

SPN加解密及线性密码分析

2112514 辛浩然

SPN加解密

SPN 加密

SPN由 Nr 轮组成，在每一轮(除了最后一轮稍有不同外)，先用异或操作混入该轮的轮密钥，再用 π_s 进行代换，然后用 π_p 进行置换。而在最后一轮，使用 π_s 进行代换后，没有用 π_p 进行置换，而是异或最后一轮轮密钥，得到密文。

伪代码如下：

Algorithm 1 SPN 加密

```
1:  $w_0 \leftarrow x$ 
2: for  $r \leftarrow 1$  to  $Nr - 1$  do
3:    $u^r \leftarrow w^{r-1} \oplus K^r$ 
4:   for  $i \leftarrow 1$  to  $m$  do
5:      $w^r \leftarrow (v_{\pi_p(1)}^r, \dots, v_{\pi_p(\ell m)}^r)$ 
6:   end for
7: end for
8:  $u^{Nr} \leftarrow w^{Nr-1} \oplus K^{Nr}$ 
9: for  $i \leftarrow 1$  to  $m$  do
10:   $v_{<i>}^{Nr} \leftarrow \pi_s(u_{<i>}^{Nr})$ 
11: end for
12:  $t \leftarrow v^{Nr} \oplus K^{Nr+1}$ 
13: output( $y$ )
```

```
#include <iostream>
#include <string>
using namespace std;
string S_box(string u, int nr)
{
    string res;
    string mapping[] = {
        "1110", "0100", "1101", "0001",
        "0010", "1111", "1011", "1000",
    }
```

```

        "0011", "1010", "0110", "1100",
        "0101", "1001", "0000", "0111"};

    for (int i = 0; i < nr; i++)
    {
        string tmp = u.substr(i * 4, 4);
        res += mapping[stoi(tmp, 0, 2)];
    }

    return res;
}

string P_box(string s, int nr)
{
    string res;
    int mapping[] = {
        0, 4, 8, 12,
        1, 5, 9, 13,
        2, 6, 10, 14,
        3, 7, 11, 15};

    for (int i = 0; i < nr * 4; i++)
    {
        res += s[mapping[i]];
    }

    return res;
}

int main()
{
    string x, w, k, u, s_res, p_res;
    cin >> x >> k;
    int len = 16;
    int nr = 4;
    w = x;
    for (int i = 0; i <= nr; i++)
    {
        int index = i * 4;
        u.clear();
        for (int j = 0; j < len; j++)
        {
            if (w[j] == k[index])
            {
                u += '0';
            }
            else

```

```

        {
            u += '1';
        }
        index++;
    }
    if (i == nr)
        break;
    s_res = S_box(u, nr);
    if (i != nr - 1)
    {
        p_res = P_box(s_res, nr);
    }
    else
    {
        p_res = s_res;
    }
    w.clear();
    w = p_res;
}
cout << u;
}

```

具体而言，使用一个循环，迭代 `nr` 次来执行加密过程：

- 状态 `w` 初始化为明文。在每轮中，
 - 首先，轮密钥 `k_i` 与 `w` 按位异或操作，并将结果存储在 `u` 中。
 - 使用 `S_box` 函数对 `u` 中的每 4 位进行S盒替代。
 - 对于S盒的结果：
 - 如果不是最后一轮，使用 `P_box` 函数对S盒替代的结果进行P盒置换。
 - 如果是最后一轮，将S盒的结果与轮密钥 `k_5` 进行异或，结束循环。
 - 将本轮结果存储在变量 `w` 中，以备下一轮使用。
- 最后一轮异或后的结果即为加密后的密文。

SPN 解密

SPN 解密即SPN 加密的逆过程，以下伪代码说明了SPN解密的基本流程：

Algorithm 2 SPN 解密

```
1:  $v^{Nr} \leftarrow y \oplus K^{Nr+1}$ 
2: for  $i \leftarrow 1$  to  $m$  do
3:    $u_{\langle i \rangle}^{Nr} \leftarrow \pi_s^{-1}(v_{\langle i \rangle}^{Nr})$ 
4: end for
5:  $w^{Nr-1} \leftarrow u^{Nr} \oplus K^{Nr}$ 
6: for  $r \leftarrow Nr - 1$  downto  $1$  do
7:    $v^r \leftarrow (w_{\pi_p^{-1}(1)}^r, \dots, w_{\pi_p^{-1}(\ell m)}^r)$ 
8:   for  $i \leftarrow 1$  to  $m$  do
9:      $u_{\langle i \rangle}^r \leftarrow \pi_s^{-1}(v_{\langle i \rangle}^r)$ 
10:  end for
11:   $w^{r-1} \leftarrow u^r \oplus K^r$ 
12: end for
13:  $x \leftarrow w_0$ 
14: output( $x$ )
```

