AES算法详解

宋林轲 2018K8009970020

2021 年 1 月 27 日

目录

第一部分　数学基础

1　有限域

在了解AES算法之前，需对其数学基础有所涉猎。

AES算法建立在有限域$GF(2^8)$之上，其运算满足域上的规则。在AES算法中主要使用的是加法运算和乘法运算，以及将二者结合后的矩阵运算。下面将在小节中分别介绍。

1.1　加法乘法运算

有限域$GF(2^8)$上的加法运算遵循模2加原则，即通过二进制表示时的逐位异或的方式计算结果。例如：$01010111 \oplus 10000011 = 11010100$。采用十六进制表示，即为：$0x57 \oplus 0x83 = 0xD4$。

有限域$GF(2^8)$上的乘法运算则相对麻烦，以下面两个数为例：$A = (a_0a_1 \cdots a_7)_2$和$B = (b_0b_1 \cdots b_7)_2$。同时，定义一个存放长为8-bits的结果向量$C$，初始值为0。若以B为被乘数，从A的高位向低位遍历，若高位为1，则对C加上一个B,与此同时将C左移一位，A则向低位遍历一位，特别注意，由于是有限域$GF(2^8)$上的模乘，在左移C的过程中若出现高位溢出为1的情况，则需要减去8次不可约多项式对应的数$0x11B$，才能继续进行移位加法操作。最终多次加法移位后的结果C即为模乘后的结果。

语言表述可能存在不精确的地方，具体细节请参阅代码部分。

1.2　矩阵运算

有限域$GF(2^8)$上的矩阵运算本质上是有限域上模乘和模2加的集合。以下面的4×4矩阵为例：

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 & c_7 \\ c_8 & c_9 & c_{10} & c_{11} \\ c_{12} & c_{13} & c_{14} & c_{15} \end{bmatrix} \tag{1}$$

$$c_{ij} = \sum_{k}^{4} a_{ik} \cdot b_{kj} \tag{2}$$

注：本公式均为有限域$GF(2^8)$之上的乘法和加法。

2 代码实现

下面分别给出了实现数学基础部分所用的函数，注释中可以提取出笔者的实现思路。

```
/* Author: Song link            *
 * ID: 2018K8009970020          *
 * MTA: module two add.         *
 * quite easy to operate, so    *
 * we use it as a preprocessor  *
 * definition.                  */

#define MTA(a, b)  (a ^ b)

/* GF(2^8) multiply:                          *
 * Use this as the multiplication method      *
 * As mentioned above, to get the result      *
 * ,we shift one of the number step by step   *
 * from the highest bit to the lowest bit.    *
 * When the result happens to be larger than  *
 * 0xff, add 0x11b to it.                     *
 * To simplify this process, we transfer it   *
 * into uint16_t beforehand.                  */

static uint8_t GFM(uint8_t a, uint8_t b)
{
        uint16_t ar, br, rt;
        ar = (uint16_t)a;
        br = (uint16_t)b;
        rt = 0;
```

```
26          while(ar)
27          {
28                  if(ar & 1)
29                  {
30                          rt = rt ^ br;
31                  }
32                  br <<= 1;
33                  if(br >= 0x100)
34                  {
35                          br = br ^ 0x11b;
36                  }
37                  ar >>= 1;
38          }
39          return (uint8_t) rt;
40  }
41
42  /* Multiplication for Matrix:              *
43   * As mentioned above, we put two arrays  *
44   * to compute the vector used in Matrix   *
45   * Multiplication.                        */
46
47  static uint8_t Matx_Multi(uint8_t c[], uint8_t a[])
48  {
49          int i;
50          uint8_t ret = 0x00;
51          for(i = 0; i < 4; i++)
52          {
53                  ret = MTA(GFM(c[i], a[i]), ret);
54          }
55          return ret;
56  }
```

