Project Report for Midsem

Design & Implementation of DES and OFB mode of encryption

SOHAN DAS
CRS 2119
M.tech student
csr2119@isical.ac.in

Department of Cryptology and Security,
Indian Statistical Institute, Kolkata

September 28, 2022

Abstract

Here we will design and implement DES encryption and DES decryption . Each line at the time of implementation will be explained . At the end we will use DES as stream cipher by the OFB mode of encryption so that we can feel the real world application of DES . We will implement that OFB mode of encryption in C also and each line will also be explained . We also see the correctness of all these three schemes .

Contents

List of Figures

1 Introduction

DES stands for Data Encryption Standard is a famous block cipher. In 1972 NBS (National Bureau of Standards) which is now known as NIST (National Institute of Standards and Technology) published a solicitation for some cryptographic algorithm on Federal Register. In response IBM developed DES in 1975, based on their own cryptosystem "Lucifer".
DES is a private-key (symmetric-key) algorithm for encryption which takes inputs a 64-bit plaintext with a 56-bit key and gives output a 64-bit ciphertext. So it's called symmetric-key block cipher of block size 64-bit.

In this project we will implement the whole DES encryption method in C-programming language.

DES was adopted as standard in the sense that it can be used as a versatile building block with which a diverse set of cryptographic mechanisms can be realized. We can use DES for realizing stream ciphers uising varios kind of " modes of operations" ,like ECB, CBC, CFB, OFB, CTR etc.

Here we will implement the OFB (output feedback) mode of encryption using DES as building block in C-programming language

2 Design & Implementation of DES in C-language

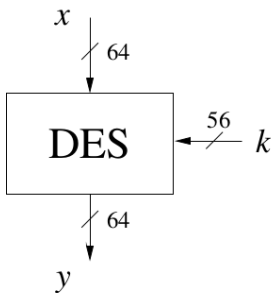


Figure 1: Basic input output overview of DES

2.1 Design of DES

We can see that it takes 64-bit plaintext x and 56-bit key k as inputs of DES anf it gives output of 64-bit ciphertext y. Now we will see the internal design of DES. There are two separate design in DES, one consists of 16-round Fiestel network and the other one is key-scheduling algorithm by which 16 numbers of round-key will be generated for using

those as input(key) of f-function. f-function is an important part of Fiestel cipher. From the below drawn figure, we can have clear concept of the design of DES. We separately design each of those Fiestel network and key-scheduling one by one and implement them using C.

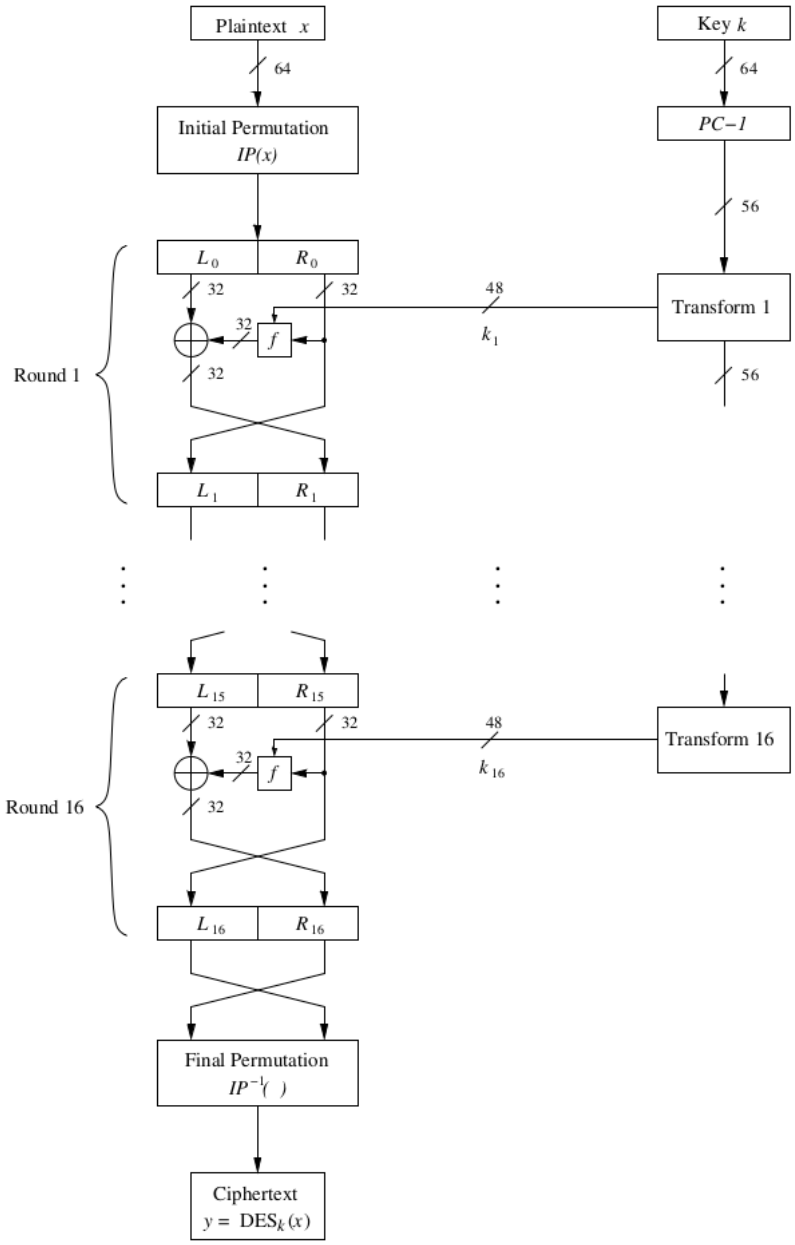


Figure 2: Design of DES

The left part of the above figure is known as 16-round Feistel network and the right part of the above figure is known as 16-round key-scheduling algorithm.

2.2 Implementation of DES

Now we will implement the Fiestel network first and then will implement the key-scheduling algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int **IP, **InvIP;
int **E;                    // Expansion matrix
unsigned char **L, **R;     // for L0,L1,...,L16 and R0,R1,... ,R16
unsigned char **ER;         // after applying Expansion() of R0.R1,....,R16
unsigned char **K;          /* K[0] will be the 64-bit key but rest of 16s will
    be round keys */
unsigned char *C, *D;
unsigned char *ciphertxt;
```

In the above code snippet we can see that we have included all required header files and declared some global variables/pointers.

1. **int **IP :** It is a double pointer which is used for storing the initial permutation matrix of size (8*8)
2. **int **InvIP :** It is also a double pointer which is used for storing matrix of inverse permutation of size (8*8)

Rests will be discused later whenever will be required.
Now we will store the initial permutation matrix as follows :

2.2.1 Initial permutation IP

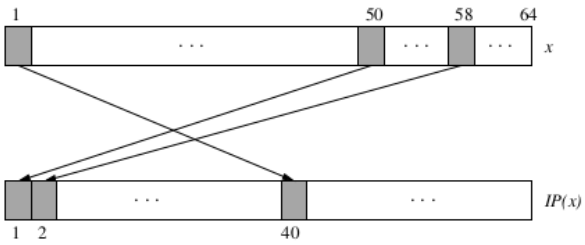


Figure 3: Initial Permutation

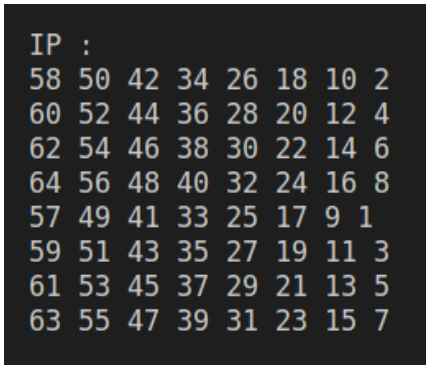


Figure 4: IPmatrix

```c
int main()
{
    /* Storing values of Initial Permutation IP */
    int i, j;
    IP = (int **)malloc(8 * sizeof(int *));
    for (i = 0; i < 8; i++)
        IP[i] = (int *)malloc(8 * sizeof(int));
    int temp_IP = 0;
    for (j = 7; j >= 0; j--)
    {
        for (i = 0; i < 4; i++)
        {
            temp_IP += 2;
            IP[i][j] = temp_IP;
        }
    }
    temp_IP = -1;
    for (j = 7; j >= 0; j--)
    {
        for (i = 4; i < 8; i++)
        {
            temp_IP += 2;
            IP[i][j] = temp_IP;
        }
    }
     /* Printing IP : */
    printf("\nIP :\n");
    for (i = 0; i < 8; i++){
        for (j = 0; j < 8; j++)
            printf("%d ", IP[i][j]);
        printf("\n");
    }
```

Here we took double pointer to store the IP matrix.So we use malloc to create space for (8*8) matrix. We noticed a beautiful pattern in that matrix so instead of putting the

values of it manually we use some formula which is quite clear from the code.
We will follow the Figure2 for implementing Fiestel structure. Now we need a plaintext and initial permutation IP wil be applied on it.

