山东大学
SHANDONG UNIVERSITY

实验二：DES 的差分分析

组员：刘舒畅，李昕，林宗茂

2023 年 10 月 10 日

成员信息及完成部分：

刘舒畅：202122460175 代码编写、差分路径分析及报告校对

李昕：202100460065 代码编写、成功率复杂度分析及报告编写

林宗茂：202100460128 代码测试，报告编写

摘　要

在本次实验中，我们分析了 des 差分攻击的原理，首先根据论文路线，构造了多个高概率的 DES 三轮差分路线，并测试了其在实际攻击中的差分频率。

同时，我们在搜索到的三轮差分路径的基础上，构造了 DES 的 6 轮密钥恢复攻击，在选择 300 个明密文对的基础上，理论攻击成功率达到 0.999991，复杂度约为 $2^7 * 1500$ 次 6 轮 des 加密。并利用恢复的 42bit 轮密钥，恢复出了 64bit 主密钥。

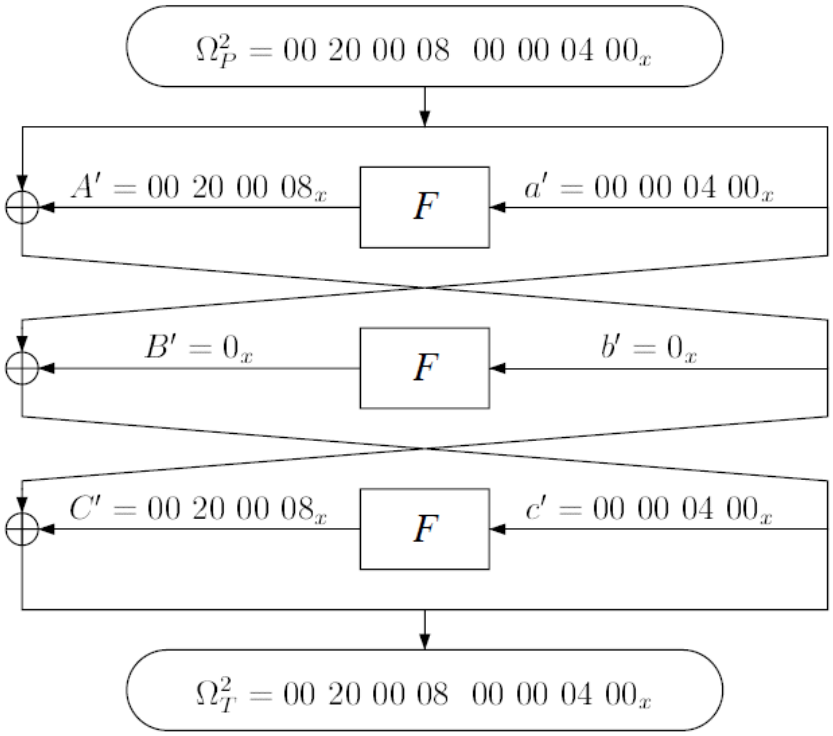
最后，我们参考论文中给出的差分路线，尝试构造了一个 5 轮高概率差分路线，并在理论上分析了 2^n 长轮数构造的方法。

关键词：DES　差分分析

目录

1 问题重述

问题一 仿照以下 3 轮 DES 算法（不考虑初始 IP 置换和最后的 IP 逆置换，最后一轮左右不交换）的迭代差分路线，程序搜索或手动推导其它 3 轮迭代差分路线（与以下示例不同的），并计算相应概率。



问题二 从 1）中搜索到的路线中选取概率最大的一条（若多条概率最大，则任选其中一条即可）。设相应概率为 2^-p，随机选取 10 组不同的密钥，随机选取 2^{-p+8} 对满足输入差分的明文对，输入 3 轮 DES 算法，得到相应的输出对，统计输出差分出现的频率（实测概率），与 1）中概率相比较，体会差分和差分路线的含义。

问题三 在 1）中搜索到的路线基础上，给出 6 轮 DES 算法的密钥恢复攻击（可以仅恢复第 6 轮的部分轮密钥），并分析成功率和复杂度。其中，选择明文可通过调用群文件中的可执行程序来实现。

问题四 参考论文中给出的路线，尝试寻找 5 轮或更长轮数的迭代差分路线。

2 实验准备

2.1 3 轮 DES 差分路线的构造

观察题目中给出的两个 3 轮差分路线，可以发现都是利用恒成立的 S 盒差分路线 0X0→0x0 构造的，对每一轮进行分析：

第一轮 将输入差分均分为左右两部分，原始输入右部经过 F 函数后与原始输入左部异或得到 0，左右交换得到第一轮输出，其右部为 0，左部为原始输入右部。

第二轮 第二轮：第一轮输出右部的 0 经第二轮 F 函数后不变，与第一轮输出左部异或后左右交换得到第二轮输出，其右部为原始输入右部，左部为 0。

第三轮 第三轮：第二轮输出的右部作为第三轮输出右部，右部经 F 函数变换后与第二轮输出左部 0 异或得到第三轮输出左部，即输出与原始输入相同。

要仿照示例构造 3 轮差分路线，首先要求 DES 每个 S 盒的差分分布表。要使差分路线概率尽可能大，构造的第一轮应只有一个活跃 S 盒，因此选定 8 个 S 盒中一个 S 盒的输入差分，其余 S 盒输入差分设置为 0 作为原始输入右部，选定 S 盒输入差分后再选择 S 盒最优输出差分，经过 P 置换后得到原始输入左部，可得到其中一个差分路线的输入。

三轮加密中每一轮的 f 函数包含三个部件：E 扩展、S 盒、P 置换，现分析每个部件对差分路线构造的影响。

E **扩展** E 扩展通过有重复元素的置换将 4*8=32bit 数据扩展到 6*8=48bit，置换表如下图，每一行为一个 S 盒的输入。观察发现第一个 S 盒的输入差分为第 32、1、2、3、4、5 位的差分，其中 32、1、4、5 位与第 2 个、第 8 个 S 盒相关，与若要做到遍历其中一个 S 盒的所有可能，且其他 7 个 S 盒输入全为 0，只能遍历每个 S 盒中间两位的取值，即 F 函数输入的 4bit 分组只能遍历 0010、0100、0110，即只能遍历 2，4，6，其他 s 盒同理。

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

S **盒** 6 位输入，4 位输出，对差分路线的影响为某一对差分的差分传播概率。

P **置换** 线性置换，将差分结果经过 P 置换即可，对差分路线的概率没有影响。

2.2 测试差分路线概率

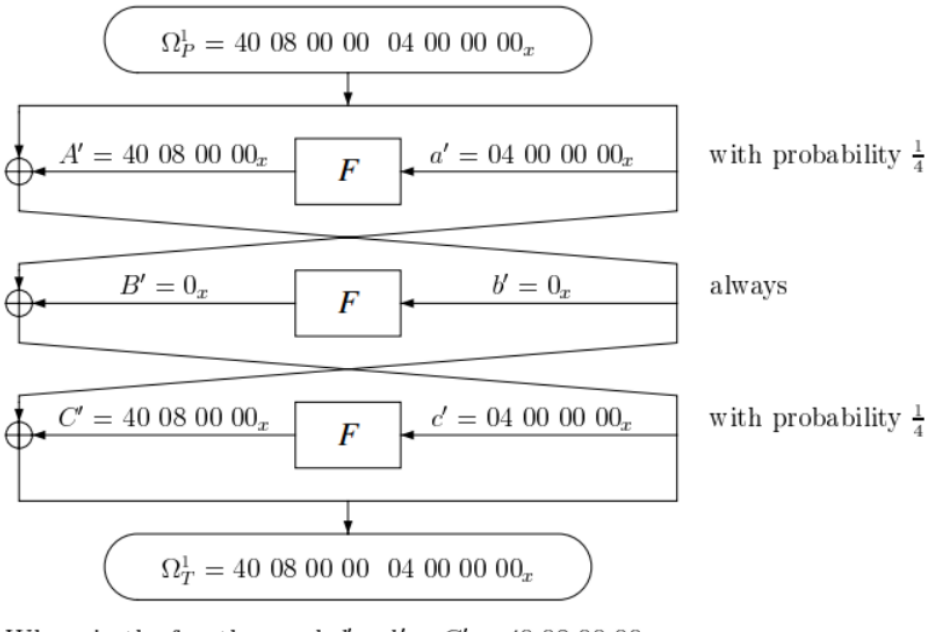
根据第一问程序运行结果，可以得到四条差分传播概率 $p = (\frac{14}{64})^2\ 2^{-4.4} = 4.785\%$ 的差分传播路线，其输入差分分别为 00808200 60000000，40004010 02000000，00004010 06000000，00100001 00000060，在这里对第一个路线进行测试，随机选取 $2^{12.4} \approx 5400$ 对满足差分的明文对，统计输出差分出现的频率，与概率作比较。

