南開大學
Nankai University

网络空间安全学院

密码学课程实验报告

分组密码算法实现：AES

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 12 月 7 日

目录

1 实验要求

1.1 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。

1.2 实验环境

运行 Windows 操作系统的 PC 机，具有 VC 等语言编译环境

1.3 实验内容

1. 算法分析：

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、列混合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.AES 实现程序的总体设计：

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. 编码方式

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. 雪崩效应

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。

2 数学基础

在介绍 AES 算法之前，我们需要知道在伽罗瓦域上的计算的方式

要弄懂 GF(2n̂)，要先明白多项式运算。这里的多项式和初中学的多项式运算有一些区别。虽然它们的表示形式都是这样的:f(x) = x6 + x4 + x2 + x + 1。下面是它的一些特点。

多项式的系数只能是 0 或者 1。当然对于 GF(pn̂)，如果 p 等于 3，那么系数是可以取：0，1，2 的合并同类项时，系数们进行异或操作，不是平常的加法操作。比如 x4 + x4 等于 0*x4。因为两个系数都为 1，进行异或后等于 0 无所谓的减法 (减法就等于加法)，或者负系数。所以，x4 –x4 就等于 x4 + x4。-x3 就是 x3。

根据前面所介绍的，伽罗瓦域上的加法就是简单的异或运算，并不会有什么特殊的。

特殊的是伽罗瓦上的乘法，由于 AES 使用的是 GF（28）所以如果将其看成多项式的话最高此项的次数为 7，但是在乘法中有可能超越这个次数，那么如何解决这个问题呢，在 AES 中，我们只需要计算其乘法结果相对于一个固定的数的模数即可，也就是我们后面会提到的 m(x), 其中 m(x) 的取值为 x8+x4+x3+x+1

下面我会给出伽罗瓦域上的基本运算的 C++ 代码：

GF 上的运算（部分代码）

```
1    string StringModb(string a) {
2    string temp = a;
```

```
    int la = temp.length();
    for (int i = 0; i <= la - 9; i++) {
      if (temp[i] == '1') {
        for (int j = 0; j <= 8; j++) {
          temp[i + j] = CharAddb(temp[i + j], mx[j]);
        }
      }
    }
    string rs = "";
    for (int i = la - 1 - 7; i <= la - 1; i++) {
      string t(1, temp[i]);
      rs.append(t);
    }
    return rs;
  }
  //伽罗瓦域上的乘法运算
  string mulGF(string a, string b) {
    if (a == "00000000" || b == "00000000") {
      return "00000000";
    }
    string temp = a;
    int lb = b.length();
    string front = "";
    string back = "";
    for (int i = lb - 1; i >= 0; i--) {
      if (b[i] == '1') {
        if (front == "") {
          string t = a;
          for (int j = 1; j <= (lb - 1) - i; j++) {
            t.append("0");
          }
          front = t;
          continue;
        }
        else {
          string t = a;
          for (int j = 1; j <= (lb - 1) - i; j++) {
            t.append("0");
          }
          back = t;
          //cout << front << "  " << back << endl;
          front = StringAddb(front, back);
          //cout << front << endl;
          continue;
        }
      }
    }
    //cout << front << endl;
    temp = StringModb(front);
```

```
52      return temp;
53    }
54    string reverseGF(string a) {
55      if (a == "00000000") { return "00000000"; }//0映射到自己
56      string s = "00000000";
57      for (int i = 1; i <= 256; i++) {
58        ;
59        if (mulGF(a, s) == "00000001") {
60          return s;
61        }
62        else {
63          s = StringInc(s);
64        }
65      }
66      return s;
67    }
```

3 AES 算法介绍

3.1 AES 算法基本原理

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。

3.2 字节代换

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(28) 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：