第二部分　算法内容

AES算法涉及多个步骤，总体而言分为完整的轮秘钥操作和最后一轮秘钥操作。根据不同大小的秘钥，采用的轮数和其它部分参数会有细微不同，本文为了方便起见，采用秘钥长度和明文长度均为16字节的情况。

3　步骤分解

以初始秘钥长度和明文长度块均为16字节为例，在此情况中AES的加密包含10次的完整加密轮和1次最后加密轮。完整的加密轮包含下面几个步骤：字节代换，行移位，列混合，秘钥加。最后加密轮则包含下面几个步骤：字节代换，行移位，秘钥加。然而，秘钥的长度本身不能支持如此多轮的秘钥加操作，因此，我们需要对输入的秘钥进行扩展，再去实现轮函数加密。

3.1　扩展

严格意义上的扩展包含两个部分。AES的加密会对明文进行分割，如果明文组出现不足16字节的情况，将会自动扩展至16字节以便之后的加密，这里便不再赘述。另一层意义上则是利用输入的16字节的秘钥扩展至多轮不相同的秘钥，以便每次进行轮函数加密使用。

下面将详细说明16字节轮秘钥的扩展：

首先，扩展秘钥需要从原秘钥中读取种子。利用二维数组$expkey[N_k][MaxLen]$来存放相关信息。二维数组的每一行均存放秘钥的其中四个字节，记作$W[i]$行，扩展则是以每一行四个字节作为基本单位进行的。

在将种子秘钥的16字节填入后，之后的扩展遵循下列法则：若行的序号不是4的倍数，如$W[i]$中$i\%N_k! = 0$的情况,则$W[i]$的取值为$W[i] = W[i - N_k] \oplus W[i - 1]$。$N_k$这里取4,表示上一个秘钥的同一行。若行的序号是4的倍数，则需要通过某个函数生成秘钥。此情况相对复杂，请参考本处所做的注释和相关的文献资料。

利用上述方式扩展至$N_b \times (N_r + 1)$行时，全部的秘钥扩展即可结束。

以下提供该环节的代码。

```
1  /* KeyExpand: Expand the key for the upcoming steps   *
2   * We first use a 2-dimension array to store all the  *
3   * information of the expand-key.                     *
4   * When i % 4 == 0, it is a little troublesome        *
5   * We get a Rcon[] array to get the key-constant here *
6   * By rotbyte and array_MTA and finally ByteSub(S_box)*
7   * We can get the result. The output is stored in exp_*
8   * key.  BTW.                                         *
```

```
 * For more information about the Rcon[], please refer*
 * to aes.h and the formal pdf .                       */

static void KeyExpand(uint8_t key[], uint8_t exp_key[][MaxLen])
{
        uint8_t i, j;
        uint8_t temp[MaxLen] = {0};
        for(i = 0; i < Nk; i++)
                for(j = 0; j < 4; j++)
                        exp_key[i][j] = key[4 * i + j];

        for(i = Nk; i < Nb * (Nr + 1); i++)
        {
                memcpy(temp, exp_key[i - 1], Width);
                if(i % Nk == 0)
                {
                        uint8_t Rcon[] = {RC_Store[i / Nk], 0x00, 0x00, 0x00};
                        RotByte(temp);
                        Array_MTA(temp, Rcon, temp, Width);
                        ByteSub(temp, Width);
                }
                Array_MTA(exp_key[i - Nk], temp, exp_key[i], Width);
                memset(temp, 0, sizeof(temp));
        }
}
```

3.2 字节代换

字节代换是加密的第一步，即通过某个$Sbox$沙盒实现对信息的置换，一般会通过构造非线性映射的方式来实现乱序，保证安全。由于置换表过大，此处文档不会给出置换表。

总而言之，对于输入的信息数组$in[]$的其中一个元素i，该步骤将会通过$Sbox$在同一个位置利用$Sbox[in[i]]$来替换原本的元素。

对应的，解密时将会使用逆数组$invSbox[]$来实现快速还原解密。

```
/* ByteSub: change the plaintxt                    *
 * very easy. BTW, s_box is put                    *
 * in the aes.h,U can refer to it                  *
 * Rv_ByteSub is another s_box contrary to s_box   *
 * We won't shown here.                            */

static void ByteSub(uint8_t in[], uint8_t len)
{
        int i;
        for(i = 0; i < len; i++)
        {
                in[i] = s_box[in[i]];
        }
}
```

