

# 南开大学

## 密码学课程实验报告

### SPN 及其线性密码攻击



学 院\_\_\_\_\_网络空间安全学院  
专 业\_\_\_\_\_信息安全  
学 号\_\_\_\_\_2112060  
姓 名\_\_\_\_\_孙露  
班 级\_\_\_\_\_信息安全 1 班

# 一、实验过程

## (一) SPN 的实现

### (1) 定义

一个 SPN 包括两个变换，分别记为  $\pi_s$  和  $\pi_p$ 。设  $l$  和  $m$  都是正整数，明文和密文都是长为  $lm$  的二元向量(即  $lm$  是该密码的分组长度)。

$$\pi_s: \{0, 1\}^l \rightarrow \{0, 1\}^l$$

$$\pi_p: \{1, \dots, lm\}^1 \rightarrow \{1, \dots, lm\}^1$$

置换  $\pi_s$  叫做 S 盒, 它用一个  $l$  比特向量来替代另一个  $l$  比特向量。置换  $\pi_p$ , 用来置换  $lm$  个比特。

给定一个  $lm$  比特的二元串  $x=(x_1, \dots, x_{lm})$ , 可将其看做是  $m$  个长为  $l$  比特的子串  $x_{\langle 1 \rangle}, \dots, x_{\langle m \rangle}$  的并联。因此,  $x=x_{\langle 1 \rangle} || \dots || x_{\langle m \rangle}$ , 其中  $x_{\langle i \rangle}=(x_{(i-1)l+1}, \dots, x_{il}), 1 \leq i \leq m$ 。将要给出的 SPN 由  $Nr$  轮组成, 在每一轮(除了最后一轮稍有不同, 没有用置换  $\pi_p$ ), 先用异或操作混入该轮的轮密钥, 再用  $\pi_s$  进行  $m$  次代换, 然后用  $\pi_p$ , 进行一次置换。

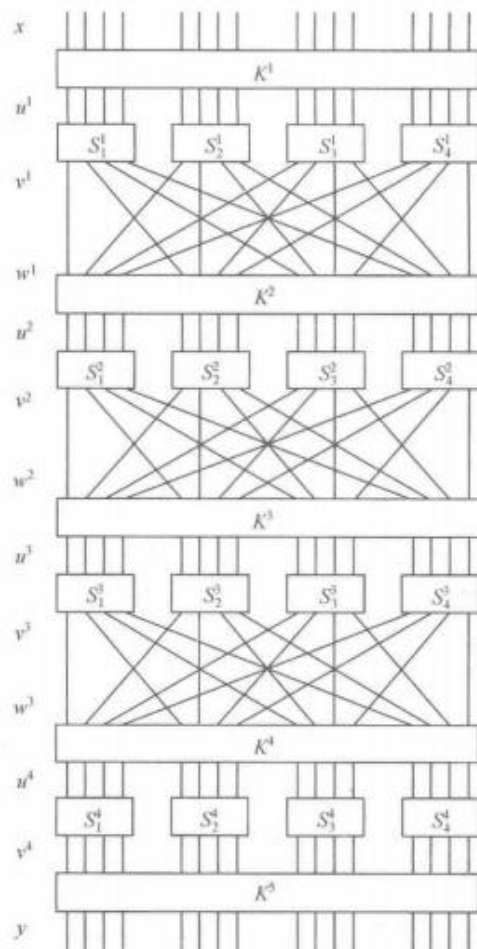


图 3.1 一个代换-置换网络

## (2) 伪代码

在算法 3.1 中,  $u^r$  是第  $r$  轮对  $S$  盒的输入,  $v^r$  是第  $r$  轮对  $S$  盒的输出。  
 $w^r$  由  $v^r$  应用置换  $\pi_p$  得到, 然后  $u^{r+1}$  由轮密钥  $K^{r+1}$  异或  $w_r$  得到 (这叫做轮密钥混合), 最后一轮没有用置换  $\pi_p$ 。如果对密钥编排方案做适当修改并用  $S$  盒的逆来取代  $S$  盒, 那么该加密算法也能用来解密。

