

# AES 算法之理论与编程结合篇

作者：hecl

看雪 ID：chence

时间：05/29/2014

## 目录

|                             |    |
|-----------------------------|----|
| 1 前言.....                   | 1  |
| 2 AES 的数学理论基础.....          | 1  |
| 3 AES 算法.....               | 3  |
| 3.1 AES 和 Rijndael.....     | 3  |
| 3.1 AES 的加解密流程图： .....      | 4  |
| 3.2 AES 状态、种子密钥和轮数.....     | 5  |
| 3.3 AES 的轮函数.....           | 5  |
| 3.3.1 阶段一：字节代换（S 盒变换） ..... | 5  |
| 3.3.2 阶段二：行移位变换.....        | 7  |
| 3.3.3 阶段三： 列混淆变换： .....     | 8  |
| 3.3.4 阶段四：轮密钥变换加.....       | 9  |
| 3.4 密钥扩展： .....             | 9  |
| 4 AES 的实现.....              | 10 |
| 5 AES 的加密模式.....            | 14 |
| 6 AES 算法的安全性.....           | 16 |
| 参考资料.....                   | 16 |

## 1 前言

AES 是现在使用最多的对称密钥分组密码算法，在逆向的过程中经常碰到，这几天处于离职期，有点时间，于是乎想细细的来研究一下它的原理，也算是离职的一个纪念吧。

网上的文章都是理论加概况性的，不好理解。

这篇文章就带领大家理论的指导下，结合编程来理解 AES 算法。由于在密码学中，我也处于会使用接口型的，就没有自己写代码了，那就选择站在巨人的肩膀上了，哈哈。代码借用了 Tu Yongce 的 AES0.2 版的代码，并在文章中多次贴出了部分代码用来理解相关的理论。代码涉及两个工程：工程 AesArrays 用来产生最终的 AES 算法所需要的各种变换数组，也实现了算法的标准流程中的各阶段算法；工程 AesCipher 实现最终的 AES 算法，它使用了 AesArays 中生成的数组数据。对于想彻底搞清楚 AES 算法是很不错的资料，在此表示感谢！

学习 AES 算法，逆向分析也是有用的，如果使用一些算法识别插件，就算识别出来了是 AES，你还是会觉得很茫然，心中不会有一种很爽朗的感觉，因为你不理解这个算法。

这也是我想深入学习 AES 算法的原因。

文章中有错误的地方恳请大家批准指出，我会进行相应的完善，谢谢~

## 2 AES 的数学理论基础

域：域 F, 有时记为  $\{F,+, \times\}$ ，是有两个二元运算的集合，这两个二元运算分别称为加法和乘法，且对于 F 中的任意元素 a、b、c，满足以下公理：

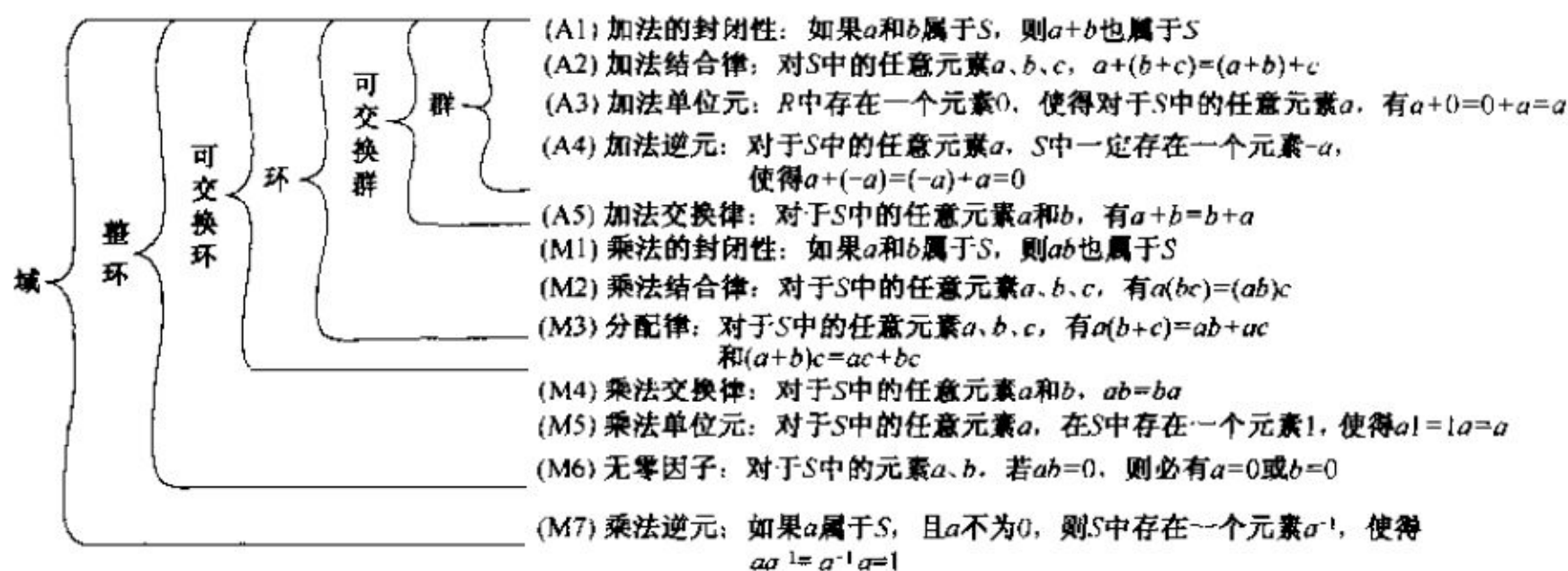


图 2.1

从图中可以看出, 域要满足 5 个加法条件和 7 个乘法条件。

有限域: 有限域许多密码编码学算法中扮演着重要的角色, 有限域的元素个数必须是一个素数的幂  $p^n$ ,  $n$  为正整数。元素个数为  $p^n$  的有限域一般记为  $GF(p^n)$ 。在

乘法逆元: 对于有限域  $GF(p^n)$ , 任意的  $w \in GF(p^n), w \neq 0$  存在  $z \in GF(p^n)$ , 使得  $w \times z \equiv 1 \pmod{p}$ , 则  $z$  为  $w$  在该有限域上的乘法逆元。

如果定义了合适的运算, 那么每一个这样的集合  $S$  都是一个有限域。定义由如下几条组成:

1. 该运算遵循基本代数规则中的普通多项式运算规则;
2. 系数运算以  $p$  为模, 即遵循有限域  $Z_p$  上的运算规则;
3. 如果乘法运算结果是次数大于  $n-1$  的多项式, 那么必须将其除以某个次数为  $n$  的既约多项式  $m(x)$  并取余式。

AES 使用有限域  $GF(2^8)$  上的运算, 为了构造有限域  $GF(2^8)$ , 必须定义一个既约多项式, 在 AES 中, 这个既约多项式为  $m(x) = x^8 + x^4 + x^3 + x + 1$ 。

下面来讲讲 AES 里的乘法逆元的求法, 由于公式太多, 输入不便, 故截图之:

### 乘法

简单的异或运算不能完成  $GF(2^n)$  上的乘法。但是可以使用一种合理且容易实现的技巧。我们将在高级加密标准使用的有限域中讨论该技巧。这个有限域是  $GF(2^8)$ , 其中模多项式为  $m(x) = x^8 + x^4 + x^3 + x + 1$ 。

这个技巧基于下面的等式:

$$x^8 \bmod m(x) = [m(x) - x^8] = (x^4 + x^3 + x + 1) \quad (4.7)$$

通过观察不难证明等式(4.7)是正确的。一般地, 在  $GF(2^n)$  上对于  $n$  次多项式  $p(x)$ , 有  $x^n \bmod p(x) = [p(x) - x^n]$ 。

现在考虑  $GF(2^8)$  上的多项式  $f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ , 将它乘以  $x$ , 可得:

$$x \times f(x) = (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod m(x) \quad (4.8)$$

如果  $b_7 = 0$ , 那么结果就是一个次数小于 8 的多项式, 不需要进一步计算。如果  $b_7 = 1$ , 那么可以通过等式(4.7)进行除  $m(x)$  取余运算:

$$x \times f(x) = (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) + (x^4 + x^3 + x + 1)$$

这表明乘以  $x$  (如 00000010) 的运算可以通过左移一位后按位异或 00011011 来实现, 其表示为  $(x^4 + x^3 + x + 1)$ 。总结如下:

$$x \times f(x) = \begin{cases} (b_6b_5b_4b_3b_2b_1b_00) & \text{若 } b_7 = 0 \\ (b_6b_5b_4b_3b_2b_1b_00) \oplus (00011011) & \text{若 } b_7 = 1 \end{cases} \quad (4.9)$$

乘以一个高于一次的多项式可以通过重复使用等式(4.9)来实现。这样一来,  $GF(2^8)$  上的乘法可以用多个中间结果相加的方法实现。

图 2.2

当我在看这里的时候也花了蛮久才看明白，主要是划红线的式子不知道是怎么变来的。其实作者省略了变化的过程，具体的变化过程如下：

$$\begin{aligned} b_7=1 \text{ 时,} \\ x \times f(x) &= (x^8 + b_6x^7 + b_5x^6 + \dots + b_0x) \bmod m(x) \Rightarrow \\ x \times f(x) &= x^8 \bmod m(x) + (b_6x^7 + b_5x^6 + \dots + b_0x) \bmod m(x) \Rightarrow \\ x \times f(x) &= (b_6x^7 + b_5x^6 + \dots + b_0x) \bmod m(x) + (x^4 + x^3 + x + 1) \Rightarrow \\ x \times f(x) &= (b_6x^7 + b_5x^6 + \dots + b_0x) + (x^4 + x^3 + x + 1) \end{aligned}$$

