# 应用密码学实验指导书

张仕斌　万武南　张金全　杨帆 编著

成都信息工程大学网络空间安全学院
中国成都

二〇二四年三月

# 目 录

# 实验一 AES 变换操作的实现

## 一、实验目的

（1）加深对 AES 算法的理解；

（2）阅读标准(fips-197)和文献，提高自学能力；

（3）加深对模块化设计的理解，提高编程实践能力。

## 二、实验内容

（1）按照 AES 算法，完成 AES 算法 S 盒、行移位、列混合、轮密钥加操作；

## 三、实验要求

（1）把 AES 的 S 盒、行移位、列混合、轮密钥加操作**写成一个函数**，然后再主程序中调用；（备注：函数名称命名方式 ****_学号后 2 位，变量名称***_学号后 2 位，例如学号为“2018011263”S 盒函数 SubBytes_63(unsigned char input_63[4][4]）

（2）输入 128bits 一个状态矩阵，输入为 16 进制数。状态矩阵的录入为一行，比如：

**32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34**

则轮密钥加的子密钥如：

**a0 fa fe 17　88 54 2c b1 23 a3 39 39　2a 6c 76 05**

(请注意这里是**列向优先**读取数据的)

（3）与**标准中的样例**进行比较,输入标准中样例，测试 S 盒、行移位、列混合和轮密钥加算法正确。

输出要求：

（1）程序的输出部分写在主程序中，可以是 4 行 4 列，也可以为一行。若为一行，输出也要求为**列向优先**。

（2）实验程序测试中，二组测试结果

第 1 组：

状态矩阵的录入为一行，比如:

19　3d　e3　be　a0　f4　e2　2b　9a　c6　8d　2a　e9　f8　48　08

则轮密钥加的子密钥如：

**a0 fa fe 17　88 54 2c b1 23 a3 39 39　2a 6c 76 05**

字节替代、行移位、列混合、轮密钥加的测试结果

第 2 组 例如学号为“2018011263”

状态矩阵的录入为一行，比如:

**20 18 01 12 63 00 00 00 00 00 00 00 00 00 0B**

则轮密钥加的子密钥如：

**a0 fa fe 17　88 54 2c b1 23 a3 39 39　2a 6c 76 05**

字节替代、行移位、列混合、轮密钥加的测试结果

（3）认真填写实验报告。

（4）截图最好有个人信息，比如输出执行的文件的路径有学号或者姓名等。

（5）截图最好前景是运行界面，背景是运行界面未遮住的一部分代码。

（6）代码附在实验报告的后面。

## 四、参考资料

**（1）按照标准，如果输入和标准相同，输出应该显示如下。可以输出矩阵，也可以输出行，建议输出行。**

**Initial state　32 88 31 e0**
**43 5a 31 37**
**f6 30 98 07**
**a8 8d a2 34**

**Round Key　2b 28 ab 09**
**7e ae f7 cf**
**15 d2 15 4f**
**16 a6 88 3c**

**Round 1**

**After**AddRoundKey　**19 a0 9a e9**
**3d f4 c6 f8**
**e3 e2 8d 48**
**be 2b 2a 08**

**After SubBytes　d4 e0 b8 1e**
**27 bf b4 41**
**11 98 5d 52**
**ae f1 e5 30**

**After ShiftRows　d4 e0 b8 1e**
**bf b4 41 27**
**5d 52 11 98**
**30 ae f1 e5**

**After MixColumns　04 e0 48 28**
**66 cb f8 06**
**81 19 d3 26**

**e5 9a 7a 4c**

**供参考的 C 语言代码**

（2） AES 的 S 盒

```
static unsigned char Sbox[16*16]=
{// populate the Sbox matrix
        /* 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,
    /*f*/   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
0xb0, 0x54, 0xbb, 0x16};
```

（3）字节替代

```
void SubBytes_83(unsigned char State_83[N][N])
{
    int i,j;
    char high;
    char low;
    for (i = 0; i < N; i ++)
    {
        for (j = 0; j < N; j ++)
        {
                low   = State[i][j] & 0x0F;   //取低 4 位
                high = (State[i][j] >> 4)&0x0f;   //取高 4 位
                State[i][j] = Sbox[16*high + low];
        }
    }
}
```