```
    // unsigned char Ptxt[8] = "SohanDas";
    unsigned char Ptxt[8] = {'S', 'o', 'h', 'a', 'n', 'D', 'a', 's'};

    unsigned char IP_Ptxt[8];
    Apply_Per_PI(Ptxt, IP_Ptxt, IP, 8, 8); /* Applying IP on Plaintext */
```

We took plaintext as a one dimension array of **unsigned char** of length 8 named a s **Ptxt[8]**, since size of unsigned char is 1 byte i,e 8-bit. We know that the input plaintext of DES shold be of 64-bit so we took 8 number of **unsigned char** each of which is of 8-bit. Now we will apply **IP** on **Ptxt** by simply applying the function **Apply_Per_PI** on it.
We now implement the above function **Apply_Per_PI**. We have to declare it first and then have to define it.

```
void Apply_Per_PI(unsigned char *, unsigned char *, int **, int, int);
```

This is the declaration part on the top of the **int main()**, whose arguments are **(unsigned char *pl, unsigned char *P_pl, int **PI, int row, int column)**.

1. **unsigned char *pl :** It takes the address of the first character pointing by pl in this way we can directly access or modify that pointed array easily. It's the array on which permutation PI will be applied. So we pass **Ptxt** in place of this argument.
2. **unsigned char *P_pl :** It's simply take the output address so that after applying permutation **PI** output is stored at the address pointed by the **unsigned char** pointer Ppl .
3. **int **PI :** We have to pass here the address the permutation PI. In this case we will pass the address of initial permutation.
4. **int row :** It's the row size of the permutation PI or we can say it the length of output P_pl, here it is 8.
5. **int column :** It's the column size of the permutation PI or we can say it the length of input pl, here it is 8.

```
    void Apply_Per_PI(unsigned char *pl, unsigned char *P_pl, int **PI, int
        row, int column)
{
    int i, j, k, m, n;
    unsigned char b;
    for (i = 0; i < row; i++)
    {
        // printf("%d. \n",i);
        P_pl[i] = 0x00;
        for (j = 0; j < column; j++)
        {
```

```
11              k = PI[i][j] - 1;
12              m = k / 8;
13              n = 7 - k % 8;
14              b = (pl[m] >> n) & 1;
15              P_pl[i] = P_pl[i] | (b << (7 - j));
16          }
17          // printf(" \n");
18      }
19  }
```

1. **line 3& 4 :** local variables.
2. **line 5 :** **for loop i** for accessing each character in P_pl
3. **line 8 :** Initialization of each charcter of P_pl to 0x00 i.e 0000 0000.
4. **line 9 :** **for loop j** for accessing each bit position in every character.
5. **linee 11 :** Since permutation matrices in DES starts from 1 instead of 0 but indexing in array starts from 0, so we negate by 1.
6. **line 12 :** **m** finds the index of the character in pl i.e. from which character we can get that bit position specified by the $(i,j)^{th}$ entry of the permutation matrix.
7. **line 13 :** **n** finds the bit position of the **pl[m]**
8. **line 14 :** **b** stores the bit value .
9. **line 15 :** Put the extracted bit value to correct place of **P_pl**.

In this way we have implemented **Apply_Per_PI** function and thus we get **IP_Ptxt** after applying Initial permutation **IP** on **Ptxt**.
Now we make 2halves of this IP_Ptxt, L_0 and R_0 , each of which is of 32bit length. Then we use it in Fiestel network. Define **L_i** and **R_i** (for i = 0,1,. . . ,16) as foloows :

```
1 unsigned char **L, **R; // for L0,L1,...,L16 and R0,R1,... ,R16
```

It is defined on the top of the **int main()** . SInce take these as global pointers so that we can use it different function easily.

```
1 /* Declaring L[0],L[1],....L[16] and R[0],R[1],...,R[16] , each are of 32 bit
      length, so we take them as character array of size 4, since 4*8=32 */
2 L = (unsigned char **)malloc(17 * sizeof(unsigned char *));
3 R = (unsigned char **)malloc(17 * sizeof(unsigned char *));
4 for (i = 0; i <= 16; i++)
5 {
6     L[i] = (unsigned char *)calloc(4, sizeof(unsigned char));
7     R[i] = (unsigned char *)calloc(4, sizeof(unsigned char));
8 }
```

Each **L_i** and **R_i** popints to a 32bit length **unsigned char** array.

2.2.2 Fiestel cipher

We will now make Fiestel ciper We will implement one round Fiestel cipher and using for

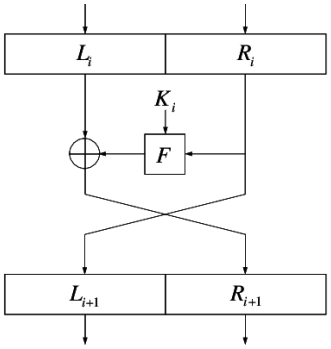


Figure 5: Fiestel-cipher

loop we will run 16 round Fiestel cipher to make the DES. For the first round we need inputs **L[0]** and **R[0]** and round key **K[1]**, which will be obtained from key-scheduling algorithm later. We will now assume that there is no key but later we will insert the proper key in the proper place.
Formula :

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

, for i = 1,2,. . . ,16
So we have to implement **f function** first.

2.2.3 f- function :

So our task is to make **L_0** and **R_0**

```
/* Initialization of L[0] and R[0] */
// printf("\n Round : 0\n\n L[0]  R[0]\n  ----   -----\n");
for (i = 0; i < 4; i++)
{
    L[0][i] = IP_Ptxt[i];
    R[0][i] = IP_Ptxt[4 + i];

