

# Cryptography 2023

学号：2111408 姓名：周钰宸 专业：信息安全IS

## Lab 2: Linear Cryptanalysis 线性攻击算法

### 1 实验内容

- 实现了教材中的线性攻击算法 (P68-69), 基于SPN网络;
- 对课本中给出的例子即推算 $K_2^5$ 和 $K_4^5$ 的线性逼近链推导进行深入理解后: 推算了 $K_1^5$ 和 $K_3^5$ 的线性逼近链公式;
- 使用了上述公式分析出 $K^5$ 轮的密钥, 并进行验证;
- 对时间消耗和性能等进行了简单分析, 思考进一步的优化提升。

### 2 实验原理

#### 2.1 SPN网络(Substitution-Permutation Network):

是一种对称密钥加密结构, 常用于构建分组密码。它是由多个轮组成, 每轮通常包括以下步骤:

- **Substitution (S-box):** 这是一个非线性变换步骤, 其中输入被分为几个较小的分组, 并通过预定义的查找表 (S-box) 进行替换。这提供了加密的混淆属性。
- **Permutation (P-box):** 这是一个线性变换步骤, 其中经过替换的分组经过一个排列步骤, 通常是按位重新排序。这提供了加密的扩散属性。
- **密钥混合:** 在最初的密钥集中会通过某种方式生成密钥编排方案, 对应于密钥编排方案会形成若干子密钥。利用子密钥可以与数据进行异或操作来进一步加密。

这种结构通过多轮的重复使用, 可以达到较强的加密效果。本次实验线性攻击算法就是基于课本中例3.1的给定网络参数结构后实现的。

#### 2.2 线性攻击

**线性攻击** 是一种用于破解分组密码的密码分析技术。其基本思想是尝试找到加密过程中的线性关系, 即明文、密文和密钥之间的某种线性表达式。这些线性关系可以是近似的, 即它们不必在所有情况下都成立。

线性攻击的目标是找到一个与实际加密函数相比更简单的线性方程, 该方程可以用来预测某些明文和密文对的特定行为。当攻击者获得足够多的不重复的明文和密文对 (通常约为8000对) 时, 他们可以使用这些线性近似来猜测加密密钥的某些部分。

这种攻击方法特别适用于那些在其加密结构中存在可利用的线性关系的分组密码。而SPN网络就是具有这样可以利用的特点。

#### 2.3 线性逼近链

不妨设最后一轮密钥组成形式为 $K = K_1K_2K_3K_4$ ,

- **K2与K4:** 基于书中的推导过程, 由活动S盒, 可以结合四个具有较大偏差绝对值的随机变量进行异或来拟合。同时不断地利用后一轮的u和v代替上一轮, 将公式化简为最后只由明文比特、倒数第二轮的u4组成:

$$z_1 = x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4$$

- **K1和K3**: 与书中分析同理, 通过查阅资料可以得到如下的线性逼近链用来分析K1和K3:

$$z_2 = x_1 \oplus x_2 \oplus x_4 \oplus u_1^4 \oplus u_5^4 \oplus u_9^4 \oplus u_{13}^4$$

$$z_3 = x_9 \oplus x_{10} \oplus x_{12} \oplus u_3^4 \oplus u_7^4 \oplus u_{11}^4 \oplus u_{15}^4$$

$$z^* = z_2 + z_3$$

## 3 实验流程

### 3.1 相关变量声明

- **l与m**: SPN网络中的明文可以看作是m个l比特长度的子串的并联。总长为lm。本题中默认l=m=4。
- **Nr**: SPN网络中或者通常的分组密码中的轮数, 本题中也默认Nr=4。
- **char unknownkey[33]**: 线性攻击待解密攻击的密钥。采用char数组的形式存储32位密钥与'\0'。
- **int substitution[16]**: SBox, 以十六进制形式存储, 不过A-F直接使用10-15。
- **int rev\_substituion[16]**: 逆SBox, 用于在第五轮使用逆SBox和v5反推u5, 进而代入线性逼近链的公式。
- **int permutation[16]**: 置换数组, 下标与书中有错位。