具体实现思路如下：

- 首先编写S盒和P盒的逆变换；
- 解密过程，具体而言，使用一个循环，`r` 从 `Nr` 到 `1` 迭代 `nr` 次来执行解密过程：
 - 如果是第一轮，则将密文与轮密钥异或得到 v^r ；
 - 如果不是第一轮，则将 w^r 逆P盒置换得到 v^r ；
 - 随后，使用 `Reverse_S_box` 函数对 v^r 进行S盒逆变换，得到 u^r ；
 - 轮密钥 k^r 与 u^r 按位异或操作，得到 w^{r-1} 的值。
- 最后一轮得到的 w^0 即为解密后的明文。

```
#include <iostream>
#include <string>
using namespace std;
string Reverse_S_box(string u, int nr)
{
    string res;
    string mapping[] = {
        "1110", "0011", "0100", "1000",
        "0001", "1100", "1010", "1111",
        "0111", "1101", "1001", "0110",
        "1011", "0010", "0000", "0101"};

    for (int i = 0; i < nr; i++)
    {
        string tmp = u.substr(i * 4, 4);
```

```

        res += mapping[stoi(tmp, 0, 2)];
    }

    return res;
}

string Reverse_P_box(string s, int nr)
{
    string res;
    int mapping[] = {
        0, 4, 8, 12,
        1, 5, 9, 13,
        2, 6, 10, 14,
        3, 7, 11, 15};

    for (int i = 0; i < nr * 4; i++)
    {
        res += s[mapping[i]];
    }

    return res;
}

int main()
{
    string y, k, u, w, v, x;
    cin >> y >> k;
    int len = 16;
    int nr = 4;
    int index = nr * 4;
    for (int i = nr; i > 0; i--)
    {
        v.clear();
        if (i != nr)
        {
            v = Reverse_P_box(w, nr);
        }
        else
        {
            for (int j = 0; j < len; j++)
            {
                if (y[j] == k[index])
                {
                    v += '0';
                }
                else
                {

```

```

        v += '1';
    }
    index++;
}
}
u = Reverse_S_box(v, nr);
int index = (i - 1) * 4;
w.clear();
for (int j = 0; j < len; j++)
{
    if (u[j] == k[index])
    {
        w += '0';
    }
    else
    {
        w += '1';
    }
    index++;
}
}
x = w;
cout << x;
}

```

线性密码分析基本原理

线性密码分析是一种已知明文攻击——攻击者已知密钥相同的多组明密文对，求解密钥。

线性分析法的基本想法是在一个**明文比特子集**与最后一轮**即将进行代换的输入状态比特子集**之间找到一个概率线性关系，即存在一个比特子集使得元素的异或表现出非随机的分布。比如： $X_{i_1} \oplus X_{i_2} \oplus \dots \oplus U_{j_1}^4 \oplus U_{j_2}^4 \oplus \dots = 0$

如果我们找到一个比特子集，使得子集中元素的异或表现出非随机的分布，也就是异或式具有偏差，即异或取值为 0 的概率 P 不等于 $1/2$ ， $|P - 1/2|$ 的值越大，则越容易使用线性分析法，利用上式通过多组明密文分析出密钥的值。

具体而言，假设拥有大量的使用**同一未知密钥K加密的明-密文对**：

- 对每一个明-密文对，将用**所有可能的候选密钥**来对**最后一轮解密密文**
- 对每一个候选密钥，计算包含在**线性关系式**中的**相关状态比特的异或值**，然后确定上述的**线性关系**是否成立
 - 如果成立，就在对应于特定候选密钥的计数器上加1

- 在这个过程的最后，我们希望计数频率离明-密文对数的一半最远的候选密钥含有那些密钥比特的正确值

分析过程

线性逼近表

对于给定S盒，输入 $X_1X_2X_3X_4$ ，得到输出 $Y_1Y_2Y_3Y_4$ 。

考虑若干输入输出变量之间的异或值，可把每一个相关的随机变量写成：

$$\left(\bigoplus_{i=1}^4 a_i X_i\right) \oplus \left(\bigoplus_{i=1}^4 b_i Y_i\right)$$

其中 $a_i \in \{0,1\}, b_i \in \{0,1\}, i = 1, 2, 3, 4$ 。为了记号的紧凑，把每一个二元向量 (a_1, a_2, a_3, a_4) 和 (b_1, b_2, b_3, b_4) 看做一个十六进制数字 (这些数字分别叫做**输入和**与**输出和**)。这样，这 256 个随机变量里的每一个就用一对十六进制数字来表示。

对于一个具有 (十六进制) 输入和 a 与输出和 b 的随机变量 (这里

$a = (a_1, a_2, a_3, a_4), b = (b_1, b_2, b_3, b_4)$), 设 $N_L(a, b)$ 表示满足如下条件的二进制 8 元组 $(x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)$ 的个数：

$$(y_1, y_2, y_3, y_4) = \pi_S(x_1, x_2, x_3, x_4) \text{ 及 } \left(\bigoplus_{i=1}^4 a_i x_i\right) \oplus \left(\bigoplus_{i=1}^4 b_i y_i\right) = 0$$

该随机变量的偏差计算公式为： $\epsilon(a, b) = (N_L(a, b) - 8)/16$

由于选择线性逼近时，需要计算相关随机变量之间的偏差，因此，需要提前列出 $N_L(a, b)$ 的表格，即线性逼近表。

具体实现思路为：

- 遍历所有输入和 a 与输出和 b ;
- 遍历所有S盒输入 $X_1X_2X_3X_4$ ，得到输出 $Y_1Y_2Y_3Y_4$;
- 将二进制表示的 a 与 $X_1X_2X_3X_4$ 按位取与，二进制表示的 b 与 $Y_1Y_2Y_3Y_4$ 按位取与，统计二者结果中1的数目，如果为偶数，说明这组随机变量异或值为0，将线性逼近表中相应位置的值加1。