算法 3.1 SPN( $x, \pi_S, \pi_P, (K^1, \dots, K^{Nr+1})$ )

```

 $w^0 \leftarrow x$ 
for  $r \leftarrow 1$  to  $Nr-1$ 
     $u^r \leftarrow w^{r-1} \oplus K^r$ 
    do
        for  $i \leftarrow 1$  to  $m$ 
             $v_{<i>}^r \leftarrow \pi_S(u_{<i>}^r)$ 
         $w^r \leftarrow (v_{\pi_P(1)}^r, \dots, v_{\pi_P(\ell_m)}^r)$ 
     $u^{Nr} \leftarrow w^{Nr-1} \oplus K^{Nr}$ 
    for  $i \leftarrow 1$  to  $m$ 
         $v_{<i>}^{Nr} \leftarrow \pi_S(u_{<i>}^{Nr})$ 
     $y \leftarrow v^{Nr} \oplus K^{Nr+1}$ 
output( $y$ )

```

### (3) C++代码

```

#include <iostream>
#include<string>
#include<bitset>
using namespace std;

int s[16] = { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 }; //S 盒
int p[16] = { 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16 }; //P 置换

//ur-->S_Box-->vr
string S_Box(string u)
{
    string v;
    for (int i = 0; i < 16; i += 4)
    {
        int index = stoi(u.substr(i, 4), 0, 2); // 将二进制字符串转换为十进制整数
        //s_output 是一个 4 位的二进制位集，将整数 index 的值以二进制形式表示，并在需要
        //时填充前导零，以确保 s_output 始终是 4 位的
        bitset<4> s_output(s[index]); // 将十进制整数转换为一个 4 位的二进制表示，并将结果
        //存储在 s_output 中。
        v += s_output.to_string(); //将 s_output 中的二进制数据转换为字符串，然后追加到
        //字符串 v 的末尾
    }
    //cout << "v: " << v << endl; //输出 v 提示
    return v;
}

//vr-->P_Box-->wr

```

```

string P_Box(string v)
{
    string w = "0000000000000000";
    for (int i = 0; i < 16; i++)
    {
        char m = v[i];
        int pos = p[i];
        w[pos-1] = m;
    }
    //cout << "w: " << w << endl;//输出 w 提示
    return w;
}

string Key_Scheduling(string K, int r)//密钥编排算法
{
    string k;
    for (int i = 4 * r - 4; i <= 4 * r +11; i++)//从 k4r-3 开始的连续 16 个 bit
    {
        k += K[i];
    }
    //cout << "k: " << k << endl;//输出 k 提示
    return k;
}

//wr-1 xor Kr-->ur
string Round_Key_Mixing(string w, string k)//轮密钥混合
{
    string u;
    bitset<16> str1(w);
    bitset<16> str2(k);
    bitset<16> u_bit = str1 ^ str2;
    u = u_bit.to_string();
    //cout << "u: " << u << endl;//输出 u 提示
    return u;
}

int main()
{
    int l = 4;
    int m = 4;
    int Nr = 4;
    string x;//明文 x
    string K;//初始密钥 K
    string y;//密文 y
    //cout << "x: ";
    cin >> x;//输入 x

```

```

//cout << "K: ";
cin >> K;//输入 K

string u;
string v;
string w = x;
string k;

for (int r = 1; r < Nr; r++)
{
    k = Key_Scheduling(K, r);
    u = Round_Key_Mixing(w, k);
    v = S_Box(u);
    w = P_Box(v);
}

k = Key_Scheduling(K, Nr);
u = Round_Key_Mixing(w, k);
v = S_Box(u);

k = Key_Scheduling(K, Nr + 1);
y = Round_Key_Mixing(v, k);
//cout << "y: ";
cout<< y;
return 0;
}

```

(a) S\_box:

S 盒是一个用于替代输入的 4 位二进制数据块的固定表。输入  $u$  是一个 16 位的二进制字符串。首先将输入的 16 位字符串按 4 位一组进行分割。然后，将每组 4 位二进制字符串转换为对应的整数值，作为 S 盒的索引。S 盒表中的值根据索引取出，并转换为 4 位的二进制字符串。最后，这些 4 位字符串连接在一起，形成输出  $v$ 。

