DES学习整理

参考文章

https://blog.csdn.net/qq_27570955/article/details/52442092

http://rk700.github.io/2016/10/13/des-details/

两篇文章结合起来，并且结合代码，最后通过程序的汇编代码，来深入理解DES加解密

原理介绍

DES是一种将64比特的明文加密成64比特的密文的对称密码算法，它的密钥长度是56比特。但从严格来说，DES的密钥长度是64比特，但由于每隔7比特会设置一个用于错误检查的比特，因此实质上其密钥长度是56比特。

DES是以64比特的明文为一个单位来进行加密的。这一个单位称为分组，一般来说以分组为单位进行处理的密码算法称为分组密码。

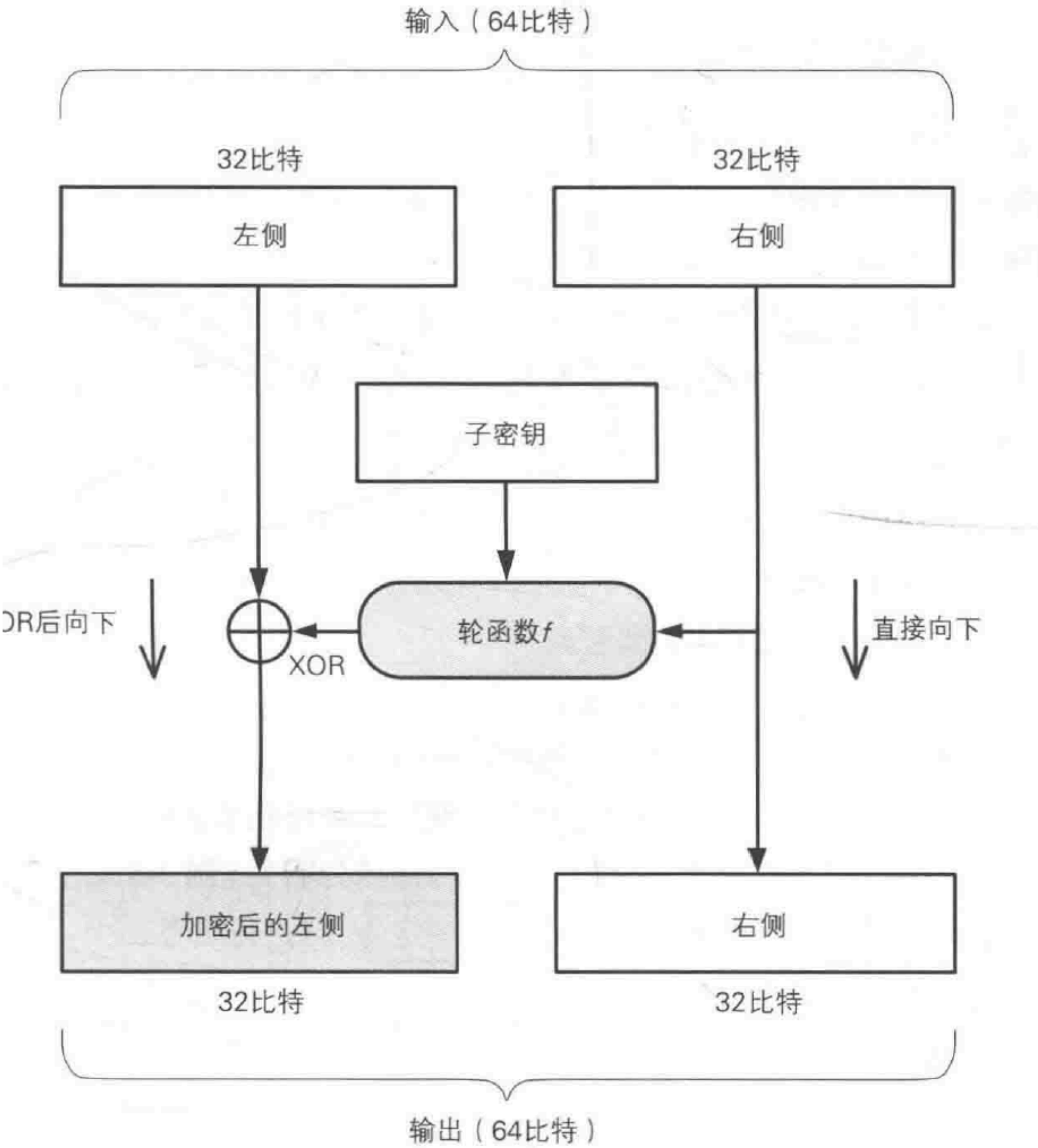
DES每次只能加密64bit的数据，如果要加密的明文比较长，就需要对DES加密进行迭代，而迭代的具体方式，就称为模式。