## 3 AES 算法

### 3.1 AES 和 Rijndael

1997 年 1 月，美国 NIST 向全世界密码学界发出征集 21 世纪高级加密标准(AES——Advanced Encryption Standard)算法的公告，并成立了 AES 标准工作研究室，1997 年 4 月 15 日的例会制定了对 AES 的评估标准。

1998 年 4 月 15 日全面征集 AES 算法的工作结束。

1998 年 8 月 20 日举行了首届 AES 讨论会，对涉及 14 个国家的密码学家所提出的候选 AES 算法进行了评估和测试，初选并公布了 15 个候选方案，供大家公开讨论。

15 个候选算法有：CAST-256，RC-6，CRYPTON-128，DEAL-128，FROG，简易布丁密码，LOKI-97，MAGENTA，MARS，Vaudenay 的抗相关快速密码 RIJNDAEL，SAFER+，SERPENT，E-2，TWO FISH。这些算法设计思想新颖，技术水平先进，算法的强度都超过 3-DES，实现速度快于 3-DES。

1999 年 8 月 9 日 NIST 宣布第二轮筛选出的 5 个候选算法为：

MARS(C.Burwick 等,IBM),RC6TM(R. Rivest 等,RSA Lab.),RIJNDEAL(J. Daemen,比),SERPENT(R. Anderson 等,英、以、挪威),TWO FISH(B. Schiener)。

2000 年 4 月 13 日，第三次 AES 会议上，对这 5 个候选算法的各种分析结果进行了讨论。

2000 年 10 月，由比利时的 Joan Daemen 和 Vincent Rijmen 提出的算法最终胜出。

2001 年 11 月，NIST 完成了评估并发布了最终标准(FIPS PUB 197)，选择 Rijndael 作为 AES 算法。

3.1 AES 的加解密流程图：

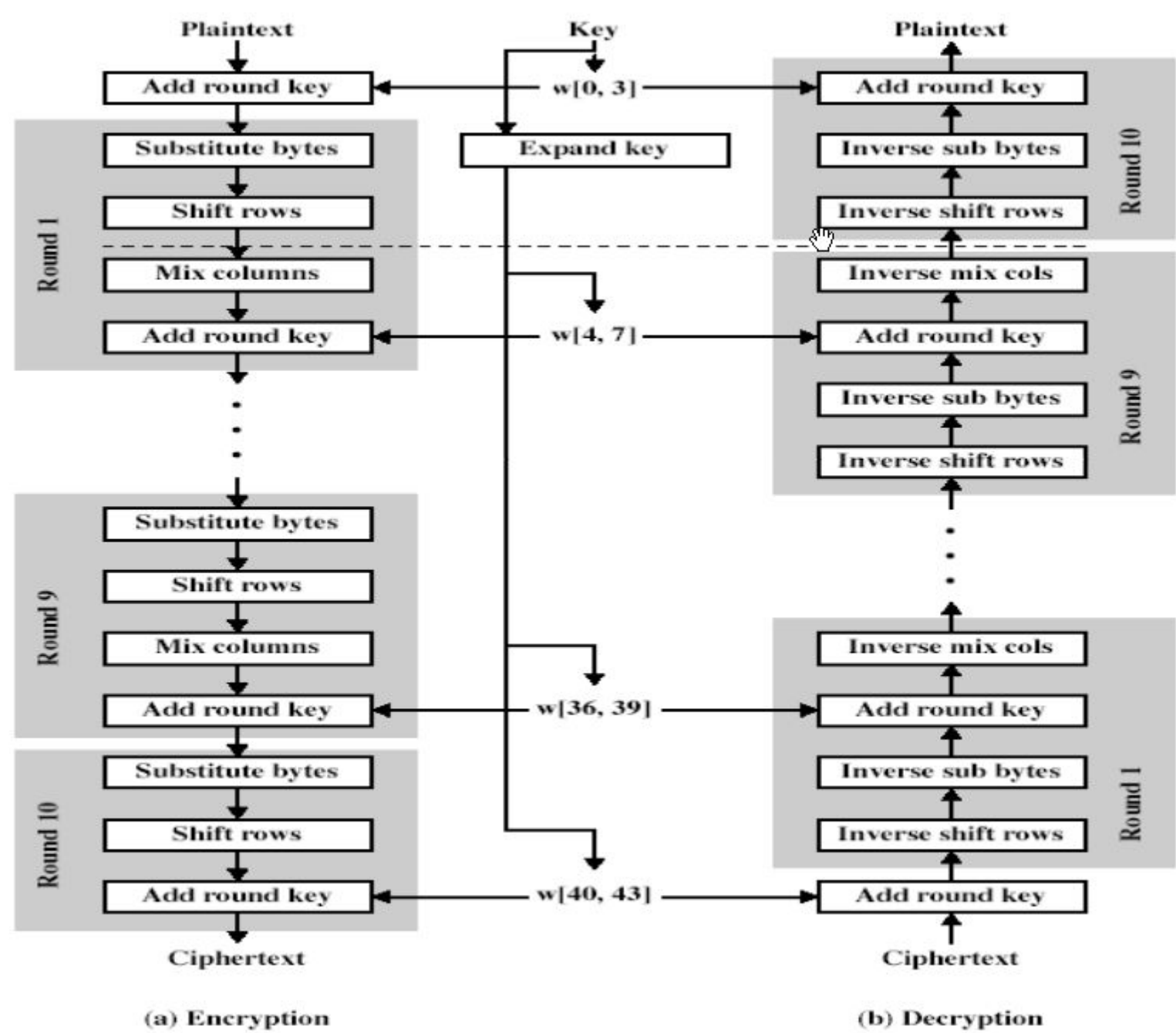


图 3.1

从图上可以看出 AES 加解密包括 10 轮，前面 9 轮包含 S 盒变换、行移位、列混淆、轮密钥加 4 个阶段，最后一轮则少了列混淆这个阶段。

Rijndael（读作 rin:dol）主要的流程如下：

```
Rijndael(State, CipherKey) {  
    //初始化  
    KeyExpansion( CipherKey, ExpandedKey );//生成子密钥  
    AddRoundKey( State, ExpandedKey );//与子密钥位与  
    // 前 Nr-1轮  
    for(i =1; i < Nr; i++) {  
        ByteSub(State);// S-盒  
        ShiftRow(State);// 行被移位  
        MixColumn(State);// 列被混叠  
        AddRoundKey (State, ExpandedKey ); //与子密钥位与  
    }  
    //最后一轮  
    ByteSub(State);  
    ShiftRow(State);  
    AddRoundKey (State, ExpandedKey );  
}
```

3.2 AES 状态、种子密钥和轮数

状态：

- ①加解密过程中的中间数据。
- ②以字节为元素的矩阵，或二维数组。

所有的操作都在状态上进行。



图 3.2

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 Nb，Nb 等于分组长度除以 4。

符号说明：

Nb 一明密文所含的数据字数。

Nk 一密钥所含的数据字数。

Nr 一迭代圈数。

3.3 AES 的轮函数

AES 的轮函数由 4 个不同的计算部件组成

分别是：

字节代换 BS （ ByteSub ）,非线性层

行移位 SR （ ShiftRow ）,线性层

列混合 MC （ MixColumn ）,线性层

密钥加 ARK （ AddRoundKey ），线性层

3.3.1 阶段一：字节代换（S 盒变换）

S 盒变换其实是一个查表的过程，分别取一个字节的 高 4 位和低 4 位作为行值和列值（因此是  $2^4 \times 2^4$ ），然后在 S 盒中找到对应的字节替换之。该变换是一个非线性变换。这个非线性就体现在 S 盒的构造上。S 盒变换是 AES 的唯一非线性变换，是 AES 安全的关键。

关于 S 盒是按如下方式构造：

- 1) 初始化 S 盒，按行升序排列的字节初始化。行 x 列 y 的字节是 xy，行号和列号从 0 开始计数。
- 2) 求出每一个元素在  $GF(2^8)$  中的逆。00 被映射为它自身。
- 3) 仿射变换。对上一步中的每一个字节的每一位作如下变换

$$Y_i = X_i + X_{(i+4) \bmod 8} + X_{(i+5) \bmod 8} + X_{(i+6) \bmod 8} + X_{(i+7) \bmod 8} + C_i$$

Ci 是字节 0x63 的第 i 位，用矩阵表示如下：

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

图 3.3

编程实现 S 盒:

```
/**
 * Calculate S-Box (Substitute Bytes) and inverse S-Box (Inverse Substitute Bytes) under polynomial GF(2^8).
 * @param sbox: Array which holds the sbox table.
 * @param isbox: Array which holds the inverse sbox table.
 * @param num: Size of the array (should be 256).
 */
void getSbox(unsigned char sbox[], unsigned char isbox[], size_t num)
{
    sbox[0] = 0x63; // 00 求逆为 00, 再异或 0x63
    isbox[0x63] = 0;
    for (size_t i = 1; i < num; ++i) {
        unsigned char tmp = getMulInverse(i); // 先求逆
        // b[i] = b[i + 4] mode 8 ^ b[i + 5] mode 8 ^ b[i + 6] mode 8 ^ b[i + 7] mode 8 ^ 0x63
        tmp = tmp ^ (tmp << 4 | tmp >> 4) ^ (tmp << 3 | tmp >> 5)
            ^ (tmp << 2 | tmp >> 6) ^ (tmp << 1 | tmp >> 7) ^ 0x63; // 仿射变换, 字节循环移位再异或
        if (sbox != 0)
            sbox[i] = tmp; // S 盒
        if (isbox != 0)
            isbox[tmp] = i; // 逆 S 盒
    }
}

/**
 * Calculate multiplicative inverse under polynomial GF(2^8).
 * @return: Multiplicative inverse of @param a.
 */
unsigned char getMulInverse(unsigned char a)
{
    unsigned char ret = 1;
    while (mul(a, ret) != 1) // 两个数在 GF(2^8)相乘得 1, 则构成互逆, 这里采用的是正向穷举的办法
        ++ret;

    return ret;
}

/**
 * Multiply two numbers under polynomial GF(2^8).
 * @return: The product of @param a and @param b.
 */
unsigned char mul(unsigned char a, unsigned char b)
{
    unsigned char ret = 0;

    for (int i = 0; i < 8; ++i) { // 循环累加 b 的每 1 位
        if (b & 0x01)
            ret ^= a;

        if (a & 0x80) { // x * f(x), b_7 = 1
            a <<= 1; // a 先左移 1 位
            a ^= 0x1B; // 再异或 0x1B
        } else
            a <<= 1; // b_7 = 0, x * f(x) 等于直接左移 1 位
    }
}
```



```
        b >>= 1;//取 b 的高 1 位
    }

    return ret;
}
```