（4）行移位

```
void ShiftRows(unsigned char State[N][N])
{
    int i,j,k;
    int shiftnum = 0;
    char tmp;
    for (i = 0; i < N; i ++)
    {
        for (j = 0; j < shiftnum; j ++)        //循环左移一次
        {
            tmp = State[i][0];
            for (k = 0; k < N-1; k ++)
            {
```

```
                State[i][k] = State[i][k+1];
            }
            State[i][k] = tmp;
        }
        shiftnum ++;                                    //移位次数+1
    }
}
```

（5）轮密钥加

```
void AddRoundKey(unsigned char State[N][N], unsigned char RoundKey[N][N])
{
    int i,j;
    for (j = 0; j < N; j ++)
    {
        for (i = 0; i < N; i ++)
            State[i][j] = State[i][j] ^ RoundKey[i][j];
    }
}
```

（6）列混合相关的 x 乘

跟 X 乘相关的：

```
unsigned char Aes::gfmultby01(unsigned char b)
{
    return b;
}
unsigned char Aes::gfmultby02(unsigned char b)
{
    if (b < 0x80)
        return (unsigned char)(int)(b <<1);
    else
        return (unsigned char)( (int)(b << 1) ^ (int)(0x1b) );
}
unsigned char Aes::gfmultby03(unsigned char b)
{
    return (unsigned char) ( (int)gfmultby02(b) ^ (int)b );
}
for(j=0;j<4;j++)
```

{ output[0][j]= gfmultby02(input[0][j]))^ gfmultby03(input[1][j]))^ gfmultby01(input[2][j]))^ gfmultby01(input[3][j]);

……..

}

**列混合中的字节乘法也可以考虑下面的代码：**

比如(不严谨的例子)：

a=0x02;b=0xEB

则调用方式为：GFMul(a,b)

```
/**
 * 有限域上的乘法 GF(2^8)
 */
byte GFMul(byte a, byte b) {
 byte p = 0;
 byte hi_bit_set;
 for (int counter = 0; counter < 8; counter++) {
  if ((b & byte(1)) != 0) {
   p ^= a;
  }
  hi_bit_set = (byte) (a & byte(0x80));
  a <<= 1;
  if (hi_bit_set != 0) {
   a ^= 0x1b; /* x^8 + x^4 + x^3 + x + 1 */
  }
  b >>= 1;
 }
 return p;
}
```

# 实验二 RSA 算法的实现

## 一、实验目的

（1）加深对 RSA 算法的理解；

（2）加深对平方乘算法和模重复平方方法的理解；

（3）加深对模块化设计的理解，提高编程实践能力。

## 二、实验内容：

（1）完成 RSA 算法加密和解密功能；

（2）按照欧几里得扩展算法求，计算 RSA 私钥；

（3）按照平方乘算法和模重复平方方法，分别计算 $a^m \bmod n$，完成 RSA 的加密和解密。

## 三、实验要求

（1）要求把欧几里得扩展算法、平方乘方法和模重复平方方法**写成一个函数**。（备注：函数名称命名方式 ****_学号后 2 位，变量名称***_学号后 2 位，例如学号为“2018011263”，平方乘算法，LRFun_63(int a_63,int m_63,int n_63）；

（2）在主程序中读入数据公钥 e，两素数 p，q，以及待加密消息 m；

（3）计算 n 值和 n 的欧拉函数，并输出显示；

（4）计算私钥 d 值；

（5）调用平方乘，使用公钥加密得到密文 c，并输出显示；

（6）调用模重复平方方法，使用解密得到明文 m，并输出显示；

（7）使用模重复平方和平方乘输出计算的中间结果及最终运算结果。

（8）认真填写实验报告。(实验报告必须有测试结果调试) 测试结果要求（三组测试）：

**测试结果要求：**

1）1 组固定的数组设置 ，RSA 算法测试，e=7 p=13 q=17 m=22

（下面的输出界面仅供参考，要求对用户友好，易读易懂）

```
请顺序输入公钥，两素数：e  p  q:7 13 17
输出模数n：221
输出模数n的欧拉函数：192
输出模数私钥d：55
请输入要加密的明文m:22
RSA加密模幂运算底数 指数 模数 ：22 7 221
RSA第0组 底数 指数 模数 ：  22 7 221
平方乘
i=2,   22
i=1,   40
i=0,   61
加密后密文为：61
RSA解密模幂运算底数 指数 模数 ：61 55 221
模重复平方
i=0,   61
i=1,   14
i=2,   22
i=3,   22
i=4,   107
i=5,   22
解密后的明文为：22

请按任意键继续. . .
```