    // printf("  %x     %x\n",L[0][i],R[0][i]);
}
```

First 4 character is taken in **L_0** and last 4 cxharacter is taken in **R_0**
We have to apply **f- function** on **R_0**. Firstly we have to expand 32-bit **R_0** to 48-bit **ER_0** (say) by applying expansion matrix **E**. Lets declare **ER_i** (for i = 0,1,2,. . . ,16) and define expansion matrix **E** :
It's declared on the top of **int main()** as :

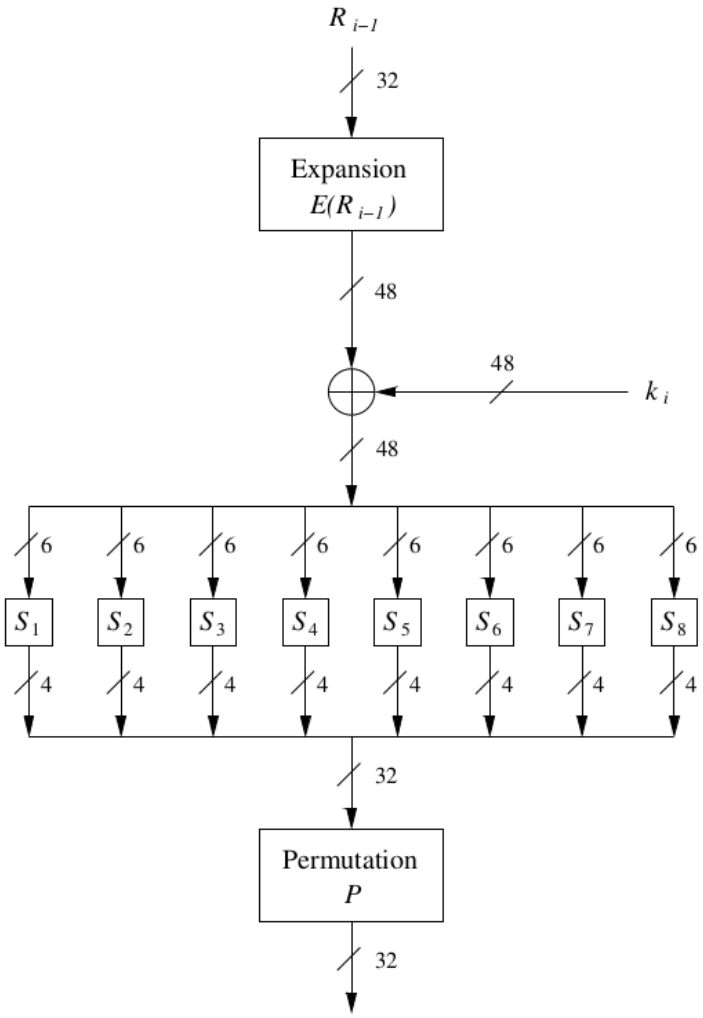


Figure 6: f-function

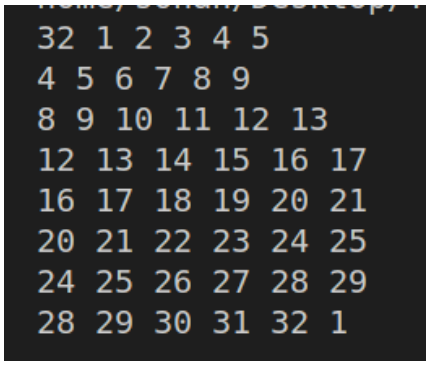
```
int **E;              // Expansion matrix
unsigned char **ER;   // after applying Expansion() of R0.R1,....,R16
```

It's written in the **int main()**.

```
/* Expansion matrix */
E = (int **)malloc(8 * sizeof(int *));
for (i = 0; i < 8; i++)
    E[i] = (int *)malloc(6 * sizeof(int));

E[0][0] = 0;
for (i = 1; i < 8; i++)
    E[i][0] = E[i - 1][0] + 4;
for (i = 0; i < 8; i++)
    for (j = 1; j < 6; j++)
        E[i][j] = E[i][j - 1] + 1;
E[0][0] = 32;
E[7][5] = 1;
```

We found some pattern in E so we didn't store E just by putting values. See the code carefully you can understood the pattern clearly.



(a) E-Matrix.

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

(b) E-Matrix.

Figure 7: E-mtrix

Formation of **ER_i**'s (for i = 1,2,...,16)

```
/* ER means expanded R from 32-bit to 48-bit */
ER = (unsigned char **)malloc(17 * sizeof(unsigned char *));
for (i = 0; i <= 16; i++)
{
    ER[i] = (unsigned char *)calloc(6, sizeof(unsigned char));
}
```

Now as per Figure6 we will apply expand **R_0** to **ER_0** by using expansion matrix **E** as follows :

Declaration of Exapnsion() :

```
    void Expansion(unsigned char *, unsigned char *, int **);
```

Defining Expansion() :

```
    void Expansion(unsigned char *R, unsigned char *MR, int **M)
{
    int i, j, k, m, n, total = -1;
    unsigned char b;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 6; j++)
        {
            k = M[i][j] - 1; /* Similar to Apply_Per_P() */
            m = k / 8;
            n = 7 - k % 8; /* Working Correctly */
            b = (R[m] >> n) & 1;

            total += 1;
            m = total / 8;
            n = 7 - total % 8;
            MR[m] = MR[m] | (b << n);
        }
    }
}
```

Arguments :(unsigned char *R, unsigned char *MR, int **M)

1. **unsigned char *R :** It takes the address of the 32bit long R_i s to whoom it will expand.
2. **unsigned char *MR :** In this address we will put the expanded 48-bit long bit-string (here we store 48-bit into 6 unsigned character).
3. **int **M :** It stores the address of expansion matrix E (actually E was taken as double pointer).

Here we are not going to describe the internal functionalities of Expansion(), since it's quite similar with **Apply_Per_PI()** in page no. 5 - 6.

Where we have applied this expansion E ?
We used it in **Fiestel()** function :

```
    void Feistel(int t)
{

    int i, j;
    Expansion(R[t - 1], ER[t - 1], E);
```

It will be discussed some later.
48-bit long **ER_(i-1)**s will be **XOR**ed with the 48-bit long round-key **K_i**'s (for i = 1,2,. . . ,16).
Below I am inserting the whole **Fiestel()** function C-code and will explain it line by line :

```
    void Feistel(int t)
{

    int i, j;
    Expansion(R[t - 1], ER[t - 1], E); // Expansion of R[t-1] is ER[t-1]
    /* do XOR of ER[t-1] and round_Key[t] and then feed it to S boxes */

    round_Key(t); // round key K[t] generated and now it can be used
    printf("\n Round key %d : ", t);
    for (i = 0; i < 6; i++)
        printf(" %x", K[t][i]);
    printf("\n");

    unsigned char *XORed_Key_ER;
    XORed_Key_ER = (unsigned char *)calloc(6, sizeof(unsigned char));
    XOR(ER[t - 1], K[t], XORed_Key_ER, 6);

    /* Break the ER character array of length 6 in another acharacter array of
        length of 8 */

    unsigned char *inputofS;
    inputofS = (unsigned char *)calloc(8, sizeof(unsigned char));
    makeInputS(XORed_Key_ER, inputofS); // this will make 6 bit input for S
        boxes as inputofS

    /* output of S box */
    unsigned char *output_S = calloc(8, sizeof(unsigned char));
    Apply_Sbox(inputofS[0], &output_S[0], S1);
    // printf("%x",output_S[0]);
    Apply_Sbox(inputofS[1], &output_S[1], S2);
    Apply_Sbox(inputofS[2], &output_S[2], S3);
    Apply_Sbox(inputofS[3], &output_S[3], S4);
    Apply_Sbox(inputofS[4], &output_S[4], S5);
    Apply_Sbox(inputofS[5], &output_S[5], S6);
    Apply_Sbox(inputofS[6], &output_S[6], S7);
```

```
34      Apply_Sbox(inputofS[7], &output_S[7], S8);
35
36      /* modify output of S boxes i.e from these 8 output_S's (of 4-bit), make 4
            8-bit blocks(of unsigned char ) */
37      unsigned char *mod_out_S = calloc(4, sizeof(unsigned char));
38      modify_Sout(output_S, mod_out_S);
39
40      /* Apply permutation P on modified output of S box */
41      unsigned char *pre_XOR_L = calloc(4, sizeof(unsigned char));
42      Apply_P_in_f(mod_out_S, pre_XOR_L);
43
44      unsigned char *post_XOR_L = calloc(4, sizeof(unsigned char));
45      XOR(pre_XOR_L, L[t - 1], post_XOR_L, 4);
46
47      /* Final swaping */
48      printf("\n  L[%d]   R[%d]\n  ----    -----\n", t, t);
49      for (int k = 0; k < 4; k++)
50      {
51          R[t][k] = post_XOR_L[k];
52          L[t][k] = R[t - 1][k];
53          printf("   %x      %x\n", L[t][k], R[t][k]);
54      }
55  }
```

Fiestel(t) is taking only one input **t** and that is the round number, because it is applied sequentially in DES for 16 rounds from rounsd-1 to round-16. At round (t-1) the **L_(t-1)** and **R_(t-1)** modified to **L_(t)** and **R_(t)** using the round key **K_t** respectively, for every t = 1,2,...,16 and then **L_(t)** and **R_(t)** will be used for the next round.

1. **line 8 :** round-key K_t is generated using the function round_Key(t). We will discuss about it later in key scheduling algorithm part.
2. **line 9 to 13 :** Just printing the round key **K_t**
3. **line 14 :** Defined local character pointer **XORed_Key_ER**
   - **XORed_Key_ER** for storing 48-bit long (6 unsigned chracter) **XOR**ed of **ER_(t-1)** and **K_t** using **XOR()** function.

   - **Explanation of XOR() :** Declaration is done on the top of the **int main()**