函数 mul 写的相当精彩，非常真实的还原了 GF(2<sup>8</sup>)上的乘法，可仿照第 2 页的理论对照看。

附上 S 盒和逆 S 盒：

```
unsignedchar sBox[] =
{ /* 0 1 2 3 4 5 6 7 8 9 a b c d e f */
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, /*0*/
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0, /*1*/
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15, /*2*/
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75, /*3*/
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84, /*4*/
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf, /*5*/
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8, /*6*/
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2, /*7*/
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73, /*8*/
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb, /*9*/
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79, /*a*/
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08, /*b*/
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a, /*c*/
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e, /*d*/
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf, /*e*/
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 /*f*/
};

unsignedchar invsBox[256] =
{ /* 0 1 2 3 4 5 6 7 8 9 a b c d e f */
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb, /*0*/
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb, /*1*/
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e, /*2*/
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25, /*3*/
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92, /*4*/
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84, /*5*/
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06, /*6*/
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b, /*7*/
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73, /*8*/
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e, /*9*/
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b, /*a*/
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4, /*b*/
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f, /*c*/
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef, /*d*/
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61, /*e*/
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d /*f*/
};
```

输入 6A ， 查找 S 盒 x=6, y=a => 02。反过来，输入 02， 查找逆 S 盒 x=0, y=2 => 6a。

### 3.3.2 阶段二：行移位变换

行移位按如下的方式进行：

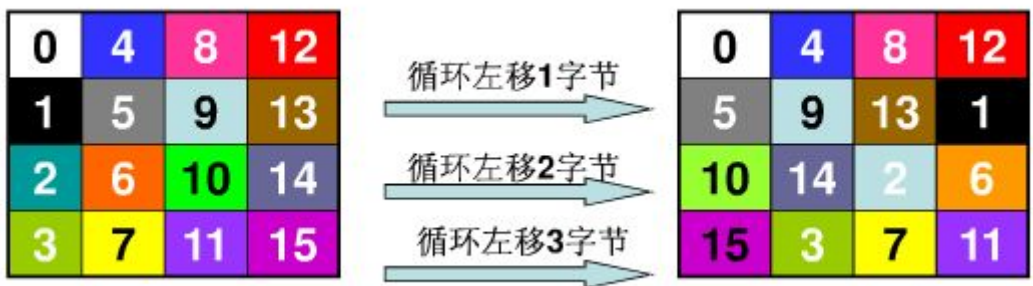


图 3.4

第一行不变，第二行循环左移 1 字节，第三行循环左移 2 字节，第三行循环左移 3 字节。这个编程应该容易实现。

### 3.3.3 阶段三： 列混淆变换：

列混淆即是用一个常矩阵乘以第二步变换后的矩阵，以达到矩阵中每一个元素都是该元素原所在列所有元素的加权和。

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

$$\begin{aligned} s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\ s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j}) \end{aligned}$$

图 3.5

逆列混淆变换：

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

图 3.6

参考代码：

```
/**
 * MixColumns transformation.
 * @param w: Machine word (int32) to be transformed.
 * @return: Machine word generated by MixColumns transformation.
 */
u32_t mixColumn(u32_t w)
{
    unsigned char byte0 = (w >> 24) & 0xFF;
    unsigned char byte1 = (w >> 16) & 0xFF;
    unsigned char byte2 = (w >> 8) & 0xFF;
    unsigned char byte3 = w & 0xFF;

    unsigned char b0 = mul(byte0, 2) ^ mul(byte1, 3) ^ byte2 ^ byte3; // S'_{0,j}

    unsigned char b1 = byte0 ^ mul(byte1, 2) ^ mul(byte2, 3) ^ byte3; // S'_{1,j}
    unsigned char b2 = byte0 ^ byte1 ^ mul(byte2, 2) ^ mul(byte3, 3);
    unsigned char b3 = mul(byte0, 3) ^ byte1 ^ byte2 ^ mul(byte3, 2);
    return (u32_t(b0) << 24) | (u32_t(b1) << 16) | (u32_t(b2) << 8) | u32_t(b3);
}

/**
 * Inverse MixColumns transformation.
 * @param w: Machine word (int32) to be transformed.
 * @return: Machine word generated by inverse MixColumns transformation.
 */
u32_t invMixColumn(u32_t w)
{
    unsigned char byte0 = (w >> 24) & 0xFF;
    unsigned char byte1 = (w >> 16) & 0xFF;
    unsigned char byte2 = (w >> 8) & 0xFF;
    unsigned char byte3 = w & 0xFF;
    unsigned char b0 = mul(byte0, 0x0E) ^ mul(byte1, 0x0B) ^ mul(byte2, 0x0D) ^ mul(byte3, 0x09);
    unsigned char b1 = mul(byte0, 0x09) ^ mul(byte1, 0x0E) ^ mul(byte2, 0x0B) ^ mul(byte3, 0x0D);
```



```
    unsigned char b2 = mul(byte0, 0x0D) ^ mul(byte1, 0x09) ^ mul(byte2, 0x0E) ^ mul(byte3, 0x0B);
    unsigned char b3 = mul(byte0, 0x0B) ^ mul(byte1, 0x0D) ^ mul(byte2, 0x09) ^ mul(byte3, 0x0E);
    return (u32_t(b0) << 24) | (u32_t(b1) << 16) | (u32_t(b2) << 8) | u32_t(b3);
}
```

3.3.4 阶段四：轮密钥变换加

状态与轮密钥（16 byte）异或相加。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 。

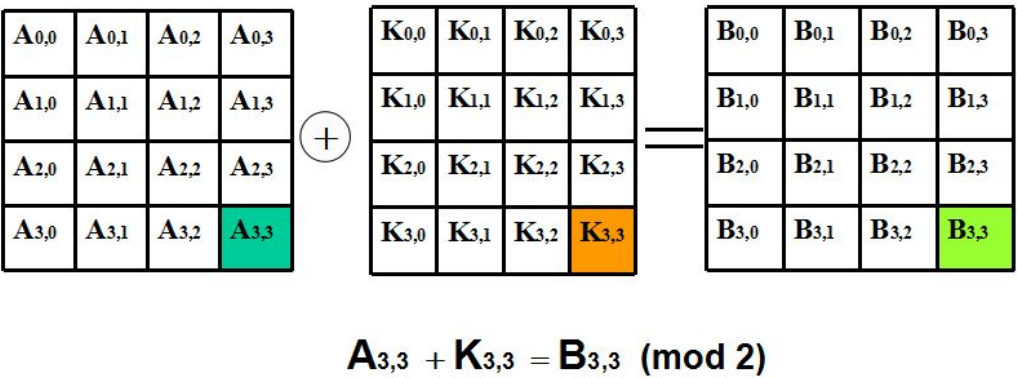


图 3.7

轮密钥加变换的逆就是其本身 (AddRoundKey)-1= AddRoundKey

3.4 密钥扩展：

用一个 4 字节字元素的一维数组  $W[Nb*(Nr+1)]$  表示扩展密钥。以输入的密钥长度为 4 字（16 字节）为例，在加密的过程中需要  $4+4*10$  字，即 176 字节数据来满足初始轮密钥阶段和 10 轮轮密钥加操作。

$W[]$  数组中最开始的  $Nk$  个字为种子密钥, 其它的字由它前面的字经过递归处理后得到。有  $Nk \leq 6$  和  $Nk > 6$  两种密钥扩展算法。

扩展算法如下：

- ①最前面的  $Nk$  个字是由种子密钥填充的。
- ②之后的每一个字  $W[j]$  等于前面的字  $W[j-1]$  的与  $Nk$  个位置之前的字  $W[j-Nk]$  的异或。
- ③而且对于  $Nk$  的整数倍的位置处的字，在异或之前，对  $W[j-1]$  的进行如下变换(就是图中的  $g$  函数)：
  - 字节的循环移位 RotByte，即当输入字为  $(a, b, c, d)$  时，输出字为  $(b, c, d, a)$
  - 用 S 盒进行变换字中的每个字节
  - 异或轮常数  $Rcon[i/Nk]$

轮常量是一个字，这个字最右边三个字节总是 0。每轮的轮常量均不同，其定义为  $Rcon[j] = (RC[j], 0, 0, 0)$ ，其中  $RC[1] = 1$ ， $RC[j] = 2*RC[j-1]$ ，且乘法定义在域  $GF(2^8)$  上。

|       |    |    |    |    |    |    |    |    |    |    |
|-------|----|----|----|----|----|----|----|----|----|----|
| j     | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
| RC[j] | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 |

