# 密码学基础实验报告——SPN 差分密码分析

## 1. 实验简述

SPN（Substitution-Permutation Network，也叫代换-置换网络）是一种密码学中常用的分组密码结构。它将明文分成固定长度的块，然后通过一系列的替换（Substitution）和置换（Permutation）操作来进行加密和解密。SPN 结构通常包括四个主要步骤：置换层、混淆层、密钥加轮和密钥混合。

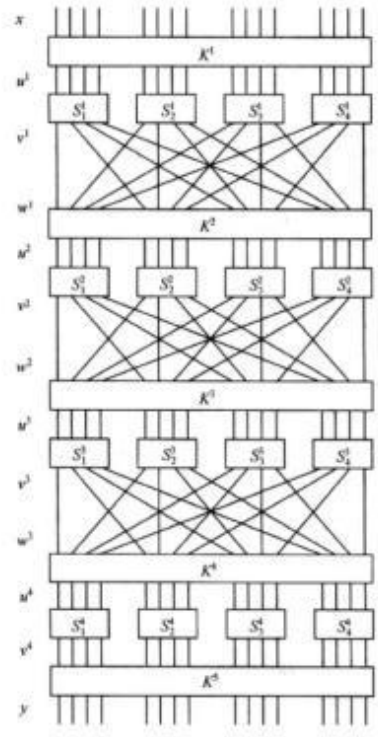
置换层(Substitution)：将输入的明文块进行替换操作，通常使用 S 盒(Substitution Box)来进行非线性替换，以增加密码算法的复杂性和安全性。

