《密码学》2024 书面作业 2

信息安全

李佳璐 2211985

1. 请利用线性密码分析确定 SPN 分组加密算法的(轮)密钥

（一）代码文件

random.cpp 生成随机明文

spn.cpp 使用spn加密算法对明文进行加密

analysis.cpp　进行线性密码分析确定轮密钥的2，4块

analysis2.cpp 进行线性密码分析确定轮密钥的1，3块

（二）线性密码分析

下面对分析过程和主要代码进行分析：

使用密钥为：0011 1010 1001 0100 1101 0110 0011 1111
第五轮密钥为：1101 0110 0011 1111

1. 生成8000组随机16位二进制明文

```
std::string generate_random_binary_string(int length) {
    std::string binary_string;
    for (int i = 0; i < length; ++i) {
        binary_string += (rand() % 2) ? '1' : '0'; // 随机生成 '0' 或 '1'
    }
    return binary_string;
}

int main() {
    srand(static_cast<unsigned int>(time(0))); // 设置随机种子
    const int num_strings = 8000;
    const int length = 16;
    std::vector<std::string> binary_strings;

    // 生成 8000 个 16 位的随机二进制明文
    for (int i = 0; i < num_strings; ++i) {
        binary_strings.push_back(generate_random_binary_string(length));
    }
    return 0;
}
```

并将这8000个明文存储到"random_ming.txt“中

2. 使用spn分组加密算法对8000组明文进行加密

使用密钥为： 0011 1010 1001 0100 1101 0110 0011 1111

实现算法与上次作业算法相同

```
void SPN(int* x, int* y, int* s, int* p, int* key)
{
    int w[16], u[16], v[16], k[16];
    for (int i = 0; i < 16; i++)
    {
        w[i] = x[i];
    }
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 16; j++)
        {
            k[j] = key[4 * i + j];
        }

        for (int j = 0; j < 16; j++)
        {
            u[j] = w[j] ^ k[j];
        }

        Substitution(u, v, s);
        Permutation(v, w, p);
    }

    for (int j = 0; j < 16; j++)
    {
        k[j] = key[4 * 3 + j];
    }

    for (int j = 0; j < 16; j++)
    {
        u[j] = w[j] ^ k[j];
    }

    Substitution(u, v, s);

    for (int j = 0; j < 16; j++)
    {
        k[j] = key[4 * 4 + j];
    }

    for (int j = 0; j < 16; j++)
    {
        y[j] = v[j] ^ k[j];
    }
}
```

并将最后生成的明密文对存储到`spn_pairs.txt`中。

3. 确定第五轮轮密钥的第2，4块

这是本次作业的核心部分

SPN线性密码分析：

- 基于S盒的线性逼近导出整个SPN的线性逼近；
- 是已知明文的分析方法，需要较多的名密文对，如果取T=8000，这个攻击通常会成功（所以设置明密文对为8000组）。
- 使用明文-密文对，统计通过线性逼近的假设条件是否成立的频率。如果实际频率和预期频率之间存在显著差异，则可以使用该偏差来推测出与该差异相关的密钥位。