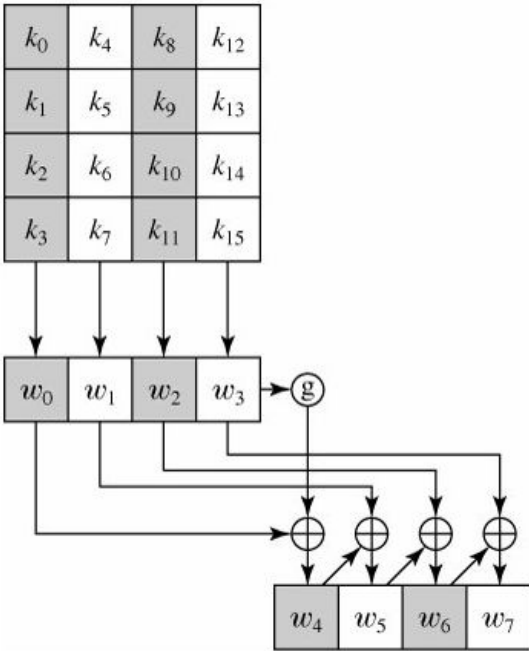


图 3.8

```
代码:
(1)  $Nk \leq 6$  的密钥扩展
KeyExpansion (byteKey[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] );
//扩展密钥的前面 4 个字由种子密钥组成

    for (i =Nk; i <Nb*(Nr+1); i++)
    {
        temp=W[i-1];
        if (i % Nk==0)          //i 是 NK 的整数倍是要特殊处理
            temp=SubByte (RotByte (temp))^Rcon[i /Nk];
        W[i]=W[i-Nk]^ temp;
    }
}

(2)  $Nk > 6$  的密钥扩展
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] );
//扩展密钥的前面 4 个字由种子密钥组成
for (i =Nk; i <Nb*(Nr+1); i++)
{
    temp=W[i -1];
    if (i % Nk==0) //i 是 NK 的整数倍是要特殊处理
        temp=SubByte (RotByte (temp))^Rcon[i /Nk];
    else if (i % Nk==4)    //i 是 4 的整数倍是要特殊处理
        temp=SubByte (temp);
    W[i]=W[i - Nk]^ temp;
}
}
```

## 4 AES 的实现

AES 实现的方式有软件和硬件之分。软件又分为基于算法描述和基于查表两种方式。

基于算法描述的软件实现

- AES 的算法描述是一种程序化的描述，便于实现
- AES 的四种基本变换都比较简单，便于实现
- 用 C 语言仿照算法描述，可方便地实现
- 这种实现的速度不是最快的

基于算法的描述代码是很容易看懂的，这里就不详述了。

基于查表的软件实现

- 用查表实现算法是一种高效的软件设计方法
- 时空折换是信息科学的基本策略
- 用查表实现算法，就是用空间换取时间
- 目前计算机系统的存储空间大、而且便宜，为查表实现算法提供了物资基础

现在的软件应该大都采用的是基于查表的方式，以空间换取时间，这样可提高软件的执行效率。在逆向的过程中如果发现了这些常数，就可以断定采用的是 AES 加密了。

查表法加密：

```
/**
 * Encrypt exactly one block of plaintext, assuming AES' block size (128-bit).
 * @param in: The plaintext.
 * @param result: The ciphertext generated from a plaintext using the key.
 */
void AesCipher::encryptBlock(const char *in, char *result)
{
```

```

// first, convert chars into words (32bit per word)
// 将16个字节转换成4个字
u32_t w1 = (u32_t)(unsigned char)*(in++) << 24;
w1 |= (u32_t)(unsigned char)*(in++) << 16;
w1 |= (u32_t)(unsigned char)*(in++) << 8;
w1 |= (u32_t)(unsigned char)*(in++);

u32_t w2 = (u32_t)(unsigned char)*(in++) << 24;
w2 |= (u32_t)(unsigned char)*(in++) << 16;
w2 |= (u32_t)(unsigned char)*(in++) << 8;
w2 |= (u32_t)(unsigned char)*(in++);

u32_t w3 = (u32_t)(unsigned char)*(in++) << 24;
w3 |= (u32_t)(unsigned char)*(in++) << 16;
w3 |= (u32_t)(unsigned char)*(in++) << 8;
w3 |= (u32_t)(unsigned char)*(in++);

u32_t w4 = (u32_t)(unsigned char)*(in++) << 24;
w4 |= (u32_t)(unsigned char)*(in++) << 16;
w4 |= (u32_t)(unsigned char)*(in++) << 8;
w4 |= (u32_t)(unsigned char)*(in++);

// AddRoundKey transformation for plaintext
w1 ^= m_ke[0][0];
w2 ^= m_ke[0][1];
w3 ^= m_ke[0][2];
w4 ^= m_ke[0][3]; // 初始异或

// round transforms
for (int i = 1; i < m_rounds; ++i) {
    u32_t t1, t2, t3, t4;

    // 字节代换, 行移位, 列混淆, 轮密钥加操作
    // sm_te 变换包括了字节代换, 行移位, 列混淆操作
    t1 = (sm_te1[w1 >> 24] ^ sm_te2[(w2 >> 16) & 0xFF] ^
        sm_te3[(w3 >> 8) & 0xFF] ^ sm_te4[w4 & 0xFF]
        ) ^ m_ke[i][0];
    t2 = (sm_te1[w2 >> 24] ^ sm_te2[(w3 >> 16) & 0xFF] ^
        sm_te3[(w4 >> 8) & 0xFF] ^ sm_te4[w1 & 0xFF]
        ) ^ m_ke[i][1];
    t3 = (sm_te1[w3 >> 24] ^ sm_te2[(w4 >> 16) & 0xFF] ^
        sm_te3[(w1 >> 8) & 0xFF] ^ sm_te4[w2 & 0xFF]
        ) ^ m_ke[i][2];
    t4 = (sm_te1[w4 >> 24] ^ sm_te2[(w1 >> 16) & 0xFF] ^
        sm_te3[(w2 >> 8) & 0xFF] ^ sm_te4[w3 & 0xFF]
        ) ^ m_ke[i][3];

    w1 = t1;
    w2 = t2;
    w3 = t3;
    w4 = t4;
}

// last round is special
int tmp = m_ke[m_rounds][0];
// Substitute Bytes, ShiftRows and AddRoundKey transformation
result[0] = sm_sbox[w1 >> 24] ^ (tmp >> 24);
result[1] = sm_sbox[(w2 >> 16) & 0xFF] ^ (tmp >> 16);
result[2] = sm_sbox[(w3 >> 8) & 0xFF] ^ (tmp >> 8);
result[3] = sm_sbox[w4 & 0xFF] ^ tmp;

tmp = m_ke[m_rounds][1];
result[4] = sm_sbox[w2 >> 24] ^ (tmp >> 24);
result[5] = sm_sbox[(w3 >> 16) & 0xFF] ^ (tmp >> 16);
result[6] = sm_sbox[(w4 >> 8) & 0xFF] ^ (tmp >> 8);
result[7] = sm_sbox[w1 & 0xFF] ^ tmp;

```

```

    tmp = m_ke[m_rounds][2];
    result[8] = sm_sbox[w3 >> 24] ^ (tmp >> 24);
    result[9] = sm_sbox[(w4 >> 16) & 0xFF] ^ (tmp >> 16);
    result[10] = sm_sbox[(w1 >> 8) & 0xFF] ^ (tmp >> 8);
    result[11] = sm_sbox[w2 & 0xFF] ^ tmp;

    tmp = m_ke[m_rounds][3];
    result[12] = sm_sbox[w4 >> 24] ^ (tmp >> 24);
    result[13] = sm_sbox[(w1 >> 16) & 0xFF] ^ (tmp >> 16);
    result[14] = sm_sbox[(w2 >> 8) & 0xFF] ^ (tmp >> 8);
    result[15] = sm_sbox[w3 & 0xFF] ^ tmp;
}

```

而 sm\_te 数组是如何生成的呢？

```

/**
 * Get transformation table including SubstituteBytes and MixColumns for encryption.
 * @param te1, te2, te3, te4: Arrays to hold the translation.
 * @param sbox: S-Box for SubstituteBytes.
 * @param num: Size of the array (should be 256).
 */
void getTE(u32_t te1[], u32_t te2[], u32_t te3[], u32_t te4[], unsigned char sbox[], size_t num)
{
    for (size_t i = 0; i < num; ++i) {
        te1[i] = (u32_t(mul(sbox[i], 2)) << 24) | (u32_t(sbox[i]) << 16) | (u32_t(sbox[i]) << 8) | u32_t(mul(sbox[i], 3)));
        te2[i] = (u32_t(mul(sbox[i], 3)) << 24) | (u32_t(mul(sbox[i], 2)) << 16) | (u32_t(sbox[i]) << 8) | u32_t(sbox[i]);
        te3[i] = (u32_t(sbox[i]) << 24) | (u32_t(mul(sbox[i], 3)) << 16) | (u32_t(mul(sbox[i], 2)) << 8) | u32_t(sbox[i]);
        te4[i] = (u32_t(sbox[i]) << 24) | (u32_t(sbox[i]) << 16) | (u32_t(mul(sbox[i], 3)) << 8) | u32_t(mul(sbox[i], 2));
    }
}

```

起初我很不解，这么复杂的算法怎么可以用查表这么轻松的给解决了，它是如何对应的，带着疑问，我就开始从理论开刀了。看下核心代码：

```

    t1 = (sm_te1[w1 >> 24] ^ sm_te2[(w2 >> 16) & 0xFF] ^
        sm_te3[(w3 >> 8) & 0xFF] ^ sm_te4[w4 & 0xFF]
    ) ^ m_ke[i][0];
    t2 = (sm_te1[w2 >> 24] ^ sm_te2[(w3 >> 16) & 0xFF] ^
        sm_te3[(w4 >> 8) & 0xFF] ^ sm_te4[w1 & 0xFF]
    ) ^ m_ke[i][1];
    t3 = (sm_te1[w3 >> 24] ^ sm_te2[(w4 >> 16) & 0xFF] ^
        sm_te3[(w1 >> 8) & 0xFF] ^ sm_te4[w2 & 0xFF]
    ) ^ m_ke[i][2];
    t4 = (sm_te1[w4 >> 24] ^ sm_te2[(w1 >> 16) & 0xFF] ^
        sm_te3[(w2 >> 8) & 0xFF] ^ sm_te4[w3 & 0xFF]
    ) ^ m_ke[i][3];