2）e，p，q 随机设置，m 为学号后 4 位整数，测试结果如下：

```
E:\课程相关资料\2019-2020（2）\应用密码学\课件\应用密码学实验资料2020\RSAtest\
输出模数n：713
输出模数n的欧拉函数：660
输出模数私钥d：283
请输入要加密的明文m:2000
RSA加密模幂运算底数 指数 模数 ：2000 7 713
RSA第0组 底数 指数 模数 ：  574 7 713
平方乘
i=2,   574
i=1,   252
i=0,   597
加密后密文为：597
RSA解密模幂运算底数 指数 模数 ：597 283 713
模重复平方
i=0,   597
i=1,   574
i=2,   574
i=3,   597
i=4,   436
i=5,   436
i=6,   436
i=7,   436
i=8,   574
解密后的明文为：574

RSA第1组 底数 指数 模数 ：  2 7 713
平方乘
i=2,   2
i=1,   8
i=0,   128
加密后密文为：128
RSA解密模幂运算底数 指数 模数 ：128 283 713
模重复平方
i=0,   128
i=1,   219
i=2,   219
i=3,   438
i=4,   326
i=5,   326
i=6,   326
i=7,   326
i=8,   2
解密后的明文为：2
```

3）e，p，q 随机设置，m 为字符串“RSA”，测试结果如下：

## 四、主要参考代码

```
//平方乘方法   a^n mod n
```

```
long int LRFun(long int a, long int n,    int key[],int klen)
{
    printf("平方乘\n");
    int s = 1;
    for (int i = klen-1;i>=0; i--)
    {
        s = (s*s) % n;
        if (key[i] == 1)
        {
            s = (s*a) % n;
        }
        printf("i=%d,    %d\n", i, s);
    }
    return s;
}
//模重复平方法
long int RLFun(long int a, long int n, int key[], int klen)
{
    printf("模重复平方 \n");
    long int s = 1;
    for (int i =0; i <klen; i++)
    {
      if (key[i] == 1)
        {
            s = (s*a) % n;
        }
        a = (a*a) % n;
        printf("i=%d,    %d\n", i, s);
    }
    return s;
}
long int Exgcd(long int a, long int b, long int &x, long int &y)        //扩展欧几里得
{
    if (b == 0)
    {
```

```
        x = 1;
        y = 0;
        return a;
    }
    long int r = Exgcd(b, a%b, x, y);    //ax1+by1=bx2+[a-(a/b)b]y2=ay2+bx2-
b(a/b)y2
    int temp = x;                        //由于a和b相同，所以有x1=y2 y1=x2-(a/b)y2
    x = y;
    y = temp - a / b * y;
    return r;
}
long int mod_reverse(long int a, long n)//求a的逆元x
{
    long int r, x, y;
    r = Exgcd(a, n, x, y);
    if (r == 1)
        return (x%n + n) % n;      //输出正数
    else
        return -1;
   }
```

# 实验三 消息摘要函数 SHA-1 算法的实现

## 一、实验目的

（1）加深对消息摘要函数 SHA-1 的理解；

（2）掌握消息摘要函数 SHA-1；

（3）提高编程实践能力。

## 二、实验内容

（1）按照标准 FIPS-180-2 中 SHA-1 算法，从文件或者屏幕中读取消息，然后对消息分组，并对最后一个分组进行填充，并对每组通过数据扩充算法扩充到 80 个字，然后执行 SHA-1 算法，并显示输出。

（2）完成填充过程，消息的长度在 1-200 个字符。

## 三、实验要求

（1）输入待 Hash 消息字符串，编码方式为 ASCII 码。例如程序的默认输入为 FIPS-180-2 中示例的“abc”， 消息的长度在 1-200 个字符。