```
1  void XOR(unsigned char *, unsigned char *, unsigned char *, int); /*
       (input1, input2, output, size) */
```

Definition of XOR() - function

```
1  void XOR(unsigned char *inp_1, unsigned char *inp_2, unsigned char
       *opt, int size) /* (input1, input2, output, size) */
2  {
3      int i;
```

```
    for (i = 0; i < size; i++)
    {
        opt[i] = inp_1[i] ^ inp_2[i];
    }
}
```

- It takes two inputs **input1** and **input2** and size of those inputs **size** and XORed them and put the XORed value in the **output** of same size.

NOTE : Now we have to make inputs of **S-boxes** from the 48-bit (consists of 6 unsigned characters) long XORed_Key_ER. There are total 8 S-boxes namely S1,S2,S3,. . . ,S8. Inputs of S-boxes are of 6-bit length. So we have declare all those S boxes and for each S boxes we have to prepare their inputs. We will take 8 unsigned characters and the least significant 6-bits will be the input of a S-box. For 8 S-boxes we will define 8 inputs namely inputofS

4. **line 20 :** Declaration of local pointer inputofS for storing 6-bit long input of s-box.
5. **line 21 :** Using calloc we initialize all the 8 inputs to 0.
6. **line 22 :** **makeInputS()** will make 8 6-bit long inputofS[i]'s.
   - Declaration of **makeInputS()** : takes two arguments, first one is XORed_Key_ER consists of 6 unsigned character and second one is the inputofS consists of 8 unsigned character

```
void makeInputS(unsigned char *, unsigned char *);
```

   - Defining the function **makeInputS()** :

```
void makeInputS(unsigned char *E_R, unsigned char *inpt_S)
{
    inpt_S[0] = 0x00 | ((E_R[0] & 0xfc) >> 2); // 0xfc = 1111 1100
    inpt_S[1] = 0x00 | ((E_R[0] & 0x03) << 4); // 0x03 = 0000 0011
    inpt_S[1] = inpt_S[1] | ((E_R[1] & 0xf0) >> 4); // 0xf0 = 1111
        0000
    inpt_S[2] = 0x00 | ((E_R[1] & 0x0f) << 2); // 0x0f = 0000 1111
    inpt_S[2] = inpt_S[2] | ((E_R[2] & 0xc0) >> 6); // 0xc0 = 1100
        0000
    inpt_S[3] = 0x00 | ((E_R[2] & 0x3f));      // 0x3f = 0011 1111

    inpt_S[4] = 0x00 | ((E_R[3] & 0xfc) >> 2); // 0xfc = 1111 1100
    inpt_S[5] = 0x00 | ((E_R[3] & 0x03) << 4); // 0x03 = 0000 0011
    inpt_S[5] = inpt_S[1] | ((E_R[4] & 0xf0) >> 4); // 0xf0 = 1111
        0000
    inpt_S[6] = 0x00 | ((E_R[4] & 0x0f) << 2); // 0x0f = 0000 1111
    inpt_S[6] = inpt_S[6] | ((E_R[5] & 0xc0) >> 6); // 0xc0 = 1100
        0000
    inpt_S[7] = 0x00 | ((E_R[5] & 0x3f));      // 0x3f = 0011 1111
}
```

- The above code is quite clear. Taking most significant 6-bit of the E_R[0] and puting it in the least significant 6-bit of inpt_S[0]. Then take the remaining least significant 2-bit of E_R[0], put them into 5-6 th bit position of inpt_S[1] and then take the most significant 4-bits of E_R[1] ,put them in least significant 4-bits of inpt_S[1],. . . , proceeds in this way.
- Since the definition of all the S-boxes are similar, so I am showing only the S-box **S1**.
- **Construction of S-box S1 :**

```
/* S_box_1 */
int S1[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}};
```

NOTE : We defined it on the top of the **int main()**. On 6-bit input **inputofS[1]** S1-box return 4-bit output

7. **line 24-25 :** A local pointer named **output_S** is defined which will point to a sequential 8-byte of memory and also initialized to 0 by using **calloc()**.
8. **line 26 - 34 :** We applied S-boxe S[i] on input inputofS[i-1] and get output output_S[i-1], for i = 1,2,. . . ,8.

Discuss about the function Apply_Sbox() :

- Declaration of function **Apply_Sbox()** :

```
void Apply_Sbox(unsigned char, unsigned char *, int (*S)[16]);
```

- Definition of function **Apply_Sbox()** :

```
void Apply_Sbox(unsigned char in, unsigned char *out, int (*S)[16])
    // Working Fine
{
    *out = 0x00;
    unsigned char m, n;                 // m - row, n- column
    n = (in & 0x1e) >> 1;               // 0x1e = 0001 1110
    m = (in & 0x01) | ((in & 0x20) >> 4); // 0x01 = 0000 0001, 0x20
        =0010 0000
    *out = S[m][n];
    // printf("%x",*out);
}
```

- **How does S-box gives output ?**
  It takes intput a 6-bit string. Let's say the input is 110101, then convert the middle 4 bit string into decimal and here in this case it is $(1010)_2 = (10)_{10}$. From left side the first bit value is 1 and the last bit value is 1 so consider 11 (most significant bit—— least significant-bit) = $(3)_{10}$ .Therefor the binary value of $(3,10)^{th}$ posion of S box will be the ouput for that S box with that input (here, 110101)

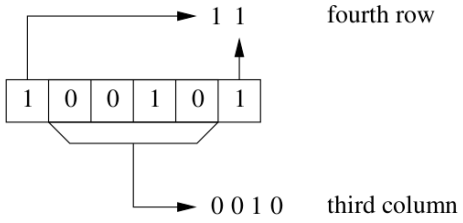


Figure 8: output of S box

9. **line 38 :** Modifying the ouput_S[i]'s to mod_out_S[i]'s
   - output_S points to a memory location for 8-sequential unsigned characters whose only 4 least significant bits are considerable for further use. So we will modify it and put these 32-bits into 4 unsigned characters namely mod_out_S using **modify_Sout()** function.
   - Declaration of **modify_Sout()** function :

```
1 void modify_Sout(unsigned char *, unsigned char *);
```

   - Declaration of **modify_Sout()** function :

```
1 void modify_Sout(unsigned char *in_4bit_8char, unsigned char
      *out_8bit_4char)
2 {
3     int i;
4     unsigned char b = 0x00; // working fine
5     for (i = 0; i < 4; i++)
6     {
7         out_8bit_4char[i] = 0x00;           // initialization (not
              needed)
8         b = (in_4bit_8char[2 * i] & 0x0f) << 4; // 0x0f = 0000 1111
9         out_8bit_4char[i] = b | ((in_4bit_8char[2 * i + 1]) & 0x0f);
10    }
11 }
```

   - It's taking 4-bit of 8 characters input and gives output of 4 unsigned characters each are of 8-bit long. So the bit values of
     - output_S[0]—output_S[1] is stored at mod_out_S[0]
     - output_S[2]—output_S[3] is stored at mod_out_S[1]
     - output_S[4]—output_S[5] is stored at mod_out_S[2]
     - output_S[6]—output_S[7] is stored at mod_out_S[3]
   - least significant 4-bit of S[0] is stored at most significant 4- bit of mod_out_S[0] and least significant 4-bit of S[1] is stored at least significant 4- bit of mod_out_S[0]. Similarly for others.
10. **line 40 - 42 :** creating local pointer **pre_XOR_L** for storing 4 unsigned chracters after applying permutation **P** in **f**-function on mod_out_S[0]
    - Declaration of **void Apply_P_in_f()** :

```
1 void Apply_P_in_f(unsigned char *, unsigned char *);
```

- Declaration of **void Apply_P_in_f()** :