```

理论上是状态图中的第一列（字 1）和轮密钥的第一个字异或。根据图 3.5，在列混淆中字是这样子来的：

$$W(j) = s'_{0,j} \ll 24 \mid s'_{1,j} \ll 16 \mid s'_{2,j} \ll 8 \mid s'_{3,j}, j = 0,1,2,3$$

展开得

$$W(j) = (2 \bullet s_{0,j} \oplus 3 \bullet s_{1,j} \oplus s_{2,j} \oplus s_{3,j}) \ll 24 \mid (s_{0,j} \oplus 2 \bullet s_{1,j} \oplus 3 \bullet s_{2,j} \oplus s_{3,j}) \ll 16 \mid (s_{0,j} \oplus s_{1,j} \oplus 2 \bullet s_{2,j} \oplus 3 \bullet s_{3,j}) \ll 8 \mid (3 \bullet s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus 2 \bullet s_{3,j}), j = 0,1,2,3$$

根据逻辑运算的分配率和结合律，变换得

$$W(j) = (2 \bullet s_{0,j} \ll 24 \mid s_{0,j} \ll 16 \mid s_{0,j} \ll 8 \mid 3 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j} \ll 24 \mid 2 \bullet s_{1,j} \ll 16 \mid s_{1,j} \ll 8 \mid s_{1,j}) \oplus (s_{2,j} \ll 24 \mid 3 \bullet s_{2,j} \ll 16 \mid 2 \bullet s_{2,j} \ll 8 \mid s_{2,j}) \oplus (s_{3,j} \ll 24 \mid s_{3,j} \ll 16 \mid 3 \bullet s_{3,j} \ll 8 \mid 2 \bullet s_{3,j}), j = 0,1,2,3$$

即

$$W(j) = te1[s_{0,j}] \wedge te2[s_{1,j}] \wedge te3[s_{2,j}] \wedge te4[s_{3,j}], j = 1,2,3$$

注意，这里的  $s_{x,j}$  是行移位完成后对应的字节。可直接根据图 3.4 来取原来的 W 的对应字节，并且上面的代码就是这么干的。。。

回顾头去看上面的代码，t1,t2,t3,t4 都是由上一步的 4 个字的某个字节查表，然后异或相加。

因此可以先计算所有字节（00~FF）的各个 te，然后取出每个字中的字节，查表，再异或相加得到变换后的字。

经过前面的 Nr-1 轮变换后，最后一轮有点特殊，少了列混淆，作者也是采用查表发来实现的，表的生成方式原理可使用上面的推导方式来推导，这

里不再赘述。

解密的过程跟加密的过程是一样的，只是使用的表变了。因为行移位和逆矩阵变了。

解密的代码如下：

```
/**
 * Decrypt exactly one block of plaintext, assuming AES' block size (128-bit).
 * @param in: The ciphertext.
 * @param result: The plaintext generated from a ciphertext using the key.
 */
void AesCipher::decryptBlock(const char *in, char *result)
{
    // first, convert chars into words (32bit per word)
    u32_t w1 = (u32_t)(unsigned char)*(in++) << 24;
    w1 |= (u32_t)(unsigned char)*(in++) << 16;
    w1 |= (u32_t)(unsigned char)*(in++) << 8;
    w1 |= (u32_t)(unsigned char)*(in++);

    u32_t w2 = (u32_t)(unsigned char)*(in++) << 24;
    w2 |= (u32_t)(unsigned char)*(in++) << 16;
    w2 |= (u32_t)(unsigned char)*(in++) << 8;
    w2 |= (u32_t)(unsigned char)*(in++);

    u32_t w3 = (u32_t)(unsigned char)*(in++) << 24;
    w3 |= (u32_t)(unsigned char)*(in++) << 16;
    w3 |= (u32_t)(unsigned char)*(in++) << 8;
    w3 |= (u32_t)(unsigned char)*(in++);

    u32_t w4 = (u32_t)(unsigned char)*(in++) << 24;
    w4 |= (u32_t)(unsigned char)*(in++) << 16;
    w4 |= (u32_t)(unsigned char)*(in++) << 8;
    w4 |= (u32_t)(unsigned char)*(in++);

    // AddRoundKey transformation for plaintext
    w1 ^= m_kd[0][0];
    w2 ^= m_kd[0][1];
    w3 ^= m_kd[0][2];
    w4 ^= m_kd[0][3];

    // round transforms
    for (int i = 1; i < m_rounds; ++i) {
        u32_t t1, t2, t3, t4;

        // substitute byte, ShiftRows, MixColumns and AddRoundKey transformation
        // sm_te transformation includes substitute byte and MixColumns transformation
        t1 = (sm_td1[w1 >> 24] ^ sm_td2[(w4 >> 16) & 0xFF] ^
            sm_td3[(w3 >> 8) & 0xFF] ^ sm_td4[w2 & 0xFF]
            ) ^ m_kd[i][0];
        t2 = (sm_td1[w2 >> 24] ^ sm_td2[(w1 >> 16) & 0xFF] ^
            sm_td3[(w4 >> 8) & 0xFF] ^ sm_td4[w3 & 0xFF]
            ) ^ m_kd[i][1];
        t3 = (sm_td1[w3 >> 24] ^ sm_td2[(w2 >> 16) & 0xFF] ^
            sm_td3[(w1 >> 8) & 0xFF] ^ sm_td4[w4 & 0xFF]
            ) ^ m_kd[i][2];
        t4 = (sm_td1[w4 >> 24] ^ sm_td2[(w3 >> 16) & 0xFF] ^
            sm_td3[(w2 >> 8) & 0xFF] ^ sm_td4[w1 & 0xFF]
            ) ^ m_kd[i][3];

        w1 = t1;
        w2 = t2;
        w3 = t3;
        w4 = t4;
    }

    // last round is special
    u32_t tmp = m_kd[m_rounds][0];
```



```
// substitute byte, ShiftRows and AddRoundKey transformation
result[0] = sm_isbox[w1 >> 24] ^ (tmp >> 24 & 0xFF);
result[1] = sm_isbox[(w4 >> 16) & 0xFF] ^ (tmp >> 16 & 0xFF);
result[2] = sm_isbox[(w3 >> 8) & 0xFF] ^ (tmp >> 8 & 0xFF);
result[3] = sm_isbox[w2 & 0xFF] ^ tmp & 0xFF;

tmp = m_kd[m_rounds][1];
result[4] = sm_isbox[w2 >> 24] ^ (tmp >> 24 & 0xFF);
result[5] = sm_isbox[(w1 >> 16) & 0xFF] ^ (tmp >> 16 & 0xFF);
result[6] = sm_isbox[(w4 >> 8) & 0xFF] ^ (tmp >> 8 & 0xFF);
result[7] = sm_isbox[w3 & 0xFF] ^ tmp & 0xFF;

tmp = m_kd[m_rounds][2];
result[8] = sm_isbox[w3 >> 24] ^ (tmp >> 24 & 0xFF);
result[9] = sm_isbox[(w2 >> 16) & 0xFF] ^ (tmp >> 16 & 0xFF);
result[10] = sm_isbox[(w1 >> 8) & 0xFF] ^ (tmp >> 8 & 0xFF);
result[11] = sm_isbox[w4 & 0xFF] ^ tmp & 0xFF;

tmp = m_kd[m_rounds][3];
result[12] = sm_isbox[w4 >> 24] ^ (tmp >> 24 & 0xFF);
result[13] = sm_isbox[(w3 >> 16) & 0xFF] ^ (tmp >> 16 & 0xFF);
result[14] = sm_isbox[(w2 >> 8) & 0xFF] ^ (tmp >> 8 & 0xFF);
result[15] = sm_isbox[w1 & 0xFF] ^ (tmp & 0xFF);
}
```

## 5 AES 的加密模式

上面讲的是 AES 的核心加密算法，数据都是被处理好以后（填充或者其它进一步的处理）后分成组再进行块加密。根据对原数据填充方式、对原始数据和块加密数据的处理的不同，可以将 AES 分成以下几种模式：

| 模式            | 描述   | 典型应用                    |
|---------------|--|-------------------------|
| 电子密码本模式（ECB）  | 用相同的密钥分别对明文组加密   | 单个数据的安全传输（如一个加密密钥）      |
| 密码分组链接模式（CBC） | 加密算法的输入是上一个密文分组和下一个明文分组的异或                               | 普通目的的面向分组的传输认证          |
| 密码反馈模式（CFB）   | 一次处理 J 位，上一个分组密文作为产生一个伪随机数输出的加密算法的输入，该输出与明文异或，作为下一个分组的输入 | 普通目的的面向分组的传输认证          |
| 输出反馈模式（OFB）   | 与 CFB 基本相同，只是加密算法的输入是上一次 AES 的输出                         | 噪声通道上的数据流的传输（如卫星通信）     |
| 记数模式（CTR）     | 每个明文分组是与加密计数器的异或，对每个后续的组，计数器是累加的                         | 普通目的的面向分组的传输；<br>用于高速需求 |