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。

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

图 3.1: 字节代换矩阵运算

从上面的介绍我们不难看出其实对一个固定的字节来说，他在 GF（28）上的乘法逆元是固定的，同理他对于矩阵运算的值也是固定的，所以其实字节替换本质上是一个 S 盒代换。当然为了更好的理解算法的步骤以及结构，在实现的时候，我依然用 C++ 实现了伽罗瓦域上的乘法以及逆元以及相应的矩阵运算。但是 S 盒是一种角度去理解字节替换这个部件。

同理，如果以 S 盒的方式理解字节替换，他也有相应的逆 S 盒，直接查表的方式会提高计算效率。

行/列	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0x63	0x7c	0x77	0x7b	0xf2	0x6b	0x6f	0xc5	0x30	0x01	0x67	0x2b	0xfe	0xd7	0xab	0x76
1	0xca	0x82	0xc9	0x7d	0xfa	0x59	0x47	0xf0	0xad	0xd4	0xa2	0xaf	0x9c	0xa4	0x72	0xc0
2	0xb7	0xfd	0x93	0x26	0x36	0x3f	0xf7	0xcc	0x34	0xa5	0xe5	0xf1	0x71	0xd8	0x31	0x15
3	0x04	0xc7	0x23	0xc3	0x18	0x96	0x05	0x9a	0x07	0x12	0x80	0xe2	0xeb	0x27	0xb2	0x75
4	0x09	0x83	0x2c	0x1a	0x1b	0x6e	0x5a	0xa0	0x52	0x3b	0xd6	0xb3	0x29	0xe3	0x2f	0x84
5	0x53	0xd1	0x00	0xed	0x20	0xfc	0xb1	0x5b	0x6a	0xcb	0xbe	0x39	0x4a	0x4c	0x58	0xcf
6	0xd0	0xef	0xaa	0xfb	0x43	0x4d	0x33	0x85	0x45	0xf9	0x02	0x7f	0x50	0x3c	0x9f	0xa8
7	0x51	0xa3	0x40	0x8f	0x92	0x9d	0x38	0xf5	0xbc	0xb6	0xda	0x21	0x10	0xff	0xf3	0xd2
8	0xcd	0x0c	0x13	0xec	0x5f	0x97	0x44	0x17	0xc4	0xa7	0x7e	0x3d	0x64	0x5d	0x19	0x73
9	0x60	0x81	0x4f	0xdc	0x22	0x2a	0x90	0x88	0x46	0xee	0xb8	0x14	0xde	0x5e	0x0b	0xdb
A	0xe0	0x32	0x3a	0x0a	0x49	0x06	0x24	0x5c	0xc2	0xd3	0xac	0x62	0x91	0x95	0xe4	0x79
B	0xe7	0xc8	0x37	0x6d	0x8d	0xd5	0x4e	0xa9	0x6c	0x56	0xf4	0xea	0x65	0x7a	0xae	0x08
C	0xba	0x78	0x25	0x2e	0x1c	0xa6	0xb4	0xc6	0xe8	0xdd	0x74	0x1f	0x4b	0xbd	0x8b	0x8a
D	0x70	0x3e	0xb5	0x66	0x48	0x03	0xf6	0x0e	0x61	0x35	0x57	0xb9	0x86	0xc1	0x1d	0x9e
E	0xe1	0xf8	0x98	0x11	0x69	0xd9	0x8e	0x94	0x9b	0x1e	0x87	0xe9	0xce	0x55	0x28	0xdf

图 3.2: 字节代换等价 S 盒

3.3 行位移变换

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 C1 个字节，第 2 行循环左移 C2 个字节，第 3 行循环左移 C3 个字节。位移量 C1、C2、C3 的取值与 Nb 有关。