```
1  void Apply_P_in_f(unsigned char *before_P, unsigned char *after_P)
2  {
3      int i, j, k, m, n;
4      unsigned char b;
5      for (i = 0; i < 4; i++)
6      {
7          // printf("%d. ",i);
8          after_P[i] = 0x00;
9          for (j = 0; j < 8; j++)
10         {
11             k = P_in_f[i][j] - 1; /* Since in P there are elements
                   between 1 and 32 not from 0 to 31 */
12             m = k / 8;          // row or block
13             n = 7 - k % 8;
14             b = (before_P[m] >> n) & 1; // Extracting the bit value
                   0/1
15             // printf(" %d. m = %d, n = %d, b = %x\n",j,m,n,b);
16             after_P[i] = after_P[i] | (b << (7 - j));
17         }
18         // printf("\n");
19     }
20 }
```

- Quite similar with **Apply_Per_PI()** in page 5 -6, so not going to explain it again.

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 9: Permutation P

- It outputs pre_XOR_L ,pointing to an array of 4 unsigned chars. Quite clear from it's name that we will XOR it with L[t-1] (t-round Fiestel cipher)

11. **line 44 :** Creating local memory space for storing the output of **XOR()**function and initializing it with 0s.
12. **line 45 :** **XOR**ing pre_XOR_L with L[t - 1] and storing the XORed output in post_XOR_L[i]'s. XOR is explained before.
13. **line 47 - 55 :** Final swaping in one round is done here, since,

$$L_t = R_{t-1} \ ,$$
$$R_t = L_{t-1} \oplus f_{K_t}(R_{t-1}) \ ,$$

for all t = 1,2,. . . ,16.
Swaping is not exactly done here . We are just storing 32-bit string of **R[t-1]** in **L[t]** and storing 32-bit string **post_XOR_L[k]**

2.2.4 Key Scheduling

we are first importing the part of key scheduling lying in **int main()** below , then will describe line by line :

```
/* Key Scheduling */
/* We have to generate 16 round keys of 48-bit from a 64-bit given key and
   then do XOR with respective ERs. We will make these round keys as global
   for further checking at the time of reverse key scheduling */
    char testkey[8] = "ThankGod";
    K = (unsigned char **)malloc(17 * sizeof(unsigned char *));
    K[0] = (unsigned char *)malloc(8 * sizeof(unsigned char));
    for (i = 0; i < 8; i++)
    {
        K[0][i] = testkey[i]; /* Just for now to check correctness */
        //  printf("\n %x",K[0][i]);
    }
    for (i = 1; i <= 16; i++)
    {
        K[i] = (unsigned char *)calloc(6, sizeof(unsigned char)); /* same as
            ER[i] as we will XOR it with that */
    }

    unsigned char *afterPC_1;
    afterPC_1 = (unsigned char *)malloc(7 * sizeof(unsigned char)); /* 56-bit
        output of PC_1 will be stored in 7 unsigned char */
    /* Due to type casting we have to copy PC_1[][] 2-D array data to a
        double-pointer PC1 */
    PC1 = (int **)malloc(7 * sizeof(int *));
    for (i = 0; i < 7; i++)
    {
        PC1[i] = (int *)malloc(8 * sizeof(int));
        // printf("\n");
        for (j = 0; j < 8; j++)
        {
            PC1[i][j] = PC_1[i][j];
            // printf(" %d",PC1[i][j]);
        }
    }

    Apply_Per_PI(K[0], afterPC_1, PC1, 7, 8);

    /* Make C and D , two 28-bit of 56-bit afterPC_1 */
    C = (unsigned char *)calloc(7, sizeof(unsigned char));
    D = (unsigned char *)calloc(7, sizeof(unsigned char));

    trans56to28(afterPC_1, C, D);
    /* Due to type casting we have to copy PC_2[][] 2-D array data to a
        double-pointer PC2 */
```

```
39    PC2 = (int **)malloc(7 * sizeof(int *));
40    for (i = 0; i < 6; i++)
41    {
42        PC2[i] = (int *)malloc(8 * sizeof(int));
43        // printf("\n");
44        for (j = 0; j < 8; j++)
45        {
46            PC2[i][j] = PC_2[i][j];
47            // printf(" %d",PC2[i][j]);
48        }
49    }
```

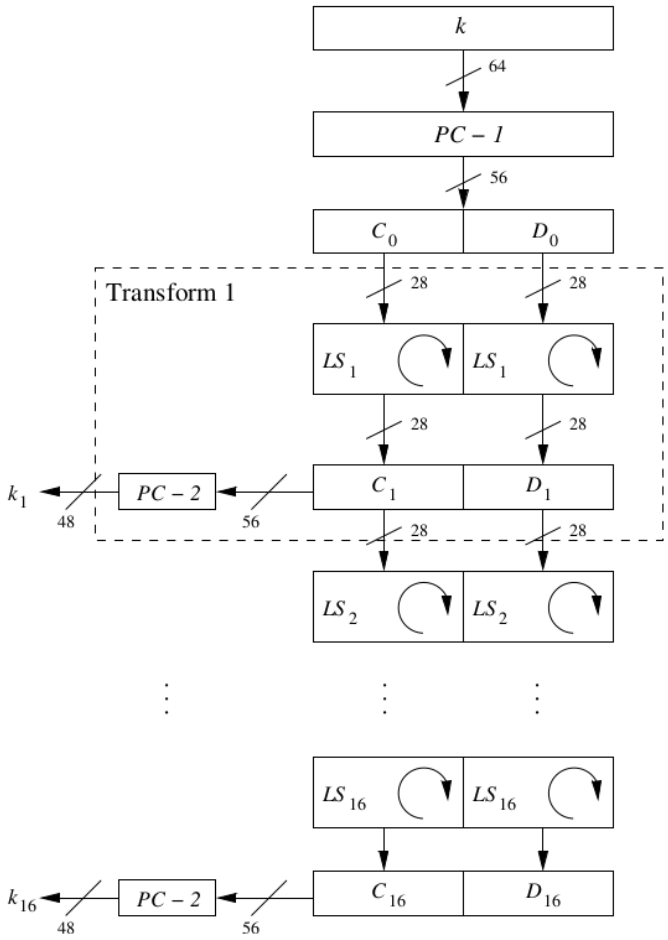


Figure 10: Key Scheduling algorithm

1. **line 3 :** Taking an character array of 8 byte, named **testkey[8]**
2. **line 4 :** Constructing 17 keys among which last 16 are roundkeys and the first one the K[0] is the given 64-bit(8 byte) key for DES.
3. **line 5 :** K[0] will contain 64-bit key so creating 8 byte space for storing the key.
4. **line 6 -10 :** Storing the key provided in **testkey** into K[0], so K[0] contains the 64-bit string values of **testkey**.

```c
/* PC_1 permutation for key scheduling */
int **PC1; /* due to some function implementation we use it same as PC_1 */
int PC_1[7][8] = {
    {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}};
```

Figure 11: PC1

5. **line 11 - 14 :** Creating spaces for storing 48-bit round-keys K[i]'s for i = 1,2,. . . ,16 Also those are initialized to 0, as we use **calloc()**
6. **line 16 - 18 :** Creating memory space for storing 56-bit key obtained from 64-bit K[0] by applying transformation PC_1
7. **line 19 - 29 :** We have defined PC_1 on the top of the **int main()** but for applying PC_1 on K[0] we want to use predefined **Apply_Per_PI()** function, so for typecasting issue we copy the 2-D array **PC_1** to double pointer (defined globally) PC1 .
8. **line 31 :** We applied PC1 on K[0] and get 56-bit (7 byte i.e. 7 unsigned chars) **afterPC_1**
9. **line 32 - 35 :** Dividing 56-bit key **afterPC_1** into two halves C and D (say). C will contain leftmost 28 bit values and rightmost 28-bit values will be contained by D
10. **line 34 - 35 :** Declaring and preparing memory spaces for C and D also we initialized them to 0.
11. **line 37 :** By applying **trans56to28()** function we divided 56-bit key **afterPC_1** into two 28-bit strings **C** and **D**, each of which contains 7 unsigned chars whose least significal 4-bit will be of our interest. (4*7 =28)
12. **line 38 - 49:** Same reason as PC1, due some typecasting we had to copy externally defined **PC_2** into **PC2**.