注： 分组密码的工作模式就是以该分组密码算法为基础构造的各种密码系统。模式适用于所有的分组密码，包括 DES、AES 和 IDEA 等。

```
典型代码：
/**
 * Encrypt multiple blocks of plaintext.
 * @param in: The plaintext to encrypt.
 * @param result: The output ciphertext.
 * @param num: Number of bytes to encrypt, must be a multiple of the blocksize.
 * @param mode: Mode to use, CBC by default.
 */
void AesCipher::encrypt(const char *in, char *result, size_t num, int mode)
{
    assert(in != 0 && result != 0 && m_bKey);
    assert(num > 0 && num % BLOCK_SIZE == 0);
```

```

// in and result are not null pointers and round key was made
if (in == 0 || result == 0 || !m_bKey)
    return ;
//n should be > 0 and multiple of BLOCK_SIZE
if (num == 0 || num % BLOCK_SIZE != 0)
    return ;

// chain working mode
if (mode == CBC || mode == CFB || mode == OFB) {
    assert(m_bIv);
    if (!m_bIv)
        return ;
}

char iv[BLOCK_SIZE];
memcpy(iv, m_iv, BLOCK_SIZE);
char counter[BLOCK_SIZE];
memcpy(counter, m_counter, BLOCK_SIZE);
size_t n;
char tmpBuf[BLOCK_SIZE];
switch (mode)
{
case ECB:
    for (n = num / BLOCK_SIZE; n > 0; --n) {
        encryptBlock(in, result);           //块加密
        in += BLOCK_SIZE;                  //移动输入指针
        result += BLOCK_SIZE;              //移动输出指针
    }

    break;

//CBC 模式是软件应用中比较常见的模式
case CBC:
    for (n = num / BLOCK_SIZE; n > 0; --n) {
        xorBlock(iv, in);                  //现将明文与上一次的密文异或，第一次加密时使用初始化的向量 iv
        encryptBlock(iv, result);          //块加密
        memcpy(iv, result, BLOCK_SIZE);    //将结果赋予 iv
        in += BLOCK_SIZE;
        result += BLOCK_SIZE;
    }

    break;

case CFB:
    for (n = num / BLOCK_SIZE; n > 0; --n) {
        memcpy(tmpBuf, in, BLOCK_SIZE);    // processing the case when in == result
        encryptBlock(iv, result);
        xorBlock(result, tmpBuf);
        memcpy(iv, result, BLOCK_SIZE);
        in += BLOCK_SIZE;
        result += BLOCK_SIZE;
    }

    break;

case OFB:
    for (n = num / BLOCK_SIZE; n > 0; --n) {
        memcpy(tmpBuf, in, BLOCK_SIZE);    // processing the case when in == result
        encryptBlock(iv, result);
        memcpy(iv, result, BLOCK_SIZE);
        xorBlock(result, tmpBuf);
        in += BLOCK_SIZE;
        result += BLOCK_SIZE;
    }

    break;
}

```

```
case CTR:
    assert(m_bCounter);
    if (!m_bCounter)
        return;
    for (n = num / BLOCK_SIZE; n > 0; --n) {
        memcpy(tmpBuf, in, BLOCK_SIZE);    // processing the case when in == result
        encryptBlock(counter, result);
        xorBlock(result, tmpBuf);
        incrCounter(counter);
        in += BLOCK_SIZE;
        result += BLOCK_SIZE;
    }

    break;
}
```

AES 算法的编程实现在这里就算告一段落了，到了这里你应该对这个 AES 算法的原理和实现有了比较全面的掌握。

## 6 AES 算法的安全性

能够抵抗目前所有的已知攻击：

- a. 穷举攻击
- b. 差分攻击
- c. 线性攻击
- d. 一致攻击

## 参考资料

1. 《密码编码学与网络安全——原理与实践（第三版）》 PDF  
2. <http://bbs.chinaunix.net/thread-971809-1-1.html>  
3. <http://www.cnblogs.com/mingcn/archive/2011/11/25/1865447.html>  
4. [http://pg.zhku.edu.cn/inforwork/kejian/COURSE/ch04/3\\_3.htm](http://pg.zhku.edu.cn/inforwork/kejian/COURSE/ch04/3_3.htm)  
5. <http://hi.baidu.com/mallor/item/89dad214d1f1fe0dd1d66d33>  
6. <http://wenku.baidu.com/view/2f486221dd36a32d737581fd.html>  
7. [http://wenku.baidu.com/link?url=P7K0-lee5\\_jkh8VYXjZmRwmQejmybZKSZSwUXlufiIzEdZTZwD2Zpw49Qe0n0L4HX48Mw5l373SiA2zr85y3GQxiV5i6St4g6G-vUII\\_ESS](http://wenku.baidu.com/link?url=P7K0-lee5_jkh8VYXjZmRwmQejmybZKSZSwUXlufiIzEdZTZwD2Zpw49Qe0n0L4HX48Mw5l373SiA2zr85y3GQxiV5i6St4g6G-vUII_ESS)