2.3 6 轮 DES 算法的密钥恢复攻击原理

2.3.1 还原第六轮轮密钥

6 轮 DES 算法的密钥恢复攻击需要利用到多个差分的不随机现象，通过搜索 7 个 S 盒从而恢复出第六轮轮密钥的 42bit 数据。其余 14bit 可以通过密钥逆扩展暴力枚举得到。

首先，我们选择概率为 $\frac{1}{16}$ 的一对差分输入输出来构造前三轮，即 40080000 04000000 以下差分路线：



前三轮差分已定后，单独观察第四轮的左右两支输入差分，注意到在第四轮时，有 5 个 S 盒的输入差分为 0（S2,S5,...,S8），这代表其输出差分也为 0，此时 0 与其他值异或时不改变其取值，从而实现了第五层的跨越，我们把第四轮轮左半支的输入差分记作 β，右半支记作 α, 将每一轮 F 函数前中间值差分记作小写阿拉伯数字，如 $c^{'}$，把 F 函数后差分记作大写阿拉伯数字，如 $C^{'}$。

则对于第六轮的右支中对应的 5 个 S 盒有，$F_5^{'} = e^{'} \oplus l^{'} = c^{'} \oplus D_5^{'} \oplus l^{'}$，其中 $L^{'}$ 为左半支输出的差分，$D_5^{'}$ 为 $D^{'}$S2,S5,...,S8 对应的 5 个 6bit 位。

此时可以针对第六轮这五个存在不随机现象的 S 盒进行选择明文下的密钥搜索，使其在指定五个 S 盒上满足 $F_5^{'} = c^{'} \oplus D_5^{'} \oplus l^{'}$（以 $\frac{1}{16}$ 的概率成立）。对于剩下的 12bit 信息，选取第二个高概率差分路线 00200008 00000400 作为前三轮，同理构造六轮差分路线，第四轮 S 盒在 S1,S2,S4,S5 和 S6 上存在 0 到 0 的置换，同样可以构造五个 S 盒的方程。由于两条路线中各自 5 个 S 盒与对方存在重叠，故一共可以恢复第六轮 7 个 S 盒共 42bit 的轮密钥信息。

2.3.2 还原主密钥

通过上述步骤可以还原出第六轮密钥的 42 比特信息，下面讨论利用第六轮部分轮密钥比特信息进行选择明文下的主密钥恢复攻击。

根据 DES 的密钥扩展算法，将主密钥的 56bit 分配到了每一轮 48bit 的轮密钥里，所以只需要枚举剩下 6+8=14 比特即可. 首先需要根据密钥扩展算法中的 PC2 盒，求出最后一轮密钥舍弃了主密钥的哪 8bit，并用一个字节串 mask 表示。如果有一位弃掉

了，就让 mask 这一位置为 1 ，然后 mask 和枚举的 2^6 个轮密钥同步逆序执行密钥扩展算法的过程，枚举 0 到 255，将 256 组猜测字节分配到 mask 所指示的字节，并与轮密钥异或起来，枚举缺失剩下的 8 bit 主密钥。

2.3.3 差分攻击的成功率 P_s 分析

定义差分分析密钥恢复阶段中，正确对与计数器的平均计数之比为 S_N 为信噪比。

若差分路线概率为 p，采样阶段随机选取的明文对数为 m，去噪阶段明密文对过滤的比例为 ϵ，去噪后每个明密文对对应的可能候选密钥个数为 η，可恢复的密钥长度为 k，则 $S_N = \frac{p*2^k}{\eta*\epsilon}$。

定义 $\mu = p * m$ 为期望的正确对数，p_r 为正确密钥由随机对得出的平均概率，则 $S_N = \frac{p}{p_r}$。我们定义 T_i 计数随机密钥 k_i，角标为其大小排序，假设 T_i 是独立且当 i!=0 时是均匀分布的 (后一种假设意味着所有错误的密钥都有相同的机会被随机对验证)，即所有 p_i 都是相同的 (p_i 是 k_i 被任意明文对验证为真的概率)，记为 p_ω，此时，计数器 T_i 具有二项式分布，T_0 为 $B(m, p_0)$，其余为 $B(m, p_\omega)$。

通常，m 非常大，因此这些二项式分布可以用正态分布近似, 用更适合概率统计的 N 来代替 m, 下式成立：

$$\begin{aligned} \mu_0 &= p_0 N, \qquad \sigma_0^2 = p_0(1-p_0)N \approx p_0 N, \\ \mu_w &= p_w N, \qquad \sigma_w^2 = p_w(1-p_w)N \approx p_w N \end{aligned}$$

当我们把计数器的前 2^{k-a} 个为正确密钥候选值时，其符合正态分布：

$$P_s = \int_{-\frac{\mu_0-\mu_q}{\sqrt{\delta_0^2+\delta_q^2}}}^{\infty} \Phi(x)dx$$

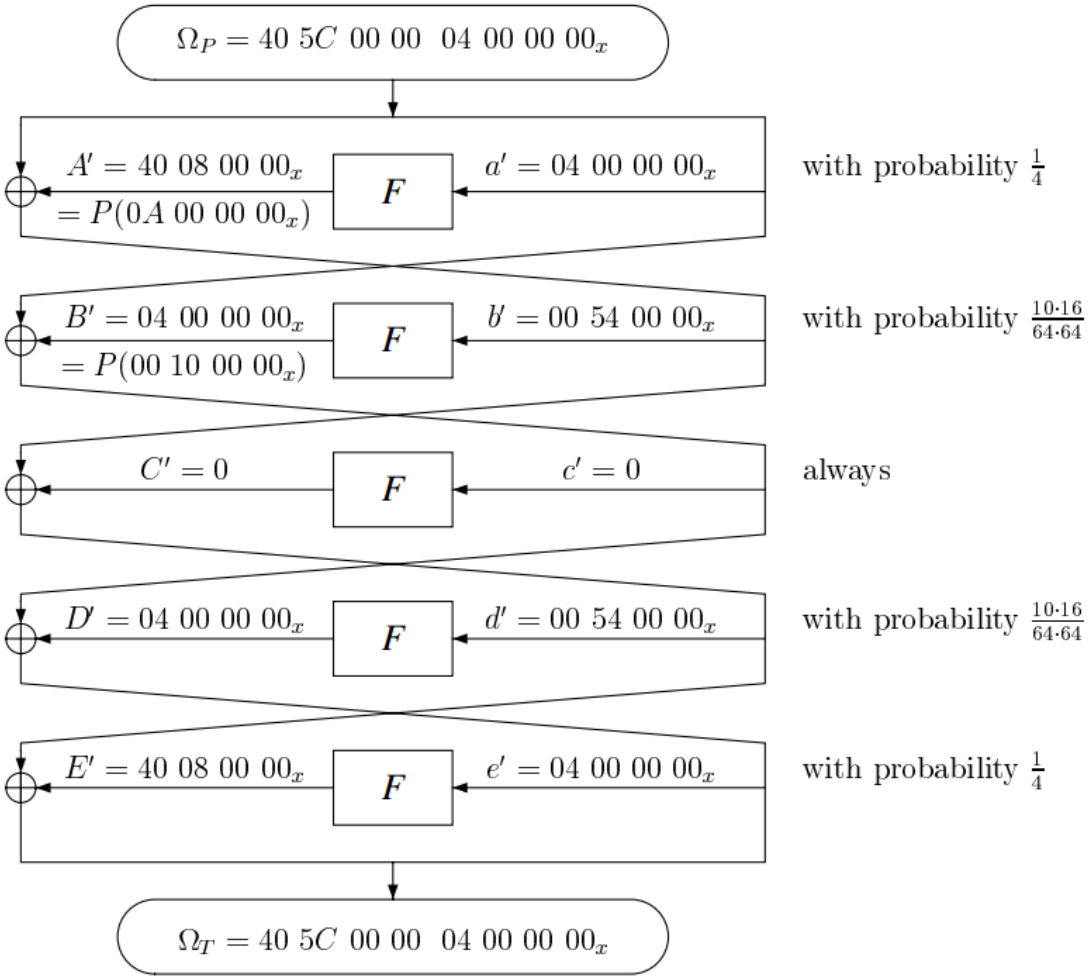
同时，$\mu_q = \mu_\omega + \delta_\omega \Phi^{-1}(1-2^{-a})$ 且 $\delta_q = \frac{\delta_\omega}{\Phi(\Phi^{-1}(1-2^{-a}))} 2^{-\frac{m+1}{2}}$.