### 3.2 解耦

#### 3.2.1 SBox Reversion

对SBox中的内容进行逆转, 求出rev\_substituion[16]并输出相关日志提示。

```
1 void getRevsbstitution() {
2     for (int i = 0; i < 16; i++) {
3         int index;
4         for (int j = 0; j < 16; j++)
5             if (substitution[j] == i)
6                 index = j;
7         rev_substituion[i] = index;
8     }
9     cout << "SBox reversed accomplished!" << endl;
10    return;
11 }
```

#### 3.2.2 BitwiseXor

对使用vector存储的两个向量进行逐位异或, 再返回结果。

```
1 vector<int> bitwiseXor(vector<int> vec1, vector<int> vec2) {
2     vector<int> result(vec1.size());
3     transform(vec1.begin(), vec1.end(), vec2.begin(), result.begin(),
4         [](int a, int b) { return a ^ b; });
5     return result;
6 }
```

### 3.2.3 Bin2Hex

为了以后方便使用，自我实现了底层解耦和二进制向十六进制的转换。输入二进制的vector bin，返回一个vector hex。

但是这里由于我的编程强迫症，每次从bin.back()位置取下次处理的二进制，因此导致是倒过来的...故每次使用该函数处理前，先需要使用3.1.4的reverseVector处理，才能得到正确的十六进制转换结果。

会用在SBox的替换中。

```
1  vector<int> Bin2Hex(vector<int> bin) {
2      vector<int> hex;
3      while (!bin.empty()) {
4          int temp = 0;
5          int i = 0;
6          if (bin.size() < 4) {
7              while (!bin.empty()) {
8                  int s = bin.back();
9                  temp += s * pow(2, 3 - i);
10                 i++;
11                 bin.pop_back();
12             }
13             hex.push_back(temp);
14         }
15         else {
16             while (i < 4) { //若小于4，特殊处理
17                 int s = bin.back();
18                 temp += s * pow(2, 3 - i);
19                 i++;
20                 bin.pop_back();
21             }
22             hex.push_back(temp);
23         }
24     }
25     return hex;
26 }
```

### 3.2.4 ReverseVector

为了能够让Bin2Hex函数正确进行处理转换为十六进制，对vector内容进行翻转。**只会在Bin2Hex使用之前调用该函数进行预处理。**

```
1  void reverseVector(vector<int>& v) {
2      reverse(v.begin(), v.end());
3  }
```

### 3.2.5 Hex2Bin

和Bin2Hex呈相反，传入vector hex后，转换为vector bin。**这次不会出现颠倒需要预处理的情况了。**

会用在SBox的替换。

```
1  vector<int> Hex2Bin(vector<int> hex) {
2      vector<int> bin;
```

```

3  while (!hex.empty()) {
4      int temp = hex.back();
5      vector<int> temp_bin;
6      while (temp) {
7          temp_bin.push_back(temp % 2);
8          temp = temp / 2;
9      }
10     if (temp_bin.size() < 4) //若长度小于4，进行补0
11         while (temp_bin.size() < 4)
12             temp_bin.push_back(0);
13     hex.pop_back();
14     reversevector(temp_bin);
15     bin.insert(bin.begin(), temp_bin.begin(), temp_bin.end());
16 }
17 return bin;
18 }

```

### 3.2.6 GeneratePairs

通过使用Mersenne Twister随机数生成器并跟踪已生成的明文，**确保能够生成指定数量的不重复的明密文对**，将明文通过SPN网络进行转换，得到对应的密文。

```

1  void generatePairs(vector<vector<int>>& plaintext, vector<vector<int>>&
   ciphertext, vector<int>* keyscheme) {
2      random_device rd; //用于获取种子
3      mt19937 gen(rd()); //使用Mersenne Twister随机数生成器
4      uniform_int_distribution<> dis(0, 1); //二进制，所以只生成0和1
5
6      set<vector<int>> generatedPlaintexts; // 用于跟踪已生成的明文
7
8      for (int i = 0; i < T; i++) {
9          vector<int> plain(16); //每个明文都有16位
10
11         do {
12             for (int j = 0; j < 16; j++) {
13                 plain[j] = dis(gen); //随机生成0或1
14             }
15             while (generatedPlaintexts.find(plain) != generatedPlaintexts.end());
16             // 保证明文是唯一的
17
18             generatedPlaintexts.insert(plain);
19             plaintext.push_back(plain);
20
21             // 使用SPN函数得到对应的密文
22             vector<int> cipher;
23             SPN(plain, keyscheme, cipher);
24             ciphertext.push_back(cipher);
25         }
26
27         cout << "Generation accomplished!" << endl;
28         return;
29     }
30 }