```c
/* PC_2 permutation for key scheduling */
int **PC2; /* due to some function
implementation we use it same as PC_1 */
int PC_2[6][8] = {
    {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}};
```

(a) Transformation PC2.

$PC-2$							
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

(b) Transformation PC2

Figure 12: Transformation PC2

We can remember in **Fiestel()** we said that we will discuss about the **round_Key()** later. Now we will discuss about it.

```
1  void round_Key(int t)
2  {
3      int i, j;
4      /* Left shifting by one bit */
5      if ((t != 1) && (t != 2) && (t != 9) && (t != 16))
6          leftshift(C, D); /* Since for round 1,2,9,16 there is only one left
               shift in DES */
7      leftshift(C, D);   /* At t-round key generation C[t-1] left-shifted to
           C[t] and D[t-1] left shifted to D[t] */
8      unsigned char *CD;
9      CD = (unsigned char *)calloc(7, sizeof(unsigned char));
10     /* We make a 56-bit string of 7 unsigned char's (8-bits each) by
           concatenating Cand D */
11     join_CD(C, D, CD);
12
13     /* Apply permutation PC_2 on this 56-bit CD (7 unsigned char of 8-bits) */
14     Apply_Per_PI(CD, K[t], PC2, 6, 8);
15 }
```

1. **line 4 - 7 :** We are performing left shift on each of C and D both by applying a function **leftshift()** . For rounds 1,2,9 and 16 only one left shift is applied and except these rounds 2 left shifts are applied.
2. **line 9 :** Create a memory space for storing 56-bit concatenated string of C and D repectively.
3. **line 11 :** Concatenate C and D by using **join()** function and store the output at 56-bit long CD
4. **line 14 :** We applied transformation PC2 on 56-bit CD by applying **Apply_Per_PI()** and got the output, 48-bit long round-key K[t] at the round t in Feistel(t)

In int main() finally we run 16 round Fiestel() and at last by swapping L[16] and R[16] we get the **pre_cipher** on which we apply the inverse permutation **InvIP** of initial permutation and the we get the required DES-ciphertext as **ciphertxt**

```
1      /* 16-round Feistel network */
2      for (i = 1; i <= 16; i++)
3      {
4          printf("\n------------------------\n Round : %d\n\n", i);
5          Feistel(i);
6      }
7
8      /* Finally invIP (inverse of initial permutation) on R[16]_L[16] */
9      unsigned char *pre_cipher = calloc(8, sizeof(unsigned char));
10     for (i = 0; i < 8; i++)
11     {
```

```
12        if (i < 4)
13            pre_cipher[i] = R[16][i];
14        else
15            pre_cipher[i] = L[16][i - 4];
16        // printf(" %x",pre_cipher[i]);
17    }
18
19    ciphertxt = (unsigned char *)calloc(8, sizeof(unsigned char));
20
21    Apply_Per_PI(pre_cipher, ciphertxt, InvIP, 8, 8);
```

1. **line 1 - 6 :** Using for loop running from 1 t 16 with step 1 we are invoking **Fiestel(i)**. At each time of invokation of **Fiestel(i)** round-key **K[i]** is generated and using this round-key **K[i]** and inputs **L[i-1], R[i-1]** , **Fiestel(i)** modifies them to **L[i], R[i]** respectively.
2. **line 8 - 9 :** Creating sequential memory allocation named **pre_cipher** for storing 64-bit long string and initialized to 0 also by using **calloc()**
3. **line 10 - 17 :** The **pre_cipher** local pointer can access a sequeces of 8 unsigned characters which will be set by combining **R[16]** and **L[16]**, each of them are consisting of 4 characters . Here the order is very important i.e. the first 4 characters of **pre_cipher** are set from **R[16]** and then the last 4 characters will be set from **L[16]** in which order they are in respective 32-bit string .
4. **line 19 :** Creating a local pointer which is pointing to the first element of consecutive 8-byte memory allocation for 8 storing 8 unsigned characters. It will be the required ciphertext, that's why we put it's name **ciphertxt**.
5. **line 21 :** Apply inverse permutation **InvIP** on **pre_cipher** to obtain the **ciphertxt** by using the **Apply_Per_PI()** function with **InvIP** permutation.
   - Inverse permutation of initial permutation **IP** is denoted by **InvIP** and defined as :
   - Declaration of **InvIP** : It is defined on the top of **int main()** function as :

```
1 int **IP, **InvIP;
```

   - Definition of **InvIP** : We defined it in **int main()**. We found some good patern in this inverse permutation of **IP** , so instead of manually typing all the entries we build the logic using some for loop.

```
1  InvIP = (int **)malloc(8 * sizeof(int *));
2      for (i = 0; i < 8; i++)
3          InvIP[i] = (int *)malloc(8 * sizeof(int));
4      int temp_InvIP = 0;
5      for (j = 1; j < 8; j = j + 2)
6      {
7          for (i = 7; i >= 0; i--)
8          {
9              temp_InvIP += 1;
10             InvIP[i][j] = temp_InvIP;
```

```
        }
    }

    for (j = 0; j < 8; j = j + 2)
    {
        for (i = 7; i >= 0; i--)
        {
            temp_InvIP += 1;
            InvIP[i][j] = temp_InvIP;
        }
    }
```

Finally we get the ciphertext **ciphertxt** of the given plaintext **Ptxt** using **DES** with private key **testkey** (here, which was **"Thankgod"**).

```
 InvIP :
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
```

(a) Inverse of IP.

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

(b) Inverse of IP.

Figure 13: Inverse of IP

Now we will decorate the printing part i.e. which should be visible in terminal on executing the DES.exe corresponding to this DES.c file . It is decorated in below part of **int main()** as follows :

```
    /* ---------------------- Decoration Part---------------------- */
    printf("\n  Plaintext  :");
    for (i = 0; i < 8; i += 1)
    {
        printf(" %c ", Ptxt[i]);
    } /* Plaintext printing */
    printf("\n (hexadecimal) :");
    for (i = 0; i < 8; i += 1)
    {
        printf(" %x ", Ptxt[i]);
```