当 δ_0^2 远大于 δ_q^2 时，计算该积分，可以得到;

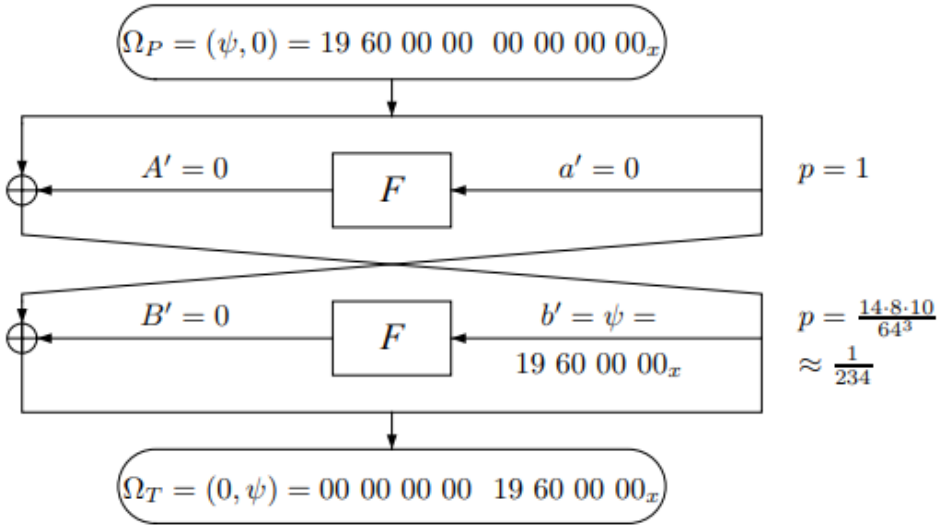
$$P_s = \Phi(\frac{\sqrt{pmS_N} - \Phi^{-1}(1-2^{-a})}{\sqrt{S_N+1}})$$

2.4 寻找 5 轮或更长轮数的迭代差分路线

观察论文中给出的 5 轮差分路线如下图：

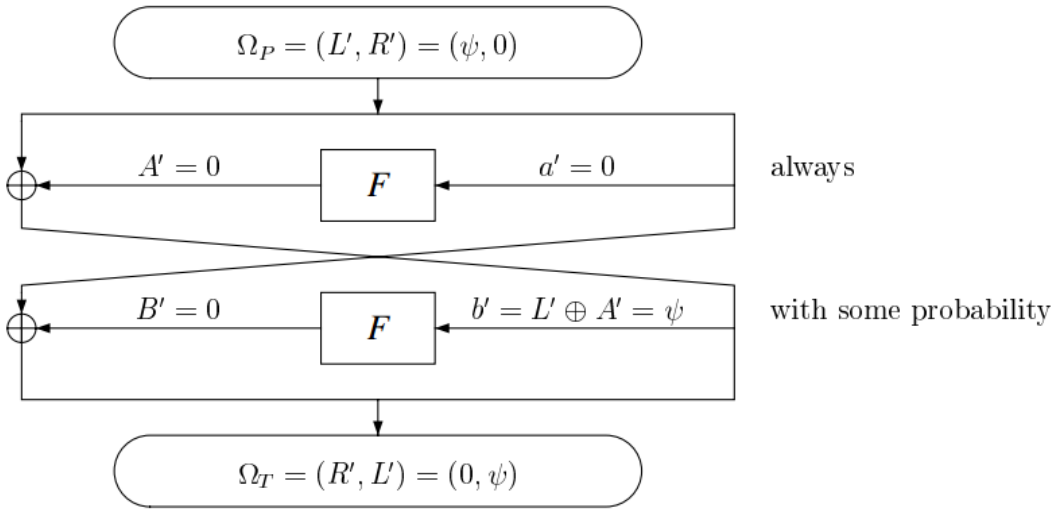


同时论文中给出了一个 2 轮的可重复差分路线：

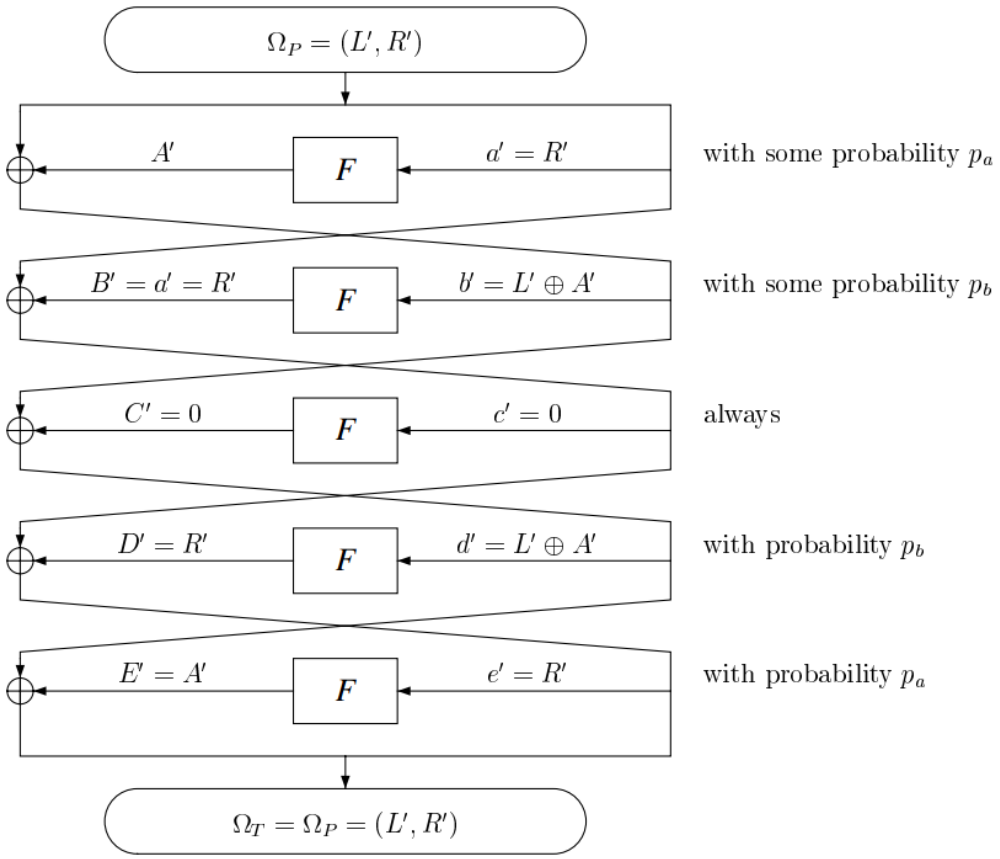


显然的，如果可以构造输入差分 $(\Phi, 0)$, 使其输出差分为 $(0, \Phi)$，则第一轮 0 差分过

F 函数，且 ϕ 第二轮过 F 盒以某概率为 0，即可交换左右支，由此构造 2^n 长的差分路线如下图：



类似的，抽象出论文中 5 轮差分路线的结果如下图：



该路线的关键是使得否可以找到 $b' -> a'$ 和 $a' -> A'$，使得 $b' = L' \oplus A'$ 和 $B' = a' = R'$，此时可以使得第三轮 F 函数的输入为 0 差分，从而使得 5 轮路线呈现对