参考文章：1，2

加密过程

DES的结构也称为Feistel网络

在**Feistel网络**中，加密的各个步骤称为**轮**，整个加密过程就是进行若干次轮的循环



每一轮都需要使用一个不同的子密钥

轮函数的作用是根据右侧和子密钥生成对左侧进行加密的比特序列，它是密码系统的核心。

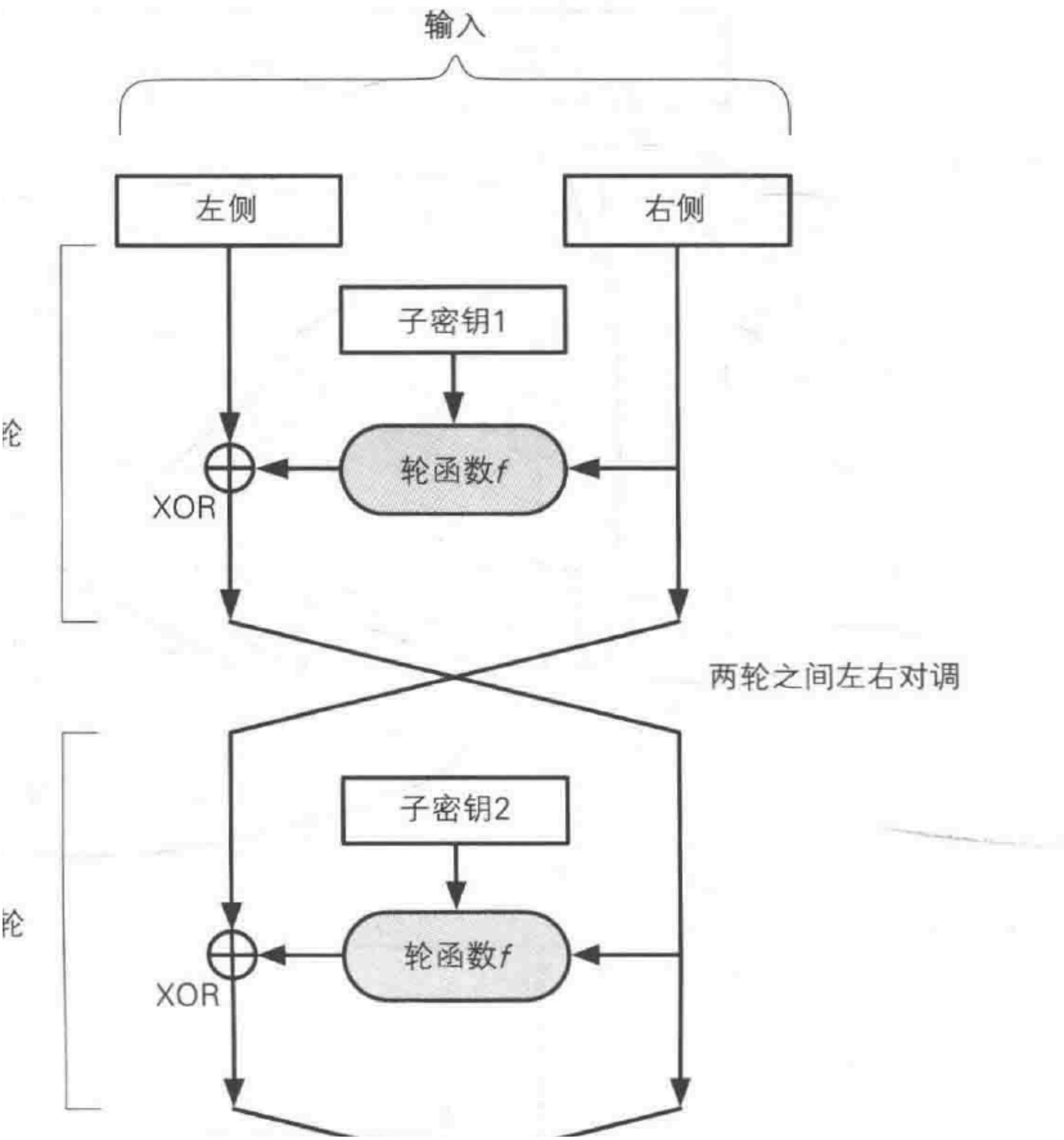
一轮的具体计算步骤：

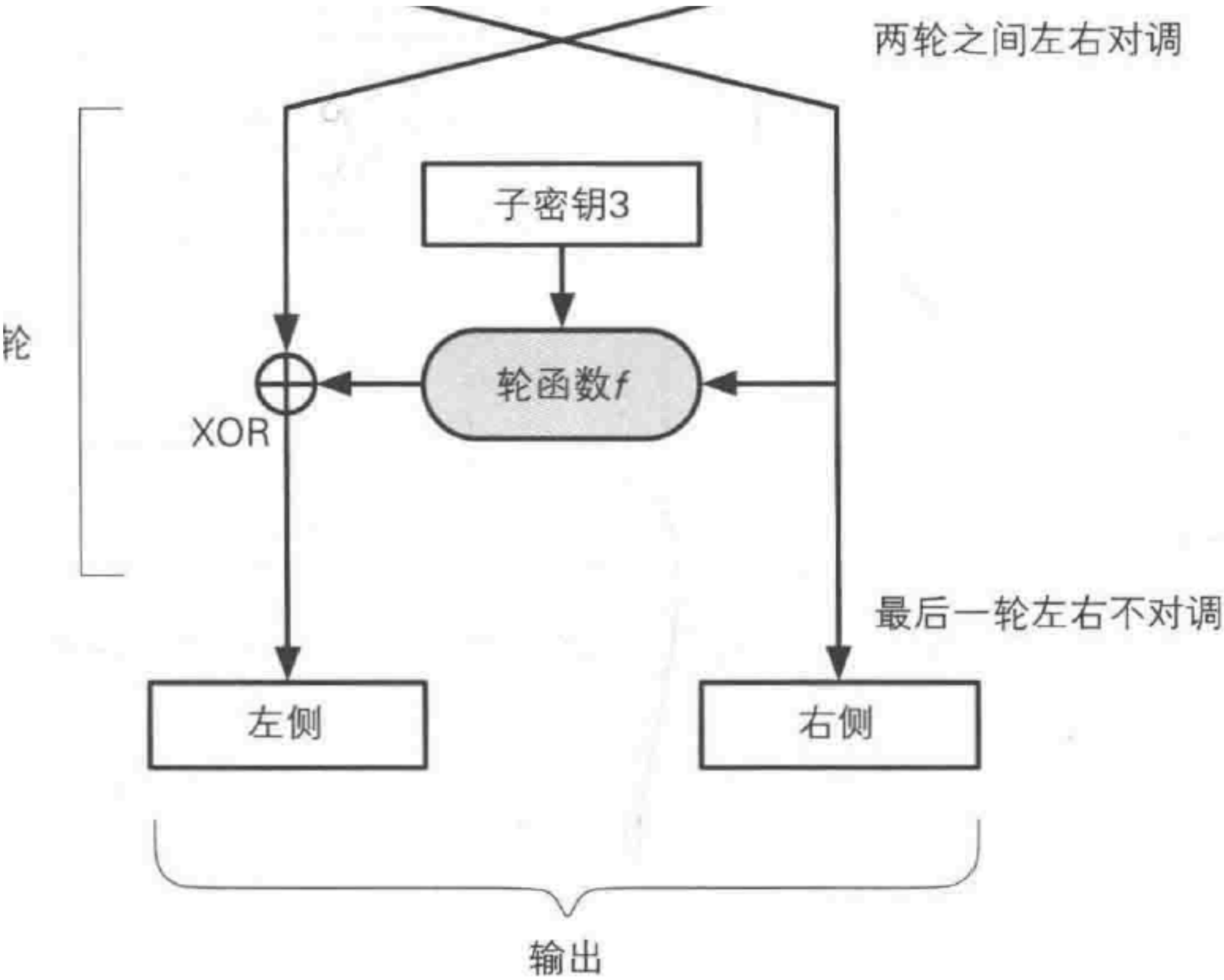
1. 将输入的数据等分为左右两部分

2. 将输入的右侧直接发送到输出的右侧
3. 将输入的右侧发送到轮函数
4. 轮函数根据右侧数据和子密钥，计算出一串看上去是随机的比特序列
5. 将上一步得到的比特序列与左侧数据进行xor运算，并将结果作为加密后的左侧

但是这样右侧的数据根本就没有加密，因此我们需要用不同的子密钥对一轮处理重复若干次，并在每次处理之间将左侧和右侧的数据对调。

如下图：





整个过程简述如下：

```
>初始明文处理阶段:
明文初始置换 64位
置换后的明文分为两组 32位*2

>子密钥产生阶段:
输入的密钥 64位
根据置换选择表得到密钥 28位*2
根据循环左移表，将密钥循环左移
将两个分开的密钥合并成56位，根据置换选择表2得到48位的子密钥
回到第三步根据轮数进行不同的左移，知道循环16轮，产生16个子密钥

>加密阶段:
对经过初始置换并分组的明文的一组进行E-盒拓展为48位
将拓展后的明文同对应的子密钥进行异或得到密文 48位
密文经过S-盒由48位变为32位
32位的密文经过P-盒乱序
交换左右两分组，进入下一轮加密阶段，整个加密阶段循环16轮

>最后密文逆置换阶段:
将以上操作所得的密文进行逆置换得到最终的密文

>以上过程就是对一个分组(64位)的DES加密
```

初始置换

第一步将64位明文根据初始置换表，进行置换，然后输入到**Feistel网络**中

初始置换表如下:

```
/* Initial Permutation Table */
static char 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
};
```

初始置换代码如下:

```
    /* initial permutation */
    for (i = 0; i < 64; i++) {

        init_perm_res <<= 1;
        init_perm_res |= (input >> (64-IP[i])) & LB64_MASK;

    }

    L = (uint32_t) (init_perm_res >> 32) & L64_MASK;
    R = (uint32_t) init_perm_res & L64_MASK;
```

IDA中如下：

```
  v27 = __readfsqword(0x28u);
  v19 = 0;
  memset(v26, 0, 0x80uLL);
  v23 = 0LL;
  v24 = 0LL;
  v25 = 0LL;
  for ( i = 0; i <= 63; ++i )
    v24 = (a1 >> (64 - byte_602060[i])) & 1 | 2 * v24;