（2）按照 SHA-1 算法进行填充，然后 512 比特分组，分为多组，然后对每组消息进行处理，数据扩充到 80 个字。

（3）输出每一分组中的 $W_0$, $W_1$，$W_{14}$，$W_{15}$，$W_{16}$, $W_{79}$ （十六进制）

（4）填充过程写成一个函数，数据扩充过程写成一个函数，数据扩充中循环移位也可以写成一个函数。函数名称命名方式 ****_学号后 2 位，变量名称***_学号后 2 位，例如学号为“2018011263”，平方乘算法，SHA1ProcessMessageBlock_63(unsigned char Message_Block）

例如：

输出为：

W0 : 61626380

W1 :00000000

W14:00000000

W15 : 00000018

….

(5) 认真填写实验报告，测试结果与标准对比。

**测试结果要求：**

（1）输入为 ASCII 码，程序的默认输入为 FIPS-180-2 中示例的“abc”，输出测试结果；

按照标准，输出如下：

W[0]=61626380

W[1]=00000000

W[14]=00000000

W[15]=00000018

W[16]=c2c4c700

W[79]=822e0879

最终的消息摘要值为：

a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d

（2）输入为 ASCII 码,程序输入为 FIPS-180-2 实例中的标准输入：

“**abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq**”

的测试结果（**不带引号**），以及每一个 512bit 块的下述子块的输出结果：

$W_0$, $W_1$，$W_{14}$，$W_{15}$，$W_{16}$，$W_{79}$ .

按照标准，输出的值如下：

第一个块：W[14]=80000000　　W[15]=00000000

W[16]=0a063a3e　　W[79]=1ff69958

第二个块：W[14]=00000000　　W[15]=000001c0

W[16]=00000000　　W[79]=04c77400

最终的消息摘要值为：

84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1

（3）输入 ASCII 码为各自学号，例如“2015011263” 的测试结果，以及每一个块的下述子块的输出结果：$W_0$, $W_1$，$W_{14}$ ，$W_{15}$.　W16，W79 .