混淆层（Permutation）：在置换操作之后，进行置换操作，打乱数据的顺序。这个步骤旨在增加密码算法的扩散性，使得每个位的变化都能影响到尽可能多的输出位。

密钥加轮（Key Addition Rounds）：将轮密钥（Round Key）与置换和混淆后的数据进行按位异或操作，以引入密钥信息并增强密码算法的安全性。

密钥混合（Key Mixing）：在每个轮次中，都会使用不同的轮密钥来进行密钥加轮操作，以确保加密过程中的密钥变化，增加密码算法的安全性。

下图为本次实验所采用的 SPN 结构。



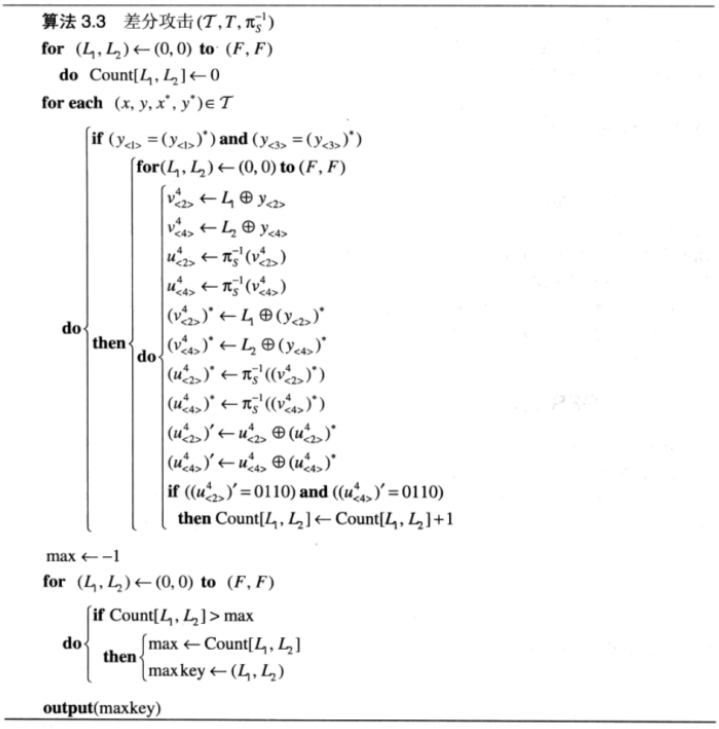
SPN 网络示意图

SPN 结构的安全性取决于 S 盒的设计、轮密钥的生成以及轮数等因素。差分密码分析是一个选择明文攻击。它通过使用大量的 4 元组（$x$、$x^*$、$y$、$y^*$）来对该 SPN 使用的密码的最后一轮进行解密，其中，异或值$x' = x \oplus x^*$是固定的，对明文$x$、$x^*$用同一个密钥 K 加密可得到$y$、$y^*$。我们对每一个候选密钥来计算某些状态的值，并确定它们的异或是否为一个确定的值，如果是，则在对应于特定候选密钥的计算器上加 1，最后选择具有最高频率的候选密钥作为我们破解得出的子密钥（一般这种情况是成立的）。

因而我们先依据书上的内容，复现书本上差分分布表、差分链和差分攻击伪代码，接着

再自己构造适当的差分，求解该 32 位密钥的低 16 位（高 16 位理论上亦能求解，但由于时间有限，故不在此过多尝试），最后根据文献资料和自我思考来研究如何构造一个好的 S 盒来减少差分攻击对其的影响。

以下图片为书上的差分攻击伪代码。




## 2. 代码分析

### i. **构造 SPN 网络**。

在进行差分密码分析之前，我们要先构造 SPN 加密算法，依据课本上的内容将 32 位密钥设置为“0011 1010 1001 0100 1101 0110 0011 1111”，主要数据的输入、加密等过程采用字符串和二进制之间的转换以提高加密效率。SPN 加密结构代码如下，其中所用到的函数均可在源代码中找到。

```
void SPN(string& w, string key)
{
    int start = 0;
    string K;
    for (int i = 0; i < 16; i++)
    {
        K += '0'; // Initialize K with zeros
    }
    for (int i = 0; i < 3; i++)
    {
        Keyupdate(start, key, K);
        Whitening(w, K);
        Substitution(w);
        Permutation(w);
    }
```

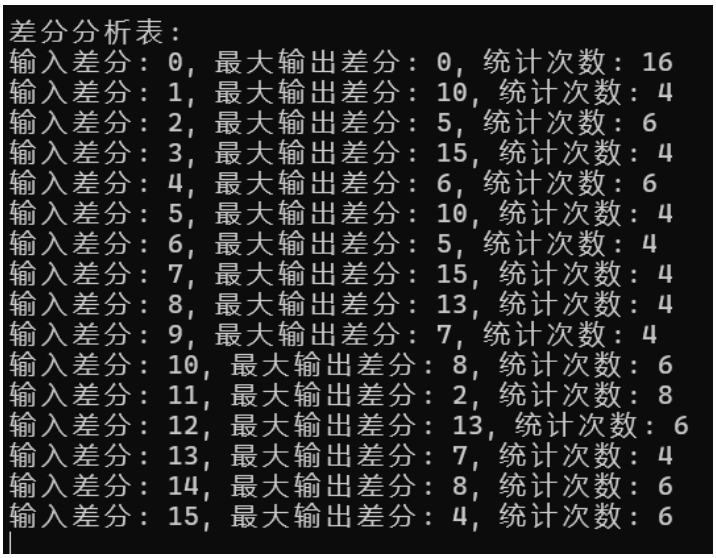
```
    Keyupdate(start, key, K);
    Whitening(w, K);
    Substitution(w);
    Keyupdate(start, key, K);
    Whitening(w, K);
}
```