```

加密处理

经过初始置换之后，便正式进入加密过程。

在这个阶段，会进行16轮相同的变化。

函数F

函数f由四步运算构成：

1. 密钥置换（Kn的生成）
2. 扩展置换
3. S-盒代替
4. P-盒置换

密钥置换--子密钥生成

DES算法由64位密钥产生16轮的48位子密钥。在每一轮的迭代过程中，使用不同的密钥。

1. 把密钥的奇偶校验位忽略不参与计算，即每个字节的第8位，将64位密钥降至56位，然后根据选择置换PC-1将这56位分成两块C0(28位)和D0(28位)；

2. 将C0和D0进行循环左移变化(注：每轮循环左移的位数由轮数决定)，变换后生成C1和D1，然后C1和D1

合并，并通过选择置换PC-2生成子密钥K1(48位)；

3. C1和D1在次经过循环左移变换，生成C2和D2，然后C2和D2合并，通过选择置换PC-2生成密钥K2(48位)；

4. 以此类推，得到K16(48位)。但是最后一轮的左右两部分不交换，而是直接合并在一起R16L16，作为逆置换的输入块。其中循环左移的位数一共是循环左移16次，其中第一次、第二次、第九次、第十六次是循环左移一位，其他都是左移两位。

密钥置换选择1---PC-1(子秘钥的生成)

操作对象是64位密钥

64位秘钥降至56位秘钥不是说将每个字节的第八位删除，而是通过缩小选择换位表1（置换选择表1）的变换变成56位。如下：

```
static char PC1[] = {
    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
};
```

再将56位秘钥分成C0和D0

代码如下：

```
    /* initial key schedule calculation */
    for (i = 0; i < 56; i++) {

        permuted_choice_1 <<= 1;
        permuted_choice_1 |= (key >> (64-PC1[i])) & LB64_MASK;

    }

    C = (uint32_t) ((permuted_choice_1 >> 28) & 0x000000000fffffff);
    D = (uint32_t) (permuted_choice_1 & 0x000000000fffffff);
```

IDA如下：

```
for ( j = 0; j <= 55; ++j )
  v23 = (a2 >> (64 - PC1[j])) & 1 | 2 * v23;
v16 = (v23 >> 28) & 0xFFFFFFF;
v17 = v23 & 0xFFFFFFF;
```

根据轮数，将Cn和Dn分别循环左移1位或2位

每轮移动的位数表：

```
/* Iteration Shift Array */
static char iteration_shift[] = {
 /* 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16 */
    1,  1,  2,  2,  2,  2,  2,  2,  1,  2,  2,  2,  2,  2,  2,  1
};
```