称的情况，得到输入差分与输出差分相同。

3 实验过程

3.1 3 轮 DES 差分路线的构造

对于 DES 的每个 S 盒，首先求其差分分布表。根据分析，S 盒中间四位的输入差分只能遍历 2，4，6。根据差分分布表内容，求出 s 盒输入差分为 2、4 和 6 时的最优输出差分（这里取第一个最优输出差分）以及相应的对数，部分代码如下，其中 S_box() 以 s 盒序号、s 盒输入作为函数输入，返回 s 盒输出结果。

```
int main(){
    memset(dif_table, 0, sizeof(dif_table));
    //首先构造差分分布表
    for(int v = 0; v < 8; v++){
        for(int d = 0; d < 64; d++){
            for(int i = 0; i < 64; i++){
                dif_table[v][d][S_box(v, i)^S_box(v, i^d)]++;
            }
        }
        printf("dif_S%d\n",v);
        for(int i = 0; i < 64; i++){
            //输入差分只选择2，4和6，即000100，001000，001100，
            if(i!=4 & i!=8 & i!=12) continue;
            printf("input_diff:[%02d]\t",i);
            int maxn = 0, num = 0;
            for(int j = 0; j < 1<<4; j++){
                if(maxn < dif_table[v][i][j]){
                    maxn = dif_table[v][i][j];
                    num = j;
                }
            }
            printf("num:%d\t out_diff:%x\t", maxn, num);
            printf("\n");
        }
        printf("\n");
    }
}
```

运行结果如下

```
dif_S0
input_diff:[04] num:10   out_diff:5
input_diff:[08] num:12   out_diff:3
input_diff:[12] num:14   out_diff:e

dif_S1
input_diff:[04] num:14   out_diff:7
input_diff:[08] num:16   out_diff:a
input_diff:[12] num:14   out_diff:5

dif_S2
input_diff:[04] num:12   out_diff:9
input_diff:[08] num:10   out_diff:3
input_diff:[12] num:12   out_diff:5

dif_S3
input_diff:[04] num:12   out_diff:6
input_diff:[08] num:8    out_diff:5
input_diff:[12] num:8    out_diff:5
```

```
dif_S4
input_diff:[04] num:10   out_diff:6
input_diff:[08] num:10   out_diff:6
input_diff:[12] num:10   out_diff:3

dif_S5
input_diff:[04] num:10   out_diff:b
input_diff:[08] num:16   out_diff:6
input_diff:[12] num:12   out_diff:3

dif_S6
input_diff:[04] num:12   out_diff:b
input_diff:[08] num:12   out_diff:a
input_diff:[12] num:14   out_diff:c

dif_S7
input_diff:[04] num:12   out_diff:7
input_diff:[08] num:12   out_diff:f
input_diff:[12] num:10   out_diff:5
```

接下来根据 s 盒的最优输出差分求解三轮 Des 输入差分的左半部分。将上一步得到的 s 盒最优输出差分结果经过 P 置换，得到 F 函数变换的结果（即输入差分右部），将该结果与右半部分链接即得到 3 轮差分路线的头尾差分，部分代码如下，其中 p() 函数以 8 位 16 进制数作为输入，输出 P 置换后的结果。

```
l = [
    [2,10,0x50000000],
    [4,12,0x30000000],
    [6,14,0xe0000000],
    [2,14,0x07000000],
    [4,16,0x0a000000],
    [6,14,0x05000000],
    [2,12,0x00900000],
    [4,10,0x00300000],
    [6,12,0x00500000],
    [2,12,0x00060000],
    [4,8 ,0x00050000],
    [6,8,0x00050000],
    [2,10,0x00006000],
    [4,10,0x00006000],
    [6,10,0x00003000],
    [2,10,0x00000b00],
```

```
    [4,16,0x00000600],
    [6,12,0x00000300],
    [2,12,0x000000b0],
    [4,12,0x000000a0],
    [6,14,0x000000c0],
    [2,12,0x00000007],
    [4,12,0x0000000f],
    [6,10,0x00000005]
]

for i in range(len(l)):
    x = l[i]
    print(x[1],'/64',p(x[2]),'0'*(i//3)+hex(x[0])[2:]+'0'*(7-(i//3)))
```

代码运行结果如下，每条三轮差分路线的头尾都是右侧的 16 位十六进制数，概率为左侧分数值的平方。

```
10 /64 00008002 20000000
12 /64 00000202 40000000
14 /64 00808200 60000000
14 /64 40004010 02000000
16 /64 40080000 04000000
14 /64 00004010 06000000
12 /64 04000100 00200000
10 /64 04000004 00400000
12 /64 04010000 00600000
12 /64 00401000 00020000
8 /64 80001000 00040000
8 /64 80001000 00060000
10 /64 00040080 00002000
10 /64 00040080 00004000
10 /64 20000080 00006000
10 /64 10202000 00000200
16 /64 00200008 00000400
12 /64 00202000 00000600
12 /64 02000401 00000020
12 /64 00000401 00000040
14 /64 00100001 00000060
12 /64 00020820 00000002
12 /64 08020820 00000004
10 /64 00000820 00000006
```

3.2 验证差分路线概率

根据题目要求，首先定义随机生成明文的函数，利用 mt19937 生成 64 位随机数，代码如下

```
uint64_t generateRandomInput()
{
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<uint64_t> dis;
    return dis(gen);
}
```

在主函数中定义各项参数,目标差分定义为 0080820060000000,利用 std::vector<std::pair> 来存储明文对，在 pair 容器中储存 5400 对符合输入差分的明文，随后随机生成十个密钥，对每个密钥都遍历一遍所有明文，计算输出差分符合条件的频率并输出，代码如下

```
int main(int argc, const char* argv[]) {
    int round = 3;  // 加密轮数
    int n = 5400; // 要生成的明文对数量
    uint64_t desiredXOR = 0x0080820060000000;//目标异或值
    std::cout << std::hex;//以16进制输出
    std::cout << "aim: " << desiredXOR << std::endl;
    std::cout << std::dec;//以16进制输出
    std::vector<std::pair<uint64_t, uint64_t>> plaintextPairs;//存储明文对的容器

    while (plaintextPairs.size() < n) //寻找明文对
    {
        uint64_t x = generateRandomInput();
        uint64_t y = x ^ desiredXOR;
        if (x ^ y == desiredXOR)
        {
            plaintextPairs.push_back(std::make_pair(x, y));
        }
    }
    for (int j = 1; j <= 10; j++)
    {
        int num = 0;//记录符合输出差分的数量
```

```
        uint64_t key = generateRandomInput();//随机取密钥
        for (const auto& pair : plaintextPairs)
        {
            uint64_t result1 = des(pair.first, key, round, 'e'); //e:表示加密  d:表示解密
            uint64_t result2 = des(pair.second, key, round, 'e');
            if ((result1 ^ result2) == desiredXOR)
            {
                num++;
            }
        }
        double ratio = static_cast<double>(num) / n * 100;
        std::cout << "key" << j << " Success rate: " << std::fixed << std::setprecision(2)
      << ratio << "%" << std::endl;// 设置输出为固定小数点后两位的浮点数，并加上百分号
    }
    exit(0);
}
```

代码运行结果如下

```
aim: 80820060000000
key1 Success rate: 4.80%
key2 Success rate: 4.19%
key3 Success rate: 4.65%
key4 Success rate: 5.33%
key5 Success rate: 4.87%
key6 Success rate: 5.00%
key7 Success rate: 4.96%
key8 Success rate: 4.54%
key9 Success rate: 4.54%
key10 Success rate: 5.11%
```

发现频率确实在 4.785% 附近，进一步增大明文对数量，结果如下，与 4.785% 距离更小，符合理论预期。

```
aim: 80820060000000
key1 Success rate: 4.76%
key2 Success rate: 4.94%
key3 Success rate: 4.77%
key4 Success rate: 4.77%
key5 Success rate: 4.78%
key6 Success rate: 4.74%
key7 Success rate: 4.85%
key8 Success rate: 4.79%
key9 Success rate: 4.88%
key10 Success rate: 4.82%
```