核心代码如下：

```
// 省略一些函数的实现，完整代码详见附件
const int size = 16;
string X[size] = {"0000", "0001", "0010", "0011",
                 "0100", "0101", "0110", "0111",
                 "1000", "1001", "1010", "1011",
                 "1100", "1101", "1110", "1111"};
string Y[size] = {"1110", "0100", "1101", "0001",
                 "0010", "1111", "1011", "1000",
```

```

        "0011", "1010", "0110", "1100",
        "0101", "1001", "0000", "0111"};

int table[size][size] = {0};
char or_bit(char a, char b){}
int countOnes(string str, int len){}
void output_table(){}
int main()
{
    string l1, l2;
    for (int i = 0; i < 16; i++)
    {
        for (int j = 0; j < 16; j++)
        {
            for (int k = 0; k < size; k++)
            {
                l1.clear();
                l2.clear();
                string i_string = bitset<4>(i).to_string();
                string j_string = bitset<4>(j).to_string();
                for (int m = 0; m < 4; m++)
                {
                    l1 += or_bit(i_string[m], X[k][m]);
                    l2 += or_bit(j_string[m], Y[k][m]);
                }
                if ((countOnes(l1, 4) + countOnes(l2, 4)) % 2 == 0)
                {
                    table[i][j]++;
                }
            }
        }
    }
    output_table();
}

```

输出线性逼近表如下：

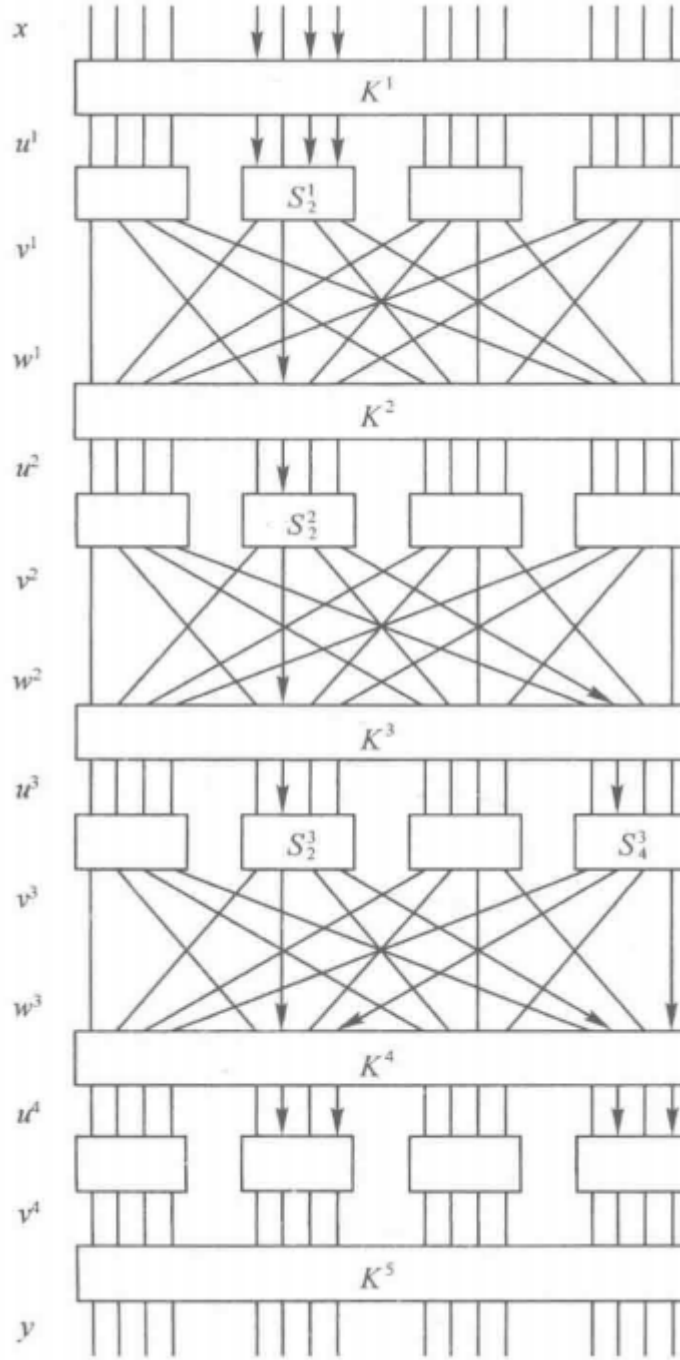
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	8	6	6	8	8	6	14	10	10	8	8	10	10	8	8
2	8	8	6	6	8	8	6	6	8	8	10	10	8	8	2	10
3	8	8	8	8	8	8	8	8	10	2	6	6	10	10	6	6
4	8	10	8	6	6	4	6	8	8	6	8	10	10	4	10	8
5	8	6	6	8	6	8	12	10	6	8	4	10	8	6	6	8
6	8	10	6	12	10	8	8	10	8	6	10	12	6	8	8	6
7	8	6	8	10	10	4	10	8	6	8	10	8	12	10	8	10
8	8	8	8	8	8	8	8	8	6	10	10	6	10	6	6	2
9	8	8	6	6	8	8	6	6	4	8	6	10	8	12	10	6
10	8	12	6	10	4	8	10	6	10	10	8	8	10	10	8	8
11	8	12	8	4	12	8	12	8	8	8	8	8	8	8	8	8
12	8	6	12	6	6	8	10	8	10	8	10	12	8	10	8	6
13	8	10	10	8	6	12	8	10	4	6	10	8	10	8	8	10
14	8	10	10	8	6	4	8	10	6	8	8	6	4	10	6	8
15	8	6	4	6	6	8	10	8	8	6	12	6	6	8	10	8