```

### 3.3 SPN网络

SPN网络，对OJ上提交的SPN进行了简单的修改，主要用于生成明文后，将其转换为对应的密文的generatePairs函数中。所以多加了一个参数vector&y。

```
1 void SPN(vector<int> x, vector<int>* ks, vector<int>& y) {
2     vector<int>* w = new vector<int>[Nr];
3     vector<int>* u = new vector<int>[Nr + 1];
4     vector<int>* v = new vector<int>[Nr + 1];
5     w[0] = x;
6     for (int r = 1; r <= Nr - 1; r++) {
7         u[r] = bitwiseXor(w[r - 1], ks[r]);
8         for (int i = 1; i <= m; i++) {
9             vector<int> s_input(u[r].begin() + 4 * (i - 1), u[r].begin() + 4 *
10 i); //获取SBox的输入
11             reverseVector(s_input); //转16进制前必须先翻转
12             int s_input_hex = Bin2Hex(s_input).back(); //转为16进制。并且由于默认
13 l=m=Nr=4, 因此转换而来的16进制只有1位
14             vector<int> s_output_hex(1, substitution[s_input_hex]); //
15             vector<int> s_output_bin = Hex2Bin(s_output_hex);
16             vector<int> vr_i = s_output_bin; //得到vr
17             v[r].insert(v[r].end(), vr_i.begin(), vr_i.end()); //插入vr
18         }
19         for (int i = 1; i <= l * m; i++)
20             w[r].push_back(v[r][permutation[i - 1] - 1]); //置换
21     }
22     u[Nr] = bitwiseXor(w[Nr - 1], ks[Nr]); //密钥混合
23     for (int i = 1; i <= m; i++) { //最后一轮
24         vector<int> s_input(u[Nr].begin() + 4 * (i - 1), u[Nr].begin() + 4 * i);
25         reverseVector(s_input);
26         int s_input_hex = Bin2Hex(s_input).back();
27         vector<int> s_output_hex(1, substitution[s_input_hex]);
28         vector<int> s_output_bin = Hex2Bin(s_output_hex);
29         vector<int> vNr_i = s_output_bin;
30         v[Nr].insert(v[Nr].end(), vNr_i.begin(), vNr_i.end());
31     }
32     //这里不进行置换，密钥混合后直接输出
33     y = bitwiseXor(v[Nr], ks[Nr + 1]);
34     return;
35 }
```

### 3.4 Linear Cryptanalysis

#### 3.4.1 K2和K4

依照书中的算法，主要流程如下：

- **构造矩阵与初始化：** candidate\_key\_pairs\_count24为计算K2和K4的z1，同理还有计算K1和K3的z2以及z3。
- **逆代换：** 利用v4\_2和v4\_4求出u4\_2和u4\_4。
- **计算线性逼近链，并计数**
- **找出最大的计数器：** 便是对应的期望密钥。