3.3 6 轮 DES 算法的密钥恢复攻击

3.3.1 轮密钥恢复

利用实验准备中的分析，分别对两条差分路线进行差分攻击，首先定义 mian 函数，对读取的两条符合差分的明文做异或运算，计算 ms 数组值，其值表示对应的密钥是否符合条件（根据给定的明文对应的索引 i 和 S 盒的索引 j），然后调用 get_key() 函数分析密钥组合的频率进而得到可能密钥，代码如下：

```
int main() {
    freopen("1.txt","r",stdin);
    uint32_t F0,F1;
    for(int i = 0; i < 300; i++){
        scanf("%lx %lx",&F0,&f0[i]);
        scanf("%lx %lx",&F1,&f1[i]);
        F[i] = F0 ^ F1 ^ c;
    }
    for(int i = 0; i < 300; i++){
        for(uint16_t j = 0; j < 5; j++){
            get_ms(i,j);
        }
    }
    get_key();
    return 0;
}
```

为了实现差分攻击，要实现 DES 的 E 扩展和 S 盒等组件，如下代码：

```
uint32_t rr(uint64_t s_input, uint64_t sub_key, int num){
    char row = 0, column = 0;
    uint32_t s_output = 0;
    uint32_t f_function_res = 0;
    s_input = s_input ^ sub_key;
    for (int j = num; j < num+1; 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 |= (uint32_t) (S[j][16*row + column] & 0x0f);
    s_output <<= 4*(7-num);
    }
    f_function_res = 0;
    for (int j = 0; j < 32; j++) {
        f_function_res <<= 1;
        f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;
    }
    // printf("%lx\n",f_function_res);
    return f_function_res;
}
uint64_t E_ex(uint32_t R){
    uint64_t s_input = 0;
    for (int j = 0; j< 48; j++) {
        s_input <<= 1;
        s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);
    }
    return s_input;
}
```

定义 get_ms(int i, int j) 函数，该函数用于计算 ms 数组中的元素值，其表示该明文在六轮差分攻击中数组索引对应的 S 盒是否有密钥符合差分条件。根据给定的明文对应的索引 i 和 S 盒的索引 j，计算得到对应的 s_output，并将与给定的 F 数组进行比较，。具体地，函数 get_ms 按照以下步骤计算得到 ms 数组的元素值：

1. 根据输入的明文对应的索引 i，从数据文件中读取该明文对应的 f0 值和 f1 值。

2. 将 f0 值和 f1 值经过 E 扩展函数得到扩展后的 f0 和 f1 值。

3. 遍历所有可能的 6 位 DES 密钥，对每个可能的 S 盒密钥进行差分分析，如果分析得该密钥符合该 S 盒差分，则记录该 ms[i][j]

通过上述步骤，筛选出哪些明密文对被 S 盒和可能的密钥匹配过，同时记录被匹配密钥的编号（通过 ms 元素 1/0 比特串为 1 的位置），得到 ms 表，代码如下图所示：

```
void get_ms(int i, int j){
    int t[5] = {0,1,3,4,5},num = 0;
    num = t[j];
```

```
    uint32_t mask = get_mask(num);
    // printf("%lx\n",mask);
    uint64_t cnt = 0;
    for(uint16_t key = 0; key < 63; key ++){
        cnt = 0;
        uint64_t f_0 = E_ex(f0[i]) & (uint64_t)((uint64_t)0b111111<<((7-num)*6));
        uint64_t f_1 = E_ex(f1[i]) & (uint64_t)((uint64_t)0b111111<<((7-num)*6));
        uint64_t sub_key = (uint64_t)((uint64_t)key<<(7-num)*6);
        // printf("%llx %llx %llx\n",f_0,f_1,sub_key);
        uint32_t F_0 = rr(f_0, sub_key, num);
        uint32_t F_1 = rr(f_1, sub_key, num);
        uint32_t F_x = F_0^F_1;
        // printf("%lx %lx %d\n",F[i]&F_x, F_x, F[i]&F_x==F_x);
        // printf("%lx\n",(F_x&mask) == F_x)
        if((F[i]&mask) == F_x){
            cnt = 1;
        }
        ms[i][j] |= cnt;
        ms[i][j] = ms[i][j] << 1;
    }
    return ;
}
```

最后实现 void get_key() 函数，首先对输入数据进行处理，剔除无效的明文对 (ms 值为 0)，并统计满足的明文对个数：

```
int cnt = 0;
for(int i = 0; i < 300; i++)
{
        int flag = 0;
        for(int j = 0; j < 5; j++)
        {
            if(ms[i][j]==0){
                flag = 1;
                break;
            }
        }
```

```
        if(!flag){
            for(int j = 0; j < 5; j++){
                ms[cnt][j] = ms[i][j];
            }
            cnt++;
        }
    }
```

对于每个同时满足五个 S 盒的明密文对，执行 get_cnt 函数，计算其存在差分特征的比特位，返回整个数据集中出现次数最多的差分特征的数量 maxn，以及对应的值 ans[]，该值即为最可能的 S 盒密钥，代码如下：

```
int get_cnt(int i, int num){
    for(int j = 0; j < 63; j++){
        if(((ms[i][num]>>(63-j))&(uint64_t)1) == 1){
            tmp[num] = j;
            // printf("ok");
            if(num == 4){
                // printf("ok\n");
                uint64_t mask[5];
                for(int k = 0; k < 5; k++) mask[k] = (uint64_t)1<<(63-tmp[k]);
                int cnt = 0;
                for(int k = 0; k < n; k++){
                    int flag = 0;
                    for(int l = 0; l < 5; l++){
                        if((mask[l] & ms[k][l]) == 0){
                            flag = 1;
                            break;
                        }
                    }
                    if(!flag) cnt++;
                }
                if(cnt>maxn){
                    maxn = cnt;
                    for(int k = 0; k < 5; k++){
                        ans[k] = tmp[k];
                    }
```

```
            }
        }else{
            get_cnt(i,num+1);
        }
    }
}
}
```

另一条差分路线同理，通过修改 get_ms 函数中 5 个 S 盒的索引 t[], 即可计算。通过 300 个数据进行选择明文攻击，得到第六轮 8 个 S 盒密钥分别为

$$3B \quad 21 \quad ** \quad 36 \quad 14 \quad 3C \quad 05 \quad 30 \quad 2S$$

3.3.2 主密钥恢复

由上文得到第六轮的部分轮密钥，下面需要恢复其主密钥。

对于每个猜测的完整第六轮轮密钥，使用 ReKeygen 函数对其进行逆扩展：使用函数 RePC1 对初始密钥进行置换操作，得到 56 位的中间结果 combined。使用函数 Rotate 对 combined 进行循环左移操作，生成每轮所需的子密钥。使用函数 RePC2 对每轮生成的子密钥进行 pc2 置换，得到 48 位的轮密钥。代码如下：

```
Bytes RePC1(Bytes l, Bytes r)
{
    int pc1[56] = {
        56, 48, 40, 32, 24, 16,  8,
         0, 57, 49, 41, 33, 25, 17,
         9,  1, 58, 50, 42, 34, 26,
        18, 10,  2, 59, 51, 43, 35,
        62, 54, 46, 38, 30, 22, 14,
         6, 61, 53, 45, 37, 29, 21,
        13,  5, 60, 52, 44, 36, 28,
        20, 12,  4, 27, 19, 11,  3
    };
    int r_pc1[64] = { 0 };
    ReBox(pc1, r_pc1, 56, 64);

    Bytes combined={};
```

```
17
18      combined |= l;
19      combined <<= 28;
20      combined |= r;
21      return Permutate(combined, r_pc1, 56, 64);
22  }
23
24  Bytes RePC2(Bytes key)
25  {
26      int pc2[48] = {
27          13, 16, 10, 23,  0,  4,
28           2, 27, 14,  5, 20,  9,
29          22, 18, 11,  3, 25,  7,
30          15,  6, 26, 19, 12,  1,
31          40, 51, 30, 36, 46, 54,
32          29, 39, 50, 44, 32, 47,
33          43, 48, 38, 55, 33, 52,
34          45, 41, 49, 35, 28, 31
35      };
36      int r_pc2[56] = { 0 };
37      ReBox(pc2, r_pc2, 48, 56);
38      return Permutate(key, r_pc2, 48, 56);
39  }
```

