# RSA算法

## 算法描述

RSA算法是1978年由R. Rivest, A. Shamir和 L. Adleman提出的一种用数论构造的、也是迄今为止理论上最为成熟完善的公钥密码体制，该体制已得到广泛的应用。

１. 密钥的产生

(1) 选两个保密的大素数p和q；

(2) 计算  ， 

其中  是  的欧拉函数值；

1. 选一整数 e ，满足 ，且  ；
2. 计算d，满足；

即d是 e 在模下的乘法逆元，因 e 与互素，由模运算可知，它的乘法逆元一定存在；

1. 以 为公开钥,  为秘密钥。

2. 加密

加密时首先将明文比特串分组，使得每个分组对应的十进制数小于，即分组长度小于 。然后对每个明文分组 m，作加密运算：

3. 解密

对密文分组的解密运算为：

下面证明RSA算法中解密过程的正确性。

证明：由加密过程，可知  ，所以

下面分两种情况：

(1) m与n互素，则由Euler定理：

 ，，。

即 。

1.  ，我们先看 的含义，由于 , 所以 意味着 m 不是 p 的倍数也不是 q 的倍数。

因此 意味着 m 是 p 的倍数或 q 的倍数，不妨设 ，其中 t 为一正整数。

此时必有gcd(m,q)=1，否则m也是q倍数，从而是的倍数，与矛盾。

由gcd(m,q)=1及Euler定理得 ，所以

,［mkφ(q)］φ(p)≡1 mod q, mkφ(n)≡1 mod q

因此存在一整数 r ，使得 ，两边同乘以得，，即所以。

## RSA算法中的计算问题

1. 加、解密

RSA的加、解密过程都为求一个整数的整数次幂，再取模。如果按其含义直接计算，则中间结果非常大，有可能超出计算机所允许的整数取值范围。

如上例中解密运算，先求再取模，则中间结果就已远远超出了计算机允许的整数取值范围。而用模运算的性质：就可减小中间结果。再者，考虑如何提高加、解密运算中指数运算的有效性。

一般，求可如下进行，其中是正整数：将表示为二进制形式 ，即因此

可得以下快速指数算法：









其中 d 是中间结果， d 的终值即为所求结果。c 在这里的作用是表示指数的部分结果，其终值即为指数 m ，c 对计算结果无任何贡献，算法中完全可将之去掉

另外一种快速算法：

（1）初试化：a＝x，b＝r，c＝1

（2）若b＝0，则输出结果c，结束

（3）若bmod2不等于0，则转移到第（5）步

（4）若bmod2＝0，则b＝b/2，a＝a^2modn，转移到第（3）步

（5）b＝b－1，c＝（c×a）modn，转第（2）步

2. 密钥的产生

产生密钥时，需要考虑两个大素数p、q的选取，以及e的选取和d的计算。因为在体制中是公开的，因此为了防止敌手通过穷搜索发现p、q，这两个素数应是在一个足够大的整数集合中选取的大数。

如果选取p和q为左右的大素数，那么 n 的阶为 ，每个明文分组可以含有664位（），即83个8比特字节，这比DES的数据分组（8个8比特字节）大得多，这时就能看出RSA算法的优越性了。

因此如何有效地寻找大素数是第一个需要解决的问题。

寻找大素数时一般是先随机选取一个大的奇数（例如用伪随机数产生器），然后用素性检验算法检验这一奇数是否为素数，如果不是则选取另一大奇数，重复这一过程，直到找到素数为止。

可见寻找大素数是一个比较繁琐的工作。然而在RSA体制中，只有在产生新密钥时才需执行这一工作。

p和q决定出后，下一个需要解决的问题是如何选取满足和 的 e ，并计算满足的 d 。

这一问题可由推广的Euclid算法完成。

Miller-Rabin(n,t)  
　　输入：一个大于3的奇整数n和一个大于等于1的安全参 数k(用于确定测试轮数)。　　   
　　将n-1表示成2st，(其 中 t是奇数)  
　　1、 随机选取整数b， 2≤b ≤n-2　　   
　　2 计算r(0)= bt mod n  
　　3.1如果r(0)=1或者r(0) =n-1则通过检验，n可能为素数，回到1，继续选取另一个随机整数； 3.2 否则，若 r(0) ≠ 1并且r(0) ≠ n-1 ，则计算 r（1）= r（0）2 mod n；   
4.1 若r（1）=n-1，则通过检验，回到1；

4.2 否则，r（1） ≠n-1，计算 r（2）= r（1）2 mod n；如此 继续下去…

(s+2).1 若r（s-1）=n-1，则通过检验，回到1；

(s+2).2若r（s-1） ≠ n-1，则n为合数。

3.3 一种改进的RSA实现方法

利用中国剩余定理，可极大地提高解密运算的速度。

方法如下：解密方计算

由中国剩余定理解即得 m 。

已证明，如果不考虑中国剩余定理的计算代价，则改进后的解密运算速度是原解密运算速度的4倍。若考虑中国剩余定理的计算代价，改进后的解密运算速度分别是原解密运算速度的3.24倍(模为768比特时)、3.32倍(模为1024比特时)、3.47倍(模为2048比特时)。

4 RSA的安全性

RSA的安全性是基于分解大整数的困难性假定，之所以为假定是因为至今还未能证明分解大整数就是 NP 问题，也许有尚未发现的多项式时间分解算法。

如果RSA的模数n被成功地分解为，则立即获得，从而能够确定e 模的乘法逆元d，即，因此攻击成功。

随着人类计算能力的不断提高，原来被认为是不可能分解的大数已被成功分解。

例如RSA-129（即 为129位十进制数，大约428个比特）已在网络上通过分布式计算历时8个月于1994年4月被成功分解，RSA-130 已于1996年4月被成功分解，RSA-140 已于1999年2月被成功分解。

RSA-155（512比特） 已于1999年8月被成功分解，得到了两个78位（十进制）的素数。

对于大整数的威胁除了人类的计算能力外，还来自分解算法的进一步改进。

分解算法过去都采用二次筛法，如对RSA-129的分解。而对RSA-130的分解则采用了一个新算法，称为推广的数域筛法，该算法在分解RSA-130时所做的计算仅比分解RSA-129多10%，对RSA-140 和RSA-155的分解，也采用的是推广的数域筛法。将来也可能还有更好的分解算法，因此在使用RSA算法时对其密钥的选取要特别注意其大小。估计在未来一段比较长的时期，密钥长度介于1024比特至2048比特之间的RSA是安全的。

是否有不通过分解大整数的其它攻击途径？下面我们证明由直接确定等价于对 n的分解。

设中，，由 ，我们有

以及

由此可得，

所以，由p、q确定和由 确定p、q是等价的。

为保证算法的安全性，还对p和q提出以下要求：

1. 要大； 由 ，如果小，则也小，因此稍大于 n ， 稍大于 。
2. 可得 n 的如下分解法：

顺序检查大于的每一整数 x 直到找到一个x 使得是某一整数（记为 y ）的平方。

由 ，得 。

1. 和都应有大素因子；
2. 这是因为RSA算法存在着可能的重复加密攻击法。设攻击者截获密文c，可如下进行重复加密：

若，即 ，则有 即 ，所以在上述重复加密的倒数第2步就已恢复出明文 m,这种攻击法只有在 t较小时才是可行的。

为抵抗这种攻击，p、q 的选取应保证使t很大。

设m 在模n下阶为k，由得所以，即， 取为满足上式的最小值（ e为在模k下的阶）。

又当e与k互素时。为使t大，k就应大且应有大的素因子。

又由，所以为使k大，p-1和q-1都应有大的素因子。

此外，研究结果表明，如果且，则d能被容易地确定。

5 对RSA的攻击

RSA存在以下两种攻击，并不是因为算法本身存在缺陷，而是由于参数选择不当造成的。

1. 共模攻击

在实现RSA时，为方便，我们可能给每一用户相同的模数 n ，虽然加解密密钥不同，然而这样做是不行的。

设两个用户的公开钥分别为e1 和e2，且e1和e2互素（一般情况都成立），明文消息是 m ，密文分别是

敌手截获c1和c2 后，可如下恢复 m 。用推广的Euclid算法求出满足 得两个整数r和s ，其中一个为负，设为r。

再次用推广的Euclid算法求出，由此得。

2. 低指数攻击

假定将RSA算法同时用于多个用户(为讨论方便，以下假定3个)，然而每个用户的加密指数（即公开钥）都很小。

设三个用户的模数分别为，当时，，否则通过 有可能得出和的分解。

设明文消息是 m ，密文分别是 

由中国剩余定理可求出。由于，可直接由开立方根得到 。

## RSA的实现

在程序流程中已经列出了部分函数，这里对没有列出的部分重要函数做以补充：

大素数类中的各种函数

* 首先有三种构造函数，一种是默认的，一种是数字的，输入字符的较难，需要逐位进行比较，如下所示：

bigint(string wxn)  
 {  
 int i, j = 0;  
 int temp;  
 for (i = 0; i < zyl\_max; i++) {  
 num[i] = 0;  
 }  
 for (i = (wxn.length() - 1); i >= 0; i--) {  
 if (wxn[i] >= 'a' && wxn[i] <= 'f') {  
 temp = wxn[i] - 'a' + 10;  
 }  
 else {  
 if (wxn[i] >= 'A' && wxn[i] <= 'F') {  
 temp = wxn[i] - 'A' + 10;  
 }  
 else {  
 temp = wxn[i] - '0';  
 }  
 }  
 if (temp / 8) {  
 num[4 \* j + 3] = 1;  
 }  
 if ((temp % 8) / 4) {  
 num[4 \* j + 2] = 1;  
 }  
 if ((temp % 4) / 2) {  
 num[4 \* j + 1] = 1;  
 }  
 if (temp % 2) {  
 num[4 \* j] = 1;  
 }  
 j++;  
 }  
 flag = 0;  
 }