线性逼近

线性密码分析要求找出一组S盒的线性逼近，这组线性逼近能够用来导出一个整个SPN(除最后一轮外)的线性逼近。

▲ 求解 $K_{<2>}^5$ 和 $K_{<4>}^5$

首先，给出用于求解 $K_{<2>}^5$ 和 $K_{<4>}^5$ 的线性逼近，如图所示。带箭头的线条对应于包含在线性逼近中的随机变量。带标号的S盒表示在这些逼近中使用了此S盒。



此线性逼近包括四个活动S盒，结合线性逼近表，可以得到：

- 在 S^1_2 中，随机变量 $\mathbf{T}_1 = \mathbf{U}^1_5 \oplus \mathbf{U}^1_7 \oplus \mathbf{U}^1_8 \oplus \mathbf{V}^1_6$ 具有偏差 $1/4$ ；
- 在 S^2_2 中，随机变量 $\mathbf{T}_2 = \mathbf{U}^2_6 \oplus \mathbf{V}^2_6 \oplus \mathbf{V}^2_8$ 具有偏差 $-1/4$ ；
- 在 S^3_2 中，随机变量 $\mathbf{T}_3 = \mathbf{U}^3_6 \oplus \mathbf{V}^3_6 \oplus \mathbf{V}^3_8$ 具有偏差 $-1/4$ ；
- 在 S^3_4 中，随机变量 $\mathbf{T}_4 = \mathbf{U}^3_{14} \oplus \mathbf{V}^3_{14} \oplus \mathbf{V}^3_{16}$ 具有偏差 $-1/4$ 。

假设这四个随机变量相互独立，根据堆积引理可以求得随机变量 $\mathbf{T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{T}_3 \oplus \mathbf{T}_4$ 具有偏差 $2^3(1/4)(-1/4)^3 = -1/32$

- 随机变量 $\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3, \mathbf{T}_4$ 具有以下性质：它们的异或可用明文比特、 u^4 的比特(S盒最后一轮的输入)以及密钥比特表示出来，即可以推导出如下关系式：

$$\mathbf{T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{T}_3 \oplus \mathbf{T}_4 = \mathbf{X}_5 \oplus \mathbf{X}_7 \oplus \mathbf{X}_8 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_8^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{16}^4 \\ \oplus \mathbf{K}_5^1 \oplus \mathbf{K}_7^1 \oplus \mathbf{K}_8^1 \oplus \mathbf{K}_6^2 \oplus \mathbf{K}_6^3 \oplus \mathbf{K}_{14}^3 \oplus \mathbf{K}_6^4 \oplus \mathbf{K}_8^4 \oplus \mathbf{K}_{14}^4 \oplus \mathbf{K}_{16}^4$$

- 由于密钥比特固定， K_i^j 具有固定的值0或1，因此，随机变量

$$\mathbf{X}_5 \oplus \mathbf{X}_7 \oplus \mathbf{X}_8 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_8^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{16}^4 \text{ 具有偏差 } \pm 1/32。$$

上式仅包含明文比特和 $u_{<2>}^4$ 、 $u_{<4>}^4$ 的比特，而上式具有偏离0的偏差这一事实允许我们进行线性密码攻击。我们可以通过线性密码攻击获得 $K_{<2>}^5$ 和 $K_{<4>}^5$ 的8比特密钥。

一般来说，一个基于偏差为 ϵ 的线性逼近的线性攻击要想获得成功，所需要的明密文对数目 T 要接近于 $c\epsilon^{-2}$ 。在该线性逼近中， $\epsilon^{-2} = 1024$ ，取 $c = 8$ ，计算可知 $T \approx 8000$ 。因此，需要大约8000对明密文对。

攻击的基本思路就是，保持对应于候选子密钥的计数器，每当随机变量 $\mathbf{X}_5 \oplus \mathbf{X}_7 \oplus \mathbf{X}_8 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_8^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{16}^4$ 取值为0时，就将对应于该子密钥的计数器加1(这些计数器的初始值全为0)。在计数过程的最后，我们希望大多数的计数器值接近于 $T/2$ ，而真正的候选子密钥对应的计数器具有接近于 $T/2 \pm T/32$ 之值，这有助于我们确定正确的8个子密钥比特。