(b) P\_Box:

P 盒是一个用于重新排列输入位的固定表。输入  $v$  是一个

16 位的二进制字符串。函数按照 P 盒置换表  $p$  对输入进行重新排列，生成输出  $w$ 。

(c) Key\_Scheduling:

这个函数用于密钥编排，以生成轮密钥。输入参数  $K$  是初始密钥， $r$  表示轮数。函数根据轮数  $r$  选择密钥的一部分，生成一个子密钥  $k$ 。从初始密钥  $K$  中选择连续的 16 位，从第  $4*r-4$  位到第  $4*r+11$  位，然后返回这个子密钥。

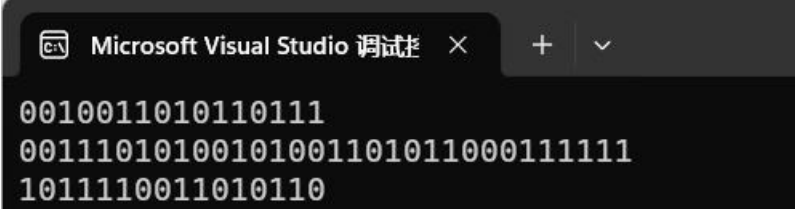
(d) Round\_Key\_Mixing

这个函数用于轮密钥混合。输入参数  $w$  是上一轮的输出， $k$  是本轮的轮密钥。函数将输入转换为 16 位的位集，然后执行异或 (XOR) 操作，将  $w$  和  $k$  进行异或运算，生成输出  $u$ 。

(e) main :

输入的明文  $x$  为  $w^0$ 。前  $Nr-1$  轮依次进行  $w^{r-1}$  与  $K^r$  异或、S\_Box、P\_Box 的运算，最后一轮不进行 P\_Box 的运算，与  $K^5$  异或得到密文。

## (4) 运行结果



```
Microsoft Visual Studio 调试器 × + v
0010011010110111
00111010100101001101011000111111
1011110011010110
```

## (二) 线性密码分析

### (1) 分析

对书中例 3.1 的 SPN 进行线性攻击

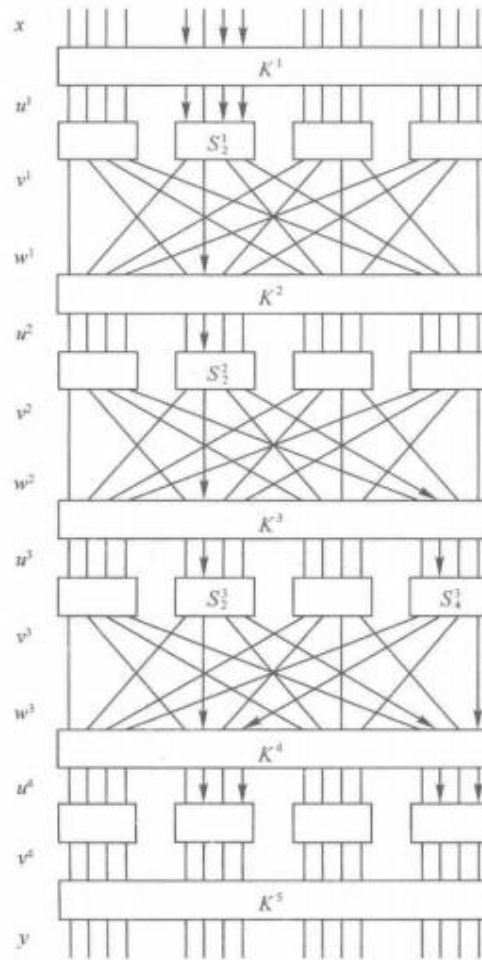


图 3.3 一个代换-置换网络的线性逼近

图 3.3 中的逼近包含如下四个活动 S 盒：

在  $S_2^1$  中，随机变量  $T_1 = U_5^1 \oplus U_7^1 \oplus U_8^1 \oplus V_6^1$  具有偏差  $1/4$