* 接下来是运算符的重载，在这里只列出加法，其余与之类似：

friend bigint operator+(bigint a, bigint b)  
 {  
 bigint result;  
 int i;  
 bool temp = 0;  
 if (a.flag == b.flag) {  
 for (i = 0; i < zyl\_max; i++) {  
 if (a[i] == 0 && b[i] == 0 && temp == 0) {  
 result.make(i, 0);  
 temp = 0;  
 }  
 else {  
 if ((a[i] == 1 && b[i] == 0 && temp == 0) ||  
 (a[i] == 0 && b[i] == 1 && temp == 0) ||  
 (a[i] == 0 && b[i] == 0 && temp == 1)) {  
 temp = 0;  
 result.make(i, 1);  
 }  
 else {  
 if ((a[i] == 1 && b[i] == 1 && temp == 0) ||  
 (a[i] == 0 && b[i] == 1 && temp == 1) ||  
 (a[i] == 1 && b[i] == 0 && temp == 1)) {  
 temp = 1;  
 result.make(i, 0);  
 }  
 else {  
 if (a[i] == 1 && b[i] == 1 && temp == 1) {  
 temp = 1;  
 result.make(i, 1);  
 }  
 }  
 }  
 }  
 }  
 result.flag = a.flag;  
 }  
 if (a.flag == 0 && b.flag == 1) {  
 b.Num\_Not();  
 return a - b;  
 }  
 if (a.flag == 1 && b.flag == 0) {  
 a.Num\_Not();  
 return b - a;  
 }  
 if (temp) {  
 cout << "Overflow" << endl;  
 }  
 return result;  
 }

* 然后是刚刚提到的利用欧几里得算法的求逆元函数和打印函数

bigint inv(bigint x)  
 {  
 bigint ZERO("0"), ONE("1");  
 bigint x1 = ONE, x2 = ZERO, x3 = x;  
 bigint y1 = ZERO, y2 = ONE, y3 = (\*this);  
 bigint t1, t2, t3;  
 if (y3 == ONE) {  
 return ONE;  
 }  
 bigint q;  
 bigint g;  
 do {  
 q = x3 / y3;  
 t1 = x1 - q \* y1;  
 t2 = x2 - q \* y2;  
 t3 = x3 - q \* y3;  
 x1 = y1;  
 x2 = y2;  
 x3 = y3;  
 y1 = t1;  
 y2 = t2;  
 y3 = t3;  
 } while (!(y3 == ONE));  
 g = y2;  
 if (!(g > ZERO))  
 g = x + g;  
 return g;  
 }  
  
 // 16进制的打印  
 void print()  
 {  
 if (this->flag == 1) {  
 cout << '-';  
 }  
 char result[zyl\_max];  
 int i;  
 for (i = zyl\_max - 1; i >= 0; i--) {  
 if ((\*this)[i] == 1) {  
 break;  
 }  
 }  
 i++;  
 int k;  
 int length = 0;  
 switch (i % 4) {  
 case 1:  
 length = i + 3;  
 break;  
 case 2:  
 length = i + 2;  
 break;  
 case 3:  
 length = i + 1;  
 break;  
 case 0:  
 length = i;  
 break;  
 }  
 for (k = 0; k < length; k = k + 4) {  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '0';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '1';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '2';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '3';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '4';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '5';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '6';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 0) {  
 result[k / 4] = '7';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 1) {  
 result[k / 4] = '8';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 1) {  
 result[k / 4] = '9';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'A';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 0 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'B';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'C';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 0 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'D';  
 }  
 if ((\*this)[k] == 0 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'E';  
 }  
 if ((\*this)[k] == 1 && (\*this)[k + 1] == 1 && (\*this)[k + 2] == 1 && (\*this)[k + 3] == 1) {  
 result[k / 4] = 'F';  
 }  
 }  
 if (i == 0) {  
 cout << '0' << endl;  
 }  
 else {  
 for (i = (k / 4) - 1; i >= 0; i--) {  
 cout << result[i];  
 }  
 cout << endl;  
 }  
 }

**判断素数和生成素数刚刚已经列出过了，这里就不再赘述**

接下来是RSA方面的各个函数

* 首先是其构造函数，计算公钥和私钥

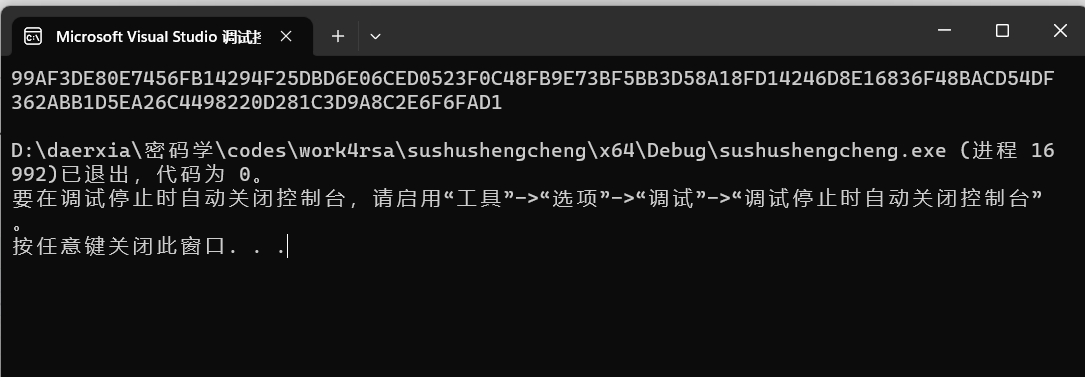
RSA(bigint a, bigint b)  
 {  
 bigint one("1");  
 p = a;  
 q = b;  
 n = p \* q;  
 f = (p - one) \* (q - one);  
 bigint curr("10001");  
 // 默认选取的公钥e为10001，和老师给的工具中相同  
 e = curr;  
 // d为私钥  
 d = e.inv(f);  
 }

* 接下来是加解密

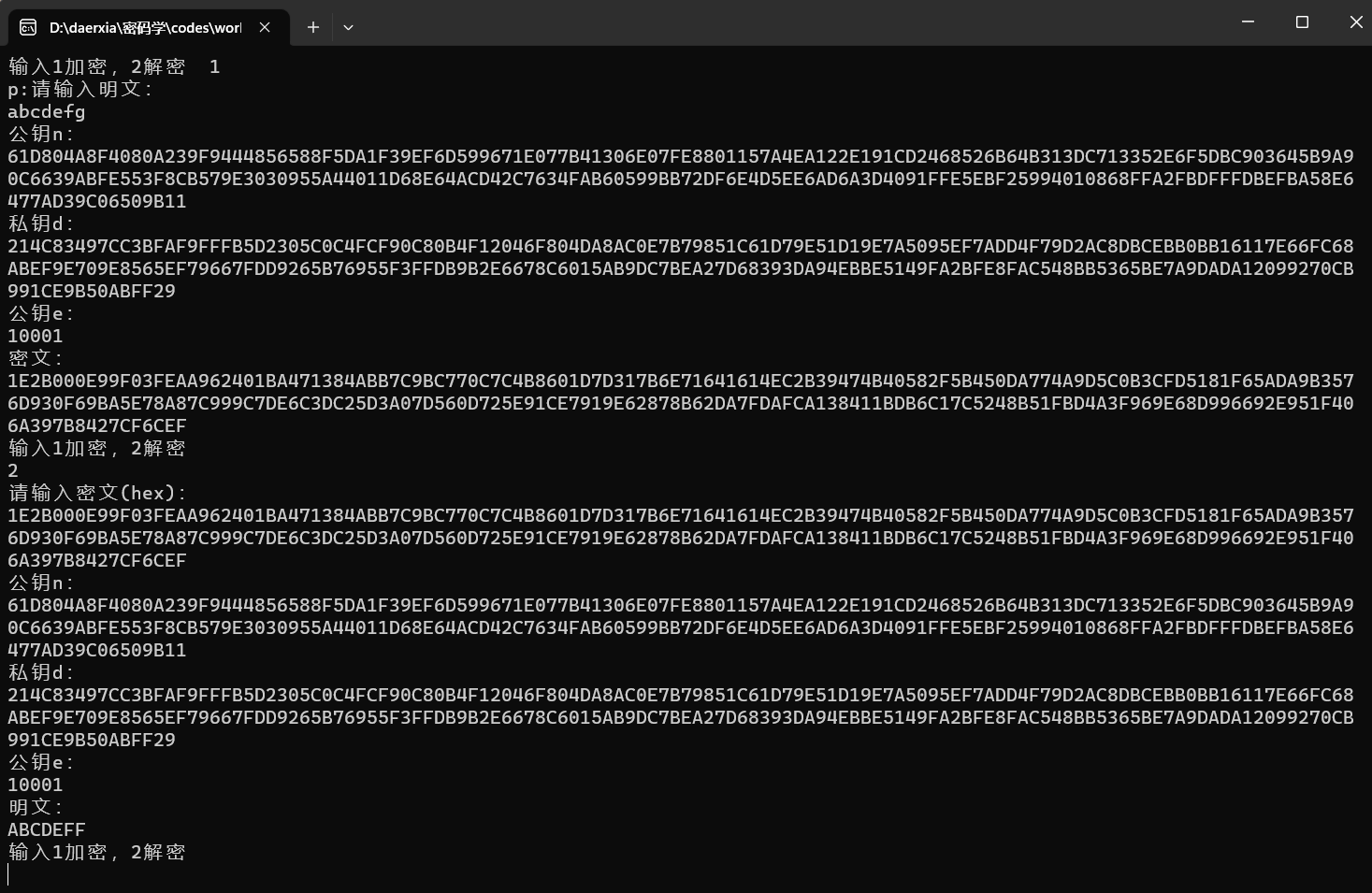
void encode(bigint m1)  
 {  
 m = m1;  
 c = m.expmod(e, n);  
 }  
 void decode(bigint c1)  
 {  
 c = c1;  
 m = c.expmod(d, n);  
 }

