南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

密码学实验

分组密码算法 AES

学号：2013018

姓名：许健

年级：2020 级

专业：信息安全

2022 年 12 月 8 日

目录

一、 引言

(一) 实验目的

通过用 AES 算法对实际的数据进行加密和解密来深刻了解 AES 的运行原理。

(二) 实验内容

1. 对课本中 AES 算法进行深入分析，对其中用到的基本数学算法、字节代换、行移位变换、列混混变换原理进行详细的分析，并考虑如何进行编程实现。对轮函数、密钥生成等环节要有清晰的了解，并考虑其每一个环节的实现过程。
2. 在第一步的基础上，对整个 AES 加密函数的实现进行总体设计，考虑数据的存储格式，参数的传递格式，程序实现的总体层次等，画出程序实现的流程图。
3. 在总体设计完成后，开始具体的编码，在编码过程中，注意要尽量使用高效的编码方式。
4. 利用 3 中实现的程序，对 AES 的密文进行雪崩效应检验。即固定密钥，仅改变明文中的一位，统计密文改变的位数；固定明文，仅改变密钥中的一位，统计密文改变的位数。

(三) 实验要求

1. 实现 AES 的加密和解密，提交程序代码和执行结果。
2. 在检验雪崩效应中，要求至少改变明文和密文中各八位，给出统计结果并计算出平均值。

二、 AES 算法详解

AES 算法本质上是一种对称分组密码体制，采用代替/置换网络，每轮由三层组成：线性混合层确保多轮之上的高度扩散，非线性层由 16 个 S 盒并置起到混淆的作用，密钥加密层将子密钥异或到中间状态。

Rijndael 是一个迭代分组密码，其分组长度和密钥长度都是可变的，只是为了满足 AES 的要求才限定处理的分组大小为 128 位，而密钥长度为 128 位、192 位或 256 位，相应的迭代轮数 N，为 10 轮、12 轮、14 轮。AES 汇聚了安全性能、效率、可实现性、灵活性等优点。最大的优点是可以给出算法的最佳差分特征的概率，并分析算法抵抗差分密码分析及线性密码分析的能力。

(一) AES 加密解密流程

DES 机密解密的流程如图1所示。加密的主要过程包括：对明文状态的一次密钥加，$N_r - 1$ 轮轮加密和末尾轮轮加密，最后得到密文。其中 $N_r - 1$ 轮轮加密每一轮有四个部件，包括字节代换部件 ByteSub、行移位变换 ShiftRow、列混合变换 MixColumn 和一个密钥加 AddRoundKey 部件，末尾轮加密和前面轮加密类似，只是少了一个列混合变换 MixColumn 部件。

AES 算法的解密过程和加密过程是相似的，也是先经过一个密钥加，然后进行 $N_r - 1$ 轮轮解密和末尾轮轮解密，最后得到明文。和加密不同的是 $N_r - 1$ 轮轮解密每一轮四个部件都需要用到它们的逆运算部件，包括字节代换部件的逆运算、行移位变换的逆变换、逆列混合变换和一个密钥加部件，末尾轮加密和前面轮加密类似，只是少了一个逆列混合变换部件。

在解密的时候，还要注意轮密钥和加密密钥的区别，设加密算法的初始密钥加、第 1 轮、第 2 轮、...、第 N_r 轮的子密钥依次为 k(0), k(1), k(2), $\cdots$, k($Nr-1$), k(N_r)。则解密算法的初

始密钥加、第 1 轮、第 2 轮、⋯、第 Nr 轮的子密钥依次为 k(N_r)，InvMixColumn(k(N_{r-1}))，InvMixColumn(N_{r-2})，...，InvMixColumn (k(1)), k(0)。