3.3 行移位

对于16字节的文本和秘钥，行移位遵循下面的原则。
利用循环的方式对后三行分别左移1位，2位，3位，即可完成该步骤。

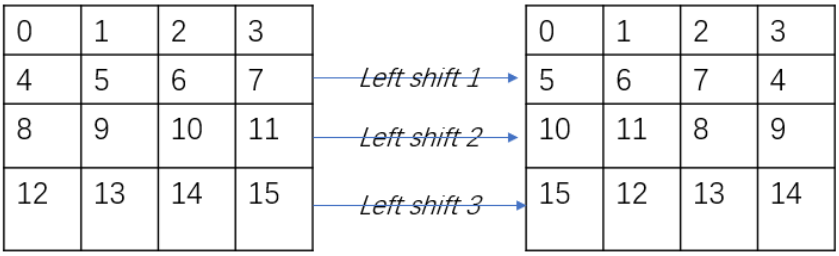

图 1: 行移位操作

```
/* ShiftRow: shift the row of the matrix *
 * count the row number, and shift it to *
 * the left due to the number we get     */

static void ShiftRow(uint8_t in[])
{
        int i, k, j;
        uint8_t tmp;
        for(i = 1; i < 4; i++)
        {
                k = 0;
                while(k < i)
                {
                        tmp = in[Nb * i];
                        for(j = 1; j < Nb; j++)
                        {
                                in[Nb * i + j - 1] = in[Nb * i + j];
                        }
                        in[Nb * i + Nb - 1] = tmp;
                        k++;
                }
        }
}
```

3.4 列混合

列混合操作涉及有限域$GF(2^8)$的加法和乘法运算。利用下面的行向量$\vec{x} = (0x02, 0x03, 0x01, 0x01)$的

移位，得到下面的M矩阵。并用先前的文本与之相乘进行加密。

$$M \cdot A = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix} \tag{3}$$

对应的，解密使用的行向量$\vec{y} = (0x0E, 0x0B, 0x0D, 0x09)$同样通过拼接和移位得到下面的N矩阵。不难验证在有限域$GF(2^8)$下这两个矩阵是互逆的，因此可以通过M矩阵和N矩阵的相乘还原原文。

$$\begin{aligned} N \cdot M \cdot A &= \begin{bmatrix} 0x0E & 0x0B & 0x0D & 0x09 \\ 0x09 & 0x0E & 0x0B & 0x0D \\ 0x0D & 0x09 & 0x0E & 0x0B \\ 0x0B & 0x0D & 0x09 & 0x0E \end{bmatrix} \cdot \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} \\ &= I \cdot A = A \end{aligned} \tag{4}$$