### 结果展示

* 首先是素数生成的结果：



* 然后是RSA加解密的结果：



# 素性检测

素性检验是指对给定的数检验其是否为素数。

1. 爱拉托斯散(Eratosthenes)筛法
2. 定理4-8 设n是一正整数，如果对所有满足的素数p，都有，那么n一定是素数。

基于这个定理，有一个寻找素数的算法，称为爱拉托斯散(Eratosthenes)筛法。

要找不大于n的所有素数，先将2到n之间的整数都列出，从中删除小于等于 的所有素数（设满足的素数有k个）的倍数，余下的整数就是所要求的所有素数

2.Miller-Rabin概率检测法

引理4-1 如果p为大于2的素数，则方程的解只有 和。

证明 由，有，因此或或 且。

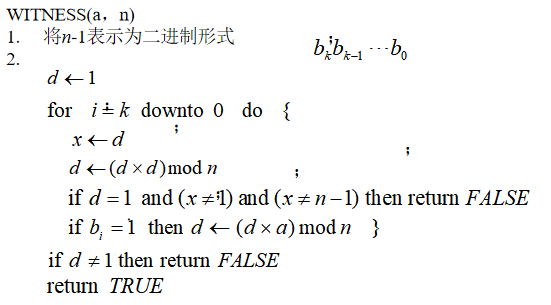
若且 ，则存在两个整数k和j，使得，两式相减得，为不可能结果。所以有或。

设，则，因此。类似地可得。

（引理4-1证毕）

引理4-1的逆否命题为：如果方程有一解，那么p不为素数。

下面介绍Miller-Rabin的素性概率检测法。其核心部分如下：



算法有两个输入，n 是待检验的数，a 是小于n 的整数。如果算法的返回值为 FALSE ，则 n 肯定不是素数，如果返回值为 TRUE，则 n 有可能是素数。

for 循环结束后，有，由 Fermat 定理知，若 n 为素数，则 d 为1。因此若，则 n 不为素数，所以返回 FALSE。

该算法有以下性质：对 s 个不同的 a ，重复调用这一算法，只要有一次算法返回为 FALSE ，就可肯定 n 不是素数。如果算法每次返回都为 TRUE ，则 n 是素数的概率至少为，因此对于足够大的 s ，就可以非常肯定地相信 n 为素数。

#pragma once

typedef long long ll;

ll Quick\_Multiply\_Mod(ll a, ll b, ll m)//快速积 ，a与b相乘，模数为m

{

ll ans = 0, temp = a;

while (b)

{

if (b & 1)//判断b为奇数还是偶数，实质是判断b二进制的最后一位是1（奇）还是0（偶数）

{

ans = (ans + temp) % m;

}

temp = (temp + temp) % m;

b >>= 1;//b向右移动一位，相当于b/2

}

return ans;

}

ll Quick\_Power\_Mod(ll a, ll b, ll m)//快速幂，a为底数，b为指数，m为模数

{

ll ans = 1, temp = a;

while (b)

{

if (b&1)

{

//ans = (ans \* temp) % m;

ans = Quick\_Multiply\_Mod(ans, temp, m);

}

//temp = (temp \* temp) % m;

temp = Quick\_Multiply\_Mod(temp, temp, m);

b >>= 1;

}

return ans;

}

bool Miller\_Rabin(ll n)//Miller-Rabing算法

{

if (n == 2)//2是素数

return true;

if (n < 2 || n % 2 == 0)//0，1和偶数不是素数

return false;

//把n-1写成2的k次方\*t的形式

int k=0, t=n-1;

while (!(t&1))//如果t不是奇数，就执行。相当于t%2

{

k++;

t >>= 1;//t向右移动一位，相当于t/2

}

//进行20轮测试，增加可靠性

for (int i = 0; i <=20; i++)

{

ll a = rand() % (n - 1) + 1;//选取底数a，1<=a<=n-1

ll b = Quick\_Power\_Mod(a, t, n);

ll y;

for (int i = 0; i < k; i++)

{

y = Quick\_Multiply\_Mod(b, b, n);

if (y == 1 && b != 1 && b != n - 1)

return false;

b = y;

}

if (y != 1)

return false;

}

return true;

}

3. AKS算法

2002年，印度数学家 Manindra Agrawal, Neeraj Kayal, Nitin Saxena 给出了一个确定性的素数判别算法，简称AKS算法。

设 N 和 I 分别是自然数集合和整数集合，且，满足 的最小正整数 k 称为模 n 下 a 的阶，记为 。

算法基于以下引理：

引理4-2

设是一自然数，，则 n 是素数的充要条件是：

证明 对，中的系数为。如果 n 是素数，则，所以的系数都为0。

如果 n 是合数，可设 q 是它的一个素数因子且，则不能除尽，而且和 互素，所以在模 n 下， 的系数不为0 ， 。

（引理4-2证毕）

引理4-2给出了一个素数检验的简单方法，然而要验证等式  是否成立，需计算 n 个系数。

为了减少系数的计算，可在等式的两边同时对一个形如的多项式取模（其中 r 是一个适当选择的小整数），即将判断等式是否成立，改为判断 是否成立。

表示在环上， 。

算法如下：

输入整数 n ，

1. 如果 ，输出“合数”；

2. 求满足的最小的 r ；

3. 如果存在 a ，满足且

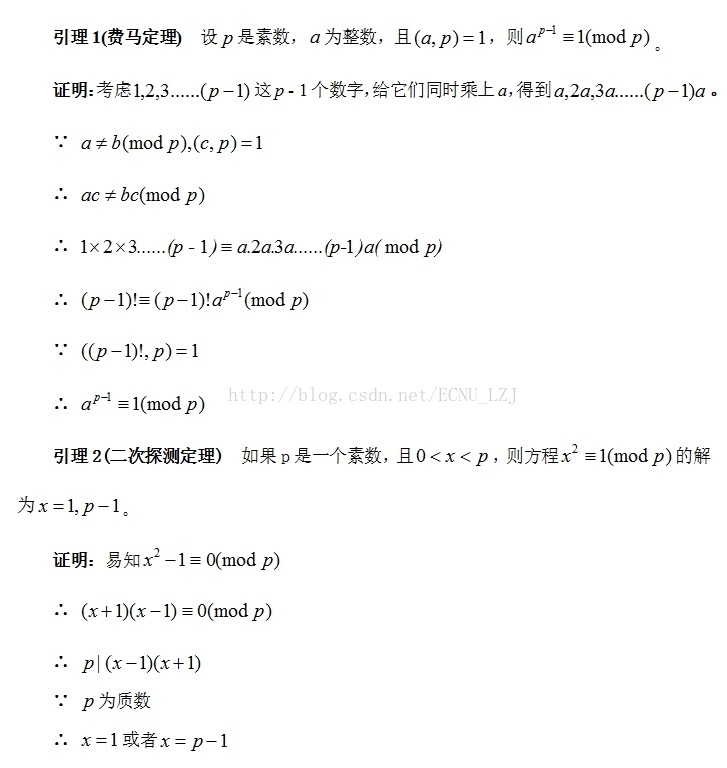
，输出“合数”；

4. 如果 ，输出“素数”；

5. 如果，输出“合数”；

6. 输出“素数”。

### **费马定理与二次探测定理**



# AES算法

## 出现的背景

在AES出现之前，DES一直是美国数据加密的标准算法,。它的密钥长度为64位,但事实上，只相当于56位参与运算。

在1999年1月的一次评审上。由于计算机计算能力的提高,DES密钥过短的问题成为了安全隐患。RSA数据安全公司宣布:该公司所发起的对56位DES的攻击已经由一个称为电子边境基金(EFF)的组织,通过互联网上的100000台计算机合作在22小时15分钟内完成。

在这种情况下,对于替代DES的要求日益增多。最终,NIST于1997年发布公告,征集新的数据加密标准作为联邦信息处理标准以代替DES。新的数据加密标准称为AES。

密码学中的高级加密标准(AdvancedEncryptionStandard,AES),又称Rijndael加密算法,由比利时密码学家JoanDaemen和VincentRijmen所设计，已经被多方分析且广为全世界所使用，并在2002年5月26日成为有效的标准。在2006年,AES成为对称密钥加密中最流行的算法之一。

AES有一个固定的128位的块大小和128,192或256位大小的密钥大小。它在软件及硬件上都能快速地加解密,相对来说较易于操作,且只需要很少的存储空间，也因此被广泛使用。

的数学基础

有限域GF(28)

有限域中的元素可以用多种不同的方式表示。对于任意素数的方幂，都有惟一的一个有限域，因此GF( )的所有表示是同构的，但不同的表示方法会影响到GF()上运算的复杂度，所以该算法采用传统的多项式表示法。