附表：

|                    |             |             |             |             |              |             |             |             |
|--------------------|-------------|-------------|-------------|-------------|--------------|-------------|-------------|-------------|
| u32_t tel[256] = { | 0xc66363a5, | 0xf87c7c84, | 0xee777799, | 0xf67b7b8d, | 0xffff2f20d, | 0xd66b6bbd, | 0xde6f6fb1, | 0x91c5c554, |
|                    | 0x60303050, | 0x02010103, | 0xce6767a9, | 0x562b2b7d, | 0xe7fefef19, | 0xb5d7d762, | 0x4dababe6, | 0xec76769a, |
|                    | 0x8fcaca45, | 0x1f82829d, | 0x89c9c940, | 0xfa7d7d87, | 0xffffafa15, | 0xb25959eb, | 0x8e4747c9, | 0xfb0f000b, |
|                    | 0x41adadec, | 0xb3d4d467, | 0x5fa2a2fd, | 0x45afafea, | 0x239c9cbf,  | 0x53a4a4f7, | 0xe4727296, | 0x9bc0c05b, |
|                    | 0x75b7b7c2, | 0xelfdfd1c, | 0x3d9393ae, | 0x4c26266a, | 0x6c36365a,  | 0x7e3f3f41, | 0xf5f7f702, | 0x83cccc4f, |
|                    | 0x6834345c, | 0x51a5a5f4, | 0xd1e5e534, | 0xf9f1f108, | 0xe2717193,  | 0xabd8d873, | 0x62313153, | 0x2a15153f, |
|                    | 0x0804040c, | 0x95c7c752, | 0x46232365, | 0x9dc3c35e, | 0x30181828,  | 0x379696a1, | 0x0a05050f, | 0x2f9a9ab5, |
|                    | 0x0e070709, | 0x24121236, | 0x1b80809b, | 0xdfe2e23d, | 0xcdebeeb26, | 0x4e272769, | 0x7fb2b2cd, | 0xea75759f, |
|                    | 0x1209091b, | 0x1d83839e, | 0x582c2c74, | 0x341a1a2e, | 0x361b1b2d,  | 0xdc6e6eb2, | 0xb45a5aee, | 0x5ba0a0fb, |
|                    | 0xa45252f6, | 0x763b3b4d, | 0xb7d6d661, | 0x7db3b3ce, | 0x5229297b,  | 0xdde3e33e, | 0x5e2f2f71, | 0x13848497, |
|                    | 0xa65353f5, | 0xb9d1d168, | 0x00000000, | 0xc1eded2c, | 0x40202060,  | 0xe3fcfc1f, | 0x79b1b1c8, | 0xb65b5bed, |
|                    | 0xd46a6abe, | 0x8dcbc46,  | 0x67bebed9, | 0x7239394b, | 0x944a4ade,  | 0x984c4cd4, | 0xb05858e8, | 0x85cfcf4a, |
|                    | 0xbbd0d06b, | 0xc5efef2a, | 0x4faaaae5, | 0xedfbfb16, | 0x864343c5,  | 0x9a4d4dd7, | 0x66333355, | 0x11858594, |
|                    | 0x8a4545cf, | 0xe9f9f910, | 0x04020206, | 0xfe7f7f81, | 0xa05050f0,  | 0x783c3c44, | 0x259f9fba, | 0x4ba8a8e3, |
|                    | 0xa25151f3, | 0x5da3a3fe, | 0x804040c0, | 0x058f8f8a, | 0x3f9292ad,  | 0x219d9dbc, | 0x70383848, | 0xf1f5f504, |
|                    | 0x63bcbcdf, | 0x77b6b6c1, | 0xafdada75, | 0x42212163, | 0x20101030,  | 0xe5ffff1a, | 0xfdf3f30e, | 0xbfd2d26d, |
|                    | 0x81cdcd4c, | 0x180c0c14, | 0x26131335, | 0xc3ecec2f, | 0xbe5f5fe1,  | 0x359797a2, | 0x884444cc, | 0x2e171739, |
|                    | 0x93c4c457, | 0x55a7a7f2, | 0xfc7e7e82, | 0x7a3d3d47, | 0xc86464ac,  | 0xba5d5de7, | 0x3219192b, | 0xe6737395, |
|                    | 0xc06060a0, | 0x19818198, | 0x9e4f4fd1, | 0xa3dc7f,   | 0x44222266,  | 0x542a2a7e, | 0x3b9090ab, | 0x0b888883, |

```

0x8c4646ca, 0xc7eeee29, 0x6bb8b8d3, 0x2814143c, 0xa7dede79, 0xbc5e5ee2, 0x160b0b1d, 0xaddbdb76,
0xdbe0e03b, 0x64323256, 0x743a3a4e, 0x140a0a1e, 0x924949db, 0x0c06060a, 0x4824246c, 0xb85c5ce4,
0x9fc2c25d, 0xbdd3d36e, 0x43acacef, 0xc46262a6, 0x399191a8, 0x319595a4, 0xd3e4e437, 0xf279798b,
0xd5e7e732, 0x8bc8c843, 0x6e373759, 0xda6d6db7, 0x018d8d8c, 0xb1d5d564, 0x9c4e4ed2, 0x49a9a9e0,
0xd86c6cb4, 0xac5656fa, 0xf3f4f407, 0xcfeaea25, 0xca6565af, 0xf47a7a8e, 0x47aeae9, 0x10080818,
0x6fbabad5, 0xf0787888, 0x4a25256f, 0x5c2e2e72, 0x381c1c24, 0x57a6a6f1, 0x73b4b4c7, 0x97c6c651,
0xcbe8e823, 0xaldddd7c, 0xe874749c, 0x3e1f1f21, 0x964b4bdd, 0x61bdbddc, 0xd8b8b86, 0xf8a8a85,
0xe0707090, 0x7c3e3e42, 0x71b5b5c4, 0xcc6666aa, 0x904848d8, 0x06030305, 0xf7f6f601, 0x1c0e0e12,
0xc26161a3, 0x6a35355f, 0xae5757f9, 0x69b9b9d0, 0x17868691, 0x99c1c158, 0x3a1d1d27, 0x279e9eb9,
0xd9e1e138, 0xebf8f813, 0x2b9898b3, 0x22111133, 0xd26969bb, 0xa9d9d970, 0x078e8e89, 0x339494a7,
0x2d9b9bb6, 0x3c1e1e22, 0x15878792, 0xc9e9e920, 0x87cece49, 0xaa5555ff, 0x50282878, 0xa5dfdf7a,
0x038c8c8f, 0x59a1a1f8, 0x09898980, 0x1a0d0d17, 0x65bfbfda, 0xd7e6e631, 0x844242c6, 0xd06868b8,
0x824141c3, 0x299999b0, 0x5a2d2d77, 0x1e0f0f11, 0x7bb0b0cb, 0xa85454fc, 0x6dbbbbd6, 0x2c16163a,
};
u32_t te2[256] = {
0xa5c66363, 0x84f87c7c, 0x99ee7777, 0x8df67b7b, 0xdfff2f2, 0xbdd66b6b, 0xb1de6f6f, 0x5491c5c5,
0x50603030, 0x03020101, 0xa9ce6767, 0x7d562b2b, 0x19e7fefe, 0x62b5d7d7, 0xe64dabab, 0x9aec7676,
0x458fcaca, 0x9d1f8282, 0x4089c9c9, 0x87fa7d7d, 0x15effafa, 0xebb25959, 0xc98e4747, 0xbfbf0f0,
0xec41adad, 0x67b3d4d4, 0xfd5fa2a2, 0xea45afaf, 0xbf239c9c, 0xf753a4a4, 0x96e47272, 0x5b9bc0c0,
0xc275b7b7, 0x1ce1fdfd, 0xae3d9393, 0x6a4c2626, 0x5a6c3636, 0x417e3f3f, 0x02f5f7f7, 0x4f83cccc,
0x5c683434, 0xf451a5a5, 0x34d1e5e5, 0x08f9f1f1, 0x93e27171, 0x73abd8d8, 0x53623131, 0x3f2a1515,
0x0c080404, 0x5295c7c7, 0x65462323, 0x5e9dc3c3, 0x28301818, 0xa1379696, 0xf0a0505, 0xb52f9a9a,
0x090e0707, 0x36241212, 0x9b1b8080, 0x3ddfe2e2, 0x26cdebeb, 0x694e2727, 0xcd7fb2b2, 0x9fea7575,
0x1b120909, 0x9e1d8383, 0x74582c2c, 0x2e341a1a, 0x2d361b1b, 0xb2dc6e6e, 0xeeb45a5a, 0xfb5ba0a0,
0xf6a45252, 0x4d763b3b, 0x61b7d6d6, 0xce7db3b3, 0x7b522929, 0x3edde3e3, 0x715e2f2f, 0x97138484,
0xf5a65353, 0x68b9d1d1, 0x00000000, 0x2cc1eded, 0x60402020, 0x1fe3fcfc, 0xc879b1b1, 0xedb65b5b,
0xbed46a6a, 0x468dcbb, 0xd967bebe, 0x4b723939, 0xde944a4a, 0xd4984c4c, 0xe8b05858, 0xa85cfcf,
0x6bbbd0d0, 0x2ac5efef, 0xe54faaaa, 0x16edfbfb, 0xc5864343, 0xd79a4d4d, 0x55663333, 0x94118585,
0xcf8a4545, 0x10e9f9f9, 0x06040202, 0x81fe7f7f, 0xf0a05050, 0x44783c3c, 0xba259f9f, 0xe34ba8a8,
0xf3a25151, 0xfe5da3a3, 0xc0804040, 0x8a058f8f, 0xad3f9292, 0xbc219d9d, 0x48703838, 0x04f1f5f5,
0xdf63bcb, 0xc177b6b6, 0x75afdada, 0x63422121, 0x30201010, 0xae5ffff, 0xefdf3f3, 0x6dbfd2d2,
0x4c81cdcd, 0x14180c0c, 0x35261313, 0x2fc3ec, 0xe1be5f5f, 0xa2359797, 0xcc884444, 0x392e1717,
0x5793c4c4, 0xf255a7a7, 0x82fc7e7e, 0x477a3d3d, 0xacc86464, 0xe7ba5d5d, 0x2b321919, 0x95e67373,
0xa0c06060, 0x98198181, 0xd19e4f4f, 0x7fa3dc, 0x66442222, 0x7e542a2a, 0xab3b9090, 0x830b8888,
0xca8c4646, 0x29c7eeee, 0xd36bb8b8, 0x3c281414, 0x79a7dede, 0xe2bc5e5e, 0x1d160b0b, 0x76addbdb,
0x3dbe0e0, 0x56643232, 0x4e743a3a, 0x1e140a0a, 0xdb924949, 0xa0c0606, 0x6c482424, 0xe4b85c5c,
0x5d9fc2c2, 0x6ebdd3d3, 0xef43acac, 0xa6c46262, 0xa8399191, 0xa4319595, 0x37d3e4e4, 0x8bf27979,
0x32d5e7e7, 0x43bc8c8, 0x596e3737, 0xb7da6d6d, 0x8c018d8d, 0x64b1d5d5, 0xd29c4e4e, 0xe049a9a9,
0xb4d86c6c, 0xfaac5656, 0x07f3f4f4, 0x25cfeaea, 0xafca6565, 0xef47a7a, 0xe947aeae, 0x18100808,
0xd56fbaba, 0x88f07878, 0x6f4a2525, 0x725c2e2e, 0x24381c1c, 0xf157a6a6, 0xc773b4b4, 0x5197c6c6,
0x23cbe8e8, 0x7caldddd, 0x9ce87474, 0x213e1f1f, 0xdd964b4b, 0xdc61bdb, 0x860d8b8b, 0x850f8a8a,
0x90e07070, 0x427c3e3e, 0xc471b5b5, 0xaacc6666, 0xd8904848, 0x05060303, 0x01f7f6f6, 0x121c0e0e,
0xa3c26161, 0x5f6a3535, 0xf9ae5757, 0xd069b9b9, 0x91178686, 0x5899c1c1, 0x273a1d1d, 0xb9279e9e,
0x38d9e1e1, 0x13ebf8f8, 0xb32b9898, 0x33221111, 0xbbd26969, 0x70a9d9d9, 0x89078e8e, 0xa7339494,
0xb62d9b9b, 0x223c1e1e, 0x92158787, 0x20c9e9e9, 0x4987cece, 0xffaa5555, 0x78502828, 0x7aa5dfdf,
0x8f038c8c, 0xf859a1a1, 0x80098989, 0x171a0d0d, 0xda65bfbf, 0x31d7e6e6, 0xc6844242, 0xb8d06868,
0xc3824141, 0xb0299999, 0x775a2d2d, 0x111e0f0f, 0xcb7bb0b0, 0xfca85454, 0xd66dbbbb, 0x3a2c1616,
};
u32_t te3[256] = {
0x63a5c663, 0x7c84f87c, 0x7799ee77, 0x7b8df67b, 0xf20dff2, 0x6bbdd66b, 0x6fb1de6f, 0xc55491c5,
0x30506030, 0x01030201, 0x67a9ce67, 0x2b7d562b, 0xfe19e7fe, 0xd762b5d7, 0xab64dab, 0x769aec76,
0xca458fca, 0x829d1f82, 0xc94089c9, 0x7d87fa7d, 0xfa15effa, 0x59ebb259, 0x47c98e47, 0xf00bfbf0,
0xadec41ad, 0xd467b3d4, 0xa2fd5fa2, 0xafea45af, 0x9cbf239c, 0xa4f753a4, 0x7296e472, 0xc05b9bc0,
0xb7c275b7, 0xfd1ce1fd, 0x93ae3d93, 0x266a4c26, 0x365a6c36, 0x3f417e3f, 0xf702f5f7, 0xcc4f83cc,
0x345c6834, 0xa5f451a5, 0xe534d1e5, 0xf108f9f1, 0x7193e271, 0xd873abd8, 0x31536231, 0x153f2a15,
0x040c0804, 0xc75295c7, 0x23654623, 0xc35e9dc3, 0x18283018, 0x96a13796, 0x050f0a05, 0x9ab52f9a,
0x07090e07, 0x12362412, 0x809b1b80, 0xe23ddfe2, 0xeb26cdeb, 0x27694e27, 0xb2cd7fb2, 0x759fea75,
0x091b1209, 0x839e1d83, 0x2c74582c, 0x1a2e341a, 0x1b2d361b, 0x6eb2dc6e, 0x5aeeb45a, 0xa0fb5ba0,
0x52f6a452, 0x3b4d763b, 0xd661b7d6, 0xb3ce7db3, 0x297b5229, 0xe33edde3, 0x2f715e2f, 0x84971384,
0x53f5a653, 0xd168b9d1, 0x00000000, 0xed2cc1ed, 0x20604020, 0xfc1fe3fc, 0xb1c879b1, 0x5bedb65b,
0x6abed46a, 0xcb468dcb, 0xbed967be, 0x394b7239, 0x4ade944a, 0x4cd4984c, 0x58e8b058, 0xcf4a85cf,
0xd06bbbd0, 0xef2ac5ef, 0xaae54faa, 0xfb16edfb, 0x43c58643, 0x4dd79a4d, 0x33556633, 0x85941185,
0x45cf8a45, 0xf910e9f9, 0x02060402, 0x7f81fe7f, 0x50f0a050, 0x3c44783c, 0x9fba259f, 0xa8e34ba8,
0x51f3a251, 0xa3fe5da3, 0x40c08040, 0x8f8a058f, 0x92ad3f92, 0x9dbc219d, 0x38487038, 0xf504f1f5,
0xbcdf63bc, 0xb6c177b6, 0xda75afda, 0x21634221, 0x10302010, 0xff1ae5ff, 0xf30efdf3, 0xd26dbfd2,
0xcd4c81cd, 0x0c14180c, 0x13352613, 0xec2fc3ec, 0x5fe1be5f, 0x97a23597, 0x44cc8844, 0x17392e17,