接下来，给出具体实现线性攻击获得 $K_{<2>}^5$ 和 $K_{<4>}^5$ 的8比特密钥的代码：

```
// plain 为输入的明文
// cipher 为输入的密文
// binaryArray 为0~16数字对应的4位01字符串
void K24(string *plain, string *cipher, string *binaryArray, string &res2,
string &res4)
{
    int count[16][16] = {0};
    string tmp1, tmp2, tmp3, tmp4;
    char z;
    char v[16] = {0}, u[16] = {0};
    for (int i = 0; i < size; i++)
    {
        for (int l1 = 0; l1 < 16; l1++)
        {
            for (int l2 = 0; l2 < 16; l2++)
            {
                for (int k = 0; k < 4; k++)
                {
                    v[k + 4] = nor(binaryArray[l1][k], cipher[i][k + 4]);
                    v[k + 12] = nor(binaryArray[l2][k], cipher[i][k + 12]);
                }
                tmp1.clear();
```

```

        tmp2.clear();
        for (int k = 0; k < 4; k++)
        {
            tmp1 += v[k + 4];
            tmp2 += v[k + 12];
        }
        tmp1 = Reverse_S_box(tmp1, 1);
        tmp2 = Reverse_S_box(tmp2, 1);
        for (int k = 0; k < 4; k++)
        {
            u[k + 4] = tmp1[k];
            u[k + 12] = tmp2[k];
        }
        z = nor(nor(nor(nor(nor(nor(plain[i][4], plain[i][6]),
plain[i][7]), u[5]), u[7]), u[13]), u[15]));
        if (z == '0')
        {
            count[l1][l2]++;
        }
    }
}
int max = -1;
int a = 0, b = 0;
for (int l1 = 0; l1 < 16; l1++)
{
    for (int l2 = 0; l2 < 16; l2++)
    {
        count[l1][l2] = abs(count[l1][l2] - 4000);
        if (count[l1][l2] > max)
        {
            max = count[l1][l2];
            a = l1;
            b = l2;
        }
    }
}
res2 = bitset<4>(a).to_string();
res4 = bitset<4>(b).to_string();
}