此部分代码相对复杂，请看注释部分。

```
1  /* coef_mult: multiplication for the matrix     *
2   * TO SIMPLIFY IT, I DIDN'T USE polynomials     *
3   * I Just count them all out.                   *
4   * please refer to the equation I given above  */
5
6  static void coef_mult(uint8_t *a, uint8_t *b, uint8_t *d)
7  {
8          d[0] = GFM(a[0], b[0]) ^ GFM(a[1], b[1]) ^ GFM(a[2], b[2]) ^ GFM(a[3], b[3]);
9          d[1] = GFM(a[3], b[0]) ^ GFM(a[0], b[1]) ^ GFM(a[1], b[2]) ^ GFM(a[2], b[3]);
10         d[2] = GFM(a[2], b[0]) ^ GFM(a[3], b[1]) ^ GFM(a[0], b[2]) ^ GFM(a[1], b[3]);
11         d[3] = GFM(a[1], b[0]) ^ GFM(a[2], b[1]) ^ GFM(a[3], b[2]) ^ GFM(a[0], b[3]);
12 }
13 /* MixColumn: Mix the column     *
14  * Mind the column and the row   *
15  * the main difference is that   *
16  * we compute column vectors for*
17  * A matrix, for N, M matrix     *
18  * we compute the row vectors,   *
19  * which is in accordance with   *
20  * Matrix multiplication...      */
21
22 static void MixColumn(uint8_t in[])
23 {
24         uint8_t a[] = {0x02, 0x03, 0x01, 0x01};
25         uint8_t i, j, col[4], res[4];
26
27         for (j = 0; j < Nb; j++) {
28                 for (i = 0; i < 4; i++) {
29                         col[i] = in[Nb * i + j];
30                 }
31
```

```
32                      coef_mult(a, col, res);
33
34                      for (i = 0; i < 4; i++) {
35                              in[Nb * i + j] = res[i];
36                      }
37              }
38      }
```

3.5 秘钥加

秘钥加环节非常简单，通过秘钥扩展后得到的秘钥与需加密的信息进行一次异或，即可实现。此处不再赘述。唯一需要注意的是秘钥的挑选。这里直接看代码部分。

```
1   /* AddRoundKey: Simply add the key here.  *
2    * Very common                            */
3
4   static void AddRoundKey(uint8_t in[], uint8_t key[], uint8_t len)
5   {
6           int i;
7           for(i = 0; i < len; i++)
8           {
9                   in[i] = MTA(in[i], key[i]);
10          }
11  }
```

4 步骤组合

以上梳理了AES加密的全部单位操作过程，下面将其组合以供使用。

4.1 加密周期

完整的加密周期包含字节代换，行移位，列混合以及秘钥加。而最后一轮则是不需要采用列混合，因此代码撰写如下。

```
1   /* OneFullRound:    *
2    * FinalRound:      *
3    * QUITE EASY       */
4
5   static void OneFullRound(uint8_t in[], uint8_t key[])
6   {
7           ByteSub(in, Nb * Width);
8           ShiftRow(in);
9           MixColumn(in);
10          AddRoundKey(in, key, Nb * Width);
11  }
12
13  static void FinalRound(uint8_t in[], uint8_t key[])
14  {
15          ByteSub(in, Nb * Width);
16          ShiftRow(in);
17          AddRoundKey(in, key, Nb * Width);
```

```
18  }
```

4.2 解密周期

完整的解密周期则是完整加密周期的逆向处理，先进行秘钥加，再通过列混合，行移位，以及字节代换。因此，代码撰写如下。

```
1   /* Rv_FinalRound:              *
2    * Rv_OneFullRound:           *
3    * Quite easy: but mind that *
4    * Rv_FinalRound is the first*
5    * Step.                     */
6
7   static void Rv_FinalRound(uint8_t in[], uint8_t key[])
8   {
9           AddRoundKey(in, key, Nb * Width);
10          Rv_ShiftRow(in);
11          Rv_ByteSub(in, Nb * Width);
12  }
13
14  static void Rv_OneFullRound(uint8_t in[], uint8_t key[])
15  {
16          AddRoundKey(in, key, Nb * Width);
17          Rv_MixColumn(in);
18          Rv_ShiftRow(in);
19          Rv_ByteSub(in, Nb * Width);
20  }
```