ii. **构建差分分析表**。

依据书上内容，对每一个输入异或进行计算，得出相应的输出异或分布，然后统计每个输入异或对应的最高频次的输出异或及其次数。按照书上给定的 S 盒，我们可构造：

```
void generate_table() {
    // 遍历所有可能的输入差分和输出差分
    for (int input_diff = 0; input_diff < 16; input_diff++) {
        for (int i = 0; i < 16; i++) {
            // 统计特定输入和输出差分的出现次数
            // 遍历所有可能的输入值
            string input_binary = bitset<4>(i).to_string();
            string output_binary = input_binary;
            Substitution_single(output_binary); // 获得 S 盒的输出
            int input_x = stoi(input_binary, nullptr, 2);
            int output_x = stoi(output_binary, nullptr, 2);
            int xor_result_input = input_x ^ input_diff;
            string input_binary_xor = bitset<4>(xor_result_input).to_string();
            string output_binary_xor = input_binary_xor;
            Substitution_single(output_binary_xor);
            int input_xor = stoi(input_binary_xor, nullptr, 2);
            int output_xor = stoi(output_binary_xor, nullptr, 2);
            int xor_result_output = output_x ^ output_xor;
            // 将统计结果存储到表中
            if (++diff_table[input_diff][xor_result_output] > max_count[input_diff]) {
                max_count[input_diff] = diff_table[input_diff][xor_result_output];
                max_output_diff[input_diff] = xor_result_output;
            }
        }
    }
}
```

根据该代码，我们可以得到以下的输出结果，分析可知与书上的差分分布表保持一致。




### iii. 构建差分链

对于给定的输入异或，我们需要在SPN中根据上述的差分分析表来找出特定的差分链，并求出它的扩散率。为了减少运算，在代码中将扩散率用差分统计总数和整体随机总数来代替，具体代码如下：

```
void diff_chain_check(string& input_diff)
{
    string result;
    int index = 0;
    for (int i = 0; i < 4; i++)
    {
        string binaryString = input_diff.substr(i * 4, 4);
        bitset<4> binary(binaryString);
        if (binary.to_ulong() != 0)
        {
            index = max_output_diff[binary.to_ulong()];
            s1[i] = max_count[binary.to_ulong()];
            if (probility != 0)
                probility *= s1[i];
            else
                probility += s1[i];
            sum *= 16;
        }
        input_diff.replace(i * 4, 4, bitset<4>(index).to_string());
        index = 0;
    }
    Permutation(input_diff);
    for (int i = 0; i < 4; i++)
    {
        string binaryString = input_diff.substr(i * 4, 4);
        bitset<4> binary(binaryString);
```

```
        if (binary.to_ulong() != 0)
        {
            index = max_output_diff[binary.to_ulong()];
            s2[i] = max_count[binary.to_ulong()];
            probility *= s2[i];
            sum *= 16;
        }
        input_diff.replace(i * 4, 4, bitset<4>(index).to_string());
        index = 0;
    }
    Permutation(input_diff);
    for (int i = 0; i < 4; i++)
    {
        string binaryString = input_diff.substr(i * 4, 4);
        bitset<4> binary(binaryString);
        if (binary.to_ulong() != 0)
        {
            index = max_output_diff[binary.to_ulong()];
            s3[i] = max_count[binary.to_ulong()];
            probility *= s3[i];
            sum *= 16;
        }
        input_diff.replace(i * 4, 4, bitset<4>(index).to_string());
        index = 0;
    }
    Permutation(input_diff);
    cout << "u4 = " << input_diff << endl;
}
```