代码如下：

```

1 void LinearCryptanalysis(vector<vector<int>> x, vector<vector<int>> y,
  vector<int> K5, int* rev_substitution) {
2   int candidate_key_pairs_count24[16][16]; //z1, 先计算K2和K4
3   int candidate_key_pairs_count13_1[16][16]; //z2
4   int candidate_key_pairs_count13_2[16][16]; //z3
5   int candidate_key_pairs_count13[16][16]; //再计算K1和K3
6
7   vector<int>* maxkey = new vector<int>[4]; //存储最终结果
8   for (int L1 = 0; L1 < 16; L1++)
9       for (int L2 = 0; L2 < 16; L2++)
10          candidate_key_pairs_count24[L1][L2] =
11          candidate_key_pairs_count13_1[L1][L2] =
12          candidate_key_pairs_count13_2[L1][L2] =
13          candidate_key_pairs_count13[L1][L2] = 0;
14   cout << "Preparations finished!" << endl;
15   cout << "-----K2 & K4 Analysis-----" << endl;
16   cout << "Begin analyzing K2 and K4:" << endl;
17   int t = 0;
18   while (t < T) {
19       vector<int> xt = x[t];
20       vector<int> yt = y[t];
21       vector<int> yt_1(yt.begin(), yt.begin() + 4);
22       vector<int> yt_2(yt.begin() + 4, yt.begin() + 8);
23       vector<int> yt_3(yt.begin() + 8, yt.begin() + 12);
24       vector<int> yt_4(yt.begin() + 12, yt.begin() + 16);
25       for (int L1 = 0; L1 < 16; L1++) {
26           for (int L2 = 0; L2 < 16; L2++) {
27               vector<int> L1_v_hex(1, L1);
28               vector<int> L1_v_bin = Hex2Bin(L1_v_hex);
29               vector<int> L2_v_hex(1, L2);
30               vector<int> L2_v_bin = Hex2Bin(L2_v_hex);
31               vector<int> v4_2 = bitwiseXor(L1_v_bin, yt_2);
32               vector<int> v4_4 = bitwiseXor(L2_v_bin, yt_4);
33               //对v4_2进行逆代换
34               reverseVector(v4_2);
35               int rev_s_input_hex1 = Bin2Hex(v4_2).back();
36               vector<int> s_output_hex1(1, rev_substitution[rev_s_input_hex1]);
37               vector<int> s_output_bin1 = Hex2Bin(s_output_hex1);
38               vector<int> u4_2 = s_output_bin1;
39               //对v4_4进行逆代换
40               reverseVector(v4_4);
41               int rev_s_input_hex2 = Bin2Hex(v4_4).back();
42               vector<int> s_output_hex2(1, rev_substitution[rev_s_input_hex2]);
43               vector<int> s_output_bin2 = Hex2Bin(s_output_hex2);
44               vector<int> u4_4 = s_output_bin2;
45               //找出等于0的位置
46               int z1 = xt[4] ^ xt[6] ^ xt[7] ^ u4_2[1] ^ u4_2[3] ^ u4_4[1] ^
u4_4[3];
47               if (z1 == 0)
48                   candidate_key_pairs_count24[L1][L2] += 1;
49           }
50       }
51       t++;
52       cout << t << "/" << T << " rounds finished" << endl;
53   }

```

```

54 //找出最大值的位置，遍历
55 int max = -1;
56 for (int L1 = 0; L1 < 16; L1++) {
57     for (int L2 = 0; L2 < 16; L2++) {
58         candidate_key_pairs_count24[L1][L2] =
abs(candidate_key_pairs_count24[L1][L2] - T / 2);
59         if (candidate_key_pairs_count24[L1][L2] > max) {
60             max = candidate_key_pairs_count24[L1][L2];
61             vector<int> L1_v_hex(1, L1);
62             vector<int> L2_v_hex(1, L2);
63             vector<int> K2 = Hex2Bin(L1_v_hex);
64             vector<int> K4 = Hex2Bin(L2_v_hex);
65             maxkey[1] = K2;
66             maxkey[3] = K4;
67         }
68     }
69 }
70 cout << "K2 and K4 analysis finished!" << endl;

```

### 3.4.2 K1和K3

同样的，结合使用上面求解得到的K2和K4，进一步求解K1和K3，主要流程类似，但仍有两点不同：

- **利用K2和K4：** v4\_2和v4\_4可以提前使用之前已经破解的K2和K4异或提前算出，避免重复计算。
- **z\*的计算：** z\*由z2和z3相加得到，而不是简单的直接判断是否为0。