对于缺失位，定义一个 mask 变量，通过 mask 来确定每轮中产生子密钥的位置。将 mask_l 和 mask_r 分别与 l 和 r 对应的部分进行循环左移，得到当前轮的 mask。定义 mask_list 数组用于记录 mask 中为 1 的位置，并根据 mask_list 更新主密钥 findkey 数组，代码如下：

```
1  final_key = RePC2(final_key);
2      int loss[8] = { 8, 17, 21, 24, 34, 37, 42, 53 };
3      Bytes mask = 0;
4      for (int i = 0; i < 8; i++)
5      {
6          mask = mask | 1ULL << (55 - loss[i]);
7      }
8      Bytes l = final_key >> 28;
```

```
    Bytes r = final_key ^ l << 28;
    Bytes mask_l = mask >> 28;
    Bytes mask_r = mask ^ mask_l << 28;
    int offset[16] = {
        1,  1,  2,  2,  2,  2,  2,  2,
        1,  2,  2,  2,  2,  2,  2,  1
    };

    for (int i = round - 1; i >= 0; i--)
    {
        l = Rotate(l, 28 - offset[i]);
        r = Rotate(r, 28 - offset[i]);
        mask_l = Rotate(mask_l, 28 - offset[i]);
        mask_r = Rotate(mask_r, 28 - offset[i]);
    }

    Bytes key = RePC1(l, r);
    mask = RePC1(mask_l, mask_r);
    int j = 0;
    for (int i = 63; i >= 0; i--)
    {
        if (mask & 1)
        {
            mask_list[j] = i;
            j++;
        }
        mask >>= 1;
    }
```

运行得到缺失的位置信息以及遍历可能的主密钥如下图所示：

```
key:001110*1*11100100010101111*0010*01**10001*11000100101010010*000
KEY:0011100111110010001010111111001000110100011110001001010100100000
j:8
61 43 37 36 33 28 9 7
39722be224589520
3b722be224589520
39f22be224589520
3bf22be224589520
39722bf224589520
3b722bf224589520
39f22bf224589520
3bf22bf224589520
39722be2a4589520
3b722be2a4589520
39f22be2a4589520
3bf22be2a4589520
```

即缺少主密钥中的第 7，9，28，33，36，37，43，61 位，通过枚举缺失的 8bit 主密钥，计算特定明文所对应的密文，与实验材料提供的 dec_enc4 程序进行对比，编写代码如下，通过设定一对来自样例加密代码的明密文对，循环加密得到主密钥：

```
1  void FIND()
2  {
3      for (int i = 0; i < 256; i++)
4      {
5          decconvertToBinary(i, rep);
6          for(int j=0;j<8;j++)
7            findkey[mask_list[j]] = rep[j];
8          long long result = binaryStringToLongLong(findkey);
9          std::memcpy(&result, &result, sizeof(uint64_t));
10         uint64_t test = 0x289b47de9785d611;
11         uint64_t in = 0x1122334455667788;
12         uint64_t res = des(in, result, 6, 'e'); //e:表示加密  d:表示解密
13         cout << hex << result << endl;
14         if (res == test)
15         {
16             cout << "find it! The master key is:" << result << endl;
17             return ;
18         }
19     }
20     cout << "no!";
21 }
```

得到以下结果：

恢复出主密钥为：

$$39F22BF234789520$$

第六轮完整轮密钥为：

$$EE1A3653C170$$

3.3.3 恢复轮密钥的成功率与复杂度分析

对于第一条差分路线 (差分概率为 0.0625)，由公式 $P_s = \Phi(\frac{\sqrt{pmS_N} - \Phi^{-1}(1-2^{-a})}{\sqrt{S_N+1}})$，$S_N = \frac{p*2^k}{\eta*\epsilon}$。易知 p=$2^{-4}$，m=300，k=a=30。由于 S 盒是 6 进 4 出的查表变换，所以差分分布表的平均值为 $2^{6-4} = 4$，而一次攻击猜测 5 个 S 盒对应的密钥，所以 $\eta = 4^5 = 2^{10}$。

在我们代码的去噪阶段，S 盒差分分布表中值为 0 的概率约为 0.02，从而每个 S 盒的过滤强度约为 0.8，5 个 S 盒的过滤强度约为 $\epsilon = 0.8^5 = 0.3277$，$S_N \approx 2^{11} * 10^2$，如果不去噪，则 $S_N = 2^{16}$。

选择去噪后的信噪比，带入公式有，成功率 $P_s \approx 0.999991$，则可以相信恢复出的密钥为正确轮密钥。

对于轮密钥恢复的数据复杂度易知为 600 个明文（需要选取 300 对明文)。

对于时间复杂度分析，在采样阶段进行 600 次 6 轮加密运算，由于本次实验中使用参考标程，故看作查找表不计算时间复杂度；在去噪阶段，计算 ms 数组需要 $2^6 * 5 * m$ 次异或和 $2^7 * 5 * m$ 次加密；在恢复密钥阶段，对去噪阶段后剩下的明密文对进行比较，约需要 5m 次比较操作和 $\epsilon * m * 2^6$ 次查 ms 表操作 (get_cnt 操作)，故总的时间复杂度约 $2^7 * 5 * 300$ 次 6 轮 des 加密运算。

对于存储复杂度，本代码中存储明文及其差分的空间大小为 $2^5 * 3 * 300$Bit，存储 ms 表需要 $2^6 * 5 * 300$Bit，总的存储复杂度约为 $2^6 * 5 * 300$Bit。