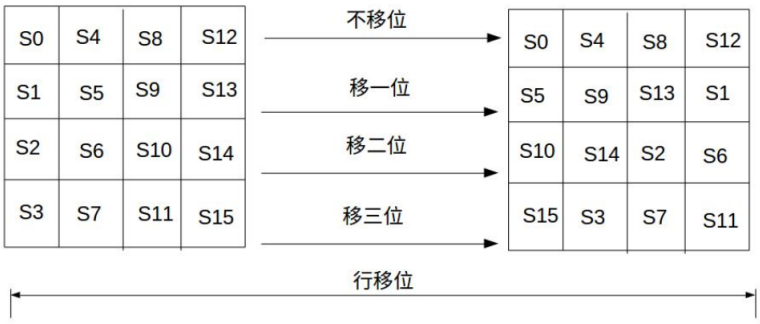

图 3.3: 行位移示意图

当然，具体的行位移的参数也需要通过查表获得，具体的表格如下图所示：

表 3-10 对应于不同分组长度的位移量

N_b	C_1	C_2	C_3
4	1	2	3
6	1	2	3
8	1	3	4

63

图 3.4: 行位移参数表

ShiftRow 的逆变换是对状态阵列的后 3 列分别以位移量 Nb-C1、Nb-C2、Nb-C3 进行循环移位，使得第 i 行第 j 列的字节移位到 (j+Nb-Ci) mod Nb。也就是说可以理解为向右移动相应的位数

3.4 列混合变换

在列混合变换中，将状态阵列的每个列视为系数为 GF(28) 上的多项式，再与一个固定的多项式 c(x) 进行模 x4+1 乘法。当然要求 c(x) 是模 x4+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)= ‘03’ x3+ ‘01’ x2+ ‘01’ x+ ‘02’

c(x) 是与 x4+1 互素的，因此是模 x4+1 可逆的。列混合运算也可写为矩阵乘法。设 b(x)= c(x)a(x)，则我们有运算矩阵

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

图 3.5: 列混合运算矩阵

这个运算需要做 GF(28) 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:

(‘03’ x3+ ‘01’ x2+ ‘01’ x+ ‘02’))d(x)= ‘01’

由此可得

d(x)= ‘0B’ x3+ ‘0D’ x2+ ‘09’ x+ ‘0E’

由此推导扩展到每一行，我们可以得到列混合运算逆运算的计算矩阵:

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

图 3.6: 列混合运算逆运算矩阵

3.5 密钥加

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。

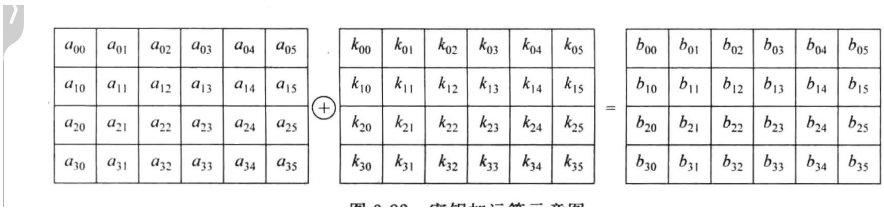

图 3.7: 密钥加

3.6 密钥扩展

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 Nb 个字，第 2 轮轮密钥取接下来的 Nb 个字，如此下去。

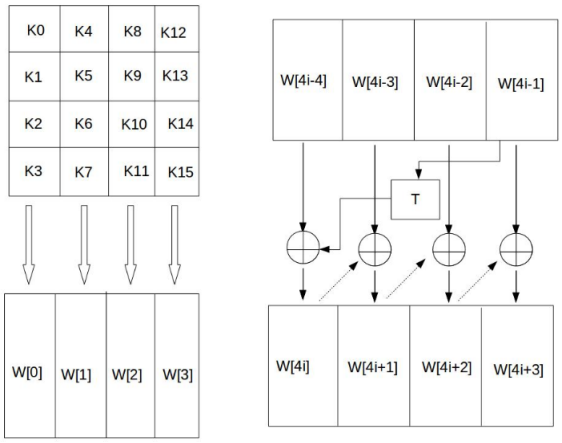

图 3.8: 密钥扩展流程图

具体的计算流程可以查看课本中的伪代码，伪代码清晰的表示了密钥扩展的流程：当密钥的长度为 128 或者 192 比特时，密钥扩展的流程如下图：

```
KeyExpansion(byte  Key[4*Nk], W[Nb*(Nr+1)])
{
    For(i=0; i<Nk; i++)
        W[i]=(Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]);
    For(i=Nk; i<Nb*(Nr+1); i++)
    {
        temp=W[i-1];
        if(i %Nk==0)
            temp=SubByte(RotByte(temp))^Rcon[i/Nk];
        W[i]=W[i-Nk]^temp;
    }
}
```