代码如下：

```
        // shifting Ci and Di
        for (j = 0; j < iteration_shift[i]; j++) {

            C = 0x0fffffff & (C << 1) | 0x00000001 & (C >> 27);
            D = 0x0fffffff & (D << 1) | 0x00000001 & (D >> 27);

        }
```

IDA中如下图：

```
for ( l = 0; iteration_shift[k] > l; ++l )
{
  v16 = 2 * v16 & 0xFFFFFFF | (v16 >> 27) & 1;
  v17 = 2 * v17 & 0xFFFFFFF | (v17 >> 27) & 1;
}
```

C1和D1分别是C0和D0经过循环左移得到的。

C1和D1合并之后，再经过置换选择表2生成48位的子秘钥K1。去掉第9、18、22、25、35、38、43、54位，从56位变成48位，再按表的位置置换。

置换选择表2(PC-2)如下：

```
/* Permuted Choice 2 Table */
static char PC2[] = {
    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
};
```

代码如下：

```
        permuted_choice_2 = 0;
        permuted_choice_2 = (((uint64_t) C) << 28) | (uint64_t) D ;

        sub_key[i] = 0;

        for (j = 0; j < 48; j++) {

            sub_key[i] <<= 1;
            sub_key[i] |= (permuted_choice_2 >> (56-PC2[j])) & LB64_MASK;

        }
```

IDA中如图：

```
    for ( m = 0; m <= 47; ++m )
    {
      v26[k] *= 2LL;
      v26[k] |= ((((unsigned __int64)v16 << 28) | v17) >> (56 - PC2[m])) & 1;
    }
```

C1和D1再次经过循环左移变换，生成C2和D2，C2和D2合并，通过PC-2生成子秘钥K2。 以此类推，得到子秘钥K1~K16。需要注意其中循环左移的位数。

扩展置换E(E位选择表)

通过扩展置换E，数据的右半部分Rn从32位扩展到48位。扩展置换改变了位的次序，重复了某些位。

扩展置换的目的：

1. 产生与秘钥相同长度的数据以进行异或运算，R0是32位，子秘钥是48位，所以R0要先进行扩展置换之后与子秘钥进行异或运算；
2. 提供更长的结果，使得在替代运算时能够进行压缩。

扩展置换E规则如下 中间为32位，两边为拓展位，拓展之后为48位

```
/*Expansion table */
static char 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
};
```

代码如下：

```
        s_input = 0;

        for (j = 0; j< 48; j++) {

            s_input <<= 1;
            s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);

        }
```

IDA中如图：

```
  for ( ii = 0; ii <= 47; ++ii )
    v21 = (HIDWORD(v18) >> (32 - E[ii])) & 1 | 2 * v21;
```

S-盒代替(功能表S盒)

Rn扩展置换之后与子秘钥Kn异或以后的结果作为输入块进行S盒代替运算 功能是把48位数据变为32位数据

代替运算由8个不同的代替盒(S盒)完成。每个S-盒有6位输入，4位输出。

所以48位的输入块被分成8个6位的分组，每一个分组对应一个S-盒代替操作。

经过S-盒代替，形成8个4位分组结果。

注意：每一个S-盒的输入数据是6位，输出数据是4位，但是每个S-盒自身是64位！！ 每个S-盒是4行16列的格式，因为二进制4位是0~15。8个S-盒的值如下：

```
/* The S-Box tables */
static char S[8][64] = {{
    /* S1 */
    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
},{
    /* S2 */
    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
},{
    /* S3 */
    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
},{
    /* S4 */
     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
},{
    /* S5 */
     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
},{
    /* S6 */
    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
},{
    /* S7 */
     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
},{
    /* S8 */
    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
}};
```

S-盒的计算过程

以S-盒8为例子

							S-盒8									
行 \ 列	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

假设S-盒8的输入(即异或函数的第43~18位)为110011。

第1位和最后一位组合形成了11(二进制)，对应S-盒8的第3行。中间的4位组成形成1001(二进制),对应S-盒8的第9列。所以对应S-盒8第3行第9列值是12。则S-盒输出是1100(二进制)。

代码如下：

```
/* S-Box Tables */
for (j = 0; j < 8; j++) {
    // 00 00 RCCC CR00 00 00 00 00 00 s_input
    // 00 00 1000 0100 00 00 00 00 00 row mask
    // 00 00 0111 1000 00 00 00 00 00 column mask

    row = (char) ((s_input & (0x0000840000000000 >> 6*j)) >> 42-6*j);
    row = (row >> 4) | row & 0x01;

    column = (char) ((s_input & (0x0000780000000000 >> 6*j)) >> 43-6*j);

    s_output <<= 4;
    s_output |= (uint32_t) (S[j][16*row + column] & 0x0f);

}
```

IDA中如图：

```
for ( jj = 0; jj <= 7; ++jj )
  v19 = S[64 * (signed __int64)jj
        + 16
        * (char)(((char)((v22 & (145135534866432LL >> 6 * (unsigned __int8)jj)) >> (-6 * (unsigned __int8)jj + 42))
        + (char)((v22 & (131941395333120LL >> 6 * (unsigned __int8)jj)) >> (-6 * (unsigned __int8)jj + 43))] & 0xF
```

P-盒置换

S-盒代替运算，每一盒得到4位，8盒共得到32位输出。这32位输出作为P盒置换的输入块。

P盒置换将每一位输入位映射到输出位。任何一位都不能被映射两次，也不能被略去。

经过P-盒置换的结果与最初64位分组的左半部分异或，然后左右两部分交换，开始下一轮迭代。

P-盒置换表(表示数据的位置)共32位

将32位的输入的第16位放在第一位，第七位放在第二位，第二十位放在第三位，以此类推

```
/* Post S-Box permutation */
static char 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
};
```

代码如下：

```
        f_function_res = 0;

        for (j = 0; j < 32; j++) {

            f_function_res <<= 1;
            f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;

        }
```

IDA中如图：

```
for ( kk = 0; kk <= 31; ++kk )
  v20 = (v19 >> (32 - P[kk])) & 1 | 2 * v20;
```