代码如下：（紧接上面部分，两部分连在一起便是完成的LinearCryptanalysis函数）

```

1 //计算K1和K3:
2 cout << "-----K1 & K3 Analysis-----" << endl;
3 cout << "Begin analyzing K1 and K3:" << endl;
4 t = 0;
5 while (t < T) {
6     vector<int> xt = x[t];
7     vector<int> yt = y[t];
8     vector<int> yt_1(yt.begin(), yt.begin() + 4);
9     vector<int> yt_2(yt.begin() + 4, yt.begin() + 8);
10    vector<int> yt_3(yt.begin() + 8, yt.begin() + 12);
11    vector<int> yt_4(yt.begin() + 12, yt.begin() + 16);
12
13    //v4_2和v4_4可以提前使用之前已经破解的K2和K4异或提前算出，避免重复计算
14
15    //异或计算v4_2
16    vector<int> v4_2 = bitwiseXor(maxkey[1], yt_2);
17    //对v4_2进行逆代换
18    reverseVector(v4_2);
19    int rev_s_input_hex2 = Bin2Hex(v4_2).back();
20    vector<int> s_output_hex2(1, rev_substitution[rev_s_input_hex2]);
21    vector<int> s_output_bin2 = Hex2Bin(s_output_hex2);
22    vector<int> u4_2 = s_output_bin2;
23
24    //异或计算v4_4
25    vector<int> v4_4 = bitwiseXor(maxkey[3], yt_4);
26    //对v4_4进行逆代换
27    reverseVector(v4_4);
28    int rev_s_input_hex4 = Bin2Hex(v4_4).back();

```

```

29     vector<int> s_output_hex4(1, rev_substituion[rev_s_input_hex4]);
30     vector<int> s_output_bin4 = Hex2Bin(s_output_hex4);
31     vector<int> u4_4 = s_output_bin4;
32
33     for (int L1 = 0; L1 < 16; L1++) {
34         for (int L2 = 0; L2 < 16; L2++) {
35             vector<int> L1_v_hex(1, L1);
36             vector<int> L1_v_bin = Hex2Bin(L1_v_hex);
37             vector<int> L2_v_hex(1, L2);
38             vector<int> L2_v_bin = Hex2Bin(L2_v_hex);
39
40             vector<int> v4_1 = bitwiseXor(L1_v_bin, yt_1);
41             vector<int> v4_3 = bitwiseXor(L2_v_bin, yt_3);
42
43             //对v4_1进行逆代换
44             reverseVector(v4_1);
45             int rev_s_input_hex1 = Bin2Hex(v4_1).back();
46             vector<int> s_output_hex1(1,
rev_substituion[rev_s_input_hex1]);
47             vector<int> s_output_bin1 = Hex2Bin(s_output_hex1);
48             vector<int> u4_1 = s_output_bin1;
49
50             //对v4_3进行逆代换
51             reverseVector(v4_3);
52             int rev_s_input_hex3 = Bin2Hex(v4_3).back();
53             vector<int> s_output_hex3(1,
rev_substituion[rev_s_input_hex3]);
54             vector<int> s_output_bin3 = Hex2Bin(s_output_hex3);
55             vector<int> u4_3 = s_output_bin3;
56
57             //找出等于0的位置
58             int z2 = xt[0] ^ xt[1] ^ xt[3] ^ u4_1[0] ^ u4_2[0] ^ u4_3[0] ^
u4_4[0];
59             if (z2 == 0)
60                 candidate_key_pairs_count13_1[L1][L2] += 1;
61
62             int z3 = xt[8] ^ xt[9] ^ xt[11] ^ u4_1[2] ^ u4_2[2] ^ u4_3[2] ^
u4_4[2];
63             if (z3 == 0)
64                 candidate_key_pairs_count13_2[L1][L2] += 1;
65         }
66     }
67     t++;
68     cout << t << "/" << T << " rounds finished" << endl;;
69 }
70 //找出最大值的位置，遍历
71 max = -1;
72 for (int L1 = 0; L1 < 16; L1++) {
73     for (int L2 = 0; L2 < 16; L2++) {
74         candidate_key_pairs_count13_1[L1][L2] =
abs(candidate_key_pairs_count13_1[L1][L2] - T / 2);
75         candidate_key_pairs_count13_2[L1][L2] =
abs(candidate_key_pairs_count13_2[L1][L2] - T / 2);