### iv. 构造四元组和进行差分攻击

通过差分链的构造，我们可以得到特定输入异或对应的$(u^4)'$，然后基于这个$(u^4)'$我们可以进行明密文对的选取的相应的差分攻击。在本次实验中，我共选取了“0000101100000000”，“0000111100000000”和“0110000000000000”3 个输入异或，得到的$(u^4)'$分别为“0000011000000110”，“0000011001100000”和“1101000011010000”。这三组输出异或涵盖了低 16 位密钥的分布，因而我们可以根据它们写出相应的明密文对构造和差分攻击代码，以书上要求的输入异或为“0000101100000000”为例，我们可得:

```
void generate_txt(string targetxor, int number) {
    // 指定特定值
    bitset<16>binaryxor(targetxor);
    short int value_1, value_2, value_xor1, value_xor2;
    int targetXOR = binaryxor.to_ulong();
    //cout << "targetXOR="<<targetXOR << endl;
    int count = 0;
    string key = "00111010100101001101011000111111";
    // 设置随机数生成器
```

```
    mt19937 rng(time(nullptr)); // 使用当前时间作为随机数种子
    uniform_int_distribution<int> distribution(0, 0xFFFF); // 生成 0 到 0xFFFF 之间的均匀分布的随机数
    // 打开文件
    ofstream outputFile("fourtuple_set.txt");
    // 写入随机生成的 16 位二进制数及其异或结果到文件
    if (outputFile.is_open()) {
        while (count < number)
        {
            // 计算与特定值异或的结果
            int randomNum = distribution(rng);
            int xorResult = randomNum ^ targetXOR;
            // 将随机数转换为 16 位的二进制字符串
            string binaryString = intToBinaryString(randomNum);
            string binaryString_out = binaryString;
            SPN(binaryString_out, key);
            // 将异或结果转换为 16 位的二进制字符串
            string xorString = intToBinaryString(xorResult);
            string xorString_out = xorString;
            SPN(xorString_out, key);
            //针对差分 0000101100000000
            value_1 = bitset<4>(binaryString_out.substr(0, 4)).to_ulong();
            value_2 = bitset<4>(binaryString_out.substr(8, 4)).to_ulong();
            value_xor1 = bitset<4>(xorString_out.substr(0, 4)).to_ulong();
            value_xor2 = bitset<4>(xorString_out.substr(8, 4)).to_ulong();
            if (value_1 == value_xor1 && value_2 == value_xor2)*/
            {
                // 写入到文件
                outputFile << binaryString << " " << xorString << " " << binaryString_out << " " << xorString_out << endl;
                count++;
            }
            randomNum++;
        }
        outputFile.close();
        cout << "Data has been written to fourtuple_set.txt." << endl;
    }
    else {
        cout << "Unable to open file for writing." << endl;
    }
}
```

我们将构造得到的四元组放入到 txt 文件中，示例如下：

文件　编辑　查看

1011010111001011 1011111011001011 0000110110100001 0000101110100100
0110100111100101 0110001011100101 1110011101000011 1110100111110101
0010100101010011 0010001001010011 0111110101101101 0100101111011010
1010111111011000 1010010011011000 1100100000010000 1100010000010111
0011100001001010 0011001101001010 0010110110111000 1010010110100001
1110101011011111 1110000111011111 1000110110000110 1000100010000011
1100000011001101 1100101111001101 0001000110011010 0000111111110111
1111100001010111 1111001101010111 1001011111010110 0011100101010100
0001001101001111 0001100001001111 1000110001010101 1111100010111101
1100100100010101 1100001000010101 0100000111101000 0100110011100101
0101111010100001 0101010110100001 0010110100001110 0010101101101011
1001010111000101 1001111011000101 0011010010110101 1010001100010110
1110111010101001 1110010110101001 0011011001110100 1001011000011011
1010110111111010 1010011011111010 0001101011011000 1100011001101101
0100100001011000 0100001101011000 0000001011010000 0011110010000110
1000001101011110 1000100001011110 0100010001101000 1110111101001100
1000001110100100 1000100010100100 0100100101100110 1011111010110011
1110111110101111 1110010010101111 0011101000011010 0110011011110011
1011111111011011 1011010011011011 0010110100111101 1010011001001011
1101110101001011 1101011001001011 1101111010110010 1101100111001101
0101001110110100 0101100010110100 1000100111100110 0101101000011010
0100101011100110 0100000111100110 1100010110000011 1100010011101110
0111010100011011 0111111000011011 0111100000010010 0100110110000010
0110000111011001 0110101011011001 0111100000001100 0111101101101111