将构成的字节b看成系数在中的多项式GF (28)上还定义了一个运算，称之为x 乘法，其定义为：xּ b(x)= b7 x 8+b6x 7+b5 x 6+b4 x 5+b3 x 4+b2 x 3+b1 x 2+b0 x (mod m(x))

如果 = 0，求模结果不变，否则为乘积结果减去m(x)，即求乘积结果与m(x)的异或。由此得出x（十六进制‘02’）乘b(x)可以先对b(x)在字节内左移一位（最后一位补0），若= 1则再与‘1B’（其二进制为 00011011）做逐比特异或来实现，该运算记b=xtime(a)。在专用芯片中，xtime只需4个异或。x的幂乘运算可以重复应用xtime来实现。而任意常数乘法可以通过对中间结果相加实现。

4个字节构成的向量可以表示为系数在GF(28)上的次数小于4的多项式。多项式的加法就是对应系数相加；换句话说，多项式的加法就是4字节向量的逐比特异或。

规定多项式的乘法运算必须要取模x4+1, 这样使得次数小于4的多项式的乘积仍然是一个次数小于4的多项式，将多项式的模乘运算记为。a(x)= a3x 3+a2x 2+a1x+a0

b(x)= b3x 3+b2x 2+b1x+b0 

由于xj mod （x4+1）= x j mod4，所以c0=a0b0⊕a3b1⊕a2b2⊕a1b3；c1= a1b0⊕a0b1⊕a3b2⊕a2b3；c2= a2b0⊕a1b1⊕a0b2⊕a3b3；c3= a3b0⊕a2b1⊕a1b2⊕a0b3。

可将上述计算表示为

注意到M(x)不是GF(28)上的不可约多项式（甚至也不是GF(2)上的不可约多项式），因此非0多项式的这种乘法不是群运算。不过在Rijndael密码中，对多项式b(x)，这种乘法运算只限于乘一个固定的有逆元的多项式a(x)= a3x3+a2x2+a1x+a0。我们有如下的定理。

定理3-1 系数在GF(28)上的多项式a3x3+a2x2+a1x+a0是模x4+1可逆的，当且仅当矩阵

在GF(28)上可逆。

c(x) = xb(x)定义为x与b(x)的模x4+1乘法,即

其矩阵表示中，除a1= ‘01’ 外，其它所有ai= ‘00’，即

因此，x（或x的幂）模乘多项式相当于对字节构成的向量进行字节循环移位。

 的设计思想

Rijndael密码的设计力求满足以下3条标准：抵抗所有已知的攻击。

在多个平台上速度快，编码紧凑。设计简单。

当前的大多数分组密码，其轮函数是Feistel结构，即将中间状态的部分比特不加改变地简单放置到其它位置。Rijndael没有这种结构，其轮函数是由三个不同的可逆均匀变换组成的，称它们为三个“层”。所谓“均匀变换”是指状态的每个比特都是用类似的方法进行处理的。不同层的特定选择大部分是建立在“宽轨迹策略”的应用基础上的；简单地说，“宽轨迹策略”就是提供抗线性密码分析和差分密码分析能力的一种设计。

为实现宽轨迹策略，轮函数三个层中的每一层都有它自己的功能：

线性混合层：确保多轮之上的高度扩散。

非线性层：将具有最优的“最坏情况非线性特性”的S盒并行使用。

密钥加层：单轮子密钥简单地异或到中间状态上，实现一次性掩盖。

在第一轮之前，用了一个初始密钥加层，其目的是在不知道密钥的情况下，对最后一个密钥加层以后的任一层（或者是当进行已知明文攻击时，对第一个密钥加层以前的任一层）可简单地剥去，因此初始密钥加层对密码的安全性无任何意义。许多密码的设计中都在轮变换之前和之后用了密钥加层，如IDEA、SAFER和Blowfish。

为了使加密算法和解密算法在结构上更加接近，最后一轮的线性混合层与前面各轮的线性混合层不同，这类似于DES的最后一轮不做左右交换一样。可以证明这种设计不以任何方式提高或降低该密码的安全性。

首先，我们给出一个AES的总体描述。该算法执行过程如下：

1.给定一个明文x，将State初始化为x，并进行AddRoundKey操作，即将轮密钥RoundKey与State异或。

2.对前Nr-1轮中的每一轮，用S盒对State进行一次代换操作SubBytes；对State做一置换ShiftRows；再对State做一次操作MixColumns；然后进行AddRoundKey操作。

3.依次进行操作SubBytes，ShiftRows，AddRoundKey。

4.将State定义为密文y。其中AES中的所有操作都是面向字节的，State用字节矩阵来表示。

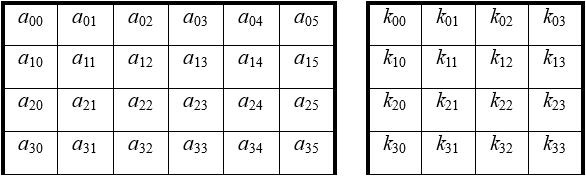
接下来我们详细展示每个过程。

状态、种子密钥和轮数

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

种子密钥：类似的，种子密钥用一个以字节为元素的矩阵阵列表示，该阵列有4行，列数记为Nk，Nk等于分组长度除以32。

轮数：轮数即迭代次数，轮数Nr依赖于密钥长度。如果密钥长度为128比特，则Nr=10；如果密钥长度为192比特，则Nr=12；如果密钥长度为256比特，则Nr=14.



有时可将这些分组当作一维数组，其每一元素是上述阵列表示中的4字节元素构成的列向量，数组唱的可为4、6、8，数组元素下标的范围分别是0~3、0~5和0~7。4字节元素构成的列向量有时也称为字。

算法的输入（包括最初的明文输入和中间过程的轮输入）都是以字节为单位按a00a10a20a30a01a11a21a31…的顺序放置到状态阵列中。同理，种子密钥以字节为单位按k00k10k20k30k01k11k21k31…的顺序放置到种子密钥阵列中。而输出（包括中间过程的轮输出和最后的密文输出），也是以字节为单位按相同的顺序从状态阵列中取出。若输入（或输出）分组中第n 个元素对应于状态阵列的第(i,j )位置上的元素，则n 和(i,j )有以下关系：

i=n mod4; j=; n=i+4j

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：

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。

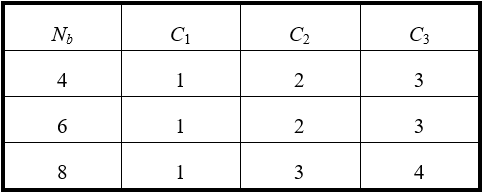
其次，对字节做如下的（GF(2)上的，可逆的）仿射变换：

上述S-盒对状态的所有字节所做的变换记为：ByteSub (State)



ByteSub的逆变换由代换表的逆表做字节代换，可通过如下两步实现，首先进行仿射变换的逆变换，再求每一字节在GF(28)上逆元。

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，第一行循环左移C1个字节，第二行循环左移C2个字节，第三行循环左移C3个字节。位移量C1、C2、C3的取值与Nb有关，由表3-10给出。



按指定的位移量对状态的行进行的行移位运算记为:ShiftRow (State)



ShiftRow的逆变换是对状态阵列的后三列分别以位移量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。

在列混合变换中，将状态阵列的每个列视为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16进制数表示）：c (x )='03'x 3+'01'x 2+'01'x+'02' c(x)是与x 4+1互素的，因此是模x 4+1可逆的。

列混合运算也可写为矩阵乘法。设b(x)= c(x)a(x)，可表示成下面的形式。





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

状态State与轮密钥RoundKey的密钥加运算表示为：AddRoundKey (State, RoundKey)



综上所述，组成Rijndael轮函数的计算部件简洁快速，功能互补。

轮函数的伪C代码如下：

Round (State, RoundKey)

{

ByteSub (State);

ShiftRow (State);

MixColumn (State);

AddRoundKey (State, RoundKey)

}

结尾轮的轮函数与前面各轮不同，将MixColumn这一步去掉。

FinalRound (State, RoundKey)

{

ByteSub (State);

ShiftRow (State);

AddRoundKey (State, RoundKey)

}

在以上伪C代码记法中，State、RoundKey 可用 指针类型，“函数” Round、FinalRound、ByteSub .ShiftRow、MixColumn、AddRoundKey都在指针State、RoundKey所指向的阵列上进行运算。

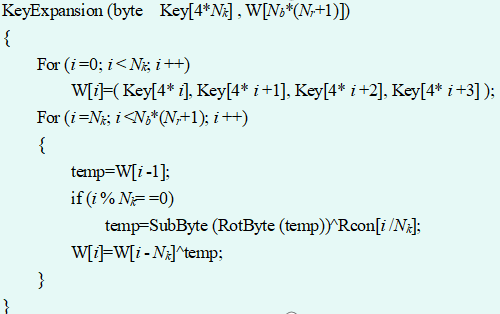
密钥编排是指从种子密钥得到轮密钥的过程，它由密钥扩展和轮密钥选取两部分组成。其基本原则如下：

轮密钥的比特数等于分组长度乘以轮数加1；

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。

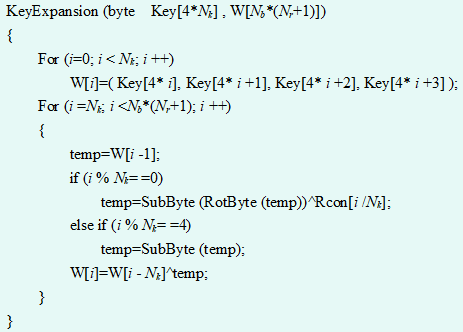
扩展密钥是以4字节字为元素的一维阵列，表示为W[Nb\* (Nr+1)]，其中前Nk个字取为种子密钥，以后每个字按递归方式定义。扩展算法根据Nk≤6和Nk>6有所不同。