```



```

76         candidate_key_pairs_count13[L1][L2] =
candidate_key_pairs_count13_1[L1][L2] + candidate_key_pairs_count13_2[L1]
[L2];
77         if (candidate_key_pairs_count13[L1][L2] > max) {
78             max = candidate_key_pairs_count13[L1][L2];
79             vector<int> L1_v_hex(1, L1);
80             vector<int> L2_v_hex(1, L2);
81             vector<int> K1 = Hex2Bin(L1_v_hex);
82             vector<int> K3 = Hex2Bin(L2_v_hex);
83             maxkey[0] = K1;
84             maxkey[2] = K3;
85         }
86     }
87 }
88 cout << "K1 and K3 analysis finished!" << endl;
89 cout << "-----Results-----" << endl;
90 //输出结果:
91 cout << "K5_1:";
92 for (int i = 0; i < 4; i++) {
93     cout << maxkey[0][i];
94 }
95 cout << endl;
96 cout << "K5_2:";
97 for (int i = 0; i < 4; i++) {
98     cout << maxkey[1][i];
99 }
100 cout << endl;
101 cout << "K5_3:";
102 for (int i = 0; i < 4; i++) {
103     cout << maxkey[2][i];
104 }
105 cout << endl;
106 cout << "K5_4:";
107 for (int i = 0; i < 4; i++) {
108     cout << maxkey[3][i];
109 }
110 cout << endl;
111 //将结果与实际的密钥对比
112 vector<int> expected_key;
113 for (int i = 0; i < 4; i++)
114     expected_key.insert(expected_key.end(), maxkey[i].begin(),
maxkey[i].end());
115
116 if (equal(expected_key.begin(), expected_key.end(), K5.begin()))
117     cout << "Success!" << endl;
118 else
119     cout << "Failure!" << endl;
120 cout << "LinearCryptanalysis accomplished!" << endl;
121 return;
122 }

```

## 3.6 main函数

主要进行一些以下功能：

- 参数的读入与密钥编排方案的生成
- 关于算法**实现流程日志的输出**便于查错
- 计算程序运行时间，以**衡量性能**
- **调用其他函数**例如LinearCryptanalysis

代码如下：

```
1  int main() {
2  l = m = Nr = 4;
3  cout << "Based on a SPN Network(l=m=Nr=4)," << endl;
4  //开始输入参数T，即明文密文对数
5  cout << "First put in T for pairs of plaintext-cipher text:" << endl;
6  cout << "T=";
7  cin >> T;
8  vector<vector<int>> plaintext;
9  vector<vector<int>> ciphertext;
10 vector<int> key;
11 vector<int>* keyscheme = new vector<int>[6];
12 for (int i = 0; i < 32; i++) // 生成密钥
13     key.push_back(unknownkey[i] - '0');
14
15 for (int i = 1; i <= 5; i++) { //生成对应密钥编排方案
16     for (int j = 4 * i - 3; j <= 4 * i + 12; j++) {
17         keyscheme[i].push_back(key[j - 1]);
18     }
19 }
20 cout << "-----Start Timing-----" << endl;
21 // 获取开始时间点
22 auto start = chrono::high_resolution_clock::now();
23 cout << "Timing started!" << endl;
24 //开始生成明文密文对
25 cout << "-----Generate Pairs-----" << endl;
26 generatePairs(plaintext, ciphertext, keyscheme);
27 //进行SBox的翻转
28 cout << "-----Reverse SBox-----" << endl;
29 getRevsubstitution();
30 cout << "Reverse SBox:" << endl;
31 for (int i = 0; i < 16; i++) {
32     cout << rev_substituion[i] << " ";
33 }
34 cout << endl;
35 //正式开始线性攻击
36 cout << "-----LinearCryptanalysis-----" << endl;
37 LinearCryptanalysis(plaintext, ciphertext, keyscheme[5], rev_substituion);
38 // 获取结束时间点
39 auto stop = chrono::high_resolution_clock::now();
40
41 // 计算花费的时间
42 auto duration = chrono::duration_cast<chrono::seconds>(stop - start);
43 int hours = duration.count() / 3600;
44 int minutes = (duration.count() % 3600) / 60;
```

```

45 int seconds = duration.count() % 60;
46
47 cout << "Time taken: " << hours << " hours " << minutes << " minutes " <<
seconds << " seconds." << endl;
48 return 0;
49 }