在  $S_2^2$  中，随机变量  $T_2 = U_6^2 \oplus V_6^2 \oplus V_8^2$  具有偏差  $-1/4$

在  $S_3^3$  中，随机变量  $T_3 = U_6^3 \oplus V_6^3 \oplus V_8^3$  具有偏差  $-1/4$

在  $S_4^4$  中，随机变量  $T_4 = U_{14}^3 \oplus V_{14}^3 \oplus V_{16}^3$  具有偏差  $-1/4$

$T_1, T_2, T_3, T_4$  这四个随机变量都具有较高的偏差绝对值，而且它们的异或会消去中间变量。

假设这四个随机变量相互独立，假定随机变量  $T_1 \oplus T_2 \oplus T_3 \oplus T_4$  具有偏差  $2^3(1/4)(-1/4)^3 = -1/32$ 。

随机变量  $T_1, T_2, T_3, T_4$  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。这一点可从以下事实看出:

由图 3.3 易验证以下关系成立:

$$T_1 = U_5^1 \oplus U_7^1 \oplus U_8^1 \oplus V_6^1 = X_5 \oplus K_5^1 \oplus X_7 \oplus K_7^1 \oplus X_8 \oplus K_8^1 \oplus V_6^1$$

$$T_2 = U_6^2 \oplus V_6^2 \oplus V_8^2 = V_6^2 \oplus K_6^2 \oplus V_6^2 \oplus V_8^2$$

$$T_3 = U_6^3 \oplus V_6^3 \oplus V_8^3 = V_6^3 \oplus K_6^3 \oplus V_6^3 \oplus V_8^3$$

$$T_4 = U_{14}^3 \oplus V_{14}^3 \oplus V_{16}^3 = V_8^2 \oplus K_{14}^3 \oplus V_{14}^3 \oplus V_{16}^3$$

将上述等式的右端相异或得到:

$$X_5 \oplus X_7 \oplus X_8 \oplus V_6^3 \oplus V_8^3 \oplus V_{14}^3 \oplus V_{16}^3 \oplus K_5^1 \oplus K_7^1 \oplus K_8^1 \oplus K_6^2 \oplus K_6^3 \oplus K_{14}^3 \quad (3.1)$$

具有偏差 $-1/32$ 。把上式中的 $V_i^3$ 用包含 $U_i^3$ 与下轮密钥比特的表达式来代替:

$$V_6^3 = U_6^4 \oplus K_6^4$$

$$V_8^3 = U_{14}^4 \oplus K_{14}^4$$

$$V_{14}^3 = U_8^4 \oplus K_8^4$$

$$V_{16}^3 = U_{16}^4 \oplus K_{16}^4$$

现在把这些式子代入式(3.1), 可得:

$$X_5 \oplus X_7 \oplus X_8 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4 \oplus K_5^1 \oplus K_7^1 \oplus K_8^1 \oplus K_6^2 \oplus K_6^3 \oplus K_{14}^3 \oplus K_6^4 \oplus K_8^4 \oplus K_{14}^4 \oplus K_{16}^4 \quad (3.2)$$

式(3.2)仅包含明文比特、 $u^4$ 的比特以及密钥比特。假设式(3.2)中的密钥比特固定, 则随机变量

$$K_5^1 \oplus K_7^1 \oplus K_8^1 \oplus K_6^2 \oplus K_6^3 \oplus K_{14}^3 \oplus K_6^4 \oplus K_8^4 \oplus K_{14}^4 \oplus K_{16}^4$$

具有固定的值 0 或 1。因此, 随机变量

$$X_5 \oplus X_7 \oplus X_8 \oplus U_6^4 \oplus U_8^4 \oplus U_{14}^4 \oplus U_{16}^4 \quad (3.3)$$

具有偏差 $\pm 1/32$ , 这里偏差的符号取决于未知密钥比特的值。注意随机变量

式(3.3)仅包含明文比特及  $u^4$  的比特。式(3.3)具有偏离 0 的偏差这一事实允许我们进行线性密码攻击。

假设拥有用同一未知密钥  $K$  加密的  $T$  对明-密文(为使攻击成功约需要  $T \approx 8000$  对明-密文)。用  $T$  来表示  $T$  对明-密文的集合。线性攻击将使我们获得  $K_{<2>}^5$  和  $K_{<4>}^5$  的 8 比特密钥, 即

$$K_5^5 \oplus K_6^5 \oplus K_7^5 \oplus K_8^5 \oplus K_{13}^5 \oplus K_{14}^5 \oplus K_{15}^5 \oplus K_{16}^5$$