```

```

0xc45793c4, 0xa7f255a7, 0x7e82fc7e, 0x3d477a3d, 0x64acc864, 0x5de7ba5d, 0x192b3219, 0x7395e673,
0x60a0c060, 0x81981981, 0x4fd19e4f, 0xdc7fa3dc, 0x22664422, 0x2a7e542a, 0x90ab3b90, 0x88830b88,
0x46ca8c46, 0xee29c7ee, 0xb8d36bb8, 0x143c2814, 0xde79a7de, 0x5ee2bc5e, 0x0b1d160b, 0xdb76addb,
0xe03bdbe0, 0x32566432, 0x3a4e743a, 0x0a1e140a, 0x49db9249, 0x060a0c06, 0x246c4824, 0x5ce4b85c,
0xc25d9fc2, 0xd36ebdd3, 0xacef43ac, 0x62a6c462, 0x91a83991, 0x95a43195, 0xe437d3e4, 0x798bf279,
0xe732d5e7, 0xc8438bc8, 0x37596e37, 0x6db7da6d, 0x8d8c018d, 0xd564b1d5, 0x4ed29c4e, 0xa9e049a9,
0x6cb4d86c, 0x56faac56, 0xf407f3f4, 0xea25cfea, 0x65afca65, 0x7a8ef47a, 0xae947ae, 0x08181008,
0xbad56fba, 0x7888f078, 0x256f4a25, 0x2e725c2e, 0x1c24381c, 0xa6f157a6, 0xb4c773b4, 0xc65197c6,
0xe823cbe8, 0xdd7ca1dd, 0x749ce874, 0x1f213elf, 0x4bdd964b, 0xbddc61bd, 0x8b860d8b, 0xa850f8a,
0x7090e070, 0x3e427c3e, 0xb5c471b5, 0x66aacc66, 0x48d89048, 0x03050603, 0xf601f7f6, 0x0e121c0e,
0x61a3c261, 0x355f6a35, 0x57f9ae57, 0xb9d069b9, 0x86911786, 0xc15899c1, 0x1d273a1d, 0x9eb9279e,
0xe138d9e1, 0xf813ebf8, 0x98b32b98, 0x11332211, 0x69bbd269, 0xd970a9d9, 0x8e89078e, 0x94a73394,
0x9bb62d9b, 0x1e223c1e, 0x87921587, 0xe920c9e9, 0xce4987ce, 0x55ffaa55, 0x28785028, 0xdf7aa5df,
0x8c8f038c, 0xa1f859a1, 0x89800989, 0x0d171a0d, 0xbfda65bf, 0xe631d7e6, 0x42c68442, 0x68b8d068,
0x41c38241, 0x99b02999, 0x2d775a2d, 0x0f111e0f, 0xb0cb7bb0, 0x54fca854, 0xbbd66dbb, 0x163a2c16,
};
u32_t te4[256] = {
0x6363a5c6, 0x7c7c84f8, 0x777799ee, 0x7b7b8df6, 0xf2f20dff, 0x6b6bbdd6, 0x6f6fb1de, 0xc5c55491,
0x30305060, 0x01010302, 0x6767a9ce, 0x2b2b7d56, 0xfefe19e7, 0xd7d762b5, 0xababe64d, 0x76769aec,
0xcaca458f, 0x82829d1f, 0xc9c94089, 0x7d7d87fa, 0xfafa15ef, 0x5959ebb2, 0x4747c98e, 0xf0f00bfb,
0xadadec41, 0xd4d467b3, 0xa2a2fd5f, 0xafafea45, 0x9c9cbf23, 0xa4a4f753, 0x727296e4, 0xc0c05b9b,
0xb7b7c275, 0xfdfd1ce1, 0x9393ae3d, 0x26266a4c, 0x36365a6c, 0x3f3f417e, 0xf7f702f5, 0xcccc4f83,
0x34345c68, 0xa5a5f451, 0xe5e534d1, 0xf1f108f9, 0x717193e2, 0xd8d873ab, 0x31315362, 0x15153f2a,
0x04040c08, 0xc7c75295, 0x23236546, 0xc3c35e9d, 0x18182830, 0x9696a137, 0x05050f0a, 0x9a9ab52f,
0x0707090e, 0x12123624, 0x80809b1b, 0xe2e23ddf, 0xebeb26cd, 0x2727694e, 0xb2b2cd7f, 0x75759fea,
0x09091b12, 0x83839e1d, 0x2c2c7458, 0x1a1a2e34, 0x1b1b2d36, 0x6e6eb2dc, 0x5a5aeeb4, 0xa0a0fb5b,
0x5252f6a4, 0x3b3b4d76, 0xd6d661b7, 0xb3b3ce7d, 0x29297b52, 0xe3e33edd, 0x2f2f715e, 0x84849713,
0x5353f5a6, 0xd1d168b9, 0x00000000, 0xeded2cc1, 0x20206040, 0xfcfc1fe3, 0xb1b1c879, 0x5b5bedb6,
0x6a6abed4, 0xcbcb468d, 0xbebed967, 0x39394b72, 0x4a4ade94, 0x4c4cd498, 0x5858e8b0, 0xcfcf4a85,
0xd0d06bbb, 0xefef2ac5, 0xaaaae54f, 0xfbf16ed, 0x4343c586, 0x4d4dd79a, 0x33335566, 0x85859411,
0x4545cf8a, 0xf9f910e9, 0x02020604, 0x7f7f81fe, 0x5050f0a0, 0x3c3c4478, 0x9f9fba25, 0xa8a8e34b,
0x5151f3a2, 0xa3a3fe5d, 0x4040c080, 0x8f8f8a05, 0x9292ad3f, 0x9d9dbc21, 0x38384870, 0xf5f504f1,
0xbcbcdf63, 0xb6b6c177, 0xdada75af, 0x21216342, 0x10103020, 0xffff1ae5, 0xf3f30efd, 0xd2d26dbf,
0xcdcd4c81, 0x0c0c1418, 0x13133526, 0xecec2fc3, 0x5f5fe1be, 0x9797a235, 0x4444cc88, 0x1717392e,
0xc4c45793, 0xa7a7f255, 0x7e7e82fc, 0x3d3d477a, 0x6464acc8, 0x5d5de7ba, 0x19192b32, 0x737395e6,
0x6060a0c0, 0x81819819, 0x4f4fd19e, 0xdcdc7fa3, 0x22226644, 0x2a2a7e54, 0x9090ab3b, 0x8888830b,
0x4646ca8c, 0xeeee29c7, 0xb8b8d36b, 0x14143c28, 0xdede79a7, 0x5e5ee2bc, 0x0b0b1d16, 0xdbdb76ad,
0xe0e03bdb, 0x32325664, 0x3a3a4e74, 0x0a0a1e14, 0x4949db92, 0x06060a0c, 0x24246c48, 0x5c5ce4b8,
0xc2c25d9f, 0xd3d36ebd, 0xacacef43, 0x6262a6c4, 0x9191a839, 0x9595a431, 0xe4e437d3, 0x79798bf2,
0xe7e732d5, 0xc8c8438b, 0x3737596e, 0x6d6db7da, 0x8d8d8c01, 0xd5d564b1, 0x4e4ed29c, 0xa9a9e049,
0x6c6cb4d8, 0x5656faac, 0xf4f407f3, 0xeaea25cf, 0x6565afca, 0x7a7a8ef4, 0xaeae947, 0x08081810,
0xbabad56f, 0x787888f0, 0x25256f4a, 0x2e2e725c, 0x1c1c2438, 0xa6a6f157, 0xb4b4c773, 0xc6c65197,
0xe8e823cb, 0xdddd7ca1, 0x74749ce8, 0x1f1f213e, 0x4b4bdd96, 0xbdbddc61, 0x8b8b860d, 0xa8a8a850f,
0x707090e0, 0x3e3e427c, 0xb5b5c471, 0x6666aacc, 0x4848d890, 0x03030506, 0xf6f601f7, 0x0e0e121c,
0x6161a3c2, 0x35355f6a, 0x5757f9ae, 0xb9b9d069, 0x86869117, 0xc1c15899, 0x1d1d273a, 0x9e9eb927,
0xe1e138d9, 0xf8f813eb, 0x9898b32b, 0x11113322, 0x6969bbd2, 0xd9d970a9, 0x8e8e8907, 0x9494a733,
0x9b9bb62d, 0x1e1e223c, 0x87879215, 0xe9e920c9, 0xcece4987, 0x5555ffaa, 0x28287850, 0xdfdf7aa5,
0x8c8c8f03, 0xa1a1f859, 0x89898009, 0x0d0d171a, 0xbfbfda65, 0xe6e631d7, 0x4242c684, 0x6868b8d0,
0x4141c382, 0x9999b029, 0x2d2d775a, 0x0f0f111e, 0xb0b0cb7b, 0x5454fca8, 0xbbbb66d, 0x16163a2c,
};

```