5 加解密代码实现

现在做好了准备，可以撰写加密和解密函数了，同时，我们可以提供16字节长度的测试用例。该程序扩展并不难，但基础功能的实现只需看16字节的情况即可验证。

下面是主体函数和调用的关键函数：

```
1
2   /* main function:                                          *
3    *      Init: alloc space and set all to 0.                    *
4    *  Read_plain: read plaintext from a file on a given path *
5    *  Read_key: read the 16 bytes key.                       *
6    *  KeyExpand: Expand the key.                             *
7    *  Rijndael: finish the encode work. Shown below.         *
8    *  Rv_Rijndael: finish the decode work. Shown below.      *
9    *  Showchar: show the output in "char" forms.             */
10
11  int main(void)
12  {
13          Init();
14          Read_plain(in);
15          Read_key(key);
16          KeyExpand(key, exp_key);
17          printf("Encode_result:\n");
18          Rijndael(in);
```

```
19          PriArray(in, Width * Nb);
20          Rv_Rijndael(in);
21          printf("Decode result:\n");
22          PriArray(in, Width * Nb);
23          Showchar(in, Width * Nb);
24  }
25
26  /* Rijndael: ENCODE                          *
27   *   Select the key for encoding.            *
28   *   AddRoundKey: first step.                *
29   *   10 times OneFullRound.                  *
30   *   1 time  FinalRound                      */
31
32  static void Rijndael(uint8_t in[])
33  {
34          int rd;
35          Key_Select(0, sel_key, exp_key);
36          AddRoundKey(in, sel_key, Nk * Width);
37          memset(sel_key, 0, sizeof(sel_key));
38          for(rd = 1; rd < Nr; rd++)
39          {
40                  Key_Select(rd, sel_key, exp_key);
41                  OneFullRound(in, sel_key);
42                  memset(sel_key, 0, sizeof(sel_key));
43          }
44          Key_Select(Nr, sel_key, exp_key);
45          FinalRound(in, sel_key);
46          return ;
47  }
48
49  /* Rv_Rijndael:                              *
50   *    Mirror image of the Rijndael           */
51  static void Rv_Rijndael(uint8_t in[])
52  {
53          int rd;
54          Key_Select(Nr, sel_key, exp_key);
55          Rv_FinalRound(in, sel_key);
56          for(rd = Nr - 1; rd > 0; rd--)
57          {
58                  Key_Select(rd, sel_key, exp_key);
59                  Rv_OneFullRound(in, sel_key);
60          }
61          Key_Select(rd, sel_key, exp_key);
62          AddRoundKey(in, sel_key, Nk * Width);
63  }
```

下面展示实验结果。

```
linka@ubuntu:~/HW/CA$ ./aes
buf:cdefasefsabcdefs
in:cdefasefsabcdefs
buf = 1234567890abcdef
key = 1234567890abcdef
Encode result:
0x73,0x22,0x14,0x69,0x38,0xda,0x6c,0xe5,0x c,0xe5,0x43,0x48,0x20,0xac,0xf9,0xcb,
Decode result:
0x63,0x64,0x65,0x66,0x61,0x73,0x65,0x66,0x73,0x61,0x62,0x63,0x64,0x65,0x66,0x73,
The string form:
c,d,e,f,a,s,e,f,s,a,b,c,d,e,f,s,
```

图 2: 16比特加密结果

第三部分　优化与性能比较

仔细查阅上述加密过程，寻找可以被优化的部分。可见最耗时的步骤是列混合阶段。若能够加快这一过程，将会大大提高程序运行的速度。之后，将利用计时器和相关的评测器来检测性能地提高水准。

6　优化

针对加密的优化，可以尝试将所有的步骤组合在一起，利用一条式子进行全部的运算。同时，为了避免系统进行乘法这一开销较大的运算，可以通过事先构造表格，通过输入进行查阅的方式，撤除乘法运算，仅通过加法和查阅，将矩阵运算的速度提高。