```

## 3.7 运行结果展示

### 3.7.1 日志展示

首先使用T=3为例，即3轮明密文对为例进行实验，主要为了能够展示完整的日志信息。

如下图所示：

```

Microsoft Visual Studio 调试控制台
//正式开始线性攻击
cout << "-----L1-----" << endl;
LinearCryptanalysis(g)
// 获取结束时间点
auto stop = chrono::high_resolution_clock::now();
// 计算花费的时间
auto duration = chrono::duration_cast<chrono::seconds>(stop - start);
int hours = duration.count() / 3600;
int minutes = (duration.count() / 60) % 60;
int seconds = duration.count() % 60;
cout << "Time taken: " << hours << " hours " << minutes << " minutes " <<
return 0;

Based on a SPN Network(l=m=Nr=4).
First put in T for pairs of plaintext-cipher text:
T=3
-----Start Timing-----
Timing started!
Generate Pairs
Generation accomplished!
Reverse SBox
SBox reversed accomplished!
Reverse SBox:
14 3 4 8 1 12 10 15 7 13 9 6 11 2 0 5
-----LinearCryptanalysis-----
Preparations finished!
K2 & K4 Analysis
Begin analyzing K2 and K4:
1/3 rounds finished
2/3 rounds finished
3/3 rounds finished
K2 and K4 analysis finished!
K1 & K3 Analysis
Begin analyzing K1 and K3:
1/3 rounds finished
2/3 rounds finished
3/3 rounds finished
K1 and K3 analysis finished!
Results
K5 1:0011
K5 2:0000
K5 3:1001
K5 4:0101
Failure!
LinearCryptanalysis accomplished!
Time taken: 0 hours 0 minutes 0 seconds.

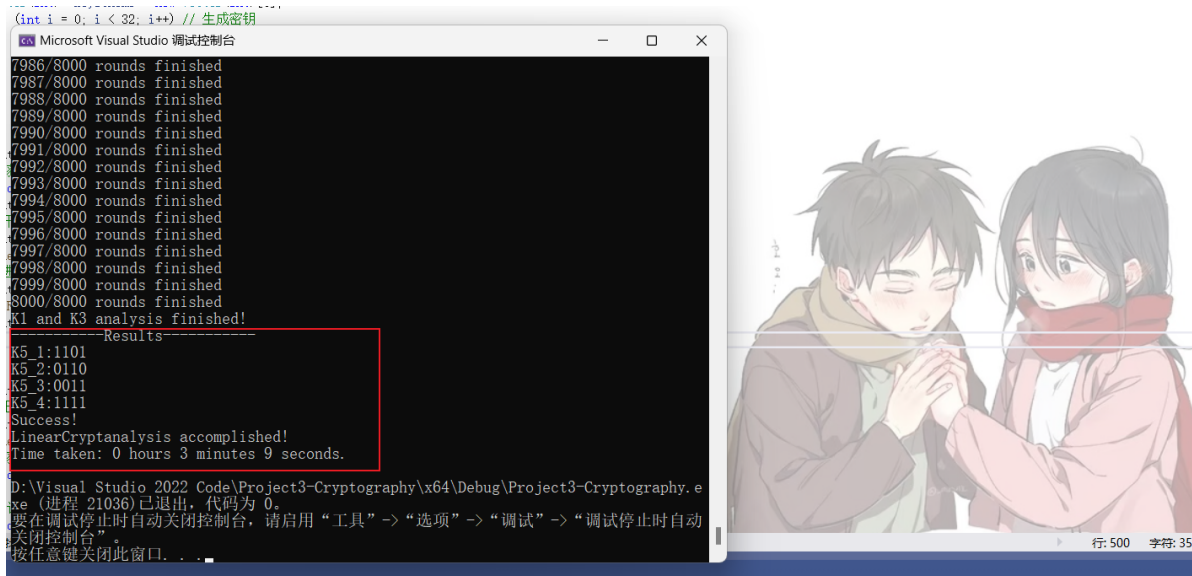
```