计算第五轮密钥的2，4块的思路：

根据4个活动S盒的4个随机变量，在假设相互独立的条件下，利用堆积定理对其计算异或的偏差。

将随机变量进行变换以及右端异或的操作，并进行代入下轮密钥或固定比特0，1值得到随机变量

具有偏离0的偏差可以进行线性密码攻击。

编写代码实现SPN线性密码分析：
参照书中伪代码：

算法 3.2　线性攻击 (T, T, π_S^{-1})
for $(L_1, L_2) \leftarrow (0,0)$ **to** (F, F)
　do $\text{Count}[L_1, L_2] \leftarrow 0$
for each $(x, y) \in T$
　do
　　for $(L_1, L_2) \leftarrow (0,0)$ **to** (F, F)
　　do
　　$v_{<2>}^4 \leftarrow L_1 \oplus y_{<2>}$
　　$v_{<4>}^4 \leftarrow L_2 \oplus y_{<4>}$
　　$u_{<2>}^4 \leftarrow \pi_S^{-1}(v_{<2>}^4)$
　　$u_{<4>}^4 \leftarrow \pi_S^{-1}(v_{<4>}^4)$
　　$z \leftarrow x_5 \oplus x_7 \oplus x_8 \oplus u_6^4 \oplus u_8^4 \oplus u_{14}^4 \oplus u_{16}^4$
　　if $z = 0$
　　then $\text{Count}[L_1, L_2] \leftarrow \text{Count}[L_1, L_2] + 1$
$\max \leftarrow -1$
for $(L_1, L_2) \leftarrow (0,0)$ **to** (F, F)
　do
　　$\text{Count}[L_1, L_2] \leftarrow |\text{Count}[L_1, L_2] - T/2|$
　　if $\text{Count}[L_1, L_2] > \max$
　　then
　　$\max \leftarrow \text{Count}[L_1, L_2]$
　　$\max\text{key} \leftarrow (L_1, L_2)$
output(maxkey)

首先通过逐行读取明文和密文，分析每个明文与密文之间的线性关系。使用二进制运算和 S-盒逆运算，进行统计分析。并将满足条件的 `j` 和 `k` 的计数结果存储在 `Count` 数组中。

```
for (int i = 0; i < n; i++)
{
    string X, Y;
    input >> X >> Y;

    for (int j = 0; j < 16; j++) {
        x[j] = X[j] - '0';
```

```
        y[j] = Y[j] - '0';
    }

    for (int j = 0; j < 16; j++)
    {
        for (int k = 0; k < 16; k++)
        {
            // for(L_1,L_2) <- (0,0) to (F,F)
            DecimalToBinary(j, L1, 3);
            DecimalToBinary(k, L2, 3);

            // 计算 v 的值
            for (int l = 0; l < 4; l++) {
                v[4 + l] = L1[l] ^ y[4 + l]; // L1 与 y 的异或
                v[12 + l] = L2[l] ^ y[12 + l]; // L2 与 y 的异或
            }

            // S-盒逆运算
            int temp1 = (v[4] << 3) | (v[5] << 2) | (v[6] << 1) | v[7];
            int temp2 = S1[temp1]; // S-盒的逆运算
            DecimalToBinary(temp2, u, 7);

            temp1 = (v[12] << 3) | (v[13] << 2) | (v[14] << 1) | v[15];
            temp2 = S1[temp1]; // S-盒的逆运算
            DecimalToBinary(temp2, u, 15);

            // 计算 z
            int z = x[4] ^ x[6] ^ x[7] ^ u[5] ^ u[7] ^ u[13] ^ u[15];
            // 更新计数
            if (z == 0) Count[j][k]++;
        }
    }
}
```

获取最大概率的轮密钥

```
int max = -1;
int LL1 = 0, LL2 = 0;
for (int i = 0; i < 16; i++)
{
    for (int j = 0; j < 16; j++)
    {
        Count[i][j] = abs(Count[i][j] - n / 2);
        if (Count[i][j] > max)
        {
            max = Count[i][j];
            LL1 = i;
            LL2 = j;
        }
    }
}
```

