left+right at round3 :15,255,200,13,136,98,194,
left+right at round4 :63,255,32,6,33,139,11,
left+right at round5 :255,252,108,8,134,44,45,
left+right at round6 :255,242,0,50,24,176,182,
left+right at round7 :255,208,0,248,98,194,216,
left+right at round8 :255,32,3,241,139,11,98,
left+right at round9 :254,64,7,243,22,22,196,
left+right at round10 :249,0,31,252,88,91,16,
left+right at round11 :228,0,127,241,97,108,67,
left+right at round12 :144,1,255,245,133,177,12,
left+right at round13 :64,7,255,230,22,196,40,
left+right at round14 :0,31,255,152,91,16,197,
left+right at round15 :0,127,254,65,108,67,22,
left+right at round16 :0,255,252,130,216,134,44,
1th Round key:240,182,102,67,96,109,
2th Round key:224,158,118,96,206,131,
3th Round key:228,242,114,30,36,27,
4th Round key:166,215,114,175,81,64,
5th Round key:238,83,83,0,227,98,
6th Round key:47,211,89,244,148,4,
7th Round key:15,81,210,200,6,218,
8th Round key:63,73,217,29,242,9,
9th Round key:31,89,153,40,14,45,
10th Round key:31,41,205,90,88,150,
11th Round key:27,108,141,5,65,185,
12th Round key:89,45,172,131,56,65,
13th Round key:208,172,173,226,131,52,
14th Round key:209,174,38,17,15,142,
15th Round key:224,190,166,92,16,145,
16th Round key:224,182,38,184,160,228,
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation/final$
```

Figure 9: output of key-scheduling

help me to get 48bits round key in each round. I have taken help of 56 bits character arrays to store 28bits each half. Then performing the shifting operation on them accordingly I merged them into 56 bits to apply pc-2.

In figure 9 we can see that we get back our initial 56 bits effective key at round 16 of left+right. So I am storing this 16 round keys to use this in the decryption algorithm in reverse direction.

5 Encryption & Decryption Function

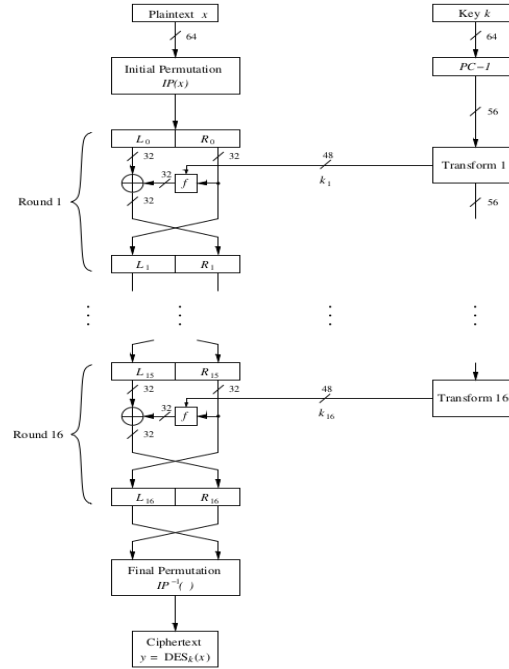


Figure 10: DES Encryption(source:Understading Cryptography: Paar,Pelzl)

Now I have all the required function to construct DES encryption function. So by following the figure 10 I ended up with whole DES encryption. In the decryption function I used the keys from the reverse direction. Initially I checked without any key the encryption and decryption function working or not by getting the same plaintext after encrypt and then decrypt. Then by key scheduling algorithm I got 16 round keys to apply on encryption function as well as decryption function.

```
sourav@sourav593: ~/Desktop/ISI_MTECH/3rd-Semester/implementation/f
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/final/main_codes$ gcc encrypt.c
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/final/main_codes$ ./a.out
Effective 56bit key is:
0,255,254,130,216,134,44,      ***V,
the plaintext is:               TamaLika
84,97,109,97,76,105,107,97,
encrypted text is:              **0t**
9,222,252,48,116,207,217,2,

sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/final/main_codes$ gcc decrypt.c
sourav@sourav593:~/Desktop/ISI_MTECH/3rd-Semester/implementation/final/main_codes$ ./a.out
Effective 56bit key is:
0,255,254,130,216,134,44,      ***V,
the ciphertext is:              **0t**
9,222,252,48,116,207,217,2,
decrypted text is:              TamaLika
84,97,109,97,76,105,107,97,
```

Figure 11: Encryption Decryption output of DES

6 Output Feedback Mode using DES

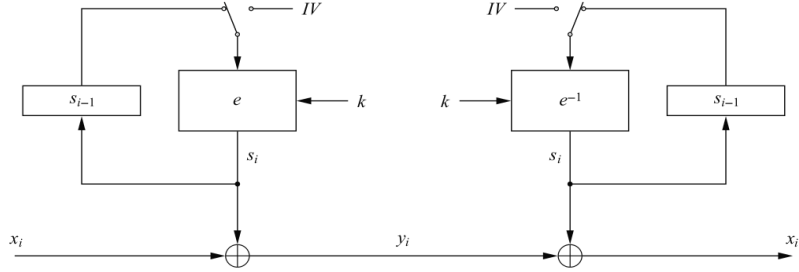


Figure 12: Encryption Decryption of OFB

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

- Encryption(first block): $y_1 = e_k(IV) \oplus x_1$
- Encryption(general block): $y_i = e_k(y_{i-1}) \oplus x_i$, $i \geq 2$
- Decryption(first block): $x_1 = e_k(IV) \oplus y_1$
- Decryption(general block): $x_i = e_k(y_{i-1}) \oplus y_i$, $i \geq 2$

Implementation Since I have already implemented the DES encryption function. I just used that function to encrypt and decryption part as stated in the definition. So taking an text file as an input now I have a ofb encryption and ofb decryption file.