可以看到如下日志信息：

- 最开始会提示本次线性攻击是基于 $l=m=Nr=4$ 的SPN网络，即对接下来的攻击具体细节进行了说明；
- 提示开始计时Start Timing，并输出完成信息；
- 提示开始生成明密文对，并输出完成信息；
- 提示开始进行SBox Reversion，并输出完成信息以及对应的逆SBox；
- 提示开始正式进行线性攻击，其中包含两部分即K2&K4与K1&K3，并输出完成轮数的信息；
- 重点输出分析的结果，期望的密钥 $K = K_1K_2K_3K_4$ 以及经过与带破解的密钥对比的结果，输出Success或者是False。并提示线性攻击完成的信息；
- 最后输出总共耗费的时间，以衡量性能；

### 3.7.2 线性攻击结果展示

选取教材中所述的令T=8000，即基于8000对明密文对进行破解，最终结果如下图所示：



```
(int i = 0; i < 32; i++) // 生成密钥
Microsoft Visual Studio 调试控制台
7986/8000 rounds finished
7987/8000 rounds finished
7988/8000 rounds finished
7989/8000 rounds finished
7990/8000 rounds finished
7991/8000 rounds finished
7992/8000 rounds finished
7993/8000 rounds finished
7994/8000 rounds finished
7995/8000 rounds finished
7996/8000 rounds finished
7997/8000 rounds finished
7998/8000 rounds finished
7999/8000 rounds finished
8000/8000 rounds finished
K1 and K3 analysis finished!
Results
K5_1:1101
K5_2:0110
K5_3:0011
K5_4:1111
Success!
LinearCryptanalysis accomplished!
Time taken: 0 hours 3 minutes 9 seconds.
D:\Visual Studio 2022 Code\Project3-Cryptography\x64\Debug\Project3-Cryptography.e
xe (进程 21036) 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. . .
```

### 3.8 结果分析

由3.7.2的线性攻击结果可以发现：

- 总共16000轮破解成功进行，K2&K4以及K1&K3都实验完成。
- 破解密钥结果为K5\_1=1101，K5\_2=0110，K5\_3=0011，K5\_4=1111，这与待破解密钥完全相同，故破解攻击成功！Success！
- 破解密钥总共用时3分9秒，相对来说花费时间较短，程序时间性能较高。

因此可以得出结论，实现的线性攻击算法LinearCrytAnalysis确实可以如教材中理论结果所述，在给定大于8000对的不重复的明文文对的前提下，在已知替换和置换规则的前提下，给定足够的时间（时间性能较强），可以成功地实现攻击。**实验总体来说非常成功！**

### 3.9 待改进与提升

实现的算法仍然有一些可以改进和提升的地方，比如：

- 避免冗余：大部分的代码即破解过程有冗余重复，有待改进；
- 加快性能：vector虽然方便，但重复的动态分配消耗时间性能较高，可以考虑使用其它数据结构作为主要载体；同时也可以考虑使用并行OpenMP等加快破解的时间；
- 解耦实现：解耦实现性能有些较慢，可以考虑使用库函数；
- 进一步破解：在拥有了第五轮全部密钥的能力下，理论上拥有足够多时间和明文文对的我们，是可以推断出全部的密钥的，包括密钥编排方案。因此可以进一步推导更高级的线性逼近链，来进行更完整的攻击。

## 4 实验总结与收获

在本次实验中，通过在OJ上的SPN网络基础上进一步实现了破解其最后一轮的全部密钥 $K = K_1K_2K_3K_4$ 的线性攻击。

这让我不仅在实现的功能的过程中强化了我使用C++编程实现密码算法的能力与输出相对应日志提示信息的编程习惯，更让我加深了对SPN网络以及线性攻击相关原理的更加深刻理解。