这些正是与  $S$  盒  $s_2^4$  和  $s_4^4$  的输出相异或的 8 比特密钥。对这 8 比特密钥来说, 共有  $2^8=256$  种可能, 把由这 8 比特密钥组成的一个二进制 8 元组叫做一个候选子密钥。

对每一个  $(x, y) \in T$  及每一个候选子密钥, 计算  $y$  的一个部分解密并获得  $u_{<2>}^4$  和  $u_{<4>}^4$ 。然后通过式(3.3)随机变量的取值, 计算

$$x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4 \quad (3.4)$$

之值。保持对应于这 256 个候选子密钥的 256 个计数器, 每当式(3.4)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, 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, 这有助于确定正确的 8 个子密钥比特。

## (2) 伪代码

算法 3.2 给出了这个特殊的线性攻击算法。集合  $T$  表示  $T$  对明-密文的集合, 变量  $L_1$  和  $L_2$  取十六进制的值, 置换  $\pi_s^{-1}$  对应于  $S$  盒的逆置换,  $\pi_s^{-1}$  被用来部分地解密密文; 输出 maxkey 包含了该攻击确定出的具有最大可能的 8 个子密钥比特。

对每一对明-密文  $(x, y) \in T$  及每一个可能的候选子密钥  $(L_1, L_2)$ , 只需计算式 (3.4)。为了实现这一点, 参考图 3.3。首先计算异或  $L_1 \oplus y_{<2>}$  和  $L_2 \oplus y_{<4>}$ , 当  $(L_1, L_2)$  是正确的子密钥时, 可分别产生  $v_{<2>}^4$  和  $v_{<4>}^4$ 。通过对  $v_{<2>}^4$  和  $v_{<4>}^4$  使用 S 盒的逆  $\pi_s^{-1}$ ; 可计算出  $u_{<2>}^4$  和  $u_{<4>}^4$ , 如果  $(L_1, L_2)$  是正确的子密钥, 这些值都是正确的。然后, 计算式 (3.4), 如果式 (3.4) 取值为 0, 就将对应于  $(L_1, L_2)$  的计数器加 1。在计算完所有相关的计数器之后, 仅找到对应于最大计数器的对  $(L_1, L_2)$ , 这就是算法 3.2 的输出。

一般来说, 一个基于偏差  $\varepsilon$  为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  $c \varepsilon^{-2}$ , 对某个“小”的常数  $c$ 。将算法 3.2 实现一下就会发现: 如果取  $T=8000$ , 这个攻击通常会成功。注意到  $T=8000$  对应于  $c \approx 8$ , 这是因为  $\varepsilon^2=1024$ 。

---

**算法 3.2** 线性攻击  $(T, T, \pi_s^{-1})$

```

for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$ 
  do  $\text{Count}[L_1, L_2] \leftarrow 0$ 
for each  $(x, y) \in T$ 
  {
    for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$ 
    {
       $v_{<2>}^4 \leftarrow L_1 \oplus y_{<2>}$ 
       $v_{<4>}^4 \leftarrow L_2 \oplus y_{<4>}$ 
       $u_{<2>}^4 \leftarrow \pi_s^{-1}(v_{<2>}^4)$ 
       $u_{<4>}^4 \leftarrow \pi_s^{-1}(v_{<4>}^4)$ 
       $z \leftarrow x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4$ 
      if  $z = 0$ 
      then  $\text{Count}[L_1, L_2] \leftarrow \text{Count}[L_1, L_2] + 1$ 
    }
  }
 $\text{max} \leftarrow -1$ 
for  $(L_1, L_2) \leftarrow (0, 0)$  to  $(F, F)$ 
  {
     $\text{Count}[L_1, L_2] \leftarrow |\text{Count}[L_1, L_2] - T/2|$ 
    do if  $\text{Count}[L_1, L_2] > \text{max}$ 
    then {
       $\text{max} \leftarrow \text{Count}[L_1, L_2]$ 
       $\text{maxkey} \leftarrow (L_1, L_2)$ 
    }
  }
output(maxkey)