执行代码得到第五轮密钥的2，4块

执行无误，这部分轮密钥分析正确。

4. 已知轮密钥2，4块计算第五轮轮密钥的1，3块

在已知第五轮轮密钥2，4块的基础上计算1，3块。

`get_spBox()`函数将4位S盒扩展为16位，并构建其对应的置换盒，为SPN加密算法提供必要的S盒和置换映射。

```
void get_spBox() {
    // 预计算 S 盒和 P 盒
    for (int i = 0; i < 65536; ++i) {
        // 计算 sBox_16[i]，将 4 位 S 盒扩展为 16 位
        sBox_16[i] = (sBox_4[i >> 12] << 12) |
            (sBox_4[(i >> 8) & 0xf] << 8) |
            (sBox_4[(i >> 4) & 0xf] << 4) |
            sBox_4[i & 0xf];

        // 计算 spBox[i]
        spBox[i] = 0; // 初始化为 0
        for (int j = 1; j <= 16; ++j) {
            // 通过位运算检查 sBox_16[i] 的对应位并更新 spBox
            if (sBox_16[i] & pos[j]) {
                spBox[i] |= pBox[j]; // 更新 spBox[i]
            }
        }
    }
}
```

遍历 8000 对明文和密文数据，对于每一组可能的 L1 和 L2（均为 0 到 15），使用逆 S 盒（`inverse_sBox`）计算出 u1、u2、u3 和 u4。这些值是通过与密钥（`key52` 和 `key54`）及 L1、L2 进行异或运算得到的。

计算了两个不同的 `z` 值，通过位与运算（`& 0x1`）提取最低位。如果结果为 0，则更新 `count13` 数组的相应计数。这两个条件分别针对不同的输入比特进行统计。

```
for (int group = 0; group < 8000; ++group) { // 从 0 开始，数组从 0 开始
    x_init = plaintext[group];
    for (int pos = 0; pos < 12; ++pos) {
        x[pos] = (x_init >> (15 - pos)) & 0x1; // pos 从 0 到 11
    }

    y_init = ciphertext[group];
    for (int pos = 0; pos < 4; ++pos) {
        y[pos] = (y_init >> (12 - 4 * pos)) & 0xf; // pos 从 0 到 3
    }

    for (int L1 = 0; L1 < 16; ++L1) {
        for (int L2 = 0; L2 < 16; ++L2) {
            // 进行 S 盒的逆运算
```

```
            u1 = inverse_sBox[y[0] ^ L1];
            u2 = inverse_sBox[y[1] ^ key52];
            u3 = inverse_sBox[y[2] ^ L2];
            u4 = inverse_sBox[y[3] ^ key54];

            // 计算 z 的值并更新计数
            if (!((x[0] ^ x[1] ^ x[3] ^ (u1 >> 3) ^ (u2 >> 3) ^ (u3 >> 3) ^ (u4
>> 3)) & 0x1)) {
                count13[0][L1][L2]++;
            }

            if (!((x[8] ^ x[9] ^ x[11] ^ (u1 >> 1) ^ (u2 >> 1) ^ (u3 >> 1) ^ (u4
>> 1)) & 0x1)) {
                count13[1][L1][L2]++;
            }
        }
    }
}

for (int L1 = 0; L1 < 16; ++L1) {
    for (int L2 = 0; L2 < 16; ++L2) {
        abs_val(count13[0][L1][L2]);
        abs_val(count13[1][L1][L2]);
        cnt13[L1][L2] = count13[0][L1][L2] + count13[1][L1][L2];
    }
}
```

确定与已知差分特性最相关的密钥位 key51 和 key53。

```
int max13 = -1;
for (int L1 = 0; L1 < 16; ++L1) {
    for (int L2 = 0; L2 < 16; ++L2) {
        if (cnt13[L1][L2] > max13) {
            max13 = cnt13[L1][L2];
            key51 = L1;
            key53 = L2;
        }
    }
}
```