然后我们从中读取数据，根据书上提供的伪代码进行特定的差分攻击：

```
void readDataFromFile(const string& filename) {
    ifstream inputFile(filename);

    if (inputFile.is_open()) {
        // 用于存储读取的四元组数据

        while (!inputFile.eof()) {
            FourTuple tuple;
            // 尝试读取四个数据项
            if (inputFile >> tuple.binaryString >> tuple.xorString >>
tuple.binaryString_out >> tuple.xorString_out) {
                // 如果成功读取四个数据项，则将其存储到 vector 中
                datas.push_back(tuple);
            }
        }
        inputFile.close();
        // 处理存储的数据

        cout << "Number of elements in vector: " << datas.size() << endl;
    }
    else {
        cout << "Unable to open file for reading." << endl;
    }
}
```

```
void differential_attack()
{
    short int count[16][16] = { 0 };
    readDataFromFile("fourtuple_set.txt");
    short int y[4] = { 0 };
    short int y_xor[4] = { 0 };
    short int v[4] = { 0 };
    short int u[4] = { 0 };
    short int v_xor[4] = { 0 };
    short int u_xor[4] = { 0 };
    short int u_mid[4] = { 0 };
    short int start = 0;
    short int key1, key2;
    unsigned long int num = 0;
    for (const auto& tuple : datas)
    {
        for (int i = 0; i < tuple.binaryString_out.size(); i += 4) {
            // 获取当前四位子串并转换为整数
            int value = bitset<4>(tuple.binaryString_out.substr(i, 4)).to_ulong();
            int value_xor = bitset<4>(tuple.xorString_out.substr(i, 4)).to_ulong();
            y[start] = value;
            y_xor[start] = value_xor;
            start++;
        }
        start = 0;
        ++num;
        //针对 0000101100000000
        for (int l1 = 0; l1 < 16; l1++)
        {
            for (int l2 = 0; l2 < 16; l2++)
            {
                v[1] = l1 ^ y[1];
                v[3] = l2 ^ y[3];
                u[1] = Substitution_single_revise(v[1]);
                u[3] = Substitution_single_revise(v[3]);
                v_xor[1] = l1 ^ y_xor[1];
                v_xor[3] = l2 ^ y_xor[3];
                u_xor[1] = Substitution_single_revise(v_xor[1]);
                u_xor[3] = Substitution_single_revise(v_xor[3]);
                u_mid[1] = u[1] ^ u_xor[1];
                u_mid[3] = u[3] ^ u_xor[3];
                if (u_mid[3] == 0x6 && u_mid[1] == 0x6)
                {
                    count[l1][l2] += 1;
```

```
            }


        }
    }
}
int max = -1;
for (int l1 = 0; l1 < 16; l1++)
{
    for (int l2 = 0; l2 < 16; l2++)
    {
        if (count[l1][l2] > max)
        {
            max = count[l1][l2];
            key1 = l1;
            key2 = l2;
        }
    }
}
cout << "max=" << max << ", key1= " << key1 << ", key2:" << key2 << "  使用的有效明密文对: " << num << endl;
}
```

书上提到的过滤操作我在构造明密文对并写入文件的时候就已进行，故在关于差分攻击的函数不需要再次重复该步骤。至此，我们完成了相关内容。

### 3. 结果展示

根据以上的步骤，我们在主函数中调用相应的函数，设置生成的符合条件的四元组个数，已知低 16 位密钥为 1101 0110 0011 1111。

对于输入异或为“0000101100000000”，$(u^4)'$=“0000011000000110”，设置四元组个数为 75 个，我们可知需要得到的最高频率的密钥分别为 0110 和 1111，转成十进制也就是 6 和 15，运行程序可以得到

```
0000101100000000
Data has been written to fourtuple_set.txt.
u4 = 0000011000000110
 差分统计总数: 1728 , 整体随机总数 : 65536
S1[0]:0 S1[1]:8 S1[2]:0 S1[3]:0
S2[0]:0 S2[1]:0 S2[2]:6 S2[3]:0
S3[0]:0 S3[1]:6 S3[2]:6 S3[3]:0
Number of elements in vector: 75
max=5, key1= 6, key2:15  使用的有效明密文对: 75
```