当Nk≤6时，扩展算法为

其中Key[4\*Nk]为种子密钥，看作以字节为元素的一维阵列。函数SubByte ( )返回4字节字，其中每一个字节都是用Rijndael的S盒作用到输入字对应的字节得到。函数RotByte ( ) 也返回4字节字，该字由输入的字循环移位得到，即当输入字为(a, b, c, d)时，输出字为 (b, c, d, a)。

可以看出，扩展密钥的前Nk个字即为种子密钥，之后的每个字W[i]等于前一个字W[i-1]与Nk个位置之前的字W[i- Nk]的异或；不过当i/Nk为整数时，须先将前一个字W[i-1]经过以下一系列的变换： 1字节的循环移位RotByte→用S盒进行变换SubByte→异或轮常数Rcon[i/Nk]。

当Nk>6时，扩展算法为



Nk>6与Nk≤6的密钥扩展算法的区别在于：当i为Nk的4的倍数时，须先将前一个字W[i -1]经过SubByte变换。

以上两个算法中，Rcon[i/Nk] 为轮常数，其值与Nk无关，定义为（字节用16进制表示，同时理解为GF(28)上的元素）：

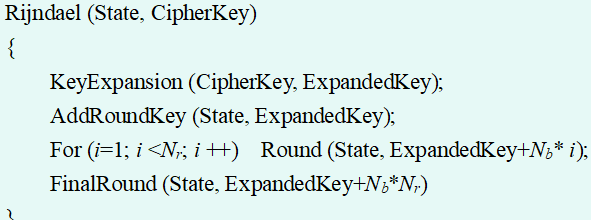
Rcon[i]=(RC[i], ‘00’, ‘00’, ‘00’)

其中RC[i] 是GF(28) 中值为xi-1的元素，因此

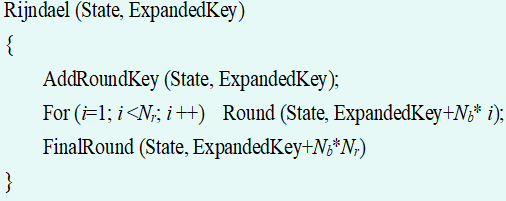
RC[1] =1（即‘01’） RC[i]= x（即‘02’）ּ RC[i-1]= xi-1

轮密钥i（即第i 个轮密钥）由轮密钥缓冲字W[Nb\* i]到 W[Nb\*(i+1)-1]给出，如图3-23所示。

加密算法为顺序完成以下操作：初始的密钥加；(Nr-1)轮迭代；一个结尾轮。即



其中CipherKey是种子密钥，ExpandedKey是扩展密钥。密钥扩展可以事先进行（预计算），且Rijndael密码的加密算法可以用这一扩展密钥来描述：



解密就是进行加密相关操作的逆操作

设字节代换（ByteSub）、行移位（ShiftRow）的逆变换分别为InvByteSub、InvShiftRow。则组合部件“ByteSub→ShiftRow”的逆变换为“InvByteSub→InvShiftRow”。

可证明组合部件“ByteSub→ShiftRow”的逆变换原本为“InvShiftRow→InvByteSub”。由于字节代换（ByteSub）是对每个字节进行相同的变换，故“InvShiftRow”与“InvByteSub”两个计算部件可以交换顺序。

设列混合（MixColumn）的逆变换为InvMixColumn。则列混合部件与密钥加部件（AddRoundKey）的组合部件 “MixColumn→AddRoundKey (·, Key)”的逆变换为 “InvMixColumn→AddRoundKey (·, InvKey)”其中密钥InvKey与Key的关系为：InvKey=InvMixColumn (Key)。

将某一轮的后两个计算部件和下一轮的前两个计算部件组成组合部件，该组合部件的程序为 MixColumn (State)；

AddRoundKey (State, Key(i));

ByteSub (State);

ShiftRow (State)

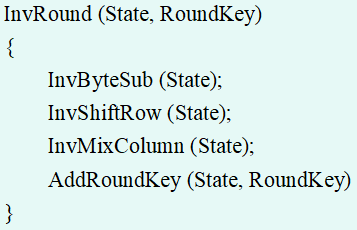
则该组合部件的逆变换程序为

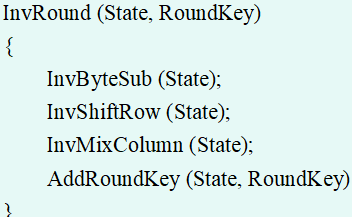
InvByteSub (State);

InvShiftRow (State);

InvMixColumn (State);

AddRoundKey (State, InvMixColumn (Key(i)))

Rijndael密码的解密算法为顺序完成以下操作：初始的密钥加；(Nr-1)轮迭代；一个结尾轮。其中解密算法的轮函数为：

解密算法的结尾轮为

## AES的应用

随着网络信息技术的迅猛发展，网络数据加密的要求也日益提高，AES的应用首先体现在网络信息安全领域中，以下四点就能很好的证明AES加密算法的广泛应用。

1. 无线 网络应用

由于无线网络的通信信道较有线网络更为开放，安全性的要求更高。目前，无线网络主要有两个标准 ：一是用于 WLAN的IEEE803．11协议 (Wi—Fi)；二是用于 WMAN的IEEE803．16协议 (WiMAXo这两个协议在制定初期所采用的安全机制分别为 RC4和DES，后来这两个协议也都将AES加入到协议的安全机制中。此外，为了保障数据传输安全性，其他的一些无线网络技术也都使用了AES。例如ZigBee技术，为确保 MAC帧的完整性、机密性、真实性和一致性，其MAC层使用AES算法进行加密，并且生成一系列的安全机制。ZigBee技术是一种近距离、低复杂度、低功耗、低数据速率，低成本的双向无线通信技术，主要适用于 自动控制和远程控制领域，可以嵌入到各种设备中。

1. 电子商务应用

在电子商务方面，主要是AES在电子商务基础平台中的密码协议和交易安全协议中的应用。例如，将AES应用在SSL(SecureSocketsLayw 安全套接层)协议中。在实施数据传输前，发送方通过身份认证后，用 SSL安全通道发送密钥到接收方的同时，使用AES算法对实时数据加密，然后基于UDP协议通过互联网发送加密的实时数据到接收方。这样接收方可以用接收到的AES密钥得到具体的实时数据。此外，还可以研究将AES与其他一些公钥加密算法 (非对称加密算法)相结合，设计出新的密码。目前比较典型的研究包括：AES与RSA相结合的混合加密体系；利用NTRU公钥密码体系分配AES密钥 ；AES与ECC(椭圆曲线加密算法)相结合的加密体系 ；AES在数据签名中的应用；AES在公钥加密体系PKI中的应用等等：

1. AES软件应用

在AES软件实现方面，其应用领域包含语音、视频信息的加密，数据库中的数据加密等。随着计算机对多媒体信息处理能力的增强，多媒体信息加密的问题 日渐凸显。由于多媒体信息的数据量很大，直接对其加密效率较低。所以，不仅要考虑数据加密算法AES的使用方法，还要设计相应的对多媒体信息进行加密的过程。关于AES在数据库方面的应用，主要在于如何在数据输入、输出中生成、分配和管理所用的密钥以及安全的数据加密策略。

1. AES硬件应用

在AES硬件实现方面，主要方向有射频 IC卡中的数据安全、智能安全卡和对硬盘数据的加密等方面。目前射频 Ic卡的应用范同很广 如公交 Ic卡、校园一卡通、门禁卡和新一代的居民身份证中都嵌入了IC芯片。其中所存储的数据通常都含有持人的私人信息，这些信息如果不经过加密处理，很可能泄露出去。因此，如何在射频 Ic卡中加入数据加密功能是AES硬件应用的一个研究方向。

## AES的实现

各类的结构体（即书本上出现的置换表以及各种扩展和置换运算）已经预先定义

各类预处理函数

* 首先是字符串，数字，以及bit流之间的各种转换，bit流的转换主要用于雪崩效应时可以检测出共有多少为不相同

// 将string类型转换为bits类型，方便后续的雪崩效应  
void getbit(string a, bitset<128>& temp)  
{  
 int num = 127;  
 for (int i = 2; i < a.length(); i++)  
 {  
 if (a[i] <= '9')  
 {  
 for (int j = 0; j < 4; j++)  
 {  
 temp[num--] = HexToBit[a[i] - 48][j];  
 }  
 }  
 else  
 {  
 for (int j = 0; j < 4; j++)  
 {  
 temp[num--] = HexToBit[a[i] - 65 + 10][j];  
 }  
 }  
 }  
}  
  
// 字符 --> 数字  
int ch\_to\_int(char& ch)  
{  
 int ans = 0;  
 // 数字的时候  
 if (ch >= 48 && ch <= 57)  
 {  
 ans = ch - '0';  
 }  
 // 16进制中a到f  
 else if (ch >= 'a' && ch <= 'f')  
 {  
 ans = ch - 'a' + 10;  
 }  
 // 16进制中A到F  
 else if (ch >= 'A' && ch <= 'F')  
 {  
 ans = ch - 'A' + 10;  
 }  
 return ans;  
}  
  