图 3.9: 密钥扩展伪代码 1

当密钥的长度为 256 比特时，可以参考下面的伪代码：

```
KeyExpansion(byte  Key[4 * Nk] , W[Nb * (Nr+1)])
{
    For(i=0; i<Nk; i++)
        W[i]=(Key[4 * i], Key[4 * i+1], Key[4 * i+2], Key[4 * i+3]);
    For(i=Nk; i<Nb * (Nr+1); i++)
    {
```

66

第 3 章 分组密码体制

```
    temp=W[i-1];
    if(i %Nk==0)
        temp=SubByte(RotByte(temp))^Rcon[i/Nk];
    else if(i %Nk==4)
        temp=SubByte(temp);
    W[i]=W[i-Nk]^temp;
  }
}
```

图 3.10: 密钥扩展伪代码 2

3.6.1 RotByte

在密钥扩展的过程中，RotByte 和行位移变换差不多，RotByte 实现的功能就是将输入的四字节字左移一个字节，也就是说，当输入为（a,b,c,d），则 RotByte 后的取值应为（b,c,d,a）

3.6.2 Rcon

Rcon 运算其实也可以理解为伽罗瓦域上的运算，Rcon(i) 指的是 x 的 i 次方对于 m(x) 的模数当 i<=8 时可以直接在 8 位二进制数中置 1，但是当 i>8 时就需要计算关于 m(x) 的模数

4 整体流程

4.1 加密流程

加密的主要过程包括：对明文状态的一次密钥加，轮轮加密和末尾轮轮加密，最后得到密文。其中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 NixColumn 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 NixColumn 部件。

总结说就是：先做密钥加，然后做常规轮变换，最后做一个末尾轮

4.2 解密流程

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，然后进行轮轮解密和末尾轮轮解密，最后得到明文。和加密不同的是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。