这样我们便完成了DES中最重要的加密部分

逆置换

将初始置换进行16次的迭代，即进行16层的加密变换，这个运算过程我们暂时称为函数f。得到L16和R16，将此作为输入块，进行逆置换得到最终的密文输出块。逆置换是初始置换的逆运算。从初始置换规则中可以看到，原始数据的第1位置换到了第40位，第2位置换到了第8位。则逆置换就是将第40位置换到第1位，第8位置换到第2位。以此类推，逆置换规则如下

```
/* Inverse Initial Permutation Table */
static char PI[] = {
    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
};
```

代码如下：

```
    pre_output = (((uint64_t) R) << 32) | (uint64_t) L;

    /* inverse initial permutation */
    for (i = 0; i < 64; i++) {

        inv_init_perm_res <<= 1;
        inv_init_perm_res |= (pre_output >> (64-PI[i])) & LB64_MASK;

    }
```

IDA中如图：

```
for ( l1 = 0; l1 <= 63; ++l1 )
  v25 = (v18 >> (64 - PI[l1])) & 1 | 2 * v25;
return v25;
```

注意：DES算法的加密密钥是根据用户输入的秘钥生成的,该算法把64位密码中的第8位、第16位、第24位、第32位、第40位、第48位、第56位、第64位作为奇偶校验位,在计算密钥时要忽略这8位.所以实际中使用的秘钥有效位是56位。详情计算看本文的3.1.2秘钥置换选择。

DES算法描述

1)、输入64位明文数据，并进行初始置换IP；

2)、在初始置换IP后，明文数据再被分为左右两部分，每部分32位，以L0，R0表示；

3)、在秘钥的控制下，经过16轮运算(f)；

4)、16轮后，左、右两部分交换，并连接再一起，再进行逆置换；

5)、输出64位密文。

DES解密

加密和解密可以使用相同的算法。加密和解密唯一不同的是秘钥的次序是相反的。就是说如果每一轮的加密秘钥分别是K1、K2、K3...K16，那么解密秘钥就是K16、K15、K14...K1。为每一轮产生秘钥的算法也是循环的。加密是秘钥循环左移，解密是秘钥循环右移。解密秘钥每次移动的位数是：0、1、2、2、2、2、2、2、1、2、2、2、2、2、2、1。

IDA动态调试跟踪数据流

IDA跟进

首先对明文进行初始置换

```
for ( i = 0; i <= 63; ++i )
    init_perm_res = (input >> (64 - byte_602060[i])) & 1 | 2 * init_perm_res;
```

其结果为 `0xFFDE6AE700FF0550`

而后对置换后的明文分成左右两部分

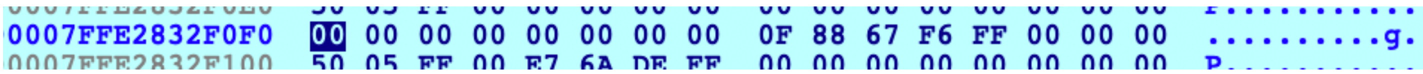
```
L = __PAIR__(init_perm_res, HIDWORD(init_perm_res));
```

左边 `0xFFDE6AE7` ,右边 `0x00FF0550`

暂时先不管左右两部分，我们先来生成16个子密钥

对密钥进行选择置换，此时会将64位的密钥转换为56位的密钥，也就是剔除了每个字节的最后一位。

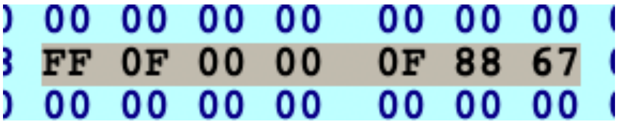
```
for ( ia = 0; ia <= 55; ++ia )
    permuted_choice_1 = (key >> (64 - PC1[ia])) & 1 | 2 * permuted_choice_1;
```



得到的结果为 `0xFFF6667880F`

而后对56位的密钥进行分组，分为C0，D0且都为28位

```
C = (permuted_choice_1 >> 28) & 0xFFFFFFF;
  D = permuted_choice_1 & 0xFFFFFFF;
```



C0为 `0xFFF`，D0为 `0x667880`

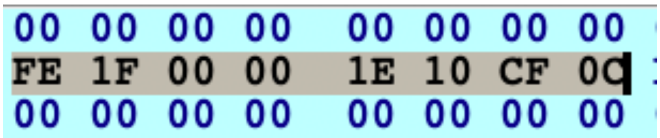
注意一下，这里都是位运算，C0和D0都是28位，但是用来存储其值的变量应该是32位的。

```
>>> 0xFFF6667880F>>28
65526
>>> hex(65526)
'0xfff6'

但是最后一字节需要去掉哦
```

而后对C0和D0分别根据 `shift table` 进行移位操作

```
for ( j = 0; iteration_shift[ib] > j; ++j )
    {
      C = 2 * C & 0xFFFFFFF | (C >> 27) & 1;
      D = 2 * D & 0xFFFFFFF | (D >> 27) & 1;
    }
```



其结果为 `0x1FFE` 和 `0xCCF101E`

而后将其结果合并组成56位的数，之后通过选择置换表2，将56位变成48位

```
sub_key[ib] = 0LL;
    for ( ja = 0; ja <= 47; ++ja )
    {
      sub_key[ib] *= 2LL;
      sub_key[ib] |= ((((unsigned __int64)C << 28) | D) >> (56 - PC2[ja])) & 1;
    }
```

```
C2 2A 57 AC 2C 50 00 00  47 A3 50 A4 AC 50 00 00
8C 84 F6 26 AC D0 00 00  CB 37 48 26 A6 E0 00 00
29 F0 3E 26 96 E0 00 00  62 5D 62 72 92 E0 00 00
3A A9 8C 72 D2 A4 00 00  50 5E E5 52 53 A6 00 00
40 9A CB 53 53 26 00 00  3C C7 D0 51 51 2F 00 00
8C 1E 19 D9 41 0F 00 00  B1 70 D8 99 41 1F 00 00
2D 6A 23 89 09 1F 00 00  92 39 B2 8D 28 1B 00 00
37 03 A5 8C 2C 19 00 00  C0 43 A7 8C 2C 51 00 00
```