// 16进制字符串 --> 数字  
long long str\_long(string str)  
{  
 long long ans = 0;  
 // 遍历字符串，将字符串的内容变为16进制数字  
 for (char ch : str)  
 {  
 ans = ans \* 16 + ch\_to\_int(ch);  
 }  
 return ans;  
}  
  
  
  
// 数字 --> 16进制字符串 (只考虑小写)  
string int\_to\_chs(long long num)  
{  
 string ans = "";  
 while (num)  
 {  
 // 通过位运算得到低四位  
 int x = num & 0xf;  
 // 根据数值进行区分  
 if (x <= 9)  
 {  
 char ch = x + '0';  
 ans += ch;  
 }  
 else  
 {  
 char ch = x - 10 + 'a';  
 ans += ch;  
 }  
 // 移位，其实相当于 / 16  
 num >>= 4;  
 }  
 // 然后反转字符  
 int left = 0, right = ans.length() - 1;  
 // 双指针实现字符串反转  
 while (left < right)  
 {  
 char ch = ans[left];  
 ans[left] = ans[right];  
 ans[right] = ch;  
 left++;  
 right--;  
 }  
 return ans;  
}

* 然后是对字节循环的实现

string loop\_wordbyte(string& wi\_1)  
{  
 string ans = wi\_1.substr(2) + wi\_1.substr(0, 2);  
 return ans;  
}

* 接下来是对字节代换的实现

string wordbyte\_sub(string& wi\_1)  
{  
 int len = wi\_1.length();  
 string ans = "";  
 for (int i = 0; i < len; i += 2)  
 {  
 // 先获取当前的下标  
 int x = ch\_to\_int(wi\_1[i]), y = ch\_to\_int(wi\_1[i + 1]);  
 // 然后获取当前的数字  
 int num = S[x][y];  
 // 先将数值转化为字符串  
 string s = int\_to\_chs(num);  
 // 然后不足的话补0  
 while (s.length() < 2)  
 {  
 s = "0" + s;  
 }  
  
 // 加起来  
 ans += s;  
 }  
 return ans;  
}

* 然后是对密钥的轮常量异或的实现

string xor\_with\_const(string& wi\_1, int rounds)  
{  
 // 先将字符串变为数字  
 long long num = 0;  
 for (int i = 0; i < 8; ++i)  
 {  
 char ch = wi\_1[i];  
 num = num \* 16 + ch\_to\_int(ch);  
 }  
 // 计算异或结果  
 num ^= Rcon[rounds];  
  
 // 将num转化为字符串  
 string res = int\_to\_chs(num);  
 while (res.length() < 8)  
 {  
 res = "0" + res;  
 }  
 return res;  
}

* 密钥拓展的时候，下标为4的倍数时，需要使用一个特殊的变换函数

string T(string& wi\_1, int round)  
{  
 // T 变换由3部分构成，用的即为上述描述的三个函数  
 // 先进行字循环  
 string ans = loop\_wordbyte(wi\_1);  
 // 然后字节代换  
 ans = wordbyte\_sub(ans);  
 // 最后是轮异或  
 ans = xor\_with\_const(ans, round);  
  
 return ans;  
}

* 密钥编排的实现

vector<string> extend\_key(string& key)  
{  
 // 先分组  
 vector<string> w\_key = group\_key(key);  
 for (int i = 0; i < 40; ++i)  
 {  
 string w = "";  
 int index = 4 + i;  
 string temp = w\_key[index - 1];  
 // 4 的倍数的时候，需要调用T函数  
 if (index % 4 == 0)  
 {  
 temp = T(temp, index / 4 - 1);  
 }  
 w = string\_xor(temp, w\_key[index - 4]);  
  
 // 压入数组中  
 w\_key.push\_back(w);  
 } **return w\_key;**}

* 行移位函数的实现

vector<string> move\_row(vector<string>& s)  
{  
 vector<string> ans = s;  
 // 字符串数组每个对应一列, 所以是对应到列进行移位  
 // 一行对应有两个16进制数，所以需要两个一起移动，相当于两列一起移动  
 for (int i = 0; i < 4; ++i)  
 {  
 int k = i \* 2;  
 // 就原本矩阵对应的行移位，对于字符串数组就是列移位  
 for (int j = 0; j < 4; ++j)  
 {  
 ans[j][k] = s[(j + i) % 4][k];  
 ans[j][k + 1] = s[(j + i) % 4][k + 1];  
 }  
 }  
 return ans;  
}

* 列混淆函数的实现

vector<string> col\_confuse(vector<string>& s)  
{  
 vector<string> ans = s;  
 // 算法中对应的是列，这边就直接变成了行，即字符  
 for (int i = 0; i < 4; ++i)  
 {  
 // 需要先将字符串拆分成两两一组，共4组  
 auto temp = split\_s(s[i]);  
 // 先转成数字  
 int s0 = str\_long(temp[0]), s1 = str\_long(temp[1]), s2 = str\_long(temp[2]),  
 s3 = str\_long(temp[3]);  
 // 计算混淆后的值  
 int t0 = power(s0) ^ power(s1) ^ s1 ^ s2 ^ s3;  
 int t1 = s0 ^ power(s1) ^ power(s2) ^ s2 ^ s3;  
 int t2 = s0 ^ s1 ^ power(s2) ^ s3 ^ power(s3);  
 int t3 = s0 ^ power(s0) ^ s1 ^ s2 ^ power(s3);  
 // 转换成字符串再相加  
 ans[i] = int\_ch2(t0) + int\_ch2(t1) + int\_ch2(t2) + int\_ch2(t3);  
 }  
 return ans;  
}

* 行移位的逆操作函数的实现

vector<string> in\_move\_row(vector<string>& s)  
{  
 vector<string> ans = s;  
 // 现在变成了逆操作  
 for (int i = 0; i < 4; ++i)  
 {  
 int k = i \* 2;  
 // 就原本矩阵对应的行移位，对于字符串数组就是列移位  
 for (int j = 0; j < 4; ++j)  
 {  
 ans[j][k] = s[(j - i + 4) % 4][k];  
 ans[j][k + 1] = s[(j - i + 4) % 4][k + 1];  
 }  
 }  
 return ans;  
}

* 逆字节代换的实现

string in\_wordbyte\_sub(string& wi\_1)  
{  
 int len = wi\_1.length();  
 string ans = "";  
 for (int i = 0; i < len; i += 2)  
 {  
 // 先获取当前的下标  
 int x = ch\_to\_int(wi\_1[i]), y = ch\_to\_int(wi\_1[i + 1]);  
 // 然后获取当前的数字  
 int num = S1[x][y];  
 // 先将数值转化为字符串  
 string s = int\_to\_chs(num);  
 // 然后不足的话补0  
 while (s.length() < 2)  
 {  
 s = "0" + s;  
 }  
  