下面具体介绍轮加密中每个部件的实现方法。

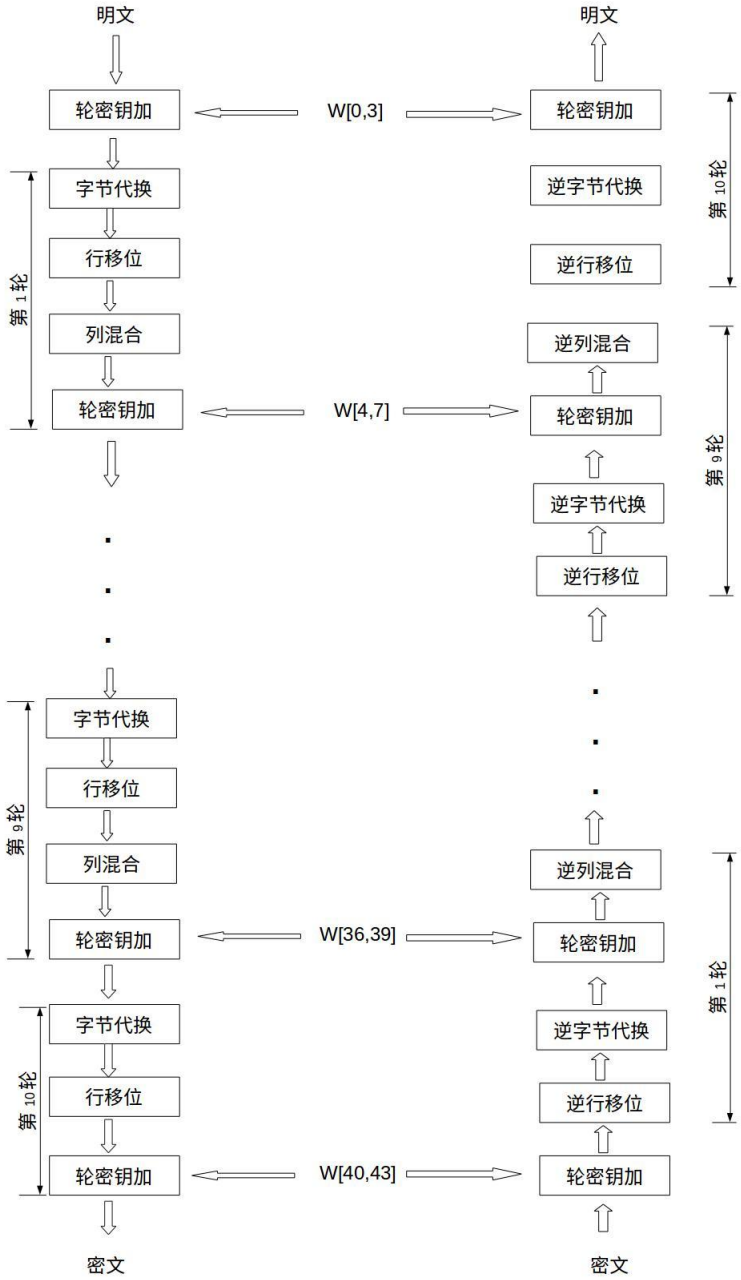

图 1: AES 加密解密流程

(二) 字节代换 ByteSub 部件

字节代换是非线形变换，独立地对状态的每个字节进行。代换表（即 S-盒）是可逆的，由以下两个变换的合成得到：

1. 首先，将字节看作 GF(2^8) 上的元素，映射到自己的乘法逆元，‘00’映射到自己。

2. 其次，对字节做如下的 (GF(2) 上的，可逆的) 仿射变换：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

图 2: GF(2) 上的仿射变换

该部件的逆运算部件就是先对自己做一个逆仿射变换，然后映射到自己的乘法逆元上。

(三) 行移位变换 ShiftRow

行移位是将状态阵列的各行进行循环移位，不同状态行的位移量不同。第 0 行不移动，第 1 行循环左移 C_1 个字节，第 2 行循环左移 C_2 个字节，第 3 行循环左移 C_3 个字节。位移量 C_1、C_2、C_3 的取值与 Nb 有关。

ShiftRow 的逆变换是对状态阵列的后 3 列分别以位移量 $N_b - C_1$、$N_b - C_2$、$N_b - C_3$ 进行循环移位，使得第 i 行第 j 列的字节移位到 $(j + N_b - C_i) mod N_b$。