这样便产生了16个48位的子密钥

但此时明文才只是经过了初始置换并进行了分组，所以需要对32位的明文分组拓展为48位。

```
s_input = 0LL;
    for ( jb = 0; jb <= 47; ++jb )
      s_input = (HIDWORD(L) >> (32 - E[jb])) & 1 | 2 * s_input;
```

```
A0 AA 80 FE 17 00 00 00
```

经过E盒的拓展之后，结果如下： `17FE80AAA0`

之后便可以将子密钥和经过拓展之后明文异或

```
s_inputa = v3 ^ s_input;
```

```
62 80 D7 52 3B 50 00 00
```

其结果为 `0x503B52D78062`

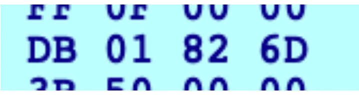
而后我们需要将异或的结果变为32位的数，这时S-盒就登场了

```
s_inputa = v3 ^ s_input;
    for ( jc = 0; jc <= 7; ++jc )
      s_output = S[64 * (signed __int64)jc
                 + 16
                 * (char)(((char)((s_inputa & (145135534866432LL >> 6 * (unsigned
__int8)jc)) >> (-6

              * (unsigned __int8)jc

              + 42)) >> 4) | ((s_inputa & (145135534866432LL >> 6 * (unsigned __in
t8)jc)) >> (-6 * (unsigned __int8)jc + 42)) & 1)
                 + (char)((s_inputa & (131941395333120LL >> 6 * (unsigned __int8)j
c)) >> (-6 * (unsigned __int8)jc + 43))] & 0xF | 16 * s_output;
```

S-盒是一个64位的数组，它的作用是：输入一个6位的数转换成4位然后输出，这样经过8个S-盒就可以将原本 `6*8` 的数据转化为 `4*8` 的数据了。



其结果为 `0x6D8201DB`

然后在将其结果经过P盒的置换

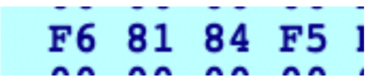
```
 for ( jd = 0; jd <= 31; ++jd )
      f_function_res = (s_output >> (32 - P[jd])) & 1 | 2 * f_function_res;
```

结果为 `0xA5AEB11`

而后将加密后的结果同经过初始置换的另一个分组R0进行异或，并将结果保存到R1中，同时将R0赋值给L1，进行下一轮

注意哦，此时子密钥已经全部生成了，因此只需要在进行E-盒拓展，子密钥异或，S盒变换，P盒置换即可，然后再次左右两边异或交换

```
temp = HIDWORD(L);
    HIDWORD(L) = f_function_res ^ L;
    LODWORD(L) = temp;
```



结果为 `0xF58481F6`

经过16轮之后，一次加密过程基本上已经完成了，最后通过逆置换便可以得到密文

```
for ( id = 0; id <= 63; ++id )
    inv_init_perm_res = (L >> (64 - PI[id])) & 1 | 2 * inv_init_perm_res;
  return inv_init_perm_res;
```

最后加密的结果为 `0xFD181E19466FE937`

到此DES的整个加密过程就是如此，其解密过程只需要将子密钥的顺序倒序即可。

不过由于DES算法已经显得非常复杂，我已经没有能力在从数据流中抽象出什么特征来说明DES算法，毕竟仅从数据的角度来看，并没有发现有用的信息，不过经过这次数据流的追踪，有助于我们更加了解DES加密的过程，就算是将DES魔改了，相信大家也是有能力写出相应的解密算法的。