 // 加起来  
 ans += s;  
 }  
 return ans;  
}

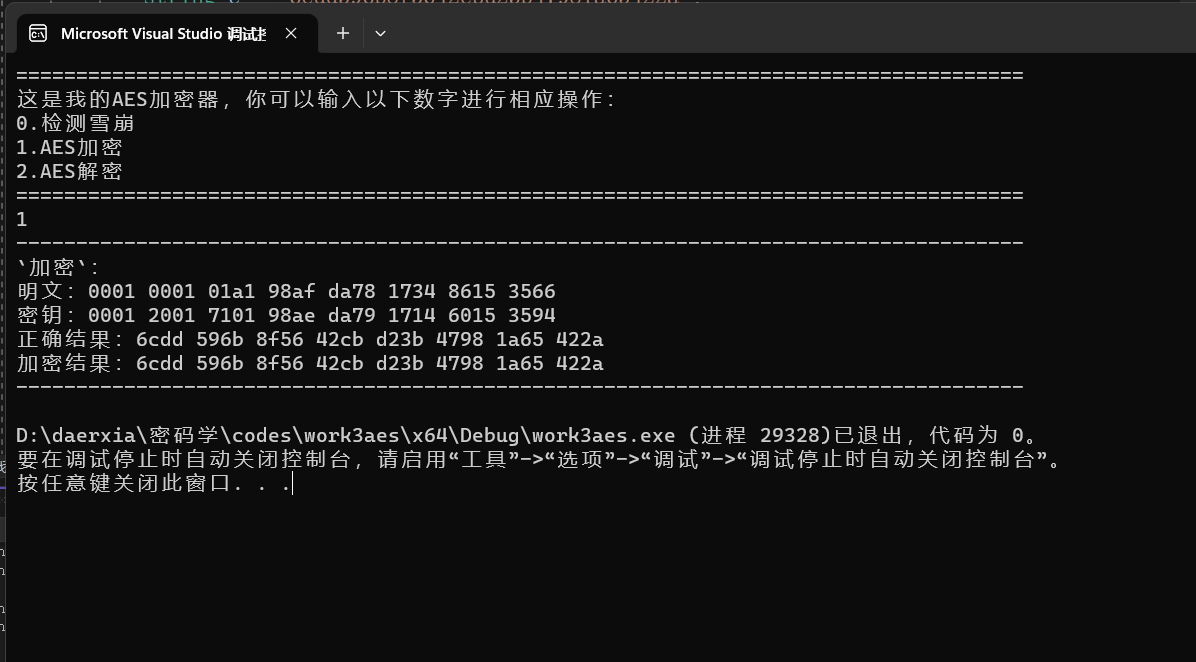
* 列混淆函数的逆变换实现

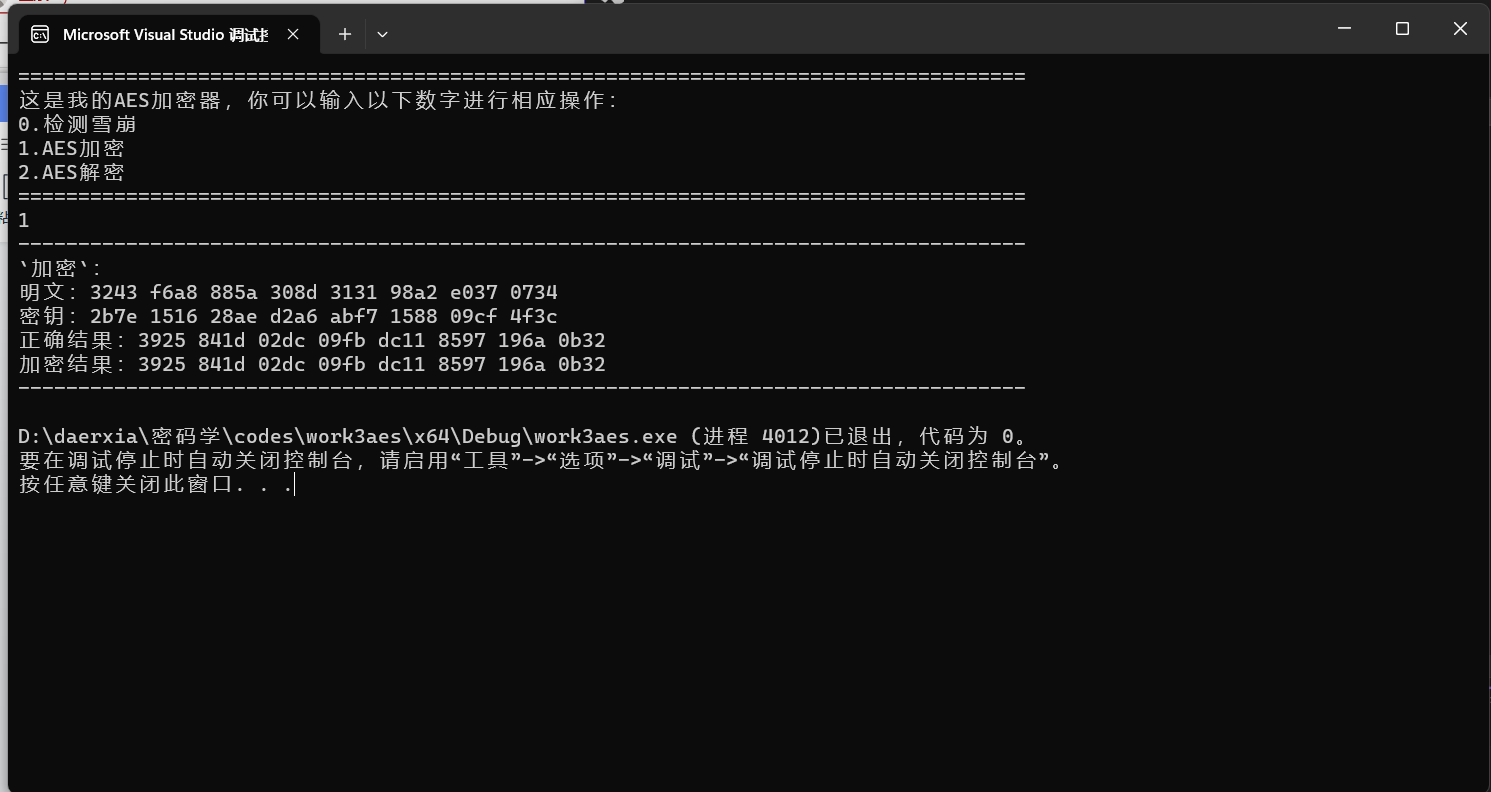
vector<string> in\_col\_confuse(vector<string>& s)  
{  
 // 逆变换其实原来的变换矩阵的逆矩阵，对应0xe, 0xb, 0xd, 0x9  
 // 4 列  
  
 vector<string> ans = s;  
 for (int i = 0; i < 4; ++i)  
 {  
 // 先分割成4个两位数字  
 auto temp = split\_s(s[i]);  
 // 转换成数字  
 vector<int> nums(4);  
 for (int j = 0; j < 4; ++j)  
 {  
 nums[j] = str\_long(temp[j]);  
 }  
 vector<int> t4(4, 0);  
 for (int j = 0; j < 4; ++j)  
 {  
 for (int t = 0; t < 4; ++t)  
 {  
 int k = (t - j + 4) % 4;  
 t4[j] ^= power(power(power(nums[t]))); // 表示8  
 switch (k)  
 {  
 case 0: // 0xe = 8 + 4 + 2  
 {  
 t4[j] ^= power(power(nums[t])) ^ power(nums[t]);  
 break;  
 }  
 case 1: // 0xb = 8 + 2 + 1  
 {  
 t4[j] ^= power(nums[t]) ^ nums[t];  
 break;  
 }  
 case 2: // 0xd = 8 + 4 + 1  
 {  
 t4[j] ^= power(power(nums[t])) ^ nums[t];  
 break;  
 }  
 default: // 0x9 = 8 + 1  
 t4[j] ^= nums[t];  
 break;  
 }  
  
 }  
 }  
 // 将数字转换成字符串存储  
 ans[i] = int\_ch2(t4[0]) + int\_ch2(t4[1]) + int\_ch2(t4[2]) + int\_ch2(t4[3]);  
 }  
  
 return ans;  
}

加密过程

* 按流程图进行实现

vector<string> aes(string& plain\_text, string& key)  
{  
 // 先拓展密钥  
 vector<string> keys = extend\_key(key);  
  
 int index = 0;  
 // 然后就是10轮迭代  
 // 需要知道明文其实是32位，所以需要搞4下  
 // 可以先把明文也分组  
  
 // 一开始的先进行一次轮密钥加  
 vector<string> texts = group\_key(plain\_text);  
 for (int i = 0; i < 4; ++i)  
 {  
 texts[i] = string\_xor(texts[i], keys[i]);  
 }  
 index += 4;  
   
 // 然后十次迭代  
 for (int k = 0; k < 10; ++k)  
 {  
 for (int j = 0; j < 4; ++j)  
 {  
 // 先是字节代换  
 texts[j] = wordbyte\_sub(texts[j]);  
 }  
 // 然后是行移位  
 texts = move\_row(texts);  
  
 if (k < 9)  
 {  
 // 再来列混淆  
 texts = col\_confuse(texts);  
 }  
  
 // 轮密钥加  
 for (int i = 0; i < 4; ++i)  
 {  
 texts[i] = string\_xor(texts[i], keys[i + index]);  
 }  
  