(四) 列混合变换 MixColumn

在列混合变换中，将状态阵列的每个列视为系数为 $GF(2^8)$ 上的多项式，再与一个固定的多项式 c(x) 进行模 x^4+1 乘法。当然要求 c(x) 是模 x^4+1 可逆的多项式，否则列混合变换就是不可逆的，因而会使不同的输入分组对应的输出分组可能相同。Rijndael 的设计者给出的 c(x) 为 (系数用十六进制数表示)：

$$c(x) = \text{‘03’}x^3 + \text{‘01’}x^2 + \text{‘01’}x + \text{‘02’}$$

c(x) 是与 $x^4 + 1$ 互素的，因此是模 $x^4 + 1$ 可逆的。列混合运算也可写为矩阵乘法。设 b(x) = c(x) ⊗ a(x)，则

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

图 3: 列混合运算

这个运算需要做 $GF(2^8)$ 上的乘法，但由于所乘的因子是 3 个固定的元素 02、03、01，所以这些乘法运算仍然是比较简单的。

列混合运算的逆运算是类似的，即每列都用一个特定的多项式 d(x) 相乘。d(x) 满足

$$\left(03x^3 + 01x^2 + 01x + 02\right) \otimes d(x) = 01$$

由此可得 $d(x) = \text{‘}0B\text{’}x^3 + \text{‘}0D\text{’}x^2 + \text{‘}09\text{’}x + \text{‘}0E\text{’}$

（五） 密钥加 AddRoundKey 部件

密钥加是将轮密钥简单地与状态进行逐比特异或。轮密钥由种子密钥通过密钥编排算法得到，轮密钥长度等于分组长度 Nb。密钥加运算的逆运算是其自身。

（六） 密钥编排

密钥编排指从种子密钥得到轮密钥的过程，它由密钥扩展和轮密钥选取两部分组成。其基本原则如下：

1. 轮密钥的字数 (4 比特 32 位的数) 等于分组长度乘以轮数加 1；
2. 种子密钥被扩展成为扩展密钥；
3. 轮密钥从扩展密钥中取，其中第 1 轮轮密钥取扩展密钥的前 N_b 个字，第 2 轮轮密钥取接下来的 N_b 个字，如此下去。

算法 3.6 KeyExpansion(key)
external RotWord, SubWord
RCon[1] ← 01000000
RCon [2] ← 02000000
RCon [3] ← 04000000
RCon [4] ← 08000000
RCon [5] ← 10000000
RCon [6] ← 20000000
RCon [7] ← 40000000
RCon [8] ← 80000000
RCon [9] ← 1B000000
RCon [10] ← 36000000
for $i \leftarrow 0$ **to** 3
  do $w[i] \leftarrow (\text{key}[4i], \text{key}[4i+1], \text{key}[4i+2], \text{key}[4i+3])$
for $i \leftarrow 4$ **to** 43
  do
    temp ← $w[i-1]$
    if $i \equiv 0 (\text{mod } 4)$
      then temp ← SubWord(RotWord(temp)) ⊕ RCon[$i/4$]
    $w[i] \leftarrow w[i-4] \oplus$ temp
return $(w[0], \cdots, w[43])$

图 4: 密钥扩展算法伪代码

轮密钥 i(即第 i 个轮密钥) 由轮密钥缓冲字 $W[N_b * i]$ 到 $W[N_b * (i+1) - 1]$ 给出

三、 AES 编码

(一) 总体设计

1. 数据的存储格式

类似于明文分组和密文分组，算法的中间结果也需分组，称算法中间结果的分组为状态，所有的操作都在状态上进行。状态可以用以字节为元素的矩阵阵列表示，该阵列有 4 行，列数记为 N_b，N_b 等于分组长度除以 32。