```

---

(3) C++代码

```

#include <iostream>
#include <string>
#include <bitset>
#include <vector>
#include <cmath>
using namespace std;

int s[16] = { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 }; // S 盒
int p[16] = { 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15, 4, 8, 12, 16 }; // P 置换
string K = "00111010100101001101011000111111";
int inverseSBox[16] = { 14, 3, 4, 8, 1, 12, 10, 15, 7, 13, 9, 6, 11, 2, 0, 5 };
int Count[16][16] = { 0 };
int best_count = -1;

//ur-->S_Box-->vr
string S_Box(string u)
{
    string v;
    for (int i = 0; i < 16; i += 4)
    {
        int index = stoi(u.substr(i, 4), 0, 2); // 将 4 位二进制子串转换为十进制整数
        bitset<4> s_output(s[index]); // 将 S 盒输出的整数值转换为 4 位二进制
        v += s_output.to_string(); // 将 S 盒输出追加到 v 中
    }
    return v;
}

//vr-->P_Box-->wr
string P_Box(string v)
{
    string w = "0000000000000000";
    for (int i = 0; i < 16; i++)
    {
        char m = v[i]; // 获取 P 置换的输入
        int pos = p[i]; // 获取 P 置换的输出位置
        w[pos - 1] = m; // 进行 P 置换
    }
    return w;
}

// 密钥编排算法
string Key_Scheduling(string K, int r)
{
    string k;

```

```

    for (int i = 4 * r - 4; i <= 4 * r + 11; i++)
    {
        k += K[i]; // 选择特定位置的密钥比特
    }
    return k;
}

// 轮密钥混合
string Round_Key_Mixing(string w, string k)
{
    string u;
    bitset<16> str1(w);
    bitset<16> str2(k);
    bitset<16> u_bit = str1 ^ str2; // 将 w 和 k 按位异或
    u = u_bit.to_string();
    return u;
}

// SPN 加密函数
string SPN(string plaintext, string K)
{
    int Nr = 4; // SPN 的轮数
    string w = plaintext;
    string k;
    string u;
    string v;
    string y;

    for (int r = 1; r <= Nr; r++)
    {
        k = Key_Scheduling(K, r);
        u = Round_Key_Mixing(w, k);
        v = S_Box(u);
        w = P_Box(v);
    }

    k = Key_Scheduling(K, Nr);
    u = Round_Key_Mixing(w, k);
    v = S_Box(u);

    k = Key_Scheduling(K, Nr + 1);
    y = Round_Key_Mixing(v, k);

    return y; // 加密结果
}

```

```

}

//pai_s 逆
void calculateInverseSBox()
{
    for (int i = 0; i < 16; i++)
    {
        inverseSBox[s[i]] = i;
    }
}

string S_Box_(string u)
{
    string v;
    int index = stoi(u.substr(0, 4), 0, 2); // 将 4 位二进制子串转换为十进制整数
    bitset<4> s_output(inverseSBox[index]); // 将 S 盒输出的整数值转换为 4 位二进制
    v += s_output.to_string(); // 将 S 盒输出追加到 v 中
    return v;
}

// 线性密码分析算法
pair<int, pair<int, int>> linearCryptanalysis(vector<pair<string, string>> data, int
num_candidate_keys)
{
    pair<int, int> best_candidate_keys = make_pair(0, 0);
    for (pair<string, string> sample : data)
    {
        // 遍历训练数据集中的每一对明文-密文
        string x = sample.first;
        string y = sample.second;

        // 遍历每个可能的子密钥 (L1, L2)
        for (int L1 = 0; L1 < num_candidate_keys; L1++)
        {
            for (int L2 = 0; L2 < num_candidate_keys; L2++)
            {
                // 计算 u64、u84、u144 和 u164
                bitset<4> binaryNumber1(L1); // 十进制数字 L1 转为二进制数字
                bitset<4> y2; // 用于存储位集合的变量
                for (int i = 0; i <= 3; i++)
                {
                    y2[i] = (y[i + 4] == '1');
                }
                bitset<4> v24_ = binaryNumber1 ^ y2;
            }
        }
    }
}