记录M和N矩阵的生成的向量如下： $\vec{x} = (0x02, 0x03, 0x01, 0x01)$以及

$\vec{y} = (0x0E, 0x0B, 0x0D, 0x09)$。

通过构造数据表，利用小程序打印所有的可能情况，并与输入一一对应，可以得到类似如下的表格。

```
static uint8_t T03_Quick[] = {
    0x00, 0x03, 0x06, 0x05, 0x0c, 0x0f, 0x0a, 0x09, 0x18, 0x1b, 0x1e, 0x1d, 0x14, 0x17, 0x12, 0x11,
    0x30, 0x33, 0x36, 0x35, 0x3c, 0x3f, 0x3a, 0x39, 0x28, 0x2b, 0x2e, 0x2d, 0x24, 0x27, 0x22, 0x21,
    0x60, 0x63, 0x66, 0x65, 0x6c, 0x6f, 0x6a, 0x69, 0x78, 0x7b, 0x7e, 0x7d, 0x74, 0x77, 0x72, 0x71,
    0x50, 0x53, 0x56, 0x55, 0x5c, 0x5f, 0x5a, 0x59, 0x48, 0x4b, 0x4e, 0x4d, 0x44, 0x47, 0x42, 0x41,
    0xc0, 0xc3, 0xc6, 0xc5, 0xcc, 0xcf, 0xca, 0xc9, 0xd8, 0xdb, 0xde, 0xdd, 0xd4, 0xd7, 0xd2, 0xd1,
    0xf0, 0xf3, 0xf6, 0xf5, 0xfc, 0xff, 0xfa, 0xf9, 0xe8, 0xeb, 0xee, 0xed, 0xe4, 0xe7, 0xe2, 0xe1,
    0xa0, 0xa3, 0xa6, 0xa5, 0xac, 0xaf, 0xaa, 0xa9, 0xb8, 0xbb, 0xbe, 0xbd, 0xb4, 0xb7, 0xb2, 0xb1,
    0x90, 0x93, 0x96, 0x95, 0x9c, 0x9f, 0x9a, 0x99, 0x88, 0x8b, 0x8e, 0x8d, 0x84, 0x87, 0x82, 0x81,
    0x9b, 0x98, 0x9d, 0x9e, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85, 0x86, 0x8f, 0x8c, 0x89, 0x8a,
    0xab, 0xa8, 0xad, 0xae, 0xa7, 0xa4, 0xa1, 0xa2, 0xb3, 0xb0, 0xb5, 0xb6, 0xbf, 0xbc, 0xb9, 0xba,
    0xfb, 0xf8, 0xfd, 0xfe, 0xf7, 0xf4, 0xf1, 0xf2, 0xe3, 0xe0, 0xe5, 0xe6, 0xef, 0xec, 0xe9, 0xea,
    0xcb, 0xc8, 0xcd, 0xce, 0xc7, 0xc4, 0xc1, 0xc2, 0xd3, 0xd0, 0xd5, 0xd6, 0xdf, 0xdc, 0xd9, 0xda,
    0x5b, 0x58, 0x5d, 0x5e, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45, 0x46, 0x4f, 0x4c, 0x49, 0x4a,
    0x6b, 0x68, 0x6d, 0x6e, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75, 0x76, 0x7f, 0x7c, 0x79, 0x7a,
    0x3b, 0x38, 0x3d, 0x3e, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25, 0x26, 0x2f, 0x2c, 0x29, 0x2a,
    0x0b, 0x08, 0x0d, 0x0e, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15, 0x16, 0x1f, 0x1c, 0x19, 0x1a
};
```

图 3: 存储的运算结果数组

声明如下：本次实验生成的表格有$T02Quick[]$,$T03Quick[]$等存储运算结果的数组，表示与数据$0x02$和$0x03$及其他数据相乘后得到的结果。通过此方法可以加快矩阵元素计算的速度。