有时可将这些分组当作一维数组，其每一元素是上述阵列表示中的 4 字节元素构成的列向量，数组长度可为 4、6、8，4 字节元素构成的列向量有时也称为 4。

本次实验中，我们定义 word 结构体，用来表示列向量，并编写用来将 byte 字节序列转为 word 数组、打印 word 数组的函数。

```
typedef unsigned char u_char;
struct word {
    u_char byte[4];
};
```

word 相关辅助函数

```
void BytetoWord(word* state, const u_char* result, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] = result[i * N + j];
        }
    }
}
void printWord(word* state, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%02x ", state[i].byte[j]);
        }
        cout << endl;
    }
    cout << endl;
}
```

2. 参数的传递格式

在传递给 AES 加密、解密函数的参数中，明文、密文对均为 word* 指针，也就是说算法的所有操作都在状态上进行，密钥为无符号类型字符数组。

```
void AES_Encryption(const word* plaintext, word* ciphertext, const u_char* key);
void AES_Decryption(word* plaintext, const word* ciphertext, const u_char* key);
```

3. 程序实现的总体层次

AES 加解密过程中会用到许多不同的部件，因此采用模块化编程的方式，对于每一个部件编写一个函数，在主函数中传递必要的参数调用它们，实现最终的功能。

从 AES 加密函数中可以看到，调用的部件包括：字节代换 ByteSub 部件、行移位变换 ShiftRow、列混合变换 MixColumn、密钥加 AddRoundKey 部件。对于密钥扩展我们也单独编写一个函数，将子密钥存储在全局变量 wordKey 中。

AES 加密函数

```
void AES_Encryption(const word* plaintext, word* ciphertext, const u_char*
    key) {
    //密钥扩展
    KeyExpansion(key, wordKey);
    //密文初始化
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            ciphertext[i].byte[j] = plaintext[i].byte[j];
        }
    }
    AddRoundKey(ciphertext, 0);
    //Nr-1轮轮函数
    for (int i = 1; i < Nr; i++) {
        ByteSub(ciphertext);
        ShiftRow(ciphertext);
        MixColumn(ciphertext);
        AddRoundKey(ciphertext, i);
    }
    //结尾轮
    ByteSub(ciphertext);
    ShiftRow(ciphertext);
    AddRoundKey(ciphertext, Nr);
}
```

AES 解密过程调用的部件是加密过程的逆部件，并没有本质的区别。

AES 解密函数

```
void AES_Decryption(word* plaintext, const word* ciphertext, const u_char*
    key) {
    //密钥扩展
    KeyExpansion(key, wordKey);
    //明文初始化
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            plaintext[i].byte[j] = ciphertext[i].byte[j];
        }
    }
    AddRoundKey(plaintext, Nr);
    //Nr-1轮轮函数
    for (int i = Nr-1; i > 0; i--) {
```

```
        InvShiftRow(plaintext);
        InvByteSub(plaintext);
        AddRoundKey(plaintext, i);
        InvMixColumn(plaintext);
    }
    //结尾轮
    InvShiftRow(plaintext);
    InvByteSub(plaintext);
    AddRoundKey(plaintext, 0);
}
```