```

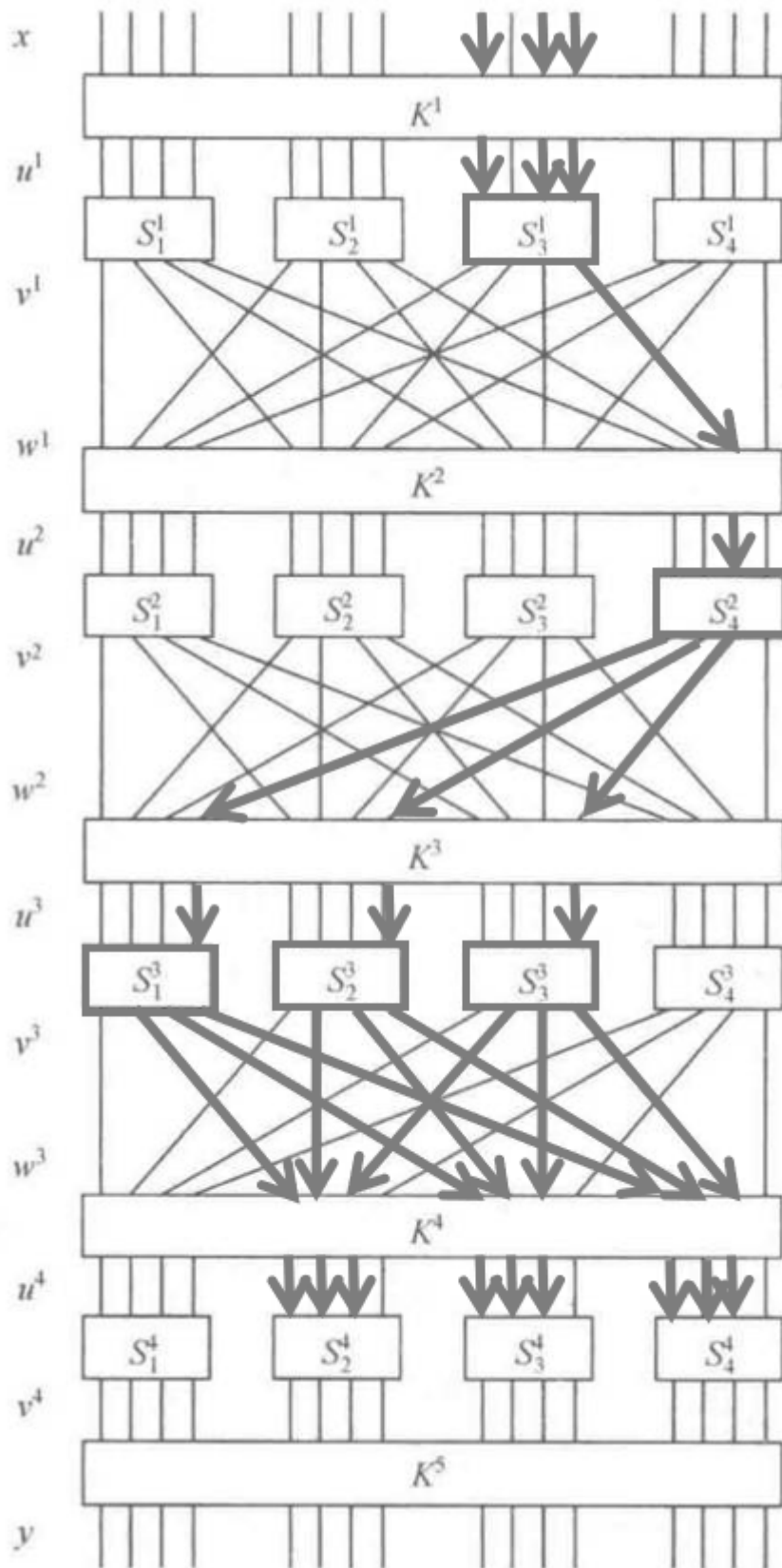
具体步骤分析如下：

1. 输入：明文数组、与之对应的密文数组；
2. 定义计数器二维数组，索引是密钥 $K_{<2>}^5$ 和 $K_{<4>}^5$ 的十进制数值，内容全部置为0；
3. 遍历所有明密文对：

- 遍历所有候选子密钥 $K_{<2>}^5$ 和 $K_{<4>}^5$:
 - 获得 $v_{<2>}^4$: 密文 $y_{<2>}$ 与密钥 $K_{<2>}^5$ 异或;
 - 获得 $v_{<4>}^4$: 密文 $y_{<4>}$ 与密钥 $K_{<4>}^5$ 异或;
 - 获得 $u_{<2>}^4$: $v_{<2>}^4$ 通过S盒逆置换;
 - 实现S盒逆置换函数, 将输入字符串 按照逆S盒映射规则进行逆变换, 并返回结果字符串
 - 获得 $u_{<4>}^4$: $v_{<4>}^4$ 通过S盒逆置换;
 - 计算异或值: 计算 $X_5 \oplus X_7 \oplus X_8 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4$ 的值
 - 如果异或值为0, 在对应于特定候选密钥的计数器上加1
- 4. 遍历计数器:
- 计数频率离明-密文对数的一半最远(即频率 $-T/2$ 最大)的候选密钥含有那些密钥比特的正确值

▲ 求解 $K_{<3>}^5$

给出用于求解 $K_{<3>}^5$ 的线性逼近, 如图所示:



此线性逼近包括五个活动S盒，结合线性逼近表，可以得到：

- 在 S_3^1 中，随机变量 $\mathbf{T}_1 = \mathbf{U}_9^1 \oplus \mathbf{U}_{11}^1 \oplus \mathbf{U}_{12}^1 \oplus \mathbf{V}_{12}^1$ 具有偏差 $1/4$ ；
- 在 S_4^2 中，随机变量 $\mathbf{T}_2 = \mathbf{U}_{15}^2 \oplus \mathbf{V}_{13}^2 \oplus \mathbf{V}_{14}^2 \oplus \mathbf{V}_{15}^2$ 具有偏差 $-3/8$ ；

- 在 S_1^3 中, 随机变量 $\mathbf{T}_3 = \mathbf{U}_4^3 \oplus \mathbf{V}_2^3 \oplus \mathbf{V}_3^3 \oplus \mathbf{V}_4^3$ 具有偏差 $3/8$;
- 在 S_2^3 中, 随机变量 $\mathbf{T}_4 = \mathbf{U}_8^3 \oplus \mathbf{V}_6^3 \oplus \mathbf{V}_7^3 \oplus \mathbf{V}_8^3$ 具有偏差 $3/8$;
- 在 S_3^3 中, 随机变量 $\mathbf{T}_5 = \mathbf{U}_{12}^3 \oplus \mathbf{V}_{10}^3 \oplus \mathbf{V}_{11}^3 \oplus \mathbf{V}_{12}^3$ 具有偏差 $3/8$.

假设这四个随机变量相互独立, 根据堆积引理可以求得随机变量 $\mathbf{T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{T}_3 \oplus \mathbf{T}_4 \oplus \mathbf{T}_5$ 具有偏差 $2^4(1/4)(-3/8)(3/8)^3 = -81/1024$

同前一部分, 可以得到随机变量

$\mathbf{X}_9 \oplus \mathbf{X}_{11} \oplus \mathbf{X}_{12} \oplus \mathbf{U}_5^4 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_7^4 \oplus \mathbf{U}_9^4 \oplus \mathbf{U}_{10}^4 \oplus \mathbf{U}_{11}^4 \oplus \mathbf{U}_{13}^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{15}^4$ 具有偏差 $\pm 81/1024$ 。

上式具有偏离0的偏差这一事实允许我们通过线性密码攻击获得 $K_{<3>}^5$ 的4比特密钥。

攻击的基本思路与前面一致, 就是保持对应于候选子密钥的计数器, 每当随机变量 $\mathbf{X}_9 \oplus \mathbf{X}_{11} \oplus \mathbf{X}_{12} \oplus \mathbf{U}_5^4 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_7^4 \oplus \mathbf{U}_9^4 \oplus \mathbf{U}_{10}^4 \oplus \mathbf{U}_{11}^4 \oplus \mathbf{U}_{13}^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{15}^4$ 取值为0时, 就将对应于该子密钥的计数器加1(这些计数器的初始值全为0)。最后统计计数器计数频率离明文对数的一半最远的候选密钥。