具体的加解密的宏观流程图可以参看下图：

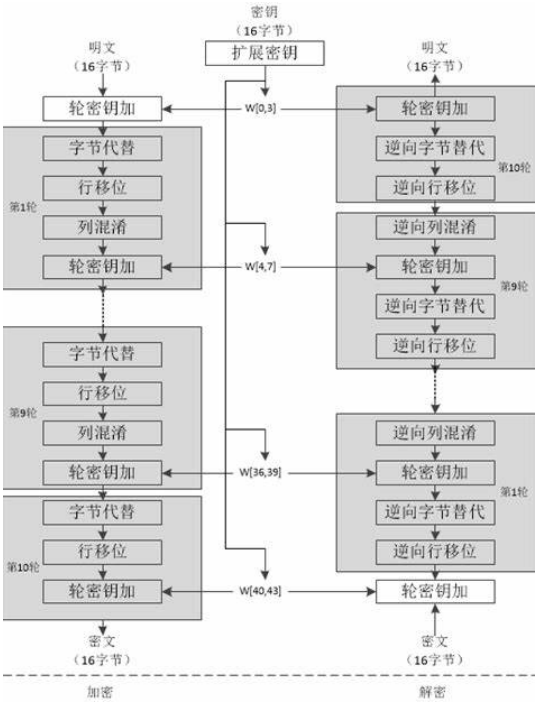

图 4.11: AES 加解密宏观流程图

上面这张图非常直观的阐述了加解密的区别，密钥使用的顺序是逆序的，其次加密和解密的轮函数是不一样的，需要对其中轮函数的具体实现进行微调。

于是到这里，我们可以给出完整的 AES 流程图：

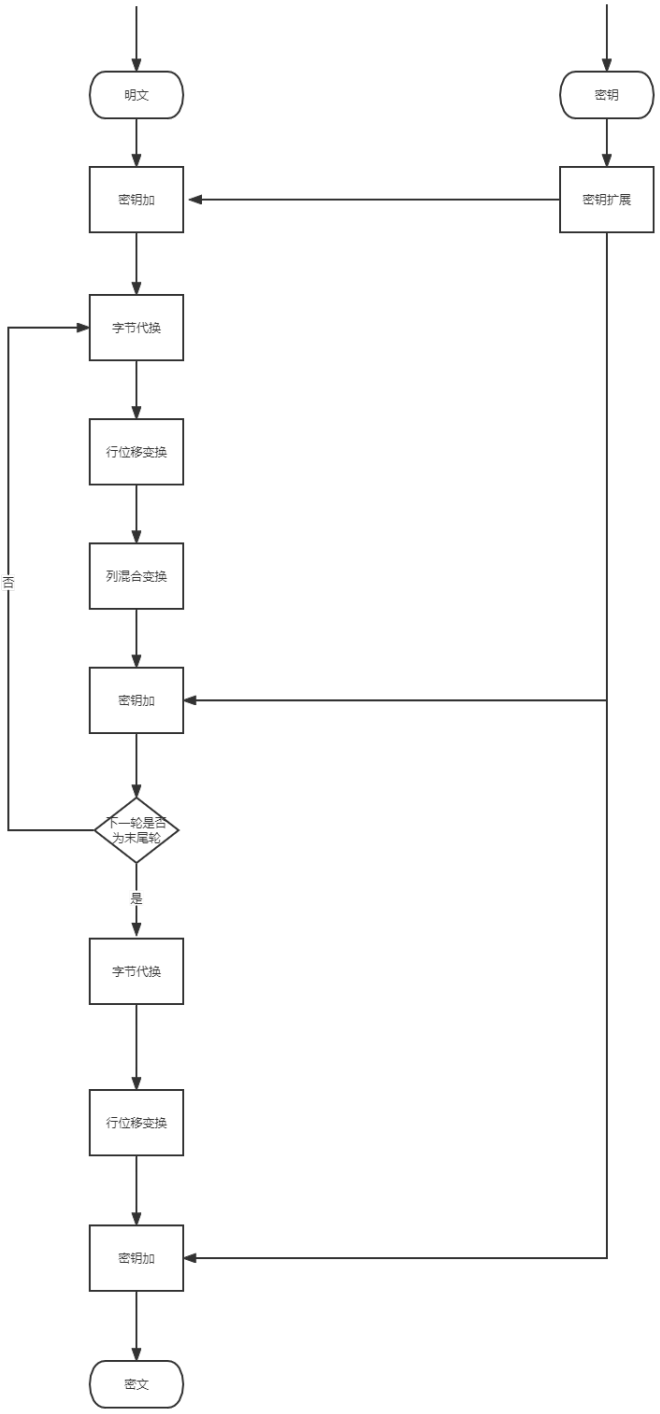

图 4.12: AES 加解密宏观流程图

5 代码实现

在本节中，会给出一些重点实现流程的具体实现代码以展示我的具体实现流程

5.1 字节替换

字节替换算法

```
string ByteSub(string s) {
string t = reverseGF(s);//得到对应的乘法逆元
string r1 = "00000000";
for (int i = 1; i <= 8; i++) {
  //每一个外层循环算出一个比特
  char tc = '0';
  for (int j = 1; j <= 8; j++) {
    char tcc;
    if (matrix1[i][j] == '0' || t[7 - (j - 1)] == '0') { tcc = '0';
        }//这里按照网上的改了一下，我也不知道为什么要这么改
    else { tcc = '1'; }
    if (tc == tcc) { tc = '0'; }
    else { tc = '1'; }
  }
  r1[i - 1] = tc;
}
for (int i = 1; i <= 8; i++) {
  if (r1[i - 1] == matrix2[i]) { r1[i - 1] = '0'; }
  else { r1[i - 1] = '1'; }
}
reverse(r1.begin(), r1.end());
return r1;
}
```