求解得到的 key1 和 key2 符合正确子密钥。

对于输入异或为“0000111100000000”，$(u^4)'$=“0000011001100000”，设置四元组个数为 100 个，我们运行程序可以得到

```
0000111100000000
Data has been written to fourtuple_set.txt.
u4 = 0000011001100000
 差分统计总数: 1296 , 整体随机总数 : 65536
S1[0]:0 S1[1]:6 S1[2]:0 S1[3]:0
S2[0]:0 S2[1]:6 S2[2]:0 S2[3]:0
S3[0]:0 S3[1]:6 S3[2]:6 S3[3]:0
Number of elements in vector: 100
max=4, key1= 6, key2:3  使用的有效明密文对: 100
```

key1 和 key2 转成二进制分别为 0110 和 0011，与我们要求解的子密钥符合。

对于输入异或为“0110000000000000”，$(u^4)'$=“1101000011010000”，同样设置四元组个数为 100 个，我们运行程序可以得到

```
0110000000000000
Data has been written to fourtuple_set.txt.
u4 = 1101000011010000
 差分统计总数: 4096 , 整体随机总数 : 16777216
S1[0]:4 S1[1]:0 S1[2]:0 S1[3]:0
S2[0]:0 S2[1]:4 S2[2]:0 S2[3]:4
S3[0]:4 S3[1]:4 S3[2]:0 S3[3]:4
Number of elements in vector: 100
max=6, key1= 13, key2:3  使用的有效明密文对: 100
```

key1 和 key2 转成二进制分别为 1101 和 0011，与我们要求解的子密钥符合。

至此，低 16 位密钥我们已全部通过差分攻击求解出来。

## 4. 实验思考

- S 盒的选取

个人认为差分攻击主要依赖于差分链的存在，而差分链的形成是由于对某个特定的输入异或，其输出异或分布不均匀导致的。在本次实验所采用的 S 盒，在之前提到的差分分析表中，我们不难看到除输入差分为 0 外，其他输入差分对应的最大输出差分统计次数从 4 到 8 不等。由此我们可以提出问题：是否可以降低除 0 外剩余输入差分统计次数来构造 S 盒?

通过查询文献可知，我们可以通过以下三步来构造 S 盒:

1) 求出有限域 GF$(2^n)$上的既约多项式对应状态字的逆元。

确定有限域后，可通过相关计算求得域 GF$(2^n)$上存在的所有 n 次既约多项式，域 GF$(2^n)$上存在 3 个 4 次既约多项式。在同一个有限域上选取不同的既约多项式可以计算得到不同的逆元集，同时改变仿射变换公式的输入变量，进而得到更多性能相似而输出不同的 S 盒。

2) 利用仿射变换公式对得到的逆元结果做仿射变换。

求得选定的既约多项式的乘法逆元后，可将求得的结果输入仿射变换公式中进行 S 盒构造。构造 S 盒的主流仿射变换公式为:

$$S(x) = C(x) + (X^{-1}) \cdot B(x) \bmod A(x)$$

在域 GF$(2^n)$上选取$A(x)$时，只需要满足最高次数为 n 且较为简单的多项式，如$A(x)$=$x^n + 1$。$B(x)$要求与$A(x)$互素，若$A(x)$为不可约多项式，则$B(x)$可任意选取。然后是$C(x)$，它的选取是为了保证得到的 S 盒不包含某些特殊点（反不动点和不动点）。在$A(x)$和$B(x)$选定的情况下，$C(x)$有多种可能的选择，进而可得到更多的 S 盒。