$K_{<3>}^5$ 虽然与 $K_{<1>}^5$ 的求解使用不同的线性逼近, 但可以一起求解, 因此具体代码实现在下部分分析。

▲ 求解 $K_{<1>}^5$

给出用于求解 $K_{<1>}^5$ 的线性逼近, 如图所示:

- 在 S_2^2 中, 随机变量 $\mathbf{T}_2 = \mathbf{U}_6^2 \oplus \mathbf{V}_6^2 \oplus \mathbf{V}_8^2$ 具有偏差 $-1/4$;
- 在 S_2^3 中, 随机变量 $\mathbf{T}_3 = \mathbf{U}_6^3 \oplus \mathbf{V}_5^3 \oplus \mathbf{V}_6^3 \oplus \mathbf{V}_8^3$ 具有偏差 $-1/4$;
- 在 S_4^3 中, 随机变量 $\mathbf{T}_4 = \mathbf{U}_{14}^3 \oplus \mathbf{V}_{14}^3 \oplus \mathbf{V}_{16}^3$ 具有偏差 $-1/4$;

假设这四个随机变量相互独立, 根据堆积引理可以求得随机变量 $\mathbf{T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{T}_3 \oplus \mathbf{T}_4$ 具有偏差 $2^3(1/4)(-1/4)^3 = -1/32$

同前一部分, 可以得到随机变量 $\mathbf{X}_5 \oplus \mathbf{X}_7 \oplus \mathbf{X}_8 \oplus \mathbf{U}_2^4 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_8^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{16}^4$ 具有偏差 $\pm 1/32$ 。

上式具有偏离0的偏差这一事实允许我们通过线性密码攻击获得 $K_{<1>}^5$ 的4比特密钥。

攻击的基本思路与前面一致, 就是保持对应于候选子密钥的计数器, 每当随机变量 $\mathbf{X}_5 \oplus \mathbf{X}_7 \oplus \mathbf{X}_8 \oplus \mathbf{U}_2^4 \oplus \mathbf{U}_6^4 \oplus \mathbf{U}_8^4 \oplus \mathbf{U}_{14}^4 \oplus \mathbf{U}_{16}^4$ 取值为0时, 就将对应于该子密钥的计数器加1(这些计数器的初始值全为0)。最后统计计数器计数频率离明密文对数的一半最远的候选密钥。

代码如下:

```
void K13(string *plain, string *cipher, string *binaryArray, string res2,
string res4, string &res1, string &res3)
{
    int count1[16] = {0}, count3[16] = {0};
    string tmp1, tmp2, tmp3, tmp4;
    char z;
    char v[16] = {0}, u[16] = {0};
    for (int i = 0; i < size; i++)
    {
        cin >> plain[i] >> cipher[i];
    }
    for (int i = 0; i < size; i++)
    {
        for (int l1 = 0; l1 < 16; l1++)
        {
            for (int k = 0; k < 4; k++)
            {
                v[k] = nor(binaryArray[l1][k], cipher[i][k]);
                v[k + 8] = nor(binaryArray[l1][k], cipher[i][k + 8]);
                v[k + 4] = nor(res2[k], cipher[i][k + 4]);
                v[k + 12] = nor(res4[k], cipher[i][k + 12]);
            }
            tmp1.clear();
            tmp2.clear();
            tmp3.clear();
            tmp4.clear();
            for (int k = 0; k < 4; k++)
```

```

    {
        tmp1 += v[k];
        tmp2 += v[k + 4];
        tmp3 += v[k + 8];
        tmp4 += v[k + 12];
    }
    tmp1 = Reverse_S_box(tmp1, 1);
    tmp2 = Reverse_S_box(tmp2, 1);
    tmp3 = Reverse_S_box(tmp3, 1);
    tmp4 = Reverse_S_box(tmp4, 1);
    for (int k = 0; k < 4; k++)
    {
        u[k] = tmp1[k];
        u[k + 4] = tmp2[k];
        u[k + 8] = tmp3[k];
        u[k + 12] = tmp4[k];
    }
    z = nor(nor(nor(nor(nor(nor(nor(nor(nor(nor(plain[i][8],
plain[i][10])), plain[i][11])), u[4]), u[5]), u[6]), u[8]), u[9]), u[10]),
u[12]), u[13]), u[14]);
    if (z == '0')
    {
        count3[l1]++;
    }
    z = nor(nor(nor(nor(nor(nor(nor(plain[i][4], plain[i][6]),
plain[i][7])), u[1]), u[5]), u[7]), u[13]), u[15]);
    if (z == '0')
    {
        count1[l1]++;
    }
}
}
int max1 = -1, max3 = -1;
int a = 0, b = 0;
for (int l1 = 0; l1 < 16; l1++)
{
    count1[l1] = abs(count1[l1] - 4000);
    count3[l1] = abs(count3[l1] - 4000);
    if (count1[l1] > max1)
    {
        max1 = count1[l1];
        a = l1;
    }
    if (count3[l1] > max3)
    {
        max3 = count3[l1];
        b = l1;
    }
}

```

```

    }
}
res1 = bitset<4>(a).to_string();
res3 = bitset<4>(b).to_string();
}