```
11      }
12      /* ----------------------------------------------*/
13      printf("\n Private Key\n (hexadecimal) :");
14      for (i = 0; i < 8; i += 1)
15      {
16          printf(" %x ", K[0][i]); /* Private Key Printing */
17      }
18      /* ---------------------------------------------- */
19      printf("\n Ciphertext\n (hexadecimal) :");
20      for (i = 0; i < 8; i++)
21      {
22          printf(" %x ", ciphertxt[i]); /* Ciphertext Printing */
23      }
24
25      return 0;
26  }
```

1. **line 2 - 6 :** Printing given pliantext **Ptxt** in **char**-form .
2. **line 7 - 11 :** Printing given pliantext **Ptxt** in **hexadecimal**-form .
3. **line 13 - 17 :** Printing given private key **K[0]** (which is same as **testkey**) in **hexadecimal**-form .
4. **line 19 - 24 :** Printing obtained ciphertext **ciphertxt** in **hexadecimal**-form .

End of DES encryption.

3 Design & Implementation of Decryption of DES in C - language

3.1 Design of Decryption of DES

Decryption of DES is same as DES except the key scheduling part. It takes 64-bit ciphertext 56-bit private key (actually here we will provide 64-bit key upon which by applying transformation **PC1** the key becomes of 56-bit) as inputs and gives output the plaintext. As we said that it's nothing but a DES except the key scheduling part. Actually we have to schedule key in reverse order so it's sometime called reverse key Scheduling. Note : We were getting ciphertext by applying **InvIP** on **R[16]**—**L[16]**. So if we consider

that ciphertext **ciphertxt** obtained from **DES** and apply initial permutation **IP** on it the we will get **R[16]**—**L[16]** . Now consider it as **L'[0]**—**R'[0]** and apply **Fiestel()** cipher with the key K[16] , then we will obtain :

$$\begin{array}{ccc} L'[0] = R[16] & \& & R'[0] = L[16] \\ L'[1] = R'[0] & \& & R'[1] = L'[0] \oplus F_{K[16]}(R'[0]) \\ L'[1] = L[16] & \& & R'[1] = R[16] \oplus F_{K[16]}(L[16]) \end{array}$$

Observe that

$$\begin{array}{ccc} L[16] = R[15] & \& & R[16] = L[15] \oplus F_{K[16]}(R[15]) \\ L'[1] = R[15] & \& & R'[1] = R[16] \oplus F_{K[16]}(L[16] \\ L'[1] = R[15] & \& & R'[1] = R[16] \oplus F_{K[16]}(R[15]) \\ L'[1] = R[15] & \& & R'[1] = L[15] \quad . \end{array}$$

In this if we apply DES on **ciphertxt** then at the end of 16 round fiestel network we will get

$$L'[16] = R[0] \qquad \& \qquad R'[16] = L[0]$$

Then by swapping them (as done in DES after 16 round of **Fiestel** network we will get L[0]—R[0] upon which applying **InvIP** we will get the original plaintext.

So, we have to think about how we can get those reverse round keys from private key K[0]. Here is the beauty of the **Key-Scheduling** algorithm . We applied left shifting on both the **C** & **D** ,each of those are 28 bit long. Now count the total number of left shift applied on both of them and that is $4 * 1 + (16 - 4) * 2 = 28$. So we applied 28 left shifts on 28 bit long string and thus it can be obsered that after all the 16 round we are basically getting the initial one . Hence

$$C[16] = C[0] \qquad \& \qquad D[16] = D[0]$$

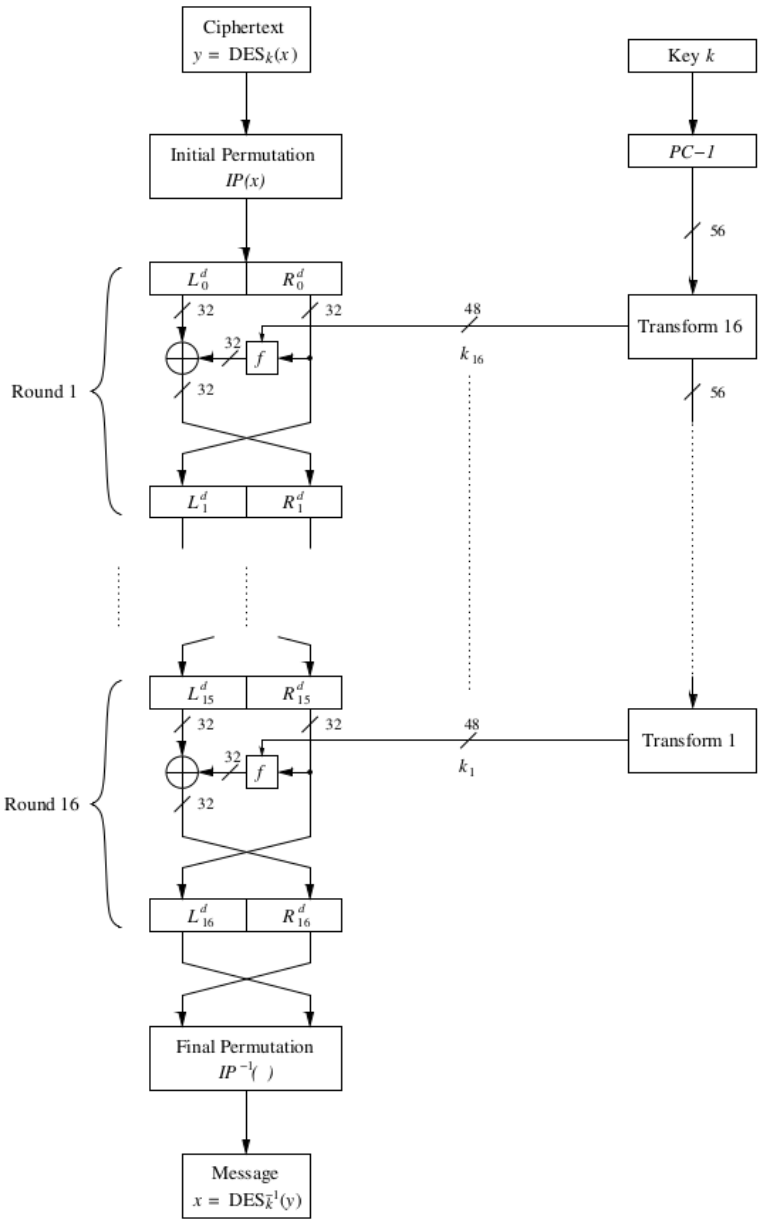


Figure 14: Design of Decryption of DES

Therefore from given **K[0]** we can get **C[16]** & **D[16]** just by applying transformation **PC1** on it and then dividing in equal half, set the first half to **C[16]** and last half to **D[16]**. So join the **C[16]**—**D[16]** and apply transformation **PC2** to get the round ket $\boldsymbol{K'[1] = K[16]}$.

Then to get $\boldsymbol{K'[2]}$ we will apply right shift once (since at the 16^{th} round of Key-Sceduling in DES we applied one left shift) on **C[16]** and also on **D[16]**, then join them and apply **PC2** on the joined **C[16]**—**D[16]** .

In this way by applying correct number of right shifts on C and D we can generate reverse round keys : $\boldsymbol{K'[t] = K[16 - t]}$,for t = 1,2,. . . ,16 sequentially.

So decryption of DES is done .

3.2 Implementation of Decryption of DES

Since all but round key generation is little different fron DES, so we will discuss only about reverse key sheduling below :
In **int main()**, we have changed **round_key()** to **Rev_round_key()** :

```
1  /* 16-round Feistel network */
2  for (i = 1; i <= 16; i++)
3  {
4      printf("\n------------------------\n Round : %d\n\n", i);
5      printf(" Reverse Round key %d : ", 17-i);
6      Rev_round_key(i); // reverse round key K[i] is generated and now it can be
           used
7      for (j = 0; j < 6; j++)
8          printf(" %x", K[i][j]);
9      printf("\n");
10     Feistel(i);
11 }
```

3.2.1 Reverse round key generation : Rev_round_key()

At each of these round i ,reverse round key **K[i]** is generated. (We denoted it by same notation as used in DES, because two files are not merged.)
Therefore we only discuss with the line no. 6 about **Rev_round_key()** function .

- **Declaration of Rev_round_key() :**

It's declared on the top of the **int main()**

```
1      void Rev_round_key(int);
```

- **Definition of Rev_round_key() :**

```
1  void Rev_round_key(int t)
2  {
3      unsigned char *CD;
4      CD = (unsigned char *)calloc(7, sizeof(unsigned char));
5      /* We make a 56-bit string of 7 unsigned char's (8-bits each) by
           concatenating Cand D */
6      join_CD(C, D, CD);
7
8      /* Apply permutation PC_2 on this 56-bit CD (7 unsigned char of 8-bits) */
9      Apply_Per_PI(CD, K[t], PC2, 6, 8);
10
```

```
11      /* Right shifting by one bit */
12
13     if((t!=1)&&(t!=8)&&(t!=15))
14         rightshift(C,D);  /* Since for round 2,9,16 there is only one right
               shift in decDES */
15
16     rightshift(C,D);/* At t-round key generation C[t-1] right-shifted to C[t]
           and D[t-1] right shifted to D[t] */
17 }
```

1. **line 3 :** Declaring local character type pointer CD .
2. **line 4 :** Allocating memory space for storing 7 unsigned characters and initializing those memory spaces with 0 .
3. **line 6 :** **join_CD()** takes input **C** & **D** and concatenate their values and put those in the above created 7 unsigned characters pointed by CD . We previously discussed line by line about this **join_CD()** function in DES encryption part.
4. **line 9 :** Using **Apply_Per_PI()** function we are transforming **CD** into reverse round key **K**[t] with the help of **(6 * 8)** transformation matrix **PC2** .
5. **line 13 - 14 :** Conditioning in which round we have to apply double right shift and in which round we have to apply single right shift.

Note : At round 1,2,9,16 we applied left shift only once in DES encryption. So we will apply right shift only once at rounds 2, 9 and 16 (corresponding to one left shift at rounds 16, 9, 2 respectively in case of DES i.e. in case of key scheduling algorithm of DES), since the last right shift is not needed as we are not going to extract reverse round key from **C**[**0**] & **D**[**0**] (index 0 w.r.t. DES) . Here we applied one right shift at rounds 1, 8, 15 , after extracting Reverse round key K[t], so in actual the effect of one right shift is applied for the next round i.e. for rounds 2, 9, 16. Except these rounds we will apply right shift twice on **C** and **D** both .

6. Now we will discuss about **rightshift()** function :

   - Declaration of **rightshift()**