5.2 行移位算法

行移位算法

```
void shiftRow() {
//构建矩阵形式的存储方式
string store[5][9];
for (int i = 1; i <= textSize / 32; i++) {//遍历列
  for (int j = 1; j <= 4; j++) {//遍历行
    string temp = "";
    for (int k = 1; k <= 8; k++) {
      temp.push_back(text[(i - 1) * 32 + (j - 1) * 8 + k]);
    }
    store[j][i] = temp;
    //cout << j << " " << i << " " << temp << endl;
  }
}
//cout << endl;
//对第2-4行进行行移位
for (int i = 2; i <= 4; i++) {
  int cross = shift[i - 1];//位移的位数
  vector<string>vs;//把这一列的所有字符串都先保存
  vs.push_back("");//占住零位
```

```
    for (int k = 1; k <= textSize / 32; k++) {
      vs.push_back(store[i][k]);
    }
    for (int k = 1; k <= textSize / 32; k++) {//移位
      if (k + cross <= textSize / 32) {
        store[i][k] = vs[k + cross];
      }
      else {
        store[i][k] = vs[k + cross - textSize / 32];
      }
    }
  }
  //cout << endl;
  //把store中的元素写回到text中
  for (int i = 1; i <= textSize / 32; i++) {
    for (int j = 1; j <= 4; j++) {
      string s = store[j][i];
      for (int k = 1; k <= 8; k++) {
        text[(i - 1) * 32 + (j - 1) * 8 + k] = s[k - 1];
      }
      //cout << j << " " << i << " " << s << endl;
    }
  }
}
```

5.3 列混合

列混合算法

```
void MixColumn(int col) {
//cout << "col = " << col << endl;
vector<string>vs;//一开始保存了text中的一列
vector<string>vt;
vector<string>rs;
//取出text文件第col列的对应的4个字节
for (int i = 1; i <= 4; i++) {
  string temp;
  for (int j = 1; j <= 8; j++) {
    temp.push_back(text[(col - 1) * 32 + (i - 1) * 8 + j]);
    //int tempi = (col - 1) * 32 + (i - 1) * 8 + j;
    //char c = text[tempi];
    //char cc = text[1];
  }
  vs.push_back(temp);
  //cout << temp <<" ";
}
//cout << endl;
//循环四次算出列混合后的四个字节
```

```
  for (int i = 1; i <= 4; i++) {
    vt.clear();
    for (int j = 1; j <= 4; j++) {
      vt.push_back(mulGF(matrix3[i][j], vs[j - 1]));
    }
    string temp = "00000000";
    for (int k = 1; k <= 4; k++) {
      temp = StringAddb2(temp, vt[k - 1]);//对vs中的内容做累加
    }
    rs.push_back(temp);
  }
  //把结果填回去
  for (int i = 1; i <= 4; i++) {
    for (int j = 1; j <= 8; j++) {
      text[(col - 1) * 32 + (i - 1) * 8 + j] = rs[i - 1][j - 1];
    }
    //cout << rs[i-1] <<" ";
  }
  //cout << endl;
}
```