```

实现思路基本与求 $K_{<2><4>}^5$ 时一致，具体如下：

1. 输入：明文数组、与之对应的密文数组、之前求出的 $K_{<2>}^5$ 和 $K_{<4>}^5$ ；
2. 定义计数器：
 - 一维数组 `count3`，索引是密钥 $K_{<3>}^5$ 的十进制数值，内容全部置为0；
 - 一维数组 `count1`，索引是密钥 $K_{<1>}^5$ 的十进制数值，内容全部置为0；
3. 遍历所有明密文对：
 - 遍历所有四位候选子密钥：
 - 获得 $v_{<1>}^4$ ：密文 $y_{<1>}$ 与密钥 $K_{<1>}^5$ 异或；
 - 获得 $v_{<2>}^4$ ：密文 $y_{<2>}$ 与已知密钥 $K_{<2>}^5$ 异或；
 - 获得 $v_{<3>}^4$ ：密文 $y_{<3>}$ 与密钥 $K_{<3>}^5$ 异或；
 - 获得 $v_{<4>}^4$ ：密文 $y_{<4>}$ 与已知密钥 $K_{<4>}^5$ 异或；
 - 获得 $u_{<1>}^4$ ： $v_{<1>}^4$ 通过S盒逆置换；
 - 获得 $u_{<2>}^4$ ： $v_{<2>}^4$ 通过S盒逆置换；
 - 获得 $u_{<3>}^4$ ： $v_{<3>}^4$ 通过S盒逆置换；
 - 获得 $u_{<4>}^4$ ： $v_{<4>}^4$ 通过S盒逆置换；
 - 计算异或值：计算 $X_5 \oplus X_7 \oplus X_8 \oplus U_2^4 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4$ 的值
 - 如果异或值为0，在 `count1` 中对应于特定候选密钥的计数器上加1
 - 计算异或值：计算 $X_9 \oplus X_{11} \oplus X_{12} \oplus U_5^4 \oplus U_6^4 \oplus U_7^4 \oplus U_9^4 \oplus U_{10}^4 \oplus U_{11}^4 \oplus U_{13}^4 \oplus U_{14}^4 \oplus U_{15}^4$ 的值
 - 如果异或值为0，在 `count3` 中对应于特定候选密钥的计数器上加1
4. 分别遍历两个计数器：
 - 计数频率离明-密文对数的一半最远(即频率 $-T/2$ 最大)的候选密钥含有那些密钥比特的正确值
 - 还是与前面一样，取8000对明密文对

求解完整密钥 K

通过以上步骤，求出第五轮密钥 K^5 的值。试图求解完整密钥，在固定后16位密钥的基础上，采用穷举的方式，遍历前16位密钥，使用明密文对加以验证(实际上只需要几对就可以，我取了16对来验证)，即可得到完整密钥 K 。

```
string K_all(string *plain, string *cipher, string *binaryArray, string K5)
{
    string K;
    for (int i = 0; i < 16; i++)
    {
        for (int j = 0; j < 16; j++)
        {
            for (int ii = 0; ii < 16; ii++)
            {
                for (int jj = 0; jj < 16; jj++)
                {
                    K = binaryArray[i] + binaryArray[j] + binaryArray[ii] +
binaryArray[jj] + K5;
                    int k = 0;
                    for (; k < 16; k++)
                    {
                        if (SPN_Encrypt(plain[k], K) != cipher[k])
                        {
                            break;
                        }
                    }
                    if (k == 16)
                    {
                        return K;
                    }
                }
            }
        }
    }
    return "failed";
}
```

攻击测试

生成明密文对

由于攻击需要大量明密文对，前面分析出来约8000对即可满足需求。因此，首先需要生成大量明密文对。

- 使用 `generate_plain.py` (见附件)构造输入数据，随机生成8000个8位01字符串作为明文，每个明文后跟一行密钥。
- 作为加密输入文件，写 `makefile` 文件运行加密函数 `SPN.cpp`，生成密文。
- 随后使用 `merge_plain_cipher.py` 将明文和密文合并为一个文件，每条明文后跟着一行相应密文，作为线性攻击的输入文件。

求解密钥

得到输入的8000个明密文对后，就可以验证线性攻击程序了。

编写 `makefile` 文件，如下：

```
CXX = g++
CXXFLAGS = -std=c++11 -Wall
TARGET = final
INPUT_FILE = input.txt
OUTPUT_FILE = output.txt

all: $(TARGET)

$(TARGET): $(TARGET).cpp
    $(CXX) $(CXXFLAGS) -o $(TARGET) $(TARGET).cpp

run: $(TARGET)
    ./$(TARGET) < $(INPUT_FILE) > $(OUTPUT_FILE)

clean:
    rm -f $(TARGET) $(OUTPUT_FILE)
```

命令行输入 `make run`，可以得到输出结果：

```
Desktop > cpp > output.txt
1 K5 is 1101011000111111
2 K is 00111010100101001101011000111111
3
```

```
K5 is 1101011000111111
K is 00111010100101001101011000111111
```

即通过线性攻击得到 $K^5 = 1101\ 0110\ 0011\ 1111$;

$K = 0011\ 1010\ 1001\ 0100\ 1101\ 0110\ 0011\ 1111$

与之前设定的加密密钥对比，发现是一致的，

$K = 0011\ 1010\ 1001\ 0100\ 1101\ 0110\ 0011\ 1111$

这证明了所编写的线性攻击程序的正确性。