3) 利用差分分析表对构造的 S 盒进行性能分析。

由于时间问题，我并未根据文献实现相应的代码来生成大量的 S 盒数据，只是依据文献

所给样例做了有限的数据测试。在域 GF($2^n$)上，我们可选择既约多项式$x^4+x^3+x^2+x+1$，仿射参数为：$A(x)=x^4+1$，$B(x)=x^3+x^2+x$，$C(x)=x^2+x+1$，构造如下的 S 盒：

```
s[] = { 7, 9, 8, 0xd, 0xa, 0xb, 2, 4, 0xc, 5, 0xe, 6, 0xf, 1, 3, 0 };
```

```
差分分析表：
输入差分： 0, 最大输出差分： 0, 统计次数： 16
输入差分： 1, 最大输出差分： 14, 统计次数： 4
输入差分： 2, 最大输出差分： 15, 统计次数： 4
输入差分： 3, 最大输出差分： 10, 统计次数： 4
输入差分： 4, 最大输出差分： 13, 统计次数： 4
输入差分： 5, 最大输出差分： 12, 统计次数： 4
输入差分： 6, 最大输出差分： 5, 统计次数： 4
输入差分： 7, 最大输出差分： 3, 统计次数： 4
输入差分： 8, 最大输出差分： 11, 统计次数： 4
输入差分： 9, 最大输出差分： 2, 统计次数： 4
输入差分： 10, 最大输出差分： 9, 统计次数： 4
输入差分： 11, 最大输出差分： 1, 统计次数： 4
输入差分： 12, 最大输出差分： 8, 统计次数： 4
输入差分： 13, 最大输出差分： 6, 统计次数： 4
输入差分： 14, 最大输出差分： 4, 统计次数： 4
输入差分： 15, 最大输出差分： 7, 统计次数： 4
```

选择既约多项式$x^4+x^3+1$，仿射参数为：$A(x)=x^4+1$，$B(x)=x^3+x+1$，$C(x)=x$，构造如下的 S 盒：

```
s[] = { 2, 9, 1, 0xf, 0xb, 0xd, 0xc, 6, 0xe, 0xa, 3, 8, 5, 4, 0, 7 };
```

```
差分分析表：
输入差分： 0, 最大输出差分： 0, 统计次数： 16
输入差分： 1, 最大输出差分： 11, 统计次数： 4
输入差分： 2, 最大输出差分： 3, 统计次数： 4
输入差分： 3, 最大输出差分： 13, 统计次数： 4
输入差分： 4, 最大输出差分： 9, 统计次数： 4
输入差分： 5, 最大输出差分： 15, 统计次数： 4
输入差分： 6, 最大输出差分： 14, 统计次数： 4
输入差分： 7, 最大输出差分： 4, 统计次数： 4
输入差分： 8, 最大输出差分： 12, 统计次数： 4
输入差分： 9, 最大输出差分： 8, 统计次数： 4
输入差分： 10, 最大输出差分： 1, 统计次数： 4
输入差分： 11, 最大输出差分： 10, 统计次数： 4
输入差分： 12, 最大输出差分： 7, 统计次数： 4
输入差分： 13, 最大输出差分： 6, 统计次数： 4
输入差分： 14, 最大输出差分： 2, 统计次数： 4
输入差分： 15, 最大输出差分： 5, 统计次数： 4
```

我们可以观察到除输入差分为 0 外，其他输入差分对应的最大输出差分统计次数均为 4，进而影响差分链的生成。

- 总结

通过本次实验，我对 SPN 的差分攻击有了系统的了解，也自主查询相关文献了解到一些关于 S 盒设计方面的知识。在差分密码分析代码实现部分，我较为系统地实现了书本上的内容，并且尝试新的输入异或来成功获取低 16 位的子密钥。由于时间、精力和自身能力等问题，在代码实现和理论部分我仍有缺陷，例如在输入异或获取$(u^4)'$时，我只能人为判断调

整过滤操作以提高正确四元组的筛选效率，对于高 16 位的子密钥分析并未去做，没有设计如何自动判别每次输入异或产生正确密钥时所需要的四元组个数，对于 S 盒的设计和优化在理论知识上了解不多等。仍有许多需要加以改进的地方，但总而言之，也算是在有限的时间内尽了自己最大的努力了。