```
1 void rightshift(unsigned char*, unsigned char*);
```

   - Definition of **rightshift()**

```
1 void rightshift(unsigned char *C, unsigned char *D)
2 {
3     unsigned char new_bc,old_bc, new_bd, old_bd ;
4     int i;
5     old_bc = (C[6]&1)<<3;
6     old_bd = (D[6]&1)<<3;
```

```
7     for (i=0; i<7; i++ )
8     {
9         new_bc = (C[i] & 1) << 3 ;
10        C[i] = (C[i]>>1) | old_bc ;
11        old_bc = new_bc;
12
13
14        new_bd = (D[i] & 1) << 3 ;
15        D[i] = (D[i]>>1) | old_bd ;
16        old_bd = new_bd;
17    }
18 }
```

6.1. **line 3 :** Taking some local unsigned char type variables for storing bit values. As per example **new_bc** will contain that bit value of **C[i]** which is need for next $(i+1)_{th}$ round and **old_bd** will contain that bit value of **D[i]** which is going to be used in the current i_{th} round .

6.2. **line 5 - 6 :** Initialization of **old_bc** to 1^{st} bit value of C[6] and similarly **old_bc** is initialized to 1^{st} bit value of D[6] . They are not only storing bit values they are storing bit value with correct position for use at the next time.

6.3. **line 7 - 17 :** for loop is running for i = 0,1,. . . ,6 . Consider i^{th} round, so **new_bc** contains 1^{st} bit value with 4^{th} position. Then C[i] is shifted once into right side and with it **old_bc** ,which contains the 1^{st} bit value with 4^{th} position of C[i-1], is **OR**ed . then **old_bc** is set to **new_bc**.
Similar things is happenning with D string .
In this way right shifting is happenning .

We have discussed all the required changes for implementing decryption scheme of DES enciphering and by changing those in our C-code of DES we have implemented Decryption of DES , namely **DecDES.c** . Here we are not going to bother with decoration part .

Therefore the Decryption of DES is done !

4 Output FeedBack mode of Encryption using DES

4.1 Introduction

Till now we have implemented DES encryption and it's decryption scheme also. Now the question is how can we use this encryption scheme in practical field or real field. DES does encryption of plaintext of size 64-bit, but in real field we have to encrypt large sized message. We read that it is used as a block cipher for encrypting large message in vast area. So, now we will see one of the technique called **Output FeedBack** mode of encryption (**OFB** in short), where we will see that the block cipher DES can be used as a stream cipher to encrypt a large message.
Also there are several modes of encryptions, like ECB, CBC, CTR, CFB etc.

4.2 Design of OFB

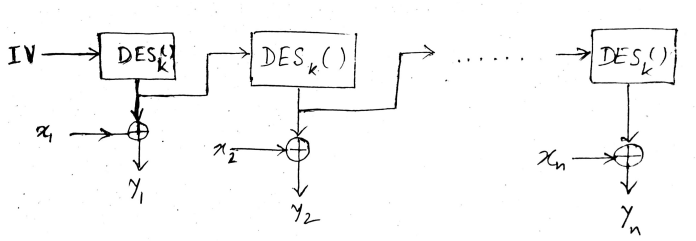


Figure 15: OFB

,where $x_1x_2x_3 \ldots x_n$ is the message text and $y_1y_2y_3 \ldots y_n$ is the ciphertext of OFB mode of encryption.

4.3 Implementation of OFB

```
int main()
{
    int i,j, m = 0;
    // unsigned char plaintxt[8] = "SohanDas";
    unsigned char plaintxt[9]; /* = {'S', 'o', 'h', 'a', 'n', 'D', 'a', 's'};
        */
    unsigned char key[8], encmsg[8], IV[8] = "hellohii";
    ciphertxt = (unsigned char *)calloc(8, sizeof(unsigned char));
    unsigned char testkey[8] = "Thankgod";
    for(i=0; i<8; i++){
        key[i] = testkey[i];
        ciphertxt[i] = 0x00;
```

```
12          // printf("IV : %x | key : %x | ciphertxt ;
                %x\n",IV[i],key[i],ciphertxt[i]);
13      }
14
15      int fd = open("msg.txt",O_RDONLY);
16      int fd_enc = open("EncryptedMsg.txt",O_WRONLY | O_CREAT | O_TRUNC, 0666);
17      while(1) {
18          for(i=0; i<8; i++){
19              plaintxt[i] = 0x00;
20              ciphertxt[i] = 0x00;
21          }
22          read(fd, plaintxt, 8*sizeof(unsigned char));
23          if(strcmp(plaintxt,ciphertxt)==0) /* while loop breaking condition */
24              break;
25          printf("\n Plaintxt : ");
26          for(i=0; i<8; i++)
27              printf(" %c",plaintxt[i]);
28          printf("\n Ciphertxt : ");
29          DES(IV, key, ciphertxt);
30          XOR(plaintxt, ciphertxt, encmsg, 8);
31          write(fd_enc, encmsg, 8*sizeof(unsigned char));
32          for(i=0; i<8; i++)
33              printf(" %c",encmsg[i]);
34          printf("\n");
35          for(i=0; i<8; i++) {
36              IV[i] = ciphertxt[i];
37          }
38      }
39      close(fd);
40      close(fd_enc);
41      return 0;
42      }
```

The above C-code is quite easy to undersatnd. We copied the whole DES encryption part in my OFB.c and then manipulating a little we successfully made **DES()** function which on 64-bit plaintext $\boldsymbol{x_1}$ and 64-bit private-key ***key*** gives the output ,the ciphertext $\boldsymbol{y_1}$.

1. **line 6 :** We have defined key[8], encmsg[8], IV[8] as for storing initial vector .
2. **line 7 :** Creating memory spaces for storing output ciphertexxt of **DES()** .
3. **line 8 :** Set testkey a 64-bit string as "Thankgod" .
4. **line 9 -13 :** Initializing **ciphertxt[8]** to all 0s and storing the values of **testkey[8]** in **key[8]** .
5. **line 15 - 16 :** Openning files **msg.txt** and **EncryptedMsg.txt** using file descriptor .
6. **line 17 :** Loop starts . Loop breaking condition is given in **line 23** . It's quite easy to understand .
7. **line 18 - 21 :** Initailization to 0s is done here .

8. **line 22 :** 8-byte of unsigned char i.e. 64-bit msg is taken and stored in plaintxt using file descriptor fd .
9. **line 23 - 24 :** while loop breaking condition, when all the plaintxt taken from file msg.txt are 0 then **strcmp()** returns 0 (since, ciphertxt is initialized to all 0), so if this is true break the while loop. So if we put 64-bit string as all 0s then the file reading will be stopped.
10. **line 25 - 28 :** Printing taken 8-byte junc of msg taken from msg.txt by palintxt .
11. **line 29 :** Apply DES() with plaintext as IV, private-key as key and gives ciphertext in ciphertxt .
12. **line 30 :** XOR the 64-bit plaintxt and 64-bit ciphertxt and XORed 64-bit string is stored in encmsg
13. **line 31 :** write the obtained encmsg into the EncryptedMsg.txt file using file descriptor .
14. **line 32 - 34 :** Printing the obtained encmsg into the terminal also.
15. **line 35 -37 :** Setting IV, to the obtained ciphertxt . This is the concept of OFB .
16. **line 38 - last :** Clossing all the file and returning 0 to int main() i.e. ending the program .

For checking correctness of OFB just change the msg.txt at line 15 by EncryptedMsg.txt after generating it from msg.txt and change the EncryptedMsg.txt by another new file checkMsg.txt . Run it an open the checkMsg.txt . It will be same as msg.txt . Therefore we can be asured that it's running fine.

5 Results

We successfully implemented DES, DecDES & OFB using DES in C-programming language . [1]

References

[1] C. P. J. Pelzl, "Understanding cryptography."