4. 程序实现的流程图

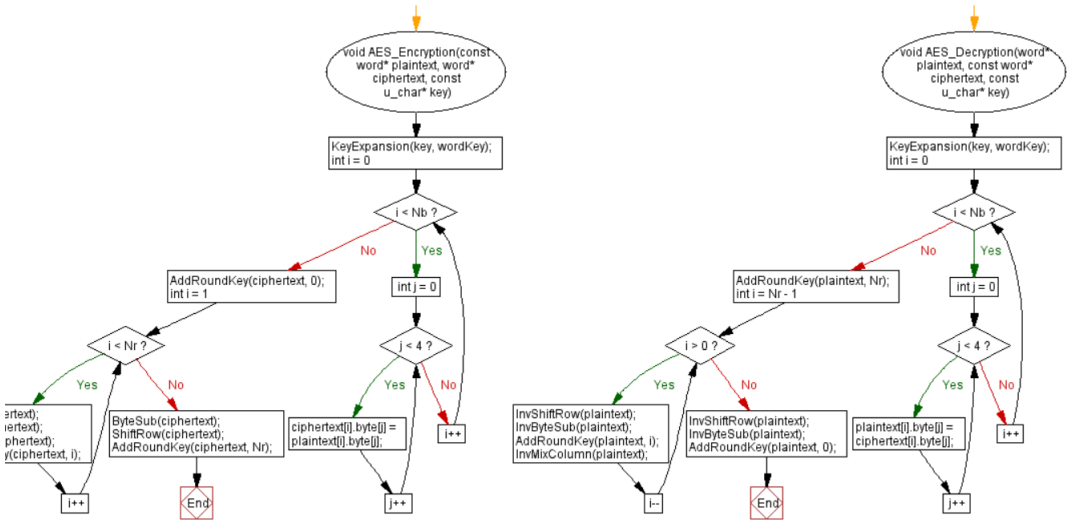

图 5: 程序实现流程图

(二) 具体实现

1. 字节代换 ByteSub

ByteSub() 通过事先编写好的 S 盒，即可将状态 state 中的每个列向量的每个字节都进行代换，其逆变换则是使用 InvS 盒，除此之外没有任何变化。对于 S 盒和 InvS 盒，不在给出具体定义。

字节代换 ByteSub

```
void ByteSub(word* state) {
    u_char L, R;
    for(int i=0;i<Nb;i++)
    for (int j = 0; j < 4; j++) {
        L = state[i].byte[j] >> 4;
        R = state[i].byte[j] & 0x0f;
        state[i].byte[j] = SBox[L][R];
    }
```

```
}
void InvByteSub(word* state) {
    u_char L, R;
    for (int i = 0; i < Nb; i++)
        for (int j = 0; j < 4; j++) {
            L = state[i].byte[j] >> 4;
            R = state[i].byte[j] & 0x0f;
            state[i].byte[j] = InvSBox[L][R];
        }
}
```

2. 行移位变换 ShiftRow

AES 行移位变换的原理如图6所示，下面给出具体的算法实现及其逆变换。

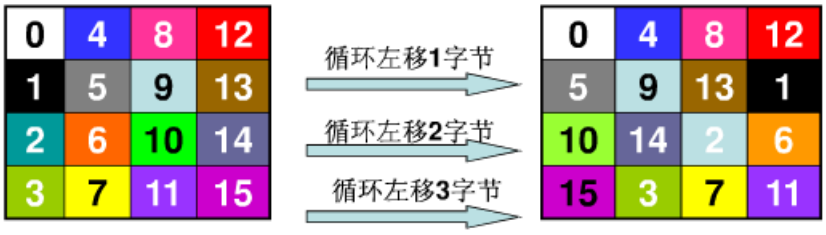

图 6: 行移位变换

行移位变换 ShiftRow

```
void ShiftRow(word* state) {
    word tmp[4];
    int C[4] = { 0,1,2,3 };
    if (Nb == 8) {
        C[2] = 3;
        C[3] = 4;
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            tmp[i].byte[j] = state[(i + C[j]) % Nb].byte[j];
        }
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] = tmp[i].byte[j];
        }
    }
}
void InvShiftRow(word* state) {
    word tmp[4];
    int C[4] = { 0,1,2,3 };
```

```
    if (Nb == 8) {
        C[2] = 3;
        C[3] = 4;
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            tmp[i].byte[j] = state[(i - C[j] + Nb) % Nb].byte[j];
        }
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] = tmp[i].byte[j];
        }
    }
}
```

3. 列混合变换 MixColumn

AES 列混合变换的原理如图7所示，下面给出具体的算法实现及其逆变换。

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} = \begin{bmatrix} s'_{00} & s'_{01} & s'_{02} & s'_{03} \\ s'_{10} & s'_{11} & s'_{12} & s'_{13} \\ s'_{20} & s'_{21} & s'_{22} & s'_{23} \\ s'_{30} & s'_{31} & s'_{32} & s'_{33} \end{bmatrix}$$