```

```

        string v24 = v24_.to_string();

        bitset<4> binaryNumber2(L2); //十进制数字 L2 转为二进制数字
        bitset<4> y4;
        for (int i = 0; i <= 3; i++)
        {
            y4[i] = (y[i + 12] == '1');
        }
        bitset<4> v44_ = binaryNumber2 ^ y4;
        string v44 = v44_.to_string();

        string u24 = S_Box_(v24);
        string u44 = S_Box_(v44);

        int z = ((x[4] == '1') + (x[6] == '1') + (x[7] == '1') + (u24[1] == '1')
+ (u24[3] == '1') + (u44[1] == '1') + (u44[3] == '1')) % 2;

        if (z == 0)
        {
            Count[L1][L2]++;
        }
    }

}

for (int L1 = 0; L1 < num_candidate_keys; L1++)
{
    for (int L2 = 0; L2 < num_candidate_keys; L2++)
    {
        Count[L1][L2] = abs(Count[L1][L2] - static_cast<int>(data.size() / 2));
        // 更新最佳候选子密钥
        if (Count[L1][L2] > best_count)
        {
            best_count = Count[L1][L2];
            best_candidate_keys = make_pair(L1, L2);
        }
    }
}

pair<int, pair<int, int>> best_keys = make_pair(best_count, best_candidate_keys);
return best_keys;
}

vector<pair<string, string>> createdata(int num_samples)
{

```

```

vector<pair<string, string>> data; // 明文-密文训练数据集，需要根据实际情况填充
for (int i = 0; i < num_samples; i++)//生成 8000 对明文-密文对
{
    string plaintext = bitset<16>(rand() & 0xFFFF).to_string();// 随机生成 16 位明文
    string ciphertext = SPN(plaintext, K);// 使用 SPN 加密函数加密明文
    data.push_back(make_pair(plaintext, ciphertext));// 存储明文和密文对
}
return data;
}

int main()
{
    int num_samples = 8000; // 用于统计的样本数量
    vector<pair<string, string>> data; // 明文-密文训练数据集，需要根据实际情况填充
    data = createdata(num_samples);
    int num_candidate_keys = 16;

    pair<int, pair<int, int>> best_keys = linearCryptanalysis(data, num_candidate_keys);
    cout << "Best Candidate Keys: L1 = " << best_keys.second.first << ", L2 = " <<
best_keys.second.second << " with count " << best_keys.first << endl;

    return 0;
}

```

(a) SPN（和第一部分一致）

(b) calculateInverseSBox 函数：

这个函数用于计算 S-DES 中的逆 S 盒。通过遍历 S 盒中的值，将 S 盒的输出值与其索引关联，以便进行逆 S 盒操作。

(c) S\_Box\_ 函数：

这个函数实现了 S-DES 中的逆 S 盒置换。输入参数 u 是 4 位的二进制字符串，被转换为对应的逆 S 盒输出。

(d) linearCryptanalysis 函数：

这个函数实现了线性密码分析算法。输入参数 data 是明文-密文训练数据对，以及 num\_candidate\_keys 表示可能的候选子密钥数量。函数遍历数据集中的每一对明文-密文，



尝试候选子密钥对明文和密文进行线性分析（和伪代码的实现一致）。最终，它返回最佳候选子密钥及其相关性计数。

(e) `createdata` 函数：

这个函数用于生成明文-密文训练数据集。输入参数 `num_samples` 表示要生成的样本数量。函数生成指定数量的随机明文，并使用 SPN 加密算法加密这些明文，将明文-密文对存储在数据集中，并返回数据集。

#### (4) 运行结果

```
Best Candidate Keys: L1 = 3, L2 = 8 with count 162
```

```
D:\A_密码学\10.22\x64\Debug\10.22.exe (进程 21732)已退出，代码为 0。  
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口。 . . .
```