> 关于魔改DES，我会进行尝试，如果成功了，会贴在最后。

DES算法的特点

1、分组加密算法：

以64位为分组。64位明文输入，64位密文输出。

2、对称算法：

加密和解密使用同一秘钥

3、有效秘钥长度为56位

秘钥通常表示为64位数，但每个第8位用作奇偶校验，可以忽略。

4、代替和置换

DES算法是两种加密技术的组合：混乱和扩散。先替代后置换。

完整代码

```
/*
 * Data Encryption Standard
 * An approach to DES algorithm
 *
 * By: Daniel Huertas Gonzalez
 * Email: huertas.dani@gmail.com
 * Version: 0.1
```

```
 *
 * Based on the document FIPS PUB 46-3
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define LB32_MASK   0x00000001
#define LB64_MASK   0x0000000000000001
#define L64_MASK    0x00000000ffffffff
#define H64_MASK    0xffffffff00000000

/* Initial Permutation Table */
static char 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
};

/* Inverse Initial Permutation Table */
static char PI[] = {
    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
};

/*Expansion table */
static char 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
};
```

```
/* Post S-Box permutation */
static char 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
};

/* The S-Box tables */
static char S[8][64] = {{
    /* S1 */
    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
},{
    /* S2 */
    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
},{
    /* S3 */
    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
},{
    /* S4 */
     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
},{
    /* S5 */
     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
},{
    /* S6 */
    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
},{
    /* S7 */
     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
},{
    /* S8 */
    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
}};

/* Permuted Choice 1 Table */
static char PC1[] = {
    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
};

/* Permuted Choice 2 Table */
static char PC2[] = {
    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
};

/* Iteration Shift Array */
static char iteration_shift[] = {
 /* 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16 */
    1,  1,  2,  2,  2,  2,  2,  2,  1,  2,  2,  2,  2,  2,  2,  1
};

/*
 * The DES function
```

```
 * input: 64 bit message
 * key: 64 bit key for encryption/decryption
 * mode: 'e' = encryption; 'd' = decryption
 */
uint64_t des(uint64_t input, uint64_t key, char mode) {

    int i, j;

    /* 8 bits */
    char row, column;

    /* 28 bits */
    uint32_t C                  = 0;
    uint32_t D                  = 0;

    /* 32 bits */
    uint32_t L                  = 0;
    uint32_t R                  = 0;
    uint32_t s_output           = 0;
    uint32_t f_function_res     = 0;
    uint32_t temp               = 0;

    /* 48 bits */
    uint64_t sub_key[16]        = {0};
    uint64_t s_input            = 0;

    /* 56 bits */
    uint64_t permuted_choice_1  = 0;
    uint64_t permuted_choice_2  = 0;

    /* 64 bits */
    uint64_t init_perm_res      = 0;
    uint64_t inv_init_perm_res  = 0;
    uint64_t pre_output         = 0;

    /* initial permutation */
    for (i = 0; i < 64; i++) {

        init_perm_res <<= 1;
        init_perm_res |= (input >> (64-IP[i])) & LB64_MASK;

    }

    L = (uint32_t) (init_perm_res >> 32) & L64_MASK;
    R = (uint32_t) init_perm_res & L64_MASK;

    /* initial key schedule calculation */
    for (i = 0; i < 56; i++) {
```

```
        permuted_choice_1 <<= 1;
        permuted_choice_1 |= (key >> (64-PC1[i])) & LB64_MASK;

    }

    C = (uint32_t) ((permuted_choice_1 >> 28) & 0x000000000fffffff);
    D = (uint32_t) (permuted_choice_1 & 0x000000000fffffff);

    /* Calculation of the 16 keys */
    for (i = 0; i< 16; i++) {

        /* key schedule */
        // shifting Ci and Di
        for (j = 0; j < iteration_shift[i]; j++) {

            C = 0x0fffffff & (C << 1) | 0x00000001 & (C >> 27);
            D = 0x0fffffff & (D << 1) | 0x00000001 & (D >> 27);

        }

        permuted_choice_2 = 0;
        permuted_choice_2 = (((uint64_t) C) << 28) | (uint64_t) D ;

        sub_key[i] = 0;

        for (j = 0; j < 48; j++) {

            sub_key[i] <<= 1;
            sub_key[i] |= (permuted_choice_2 >> (56-PC2[j])) & LB64_MASK;

        }

    }

    for (i = 0; i < 16; i++) {

        /* f(R,k) function */
        s_input = 0;

        for (j = 0; j< 48; j++) {

            s_input <<= 1;
            s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);

        }

        /*
```

```
         * Encryption/Decryption
         * XORing expanded Ri with Ki
         */
        if (mode == 'd') {
            // decryption
            s_input = s_input ^ sub_key[15-i];

        } else {
            // encryption
            s_input = s_input ^ sub_key[i];

        }

        /* S-Box Tables */
        for (j = 0; j < 8; j++) {
            // 00 00 RCCC CR00 00 00 00 00 00 s_input
            // 00 00 1000 0100 00 00 00 00 00 row mask
            // 00 00 0111 1000 00 00 00 00 00 column mask

            row = (char) ((s_input & (0x0000840000000000 >> 6*j)) >> 42-6*j);
            row = (row >> 4) | row & 0x01;

            column = (char) ((s_input & (0x0000780000000000 >> 6*j)) >> 43-6*j);

            s_output <<= 4;
            s_output |= (uint32_t) (S[j][16*row + column] & 0x0f);

        }

        f_function_res = 0;

        for (j = 0; j < 32; j++) {

            f_function_res <<= 1;
            f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;

        }

        temp = R;
        R = L ^ f_function_res;
        L = temp;

    }

    pre_output = (((uint64_t) R) << 32) | (uint64_t) L;

    /* inverse initial permutation */
    for (i = 0; i < 64; i++) {
```

```
        inv_init_perm_res <<= 1;
        inv_init_perm_res |= (pre_output >> (64-PI[i])) & LB64_MASK;

    }

    return inv_init_perm_res;

}

void str2hex(char *source,char *dest,int keyLen){
    uint8_t i;
    uint8_t highByte, lowByte;

    for (i = 0; i < keyLen; i++)
    {
        highByte = source[i] >> 4;
        lowByte = source[i] & 0x0f ;

        highByte += 0x30;

        if (highByte > 0x39)
                dest[i * 2] = highByte + 0x07;
        else
                dest[i * 2] = highByte;

        lowByte += 0x30;
        if (lowByte > 0x39)
            dest[i * 2 + 1] = lowByte + 0x07;
        else
            dest[i * 2 + 1] = lowByte;
    }
    return ;
}


int main(int argc, const char * argv[]) {

    int i;

    uint64_t input = 0x7177657274797569;
    uint64_t key = 0x3132333435363738;
    uint64_t result = 0x0000000000000000;

    // char * in = "qwertyui";
    // char in_hex[17];
    // in_hex[16]=0;
    // str2hex(in,in_hex,8);
```

```
    // printf("0x%s",in_hex);

    result = des(input, key, 'e');
    printf ("E: 0x%016llx\n", result);//0x71d05d44594773b0

    result = des(result, key, 'd');
    printf ("D: 0x%016llx\n", result);

    exit(0);

}
```

魔改DES

其实也没怎么更改，但至少对于不懂DES原理的人来说，这其中还是有困难的。

很明显有明文长度变换的地方我们无法修改，子密钥的产生过程由于存在密钥长度的变换，所以也不太适合。所以整个算法过程看下来，我最后把P-盒加密的部分给删除了。相应的解密也是同样的。

也就是最后一部分将32位的密文进行P-盒置换，运行结果如下：

最后的代码如下：

```
/*
 * Data Encryption Standard
 * An approach to DES algorithm
 *
 * By: Daniel Huertas Gonzalez
 * Email: huertas.dani@gmail.com
 * Version: 0.1
 *
 * Based on the document FIPS PUB 46-3
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

```
#define LB32_MASK   0x00000001
#define LB64_MASK   0x0000000000000001
#define L64_MASK    0x00000000ffffffff
#define H64_MASK    0xffffffff00000000