与此同时，记录移位和进行$SubByte$查阅操作对原输入的影响，记录各个字节的位置，利用枚举法将16个字节的变换关系给出。示例如图4所示。

7　性能测试比较

由于并不涉及较复杂的乘法运算，计算秘钥拓展所耗费的时间则忽略不计，利用计时器

```
static void OneFullRound(uint8_t in[], uint8_t key[])
{
    int i;

    for(i = 0; i < Nb * Width; i++)
    {
        temp[i] = in[i];
    }
/* Since the whole input length is below or equal to Nb * Width             *
 * That's trully a very small number, so we simply count all the output here *
 * from in[0] to in[15] , enjoy yourself                                     */

    in[0]  = T02_Quick[s_box[temp[0]]]   ^ T03_Quick[s_box[temp[5]]] ^
             T01_Quick[s_box[temp[10]]]  ^ T01_Quick[s_box[temp[15]]];
    in[1]  = T02_Quick[s_box[temp[1]]]   ^ T03_Quick[s_box[temp[6]]] ^
             T01_Quick[s_box[temp[11]]]  ^ T01_Quick[s_box[temp[12]]];
    in[2]  = T02_Quick[s_box[temp[2]]]   ^ T03_Quick[s_box[temp[7]]] ^
             T01_Quick[s_box[temp[8]]]   ^ T01_Quick[s_box[temp[13]]];
    in[3]  = T02_Quick[s_box[temp[3]]]   ^ T03_Quick[s_box[temp[4]]] ^
             T01_Quick[s_box[temp[9]]]   ^ T01_Quick[s_box[temp[14]]];
```

图 4: 枚举法示例图

机制来计算运行的总时间。在加密前开始计时，在解密后结束计时，打印相关的信息。

部分代码如下：

```
1  int main(void)
2  {
3          Init();
4          Read_plain(in);
5          Read_key(key);
6          KeyExpand(key, exp_key);
7          clock_t start, end;
8          printf("Encode_result:\n");
9          start = clock();
10         Rijndael(in);
11         //PriArray(in, Width * Nb);
12         Rv_Rijndael(in);
13         end = clock();
14         printf("clock_tick:_%ld\n", end -start);
15         printf("Decode_result:\n");
16         PriArray(in, Width * Nb);
17         Showchar(in, Width * Nb);
18 }
```

利用此代码对16字节的加解密速度进行比较，得到结果如下图5和图6所示，分别是正常运算流程和压缩查表运算流程：

事先已经验证二者在相同输入的情况下结果是一致的，从得到的clock　tick数值来看，二者运行的结果相同，即便对应的只有16字节数据的加解密流程，也能看到后者通过查表的方式其运算速率有明显的提高。在第二个程序中，笔者为了一致性和强迫症，将本来不需要查表的$0x01$情况也通过查表进行处理了。在这么做的情况下依旧能够看到速度有巨大的提高。

本实验的普遍性情况其实并不需要证明，利用类似的方式对长字节进行加密和解密，依据算法判断，其运行时间是16字节作为单元的简单相加，因此扩展该程序为输入任意长度字

```
linka@ubuntu:~/HW/CA$ ./aes
buf:cdefasefsabcdefs
in:cdefasefsabcdefs
buf = 1234567890abcdef
key = 1234567890abcdef
Encode result:
clock tick: 33
Decode result:
0x63,0x64,0x65,0x66,0x61,0x73,0x65,0x66,0x73,0x61,0x62,0x63,0x64,0x65,0x66,0x73,
The string form:
c,d,e,f,a,s,e,f,s,a,b,c,d,e,f,s,
```

图 5: 正常运算流程

```
linka@ubuntu:~/HW/CAadv$ ./aes
buf:cdefasefsabcdefs
in:cdefasefsabcdefs
buf = 1234567890abcdef
key = 1234567890abcdef
Encode result:
clock tick: 4
Decode result:
0x63,0x64,0x65,0x66,0x61,0x73,0x65,0x66,0x73,0x61,0x62,0x63,0x64,0x65,0x66,0x73,
The string form:
c,d,e,f,a,s,e,f,s,a,b,c,d,e,f,s,
```

图 6: 压缩查表运算流程

节均可进行加密的情况并不会影响最终的速率对比关系。

至此，实验结束。

参考资料：

《现代密码学基础》杨波（第四版）