（4）截图最好有个人信息，如输出执行的文件的路径有学号或者姓名等。

（5）截图最好前景是运行界面，背景是运行界面未遮住的一部分代码。

（6）代码附在实验报告的后面。

## 四、参考代码

代码如下：

```
#include <stdio.h>
#include <string.h>
//SHA 函数的结构体
typedef struct SHA1Context
{
    unsigned Length_Low;//消息长度低 32 位
    unsigned Length_High;//消息长度高 32 位
    unsigned char Message_Block[64];// 512bits 的块，总共 64 个字节
    int Message_Block_Index; //512bits 块索引号
    int Computed;
```

```
        int Corrupted;
    } SHA1Context;
    //循环左移 bits 位
    #define SHA1CircularShift(bits,word) ((((word) << (bits)) & 0xFFFFFFFF) |
((word) >> (32-(bits))))
    //赋初值函数
    void SHA1Reset(SHA1Context *context)
    {
        context->Length_Low = 0;
        context->Length_High = 0;
        context->Message_Block_Index = 0;
        context->Computed = 0;
        context->Corrupted = 0;
    }
    //每 512bits 数据块处理
    void SHA1ProcessMessageBlock(SHA1Context *context)
    {
        int t;
        unsigned W[80];
        for(t = 0; t < 16; t++)
    {
        W[t] = ((unsigned) context->Message_Block[t * 4]) << 24;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 1]) << 16;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 2]) << 8;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 3]);
    }
    for(t = 16; t < 80; t++)
    {
        W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }
      //每处理一块之后，SHAContext 块索引 0
        context->Message_Block_Index = 0;
    }
    //填充函数
    void SHA1PadMessage(SHA1Context *context)
```

```
    {
        if (context->Message_Block_Index > 55)
        {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
while(context->Message_Block_Index < 64)
    {
        context->Message_Block[context->Message_Block_Index++] = 0;
    }
SHA1ProcessMessageBlock(context);
while(context->Message_Block_Index < 56)
    {
        context->Message_Block[context->Message_Block_Index++] = 0;
    }
}
else
    {
context->Message_Block[context->Message_Block_Index++] = 0x80;
while(context->Message_Block_Index < 56)
    {
        context->Message_Block[context->Message_Block_Index++] = 0;
    }
}
    context->Message_Block[56] = (context->Length_High >> 24) & 0xFF;
    context->Message_Block[57] = (context->Length_High >> 16) & 0xFF;
    context->Message_Block[58] = (context->Length_High >> 8) & 0xFF;
    context->Message_Block[59] = (context->Length_High) & 0xFF;
    context->Message_Block[60] = (context->Length_Low >> 24) & 0xFF;
    context->Message_Block[61] = (context->Length_Low >> 16) & 0xFF;
    context->Message_Block[62] = (context->Length_Low >> 8) & 0xFF;
    context->Message_Block[63] = (context->Length_Low) & 0xFF;
    SHA1ProcessMessageBlock(context);
}
int SHA1Result(SHA1Context *context)
{
    if (context->Corrupted)
```

```
    {
        return 0;
    }
  if (!context->Computed)
  {
      SHA1PadMessage(context);
      context->Computed = 1;
  }
  return 1;
  }
  void SHA1Input(SHA1Context *context, const unsigned char *message_array,
unsigned length)
  {
      if (!length)
      {
      return;
      }
  if (context->Computed || context->Corrupted)
  {
      context->Corrupted = 1;
      return;
  }
  while(length-- && !context->Corrupted)
  {
      // 每 8bits 的存放
      context->Message_Block[context->Message_Block_Index++]          =
(   *message_array & 0xFF);
  context->Length_Low += 8;
  context->Length_Low &= 0xFFFFFFFF;
  if (context->Length_Low == 0)
  {
  context->Length_High++;
  context->Length_High &= 0xFFFFFFFF;
  if (context->Length_High == 0)
    {
```

```
    context->Corrupted = 1;
    }
}
if (context->Message_Block_Index == 64)
    {
SHA1ProcessMessageBlock(context);
    }
message_array++;
    }
}
int main()
{
        SHA1Context sha;
        char input[64];
        printf("ASCII string：");
        scanf("%s", input);
        SHA1Reset(&sha);
        SHA1Input(&sha, (const unsigned char *)input, strlen(input));
        SHA1Result(&sha);
        return 0;
}
```

# 实验四 RSA 数字签名方案的实现

## 一、实验目的

（1）加深对数字签名算法的理解；

（2）加深消息摘要函数 SHA-1 的掌握；

（3）提高编程实践能力。

## 二、实验内容

（1）输入字符作为原始消息，采用 SHA-1 算法，输出固定长度 160 比特的消息摘要；

（2）将输出的消息摘要作为输入，采用 RSA 算法的私钥进行签名，得到签名消息；

（3）将原始消息和签名消息作为输入，采用 RSA 算法的公钥进行签名验证。

## 三、实验要求

（1）把实验三完成的 SHA-1 杂凑函数设计成可以调用的函数；

（2）原始消息输入字符，调用 SHA-1 杂凑函数，摘要值输出采用十六进制；

（3）在 RSA 密钥生成过程中，采用自选的小素数 p 和 q，生成 RSA 私钥 d 和公钥 e，p 和 q 的乘积大于 256（十进制）；

（4）用 RSA 私钥 d 签名，公钥 e 验证，每 8 比特签名一次，总共做 20 次签名和验证；

（5）输出界面美观。

**测试结果要求：**

（1） 第 1 组测试要求：1 组固定的数值设置，RSA 算法测试，**e=5**, p=23, q=29; 求出私钥 d=493，

（2）输入为 ASCII 码，程序的默认输入为 FIPS-180-2 中示例的“abc”，最终的消息摘要值为：

a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d

（3）对0xa9签名过程：0xa9=169，$169^{493}\bmod(23*29)= 169^{493}\bmod(667)=78$。后面的分组以此类推。

（4）第 2 组输入的 ASCII 码为各自学号，例如“2015011263”，计算其消息摘要值并进行签名。

（5）截图最好有个人信息，如输出执行的文件的路径有学号或者姓名等。

（6）截图最好前景是运行界面，背景是运行界面未遮住的一部分代码。

（7）代码附在实验报告的后面。