/* Initial Permutation Table */
static char 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
};

/* Inverse Initial Permutation Table */
static char PI[] = {
    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
};

/*Expansion table */
static char 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
};

/* Post S-Box permutation */
static char 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
};

/* The S-Box tables */
static char S[8][64] = {{
    /* S1 */
    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
},{
    /* S2 */
    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
},{
    /* S3 */
    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
},{
    /* S4 */
     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
},{
    /* S5 */
     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
},{
    /* S6 */
    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
},{
    /* S7 */
     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
},{
    /* S8 */
    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
}};

/* Permuted Choice 1 Table */
static char PC1[] = {
    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
};

/* Permuted Choice 2 Table */
static char PC2[] = {
    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
};

/* Iteration Shift Array */
static char iteration_shift[] = {
 /* 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16 */
    1,  1,  2,  2,  2,  2,  2,  2,  1,  2,  2,  2,  2,  2,  2,  1
};

/*
 * The DES function
 * input: 64 bit message
 * key: 64 bit key for encryption/decryption
 * mode: 'e' = encryption; 'd' = decryption
 */
uint64_t des(uint64_t input, uint64_t key, char mode) {
```

```
    int i, j;

    /* 8 bits */
    char row, column;

    /* 28 bits */
    uint32_t C                  = 0;
    uint32_t D                  = 0;

    /* 32 bits */
    uint32_t L                  = 0;
    uint32_t R                  = 0;
    uint32_t s_output           = 0;
    uint32_t f_function_res     = 0;
    uint32_t temp               = 0;

    /* 48 bits */
    uint64_t sub_key[16]        = {0};
    uint64_t s_input            = 0;

    /* 56 bits */
    uint64_t permuted_choice_1  = 0;
    uint64_t permuted_choice_2  = 0;

    /* 64 bits */
    uint64_t init_perm_res      = 0;
    uint64_t inv_init_perm_res  = 0;
    uint64_t pre_output         = 0;

    /* initial permutation */
    for (i = 0; i < 64; i++) {

        init_perm_res <<= 1;
        init_perm_res |= (input >> (64-IP[i])) & LB64_MASK;

    }

    L = (uint32_t) (init_perm_res >> 32) & L64_MASK;
    R = (uint32_t) init_perm_res & L64_MASK;

    /* initial key schedule calculation */
    for (i = 0; i < 56; i++) {

        permuted_choice_1 <<= 1;
        permuted_choice_1 |= (key >> (64-PC1[i])) & LB64_MASK;

    }
```

```
    C = (uint32_t) ((permuted_choice_1 >> 28) & 0x000000000fffffff);
    D = (uint32_t) (permuted_choice_1 & 0x000000000fffffff);

    /* Calculation of the 16 keys */
    for (i = 0; i< 16; i++) {

        /* key schedule */
        // shifting Ci and Di
        for (j = 0; j < iteration_shift[i]; j++) {

            C = 0x0fffffff & (C << 1) | 0x00000001 & (C >> 27);
            D = 0x0fffffff & (D << 1) | 0x00000001 & (D >> 27);

        }

        permuted_choice_2 = 0;
        permuted_choice_2 = (((uint64_t) C) << 28) | (uint64_t) D ;

        sub_key[i] = 0;

        for (j = 0; j < 48; j++) {

            sub_key[i] <<= 1;
            sub_key[i] |= (permuted_choice_2 >> (56-PC2[j])) & LB64_MASK;

        }

    }

    for (i = 0; i < 16; i++) {

        /* f(R,k) function */
        s_input = 0;

        for (j = 0; j< 48; j++) {

            s_input <<= 1;
            s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);

        }

        /*
         * Encryption/Decryption
         * XORing expanded Ri with Ki
         */
        if (mode == 'd') {
            // decryption
            s_input = s_input ^ sub_key[15-i];
```

```
        } else {
            // encryption
            s_input = s_input ^ sub_key[i];

        }

        /* S-Box Tables */
        for (j = 0; j < 8; j++) {
            // 00 00 RCCC CR00 00 00 00 00 00 s_input
            // 00 00 1000 0100 00 00 00 00 00 row mask
            // 00 00 0111 1000 00 00 00 00 00 column mask

            row = (char) ((s_input & (0x0000840000000000 >> 6*j)) >> 42-6*j);
            row = (row >> 4) | row & 0x01;

            column = (char) ((s_input & (0x0000780000000000 >> 6*j)) >> 43-6*j);

            s_output <<= 4;
            s_output |= (uint32_t) (S[j][16*row + column] & 0x0f);

        }

        /*
        f_function_res = 0;

        for (j = 0; j < 32; j++) {

            f_function_res <<= 1;
            f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;

        }
        */
        temp = R;
        R = L ^ s_output;
        L = temp;

    }

    pre_output = (((uint64_t) R) << 32) | (uint64_t) L;

    /* inverse initial permutation */
    for (i = 0; i < 64; i++) {

        inv_init_perm_res <<= 1;
        inv_init_perm_res |= (pre_output >> (64-PI[i])) & LB64_MASK;

    }
```

```
    return inv_init_perm_res;

}

int main(int argc, const char * argv[]) {

    int i;

    uint64_t input = 0x7177657274797569;
    uint64_t key = 0x3132333435363738;
    uint64_t result = 0x0000000000000000;


    char a[]="qwertyui";
    char * reset;


    uint64_t * b = a;

    uint64_t re = *b;
    printf("0x%016llx\n",re);

    result = des(re, key, 'e');
    printf ("E: 0x%016llx\n", result);//0x450c1d3608c12d52

    result = des(result, key, 'd');
    printf ("D: 0x%016llx\n", result);

    exit(0);

}
```

补充

我们在这整个过程中，并没有涉及到加密模式的概念，因为这不是DES加密算法的重点内容。

有关模式的内容我在密码学笔记中有写，这里是针对当个8字节也就是64位明文进行加密的，因此不涉及模式。