图 7: 列混合变换

列混合变换 MixColumn

```
void MixColumn(word* state) {
    word* temp = new word[Nb];
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            temp[i].byte[j] = GFMultiplyByte(MixColumnMatrix[j][0], state[i].
                byte[0]);
            for (int k = 1; k < 4; k++) {
                temp[i].byte[j] ^= GFMultiplyByte(MixColumnMatrix[j][k],
                    state[i].byte[k]);
            }
        }
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] = temp[i].byte[j];
        }
    }
}
```

```
void InvMixColumn(word* state) {
    word* temp = new word[Nb];
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            temp[i].byte[j] = GFMultiplyByte(InvMixColumnMatrix[j][0], state[
                i].byte[0]);
            for (int k = 1; k < 4; k++) {
                temp[i].byte[j] ^= GFMultiplyByte(InvMixColumnMatrix[j][k],
                    state[i].byte[k]);
            }
        }
    }
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] = temp[i].byte[j];
        }
    }
}
//GFMultiplyByte函数返回两个字节在GF(2^8)上相乘的结果
u_char GFMultiplyByte(u_char L, u_char R) {
    u_char temp[8];
    u_char result = 0x00;
    temp[0] = L;
    for (int i = 1; i < 8; i++) {
        if (temp[i - 1] >= 0x80) {
            temp[i] = (temp[i - 1] << 1) ^ 0x1b;
        }else {
            temp[i] = temp[i - 1] << 1;
        }
    }
    for (int i = 0; i < 8; i++) {
        if (((R >> i) & 0x01) == 1) {
            result ^= temp[i];
        }
    }
    return result;
}
```

4. 密钥加 AddRoundKey

密钥加用来将状态 state 与轮密钥做字节异或操作，下面给出具体的算法实现。

密钥加 AddRoundKey

```
void AddRoundKey(word* state, int round) {
    for (int i = 0; i < Nb; i++) {
        for (int j = 0; j < 4; j++) {
            state[i].byte[j] ^= wordKey[i + Nb * round].byte[j];
        }
```

```
    }
}
```

5. 轮密钥生成

对于 $N_k <= 6$ 和 $N_k > 6$，AES 扩展算法会有所差异，密钥扩展过程中会使用循环移位 RotWord、S 盒变换 SubWord、异或轮常数 WordXOR 等部件，在此给出具体的算法实现以及每个部件的实现代码。

轮密钥生成

```
//密钥扩展算法
void KeyExpansion(const u_char* key, word* w) {
        initRcon();
        for (int i = 0; i < Nk; i++) {
                for (int j = 0; j < 4; j++) {
                        w[i].byte[j] = key[4 * i + j];
                }
        }
        for (int i = Nk; i < Nb * (Nr + 1); i++) {
                word temp = w[i - 1];
                if (i % Nk == 0) {
                        temp = RotWord(temp);
                        temp = SubWord(temp);
                        temp = WordXOR(temp, Rcon[i / Nk]);
                }else if (Nk > 6 && (i % Nk) == 4) {
                        temp = SubWord(temp);
                }
                w[i] = WordXOR(w[i - Nk], temp);
        }
}
void initRcon() {//初始化轮常数Rcon
    for (int i = 0; i < Nr + 1; i++) {
        for (int j = 0; j < 4; j++) {
            Rcon[i].byte[j] = 0x0;
        }
    }
    Rcon[1].byte[0] = 0x01;
    Rcon[2].byte[0] = 0x02;
    Rcon[3].byte[0] = 0x04;
    Rcon[4].byte[0] = 0x08;
    Rcon[5].byte[0] = 0x10;
    Rcon[6].byte[0] = 0x20;
    Rcon[7].byte[0] = 0x40;
    Rcon[8].byte[0] = 0x80;
    Rcon[9].byte[0] = 0x1b;
    Rcon[10].byte[0] = 0x36;
}
word RotWord(word w) {//循环移位RotWord
```

```
    word temp;
    for (int i = 0; i < 4; i++) {
        temp.byte[(i + 3) % 4] = w.byte[i];
    }
    return temp;
}
word SubWord(word w) {//S盒变换SubWord
    u_char L, R;
    for (int i = 0; i < 4; i++) {
        L = w.byte[i] >> 4;
        R = w.byte[i] & 0x0f;
        w.byte[i] = SBox[L][R];
    }
    return w;
}
word WordXOR(word w1, word w2) {//异或轮常数WordXOR
    word temp;
    for (int i = 0; i < 4; i++) {
        temp.byte[i] = w1.byte[i] ^ w2.byte[i];
    }
    return temp;
}
```