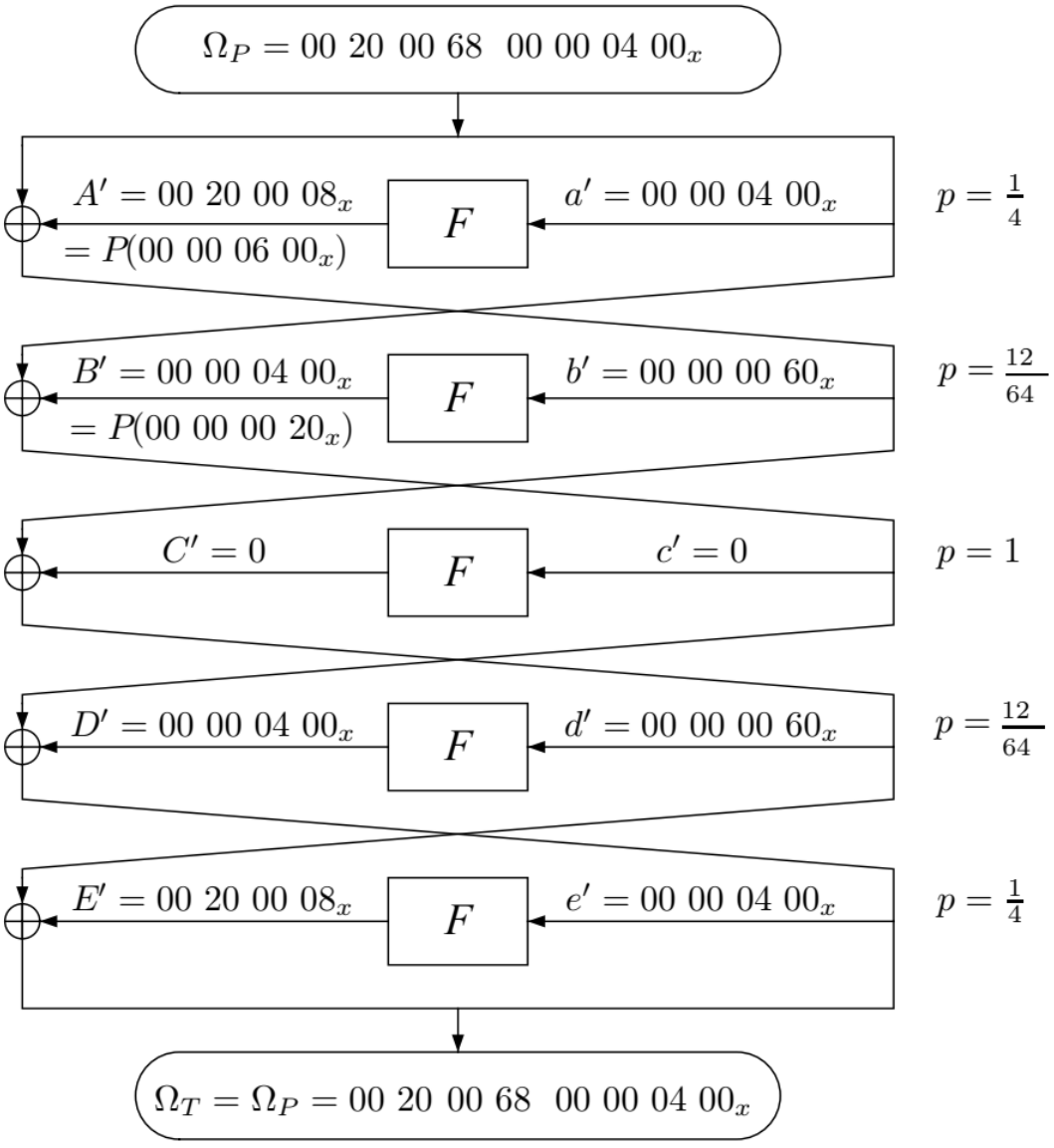
3.3.4 恢复主密钥的成功率与复杂度分析

对于恢复主密钥，由于我们采取枚举 6+8bit 的强力攻击方法，故其成功率为 1。

在恢复主密钥的过程中，对于时间复杂度，可以将其看作暴力枚举 2^{8+6} 个主密钥以及 2^{8+6} 次加密计算，故其时间复杂度约为 2^{14} 次 des 加密。对于空间复杂度，需要存储 2^{8+6} 个主密钥，即 $2^{8+6+6} = 2^{20}$bit 数据。

3.4 5 轮或更长轮数的迭代差分路线

按照论文中给出的 5 轮差分路线构造方法，检索 DES 的差分分布表，寻求中间值 $b' -> a'$ 和 $a' -> A'$，得到与论文中不同的第二条 5 轮差分路径（以 $\frac{9}{4096}$ 的概率成立）：



参考文献

[1] 王美琴等. 密码分析学 [M]. 2023.

[2] Biham E, Shamir A. Dierential Cryptanalysis of DES like Cryptosystems[J]. 1990.

[3] Ali Aydın Selçuk. On Probability of Success in Linear and Differential Cryptanalysis[J]. 2007.

A 附录

A.1 t1_diff_table 主要代码

```
uint8_t S_box(int v,int s_input){
    char row, column;
    row = (s_input & 1)|(s_input >> 4 & 2);
    column = (s_input >> 1 & 15);
    return S[v][16*row + column];
}

int main(){
    memset(dif_table, 0, sizeof(dif_table));
    for(int v = 0; v < 8; v++){
        for(int d = 0; d < 64; d++){
            for(int i = 0; i < 64; i++){
                dif_table[v][d][S_box(v, i)^S_box(v, i^d)]++;
            }
        }
        printf("dif_S%d\n",v);
        for(int i = 0; i < 64; i++){
            if(i!=4 & i!=8) continue;
            printf("[%02d]\t",i);
            int maxn = 0, num = 0;
            for(int j = 0; j < 1<<4; j++){
                if(maxn < dif_table[v][i][j]){
                    maxn = dif_table[v][i][j];
                    num = j;
                }
            }
            printf("%d\t%x\t", maxn, num);
            printf("\n");
        }
        printf("\n");
    }
}
```

A.2 t2_test_diff 主要代码

```
int main(int argc, const char* argv[]) {
    std::cout << std::hex;//以16进制输出
    int round = 3;  // 加密轮数
    int n = 10000; // 要生成的明文对数量
    int num = 0;//记录符合输出差分的数量
    uint64_t desiredXOR = 0x0020000800000400;//目标异或值
    uint64_t key = 0x0123456789ABCDEF;    // 主密钥
    std::cout << "aim: " << desiredXOR<<std::endl;
    std::vector<std::pair<uint64_t, uint64_t>> plaintextPairs;//存储明文对的容器
    while (plaintextPairs.size() < n) //寻找明文对
    {
        uint64_t x = generateRandomInput();
        uint64_t y = x ^ desiredXOR;
        if (x ^ y == desiredXOR) {
            plaintextPairs.push_back(std::make_pair(x, y));
        }
    }
    for (const auto& pair : plaintextPairs)
    {
        uint64_t result1 = des(pair.first, key, round, 'e'); //e:表示加密  d:表示解密
        uint64_t result2 = des(pair.second, key, round, 'e'); //e:表示加密  d:表示解密
        if ((result1 ^ result2) == desiredXOR)
        {
            num++;
            std::cout << "plaintext pair: " << pair.first << " and " << pair.second << std::
      endl;
            std::cout << "p: " << (pair.first ^ pair.second);
            std::cout << std::endl;
            std::cout <<"c: " << (result1 ^ result2)<<std::endl;
        }
     }
    double ratio = static_cast<double>(num) / n * 100;
    std::cout << std::fixed << std::setprecision(2) << ratio << "%" << std::endl;
    exit(0);
}
```

A.3 t3_ 利用第一组 S 盒进行六轮差分攻击

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include<iostream>
#include <random>
#include <algorithm>
#include <bitset>
#include"Des.h"
typedef unsigned char Byte;
typedef unsigned long long Bytes;
using namespace std;
char findkey[64];
char rep[8];
int mask_list[64] = { 0 };
Bytes KEY = 0x39F22BF234789520;
uint64_t E_ex(uint32_t R){
    uint64_t s_input = 0;

    for (int j = 0; j< 48; j++) {
        s_input <<= 1;
        s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);

    }
    return s_input;
}

uint32_t get_mask(int num){
    /* 8 bits */
    char row = 0, column = 0;
    uint32_t s_output = 0;
    uint32_t f_function_res = 0;

    s_output = (uint32_t)((uint32_t)0b1111 <<(uint32_t)((7-num)*4));

    f_function_res = 0;

```

```
37      for (int j = 0; j < 32; j++) {
38
39          f_function_res <<= 1;
40          f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;
41
42      }
43      // printf("%lx\n",f_function_res);
44      return f_function_res;
45  }
46
47  uint32_t rr(uint64_t s_input, uint64_t sub_key, int num){
48      /* 8 bits */
49      char row = 0, column = 0;
50      uint32_t s_output = 0;
51      uint32_t f_function_res = 0;
52
53      s_input = s_input ^ sub_key;
54      for (int j = num; j < num+1; j++) {
55      // 00 00 RCCC CR00 00 00 00 00 00 s_input
56      // 00 00 1000 0100 00 00 00 00 00 row mask
57      // 00 00 0111 1000 00 00 00 00 00 column mask
58
59      row = (char) ((s_input & (0x0000840000000000 >> 6*j)) >> 42-6*j);
60      row = (row >> 4) | row & 0x01;
61
62      column = (char) ((s_input & (0x0000780000000000 >> 6*j)) >> 43-6*j);
63
64      s_output |= (uint32_t) (S[j][16*row + column] & 0x0f);
65      s_output <<= 4*(7-num);
66
67      }
68      f_function_res = 0;
69
70      for (int j = 0; j < 32; j++) {
71
72          f_function_res <<= 1;
73          f_function_res |= (s_output >> (32 - P[j])) & LB32_MASK;
74
```