5.4 密钥扩展

密钥扩展

```
  void KeyExpansion() {
  vks.clear();
  //把元密钥放进去
  for (int i = 0; i < keySize / 32; i++) {
    vector<string>tvs;
    for (int j = 1; j <= 4; j++) {
      string temp = "";
      for (int k = 1; k <= 8; k++) {
        temp.push_back(key[(i) * 32 + (j - 1) * 8 + k]);
      }
      tvs.push_back(temp);
    }
    KeyStore ks(tvs[0], tvs[1], tvs[2], tvs[3]);
    vks.push_back(ks);
  }
  //还要弄这么多列出来
  for (int i = keySize / 32; i <= (textSize / 32) * (Myround + 1); i++) {
    KeyStore temp = vks[i - 1];
    if ((i % (keySize / 32)) == 0) {
      RotByte(temp);
      temp.l1 = ByteSub(temp.l1);
      temp.l2 = ByteSub(temp.l2);
      temp.l3 = ByteSub(temp.l3);
```

```
        temp.l4 = ByteSub(temp.l4);
        string  s = GenerateRC(i / (keySize / 32));
        temp.l1 = StringAddb2(temp.l1, s);
        //cout << temp.l1 << " " << temp.l2 << " " << temp.l3 << " " << temp.l4 << endl;
    }//更新temp
    KeyStore t = vks[i - keySize / 32];//取出前面的一个周期的对应位置的来
    //cout << "temp:" << temp.l1 << " " << temp.l2 << " " << temp.l3 << " " <<
        temp.l4 << endl;
    //cout << "t:" << t.l1 << " " << t.l2 << " " << t.l3 << " " << t.l4 << endl;
    string s1 = StringAddb2(temp.l1, t.l1);
    string s2 = StringAddb2(temp.l2, t.l2);
    string s3 = StringAddb2(temp.l3, t.l3);
    string s4 = StringAddb2(temp.l4, t.l4);
    KeyStore rs(s1, s2, s3, s4);
    //cout << s1 << " " << s2 << " " << s3 <<" "<< s4 << endl;
    vks.push_back(rs);
  }
}
```

6 雪崩效应实验

雪崩效应就是一种不稳定的平衡状态也是加密算法的一种特征，它指明文或密钥的少量变化会引起密文的很大变化，就像雪崩前，山上看上去很平静，但是只要有一点问题，就会造成一片大崩溃。可以用在很多场合对于 Hash 码，雪崩效应是指少量消息位的变化会引起信息摘要的许多位变化。

在密码学中，雪崩效应（Avalanche effect）指加密算法（尤其是块密码和加密散列函数）的一种理想属性。雪崩效应是指当输入发生最微小的改变（例如，反转一个二进制位）时，也会导致输出的剧变（如，输出中一半的二进制位发生反转）。在高品质的块密码中，无论密钥或明文的任何细微变化都应当引起密文的剧烈改变。该术语最早由 Horst Feistel 使用，尽管其概念最早可以追溯到克劳德 · 香农提出的。

在本次实验中，我运用老师提供的测试样例 1 进行雪崩效应实验，通过逐比特改变明文（128 位）以及密钥（128）位观察密文的变化。

雪崩效应的检测代码如下给出：

雪崩效应

```
void Avalanche() {
  cout << "下面开始雪崩效应测试，具体使用的数据为：" << endl;
  cout << "明文为：" << td1.text << endl;
  cout << "密钥为：" << td1.key << endl;
  cout << "首先逐个改变明文的128个比特位并查看密文的改变情况" << endl;
  encrypt(td1);
  //把td1的加密结果保存在text0中
  char* text0 = new char[129];
  memcpy(text0, text, 129);
  int count=0;
  for (int i = 1; i <= 128; i++) {
    //直接把encrypt的函数内容copy过来
```

```
13      gettext(td1);//初始化明文
14      if (text[i] == '0') { text[i] = '1'; }
15      else { text[i] = '0'; }
16      getkey(td1);//初始化密文
17      SetValue();//初始化轮数等信息
18      KeyExpansion();//完成轮密钥的初始化
19      //test1();
20      //printCStar(key, 1, 129);
21      string s = generateUsedKey(0);
22      //cout << s << endl;
23      //printCStar(text, 1, 129);
24      AddRoundKey(s);
25      //cout << "after add" << endl;
26      //printCStar(text, 1, 129);
27      for (int i = 1; i <= Myround; i++) {
28        string s = generateUsedKey(i * textSize / 32);
29        //cout << s << endl;
30        if (i == Myround) { FinalRound(s); break; }
31        Round(s);
32      }
33      cout << "改变第" << i << "位的结果为" << endl;
34      CStar2HEX(text, 1, 129);
35      cout << "改变了" << findDifference(text0, text, 1, 129) << "位" << endl;;
36      count += findDifference(text0, text, 1, 129);
37    }
38    double d = (double)count / 128.0;
39    cout << "在128次的明文逐比特改变中, 平均每次密文改变的比特数为" << d << endl;
40  }
```