运行结果如下

计算出第五轮密钥的1，3块，计算无误。

2. 请计算国密分组加密算法 SM4 的 SBox 差分分布表。

sm4.cpp

```
// SM4 的 S-Box，以二维数组形式表示
const unsigned char SBox[16][16] = {
```

```
    {0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14, 0xc2,
0x28, 0xfb, 0x2c, 0x05},
    {0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13, 0x26,
0x49, 0x86, 0x06, 0x99},
    {0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b, 0x43,
0xed, 0xcf, 0xac, 0x62},
    {0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94, 0xfa,
0x75, 0x8f, 0x3f, 0xa6},
    {0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c, 0x19,
0xe6, 0x85, 0x4f, 0xa8},
    {0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f, 0x4b,
0x70, 0x56, 0x9d, 0x35},
    {0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c, 0x3b,
0x01, 0x21, 0x78, 0x87},
    {0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02, 0xe7,
0xa0, 0xc4, 0xc8, 0x9e},
    {0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2, 0xce,
0xf9, 0x61, 0x15, 0xa1},
    {0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32, 0x30,
0xf5, 0x8c, 0xb1, 0xe3},
    {0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23, 0xab,
0x0d, 0x53, 0x4e, 0x6f},
    {0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a, 0x72,
0x6d, 0x6c, 0x5b, 0x51},
    {0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c, 0x41,
0x1f, 0x10, 0x5a, 0xd8},
    {0x0a, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0, 0x12,
0xb8, 0xe5, 0xb4, 0xb0},
    {0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1, 0x09,
0xc5, 0x6e, 0xc6, 0x84},
    {0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f, 0x3e,
0xd7, 0xcb, 0x39, 0x48}
};
```

创建一个 256x256 的表，记录每对输入和输出差分的出现次数，对于每一个可能的输入差分（0-255），遍历所有输入值（0-255）。
对每个输入值计算其 S-Box 输出以及其与当前输入差分异或后的 S-Box 输出。通过异或操作得到两个输出值之间的差分。

```
// 计算差分分布表
void computeDDT(int ddt[256][256]) {
    // 初始化 DDT
    for (int i = 0; i < 256; i++) {
        for (int j = 0; j < 256; j++) {
            ddt[i][j] = 0;
        }
    }

    // 遍历所有可能的输入差分
    for (int deltaX = 0; deltaX < 256; deltaX++) {
        for (int x = 0; x < 256; x++) {
            // 计算输入值 y
            int input = x;
            int inputXorDelta = input ^ deltaX;

            // 计算 S-Box 的输出
            unsigned char output = SBox[input >> 4][input & 0x0F];
```

```
            unsigned char outputXorDelta = SBox[inputXorDelta >> 4]
[inputXorDelta & 0x0F];

            // 计算输出差分
            int deltaY = output ^ outputXorDelta;

            // 更新 DDT
            ddt[deltaX][deltaY]++;
        }
    }
}
```

并将差分表输出为csv文件。

差分表为256×256的表格见文件 DDT.csv， 部分如下：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	256	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	2	2	2	2	0	0	2	0	2	2	0	0	0	2	2	0	2
2	0	0	2	2	2	0	2	2	0	2	2	0	2	0	0	0	2	2
3	0	0	0	0	0	2	0	0	2	0	2	0	2	0	0	0	0	0
4	0	2	0	2	2	2	2	0	0	0	0	2	2	2	2	2	0	0
5	0	2	0	0	2	2	2	4	2	0	2	2	2	2	2	2	0	2
6	0	2	0	0	2	0	2	2	0	0	2	2	2	0	0	0	0	0
7	0	2	2	0	0	0	2	0	2	0	0	0	0	0	0	2	2	0
8	0	2	2	0	0	0	2	0	0	0	0	0	0	2	0	2	0	2
9	0	2	2	2	0	2	0	0	0	2	2	2	2	2	0	0	0	0
10	0	2	2	2	2	2	0	2	0	2	2	0	0	0	0	0	2	2
11	0	2	2	0	2	2	0	0	0	2	0	0	0	2	0	0	0	2
12	0	0	2	0	2	2	0	2	0	0	0	2	0	0	0	2	2	0
13	0	2	0	2	2	0	0	0	2	0	0	0	0	0	2	2	2	0
14	0	2	0	0	2	2	0	0	2	2	2	0	0	0	0	2	0	0
15	0	0	0	2	2	2	2	0	0	0	0	0	2	2	2	2	0	2