```
    }
    return f_function_res;
}

uint32_t F[300],f0[300],f1[300];
uint32_t c = 0x04000000;

int get_cnt(int num, uint16_t key){
    uint32_t mask = get_mask(num);
    int cnt = 0;
    for(int i = 0; i < 300; i++){
        uint64_t f_0 = E_ex(f0[i]) & (uint64_t)((uint64_t)0b111111<<((7-num)*6));
        uint64_t f_1 = E_ex(f1[i]) & (uint64_t)((uint64_t)0b111111<<((7-num)*6));
        uint64_t sub_key = (uint64_t)((uint64_t)key<<(7-num)*6);
        uint32_t F_0 = rr(f_0, sub_key, num);
        uint32_t F_1 = rr(f_1, sub_key, num);
        uint32_t F_x = F_0^F_1;
        if((F[i]&mask) == F_x){
            cnt++;
        }
    }
    return cnt;
}
int main() {
    freopen("1.txt","r",stdin);
    uint32_t F0,F1;
    for(int i = 0; i < 300; i++){
        scanf("%lx %lx",&F0,&f0[i]);
        scanf("%lx %lx",&F1,&f1[i]);
        F[i] = F0 ^ F1 ^ c;
    }
    int t[5] = {1,4,5,6,7},num = 0;
    for(int i = 0; i < 5; i++){
        num = t[i];
        int maxn = 0,key;
        for(uint16_t j = 0; j < 64; j++){
            int cnt = get_cnt(num,j);
            if(maxn < cnt){
```

```
113                 maxn = cnt;
114                 key = j;
115             }
116         }
117         printf("%lx, %d\n",key,maxn);
118     }
119     return 0;
120 }
```

A.4 t3_ 恢复主密钥 _ 主要代码

```
1  Bytes Rotate(Bytes a, int n)
2  {
3      for (int i = 0; i < n; i++)
4          a = (a << 1 | (a >> 27 & 1)) & ((1 << 28) - 1);
5      return a;
6  }
7  void convertToBinary(long long key, char* binaryKey)
8  {
9      for (int i = 0; i < 64; i++) {
10         binaryKey[i] = (key >> (63 - i)) & 1 ? '1' : '0';
11     }
12 }
13 void decconvertToBinary(long long key, char* binaryKey)
14 {
15     for (int i = 0; i < 8; i++) {
16         binaryKey[i] = (key >> (7 - i)) & 1 ? '1' : '0';
17     }
18 }
19 Bytes Distribute(Bytes a, int b[], int m, int n) {
20     Bytes c = 0;
21     for (int i = m - 1; i >= 0; i--)
22     {
23         c = c | ((a & 1) << (n - b[i] - 1));
24         a >>= 1;
25     }
```

```
26      return c;
27  }
28  void ReBox(int a[], int b[], int m, int n)
29  {
30      for (int i = 0; i < n; i++)
31      {
32          b[i] = -1;
33      }
34      for (int i = 0; i < m; i++)
35      {
36          b[a[i]] = i;
37      }
38  }
39  Bytes Permutate(Bytes a, int b[], int m, int n) {
40      Bytes c = 0;
41      for (int i = 0; i < n; i++) {
42          c = c << 1 | (a >> (m - b[i] - 1) & 1);
43      }
44      return c;
45  }
46  Byte Permutate(Byte a, Byte b[], int m, int n) {
47      Byte c = 0;
48      for (int i = 0; i < n; i++) {
49          c = c << 1 | (a >> (m - b[i] - 1) & 1);
50      }
51      return c;
52  }
53  Bytes RePC1(Bytes l, Bytes r)
54  {
55      int pc1[56] = {
56          56, 48, 40, 32, 24, 16,  8,
57           0, 57, 49, 41, 33, 25, 17,
58           9,  1, 58, 50, 42, 34, 26,
59          18, 10,  2, 59, 51, 43, 35,
60          62, 54, 46, 38, 30, 22, 14,
61           6, 61, 53, 45, 37, 29, 21,
62          13,  5, 60, 52, 44, 36, 28,
63          20, 12,  4, 27, 19, 11,  3
```

```
64      };
65      int r_pc1[64] = { 0 };
66      ReBox(pc1, r_pc1, 56, 64);
67      Bytes combined={};
68      combined |= l;
69      combined <<= 28;
70      combined |= r;
71      //cout << "num:" << combined << " ";
72      return Permutate(combined, r_pc1, 56, 64);
73  }
74  Bytes RePC2(Bytes key)
75  {
76      int pc2[48] = {
77          13, 16, 10, 23,  0,  4,
78           2, 27, 14,  5, 20,  9,
79          22, 18, 11,  3, 25,  7,
80          15,  6, 26, 19, 12,  1,
81          40, 51, 30, 36, 46, 54,
82          29, 39, 50, 44, 32, 47,
83          43, 48, 38, 55, 33, 52,
84          45, 41, 49, 35, 28, 31
85      };
86      int r_pc2[56] = { 0 };
87      ReBox(pc2, r_pc2, 48, 56);
88      return Permutate(key, r_pc2, 48, 56);
89  }
90  Bytes ReKeygen(Bytes final_key, int round)
91  {
92      final_key = RePC2(final_key);
93      int loss[8] = { 8, 17, 21, 24, 34, 37, 42, 53 };
94      Bytes mask = 0;
95      for (int i = 0; i < 8; i++) {
96          mask = mask | 1ULL << (55 - loss[i]);
97      }
98      Bytes l = final_key >> 28;
99      Bytes r = final_key ^ l << 28;
100     Bytes mask_l = mask >> 28;
101     Bytes mask_r = mask ^ mask_l << 28;
```

```
102     int offset[16] = {
103         1,  1,  2,  2,  2,  2,  2,  2,
104         1,  2,  2,  2,  2,  2,  2,  1
105     };
106     for (int i = round - 1; i >= 0; i--) {
107         l = Rotate(l, 28 - offset[i]);
108         r = Rotate(r, 28 - offset[i]);
109         mask_l = Rotate(mask_l, 28 - offset[i]);
110         mask_r = Rotate(mask_r, 28 - offset[i]);
111     }
112     Bytes key = RePC1(l, r);
113     mask = RePC1(mask_l, mask_r);
114     int j = 0;
115     for (int i = 63; i >= 0; i--)
116     {
117         if (mask & 1)
118         {
119             mask_list[j] = i;
120             j++;
121         }
122         mask >>= 1;
123     }
124     convertToBinary(key, findkey);
125     cout << "key:" << bitset<sizeof(key) * 8>(key) << endl;
126     cout << "key:";
127     for (int i = 0; i < 64; i++)
128     {
129         int ch = 0;
130         for (int k = 0; k < j; k++)
131         {
132             if (i == mask_list[k])
133                 findkey[i]='*';
134         }
135         if ((i+1) % 8 == 0 && i >= 3&&i!=7&&i!=23&&i!=55)
136             findkey[i] = '0';
137     }
138     findkey[7] = '1';
139     findkey[23] = '1';
```

```
140     findkey[55] = '1';
141     for (int i = 0; i < 64; i++)
142         cout << findkey[i];
143     cout << endl;
144     cout << "KEY:" << bitset<sizeof(KEY) * 8>(KEY)<<endl;
145     cout << "j:" << j << endl;
146     for (int i = 0; i < j; i++)
147         cout <<dec<< mask_list[i]+1 << " ";
148     cout << endl;
149     Bytes guess_key = 0;
150     return 0;
151 }
152 long long binaryStringToLongLong(const char* binaryStr)
153 {
154     long long result = 0;
155 
156     for (int i = 0; i < 64; i++)
157     {
158         result <<= 1;  // 左移一位，为下一位二进制数腾出空间
159         result += binaryStr[i] - '0';  // 将字符 '0' 或 '1' 转换为对应的数字
160     }
161     return result;
162 }
163 void FIND()
164 {
165     for (int i = 0; i < 256; i++)
166     {
167         decconvertToBinary(i, rep);
168         for(int j=0;j<8;j++)
169           findkey[mask_list[j]] = rep[j];
170         long long result = binaryStringToLongLong(findkey);
171         std::memcpy(&result, &result, sizeof(uint64_t));
172         uint64_t test = 0x289b47de9785d611;
173         uint64_t in = 0x1122334455667788;
174         uint64_t res = des(in, result, 6, 'e'); //e:表示加密  d:表示解密
175         cout << hex << result << endl;
176         if (res == test)
177         {
```

```
178             cout << "find it! The master key is:" << result << endl;
179             return ;
180         }
181     }
182     cout << "no!";
183 }
184 int main()
185 {
186     ReKeygen(0xee1a3653c170, 6);
187     //Can be modified to traverse the complete round key
188     FIND();
189     return 0;
190 }
```