6.1 雪崩效应结果

:

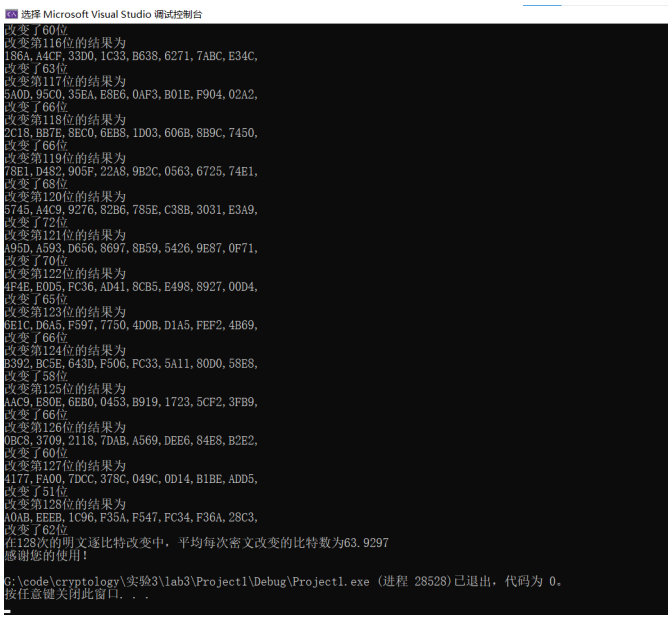

图 6.13: 改变明文的雪崩效应

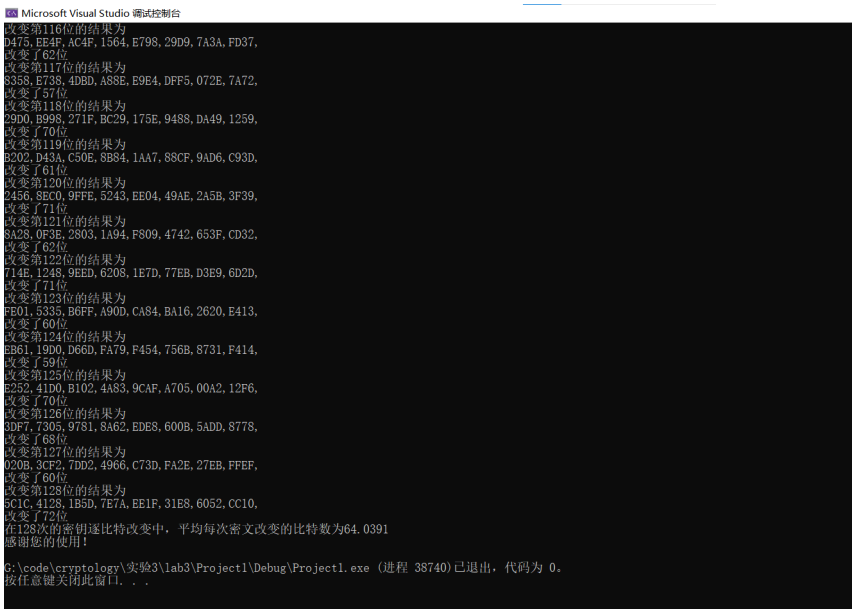

图 6.14: 改变密文的雪崩效应

可以看到，在最终的统计结果中，128 比特的密文最终的改变位数在 64 位上下，符合雪崩效应，证明了 AES 算法的高度安全性。

7 总结

通过本次实验，我动手实现了一个 AES 加密算法。在这个过程中，我不仅学习了如何实现一个伽罗瓦域上的 C++ 计算代码，同样对分组密码有了更加深刻的理解。