(三) 结果展示

使用给定的测试样例进行测试，加解密均正确。

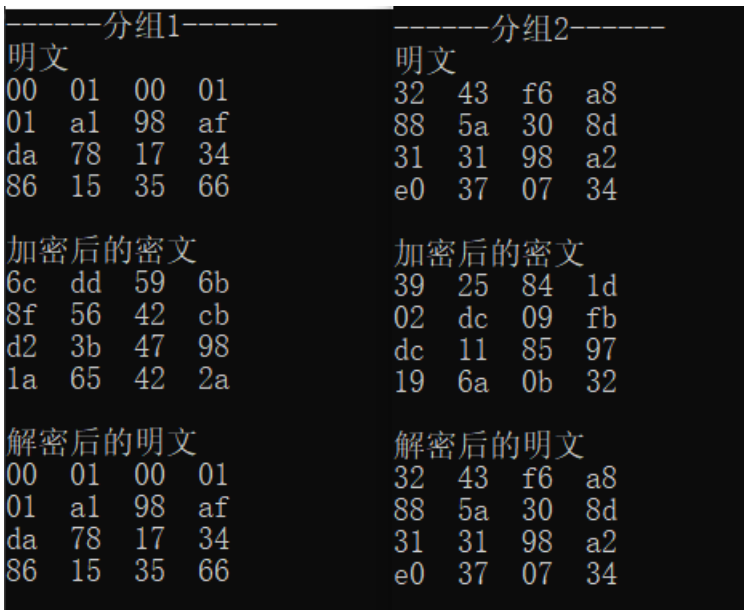

图 8: AES 加解密测试

四、 雪崩效应

(一) 改变明文分组

对每一组测试样例进行测试，分别将明文的第 0 位、第 8 位... 第 120 位取反，进行 16 次测试查看加密后的密文分组与原始密文分组不一致的位数。

```
----------------检测雪崩效应----------------
----------------固定密钥，改变明文----------------
分组1
改变明文的第0位，密文变化66位
改变明文的第8位，密文变化72位
改变明文的第16位，密文变化68位
改变明文的第24位，密文变化55位
改变明文的第32位，密文变化65位
改变明文的第40位，密文变化66位
改变明文的第48位，密文变化60位
改变明文的第56位，密文变化61位
改变明文的第64位，密文变化69位
改变明文的第72位，密文变化63位
改变明文的第80位，密文变化74位
改变明文的第88位，密文变化66位
改变明文的第96位，密文变化70位
改变明文的第104位，密文变化67位
改变明文的第112位，密文变化67位
改变明文的第120位，密文变化58位
平均变化65.4375位
```

图 9: 分组 1 改变明文分组

```
分组2
改变明文的第0位，密文变化59位
改变明文的第8位，密文变化60位
改变明文的第16位，密文变化68位
改变明文的第24位，密文变化74位
改变明文的第32位，密文变化66位
改变明文的第40位，密文变化62位
改变明文的第48位，密文变化69位
改变明文的第56位，密文变化62位
改变明文的第64位，密文变化58位
改变明文的第72位，密文变化63位
改变明文的第80位，密文变化66位
改变明文的第88位，密文变化64位
改变明文的第96位，密文变化60位
改变明文的第104位，密文变化56位
改变明文的第112位，密文变化61位
改变明文的第120位，密文变化72位
平均变化63.75位
```

图 10: 分组 2 改变明文分组

(二) 改变密钥分组

对每一组测试样例进行测试，分别将密钥的第 0 位、第 8 位... 第 120 位取反，进行 16 次测试。查看加密后的密文分组与原始密文分组不一致的位数。

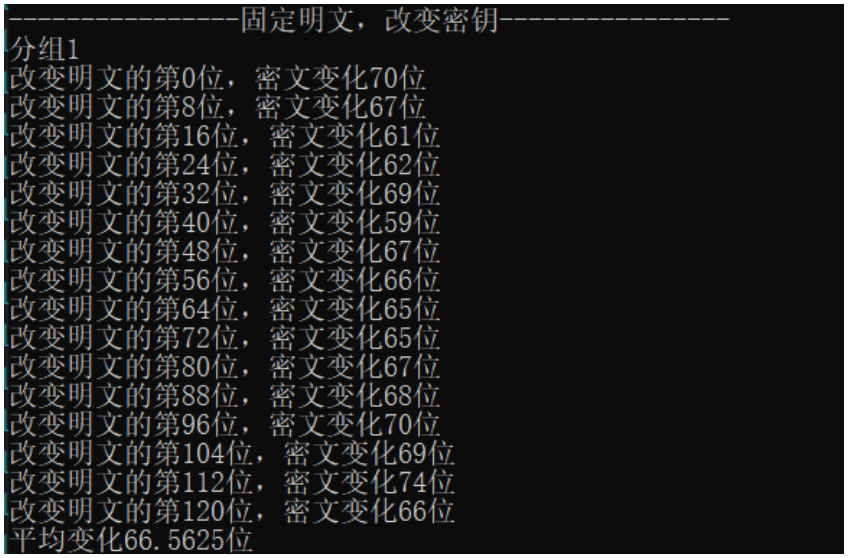

图 11: 分组 1 改变密钥分组

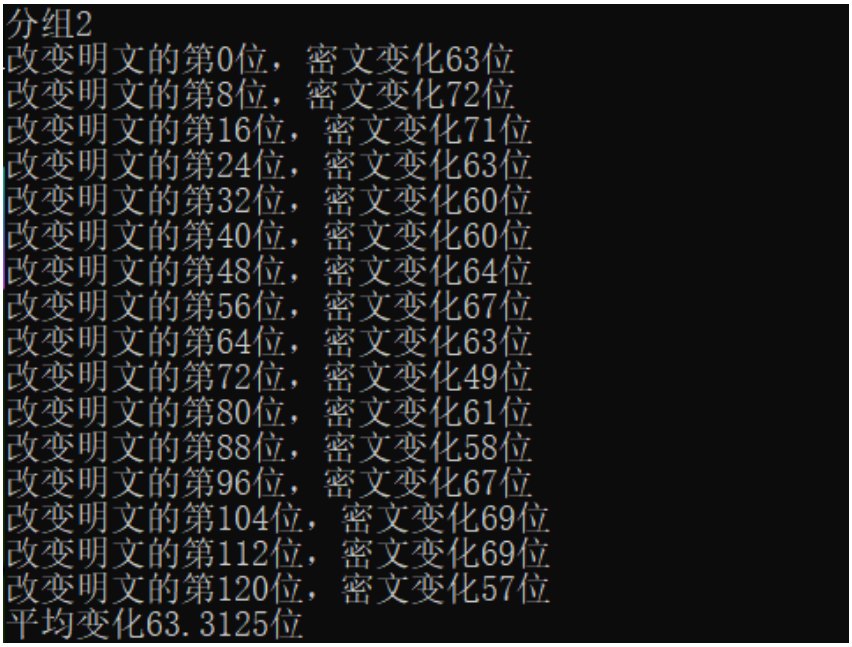

图 12: 分组 2 改变密钥分组

(三) 结果分析

从测试的结果可以看出，我们仅仅是改变了明文或密钥的一位，相应的密文大约有一半的位数发生了变化 (大约 64 位)，有了雪崩效应的存在，密码分析者就无法仅仅从输出推测输入，从而导致该算法部分乃至整个算法被全部破解。