 index += 4;  
 }  
  
 // 最后进行输出  
// show(texts);  
 return texts;  
}

* 最后的加密输出结果如下：

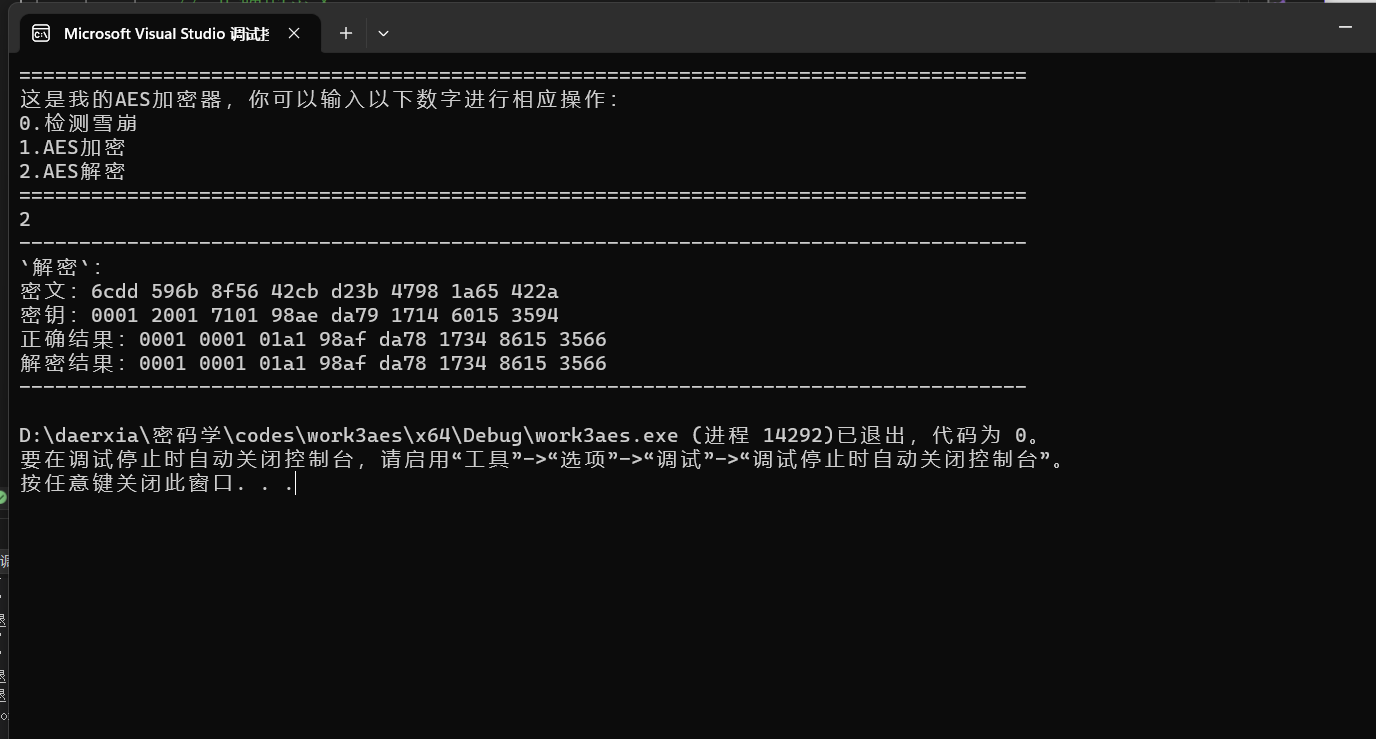


**可以看到加密结果与正确结果一样**

解密过程

* 按流程图进行实现

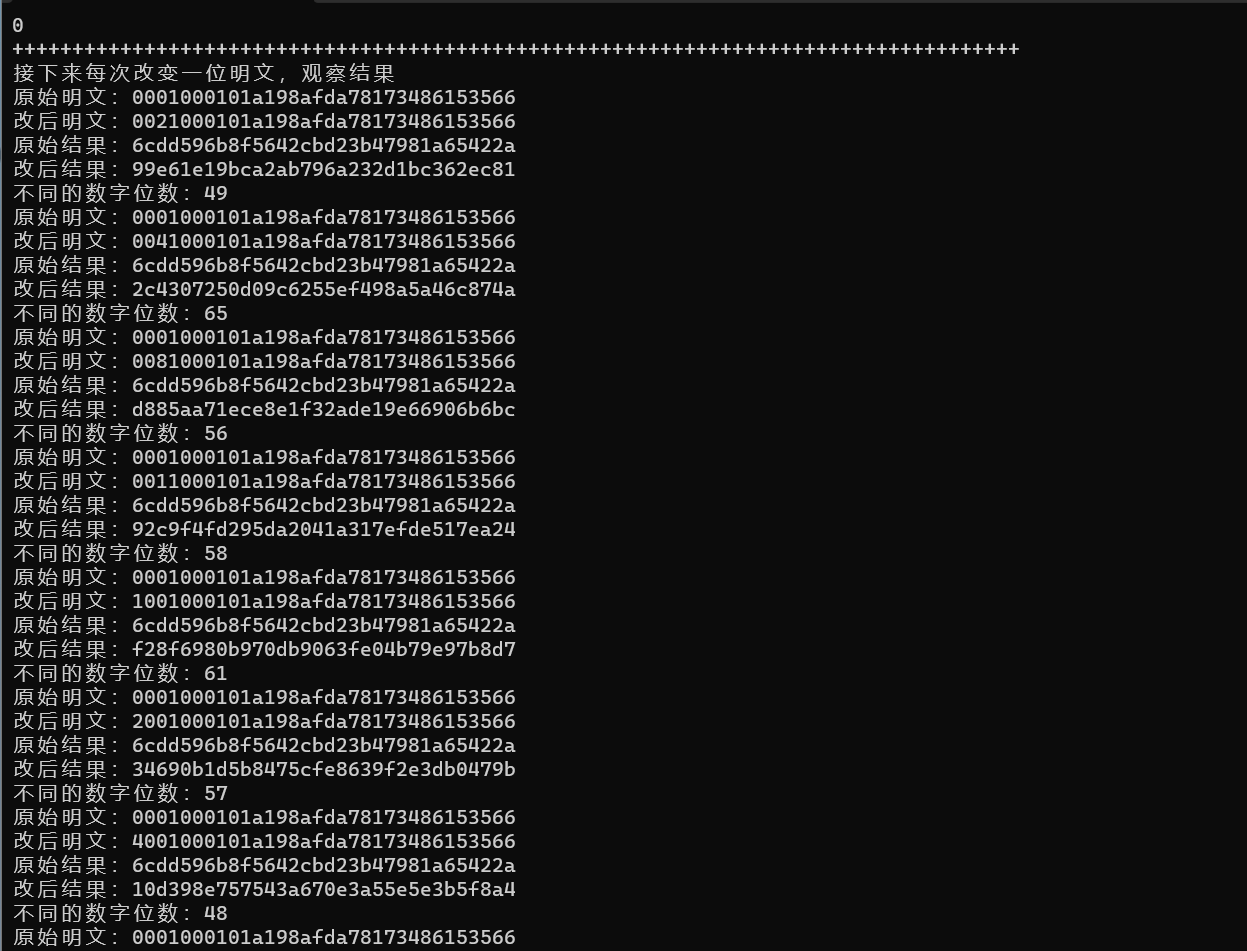
vector<string> in\_aes(string& text, string& key)  
{  
 // 先拓展密钥  
 auto keys = extend\_key(key);  
  
 // 初始下标  
 int index = 40;  
  
 // 对密文分组  
 vector<string> texts = group\_key(text);  
   
 // 一开始先进行依次轮密钥加  
 for (int i = 0; i < 4; ++i)  
 {  
 texts[i] = string\_xor(texts[i], keys[index + i]);  
 }  
 index -= 4;  
  
 // 然后十次迭代  
 for (int i = 0; i < 10; ++i)  
 {  
 // 先逆行移位  
 texts = in\_move\_row(texts);  
  
 // 然后是字节代换逆操作  
 for (int j = 0; j < 4; ++j)  
 {  
 texts[j] = in\_wordbyte\_sub(texts[j]);  
 }  
  
 // 轮密钥加  
 for (int j = 0; j < 4; ++j)  
 {  
 texts[j] = string\_xor(texts[j], keys[index + j]);  
 }  
  
 // 除了最后一轮，都要列混淆逆变换  
 if (i < 9)  
 {  
 texts = in\_col\_confuse(texts);  
 }  
 index -= 4;  
 }  
  
// show(texts);  
   
 return texts;  
}

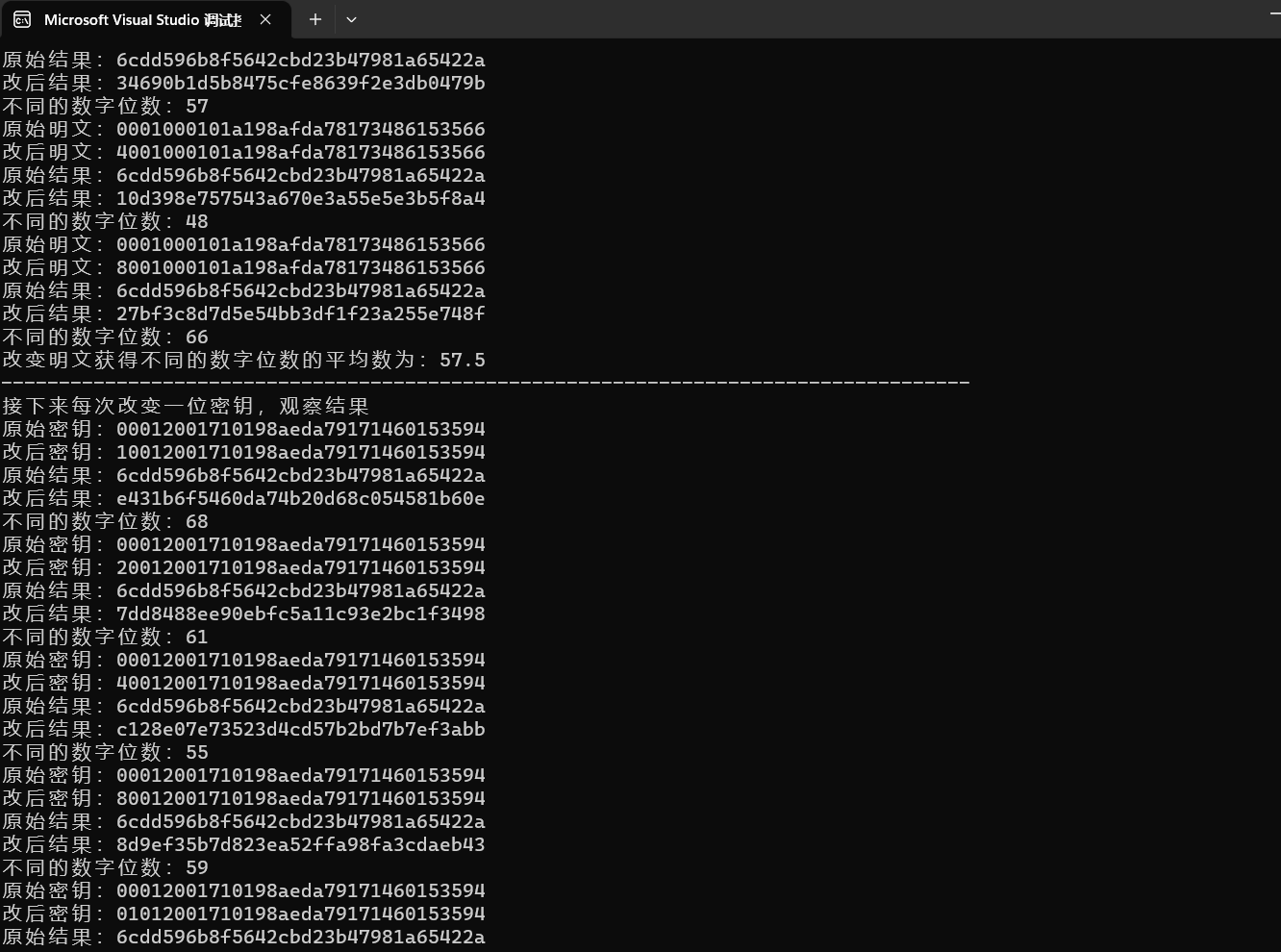
* 最后的解密输出结果如下：



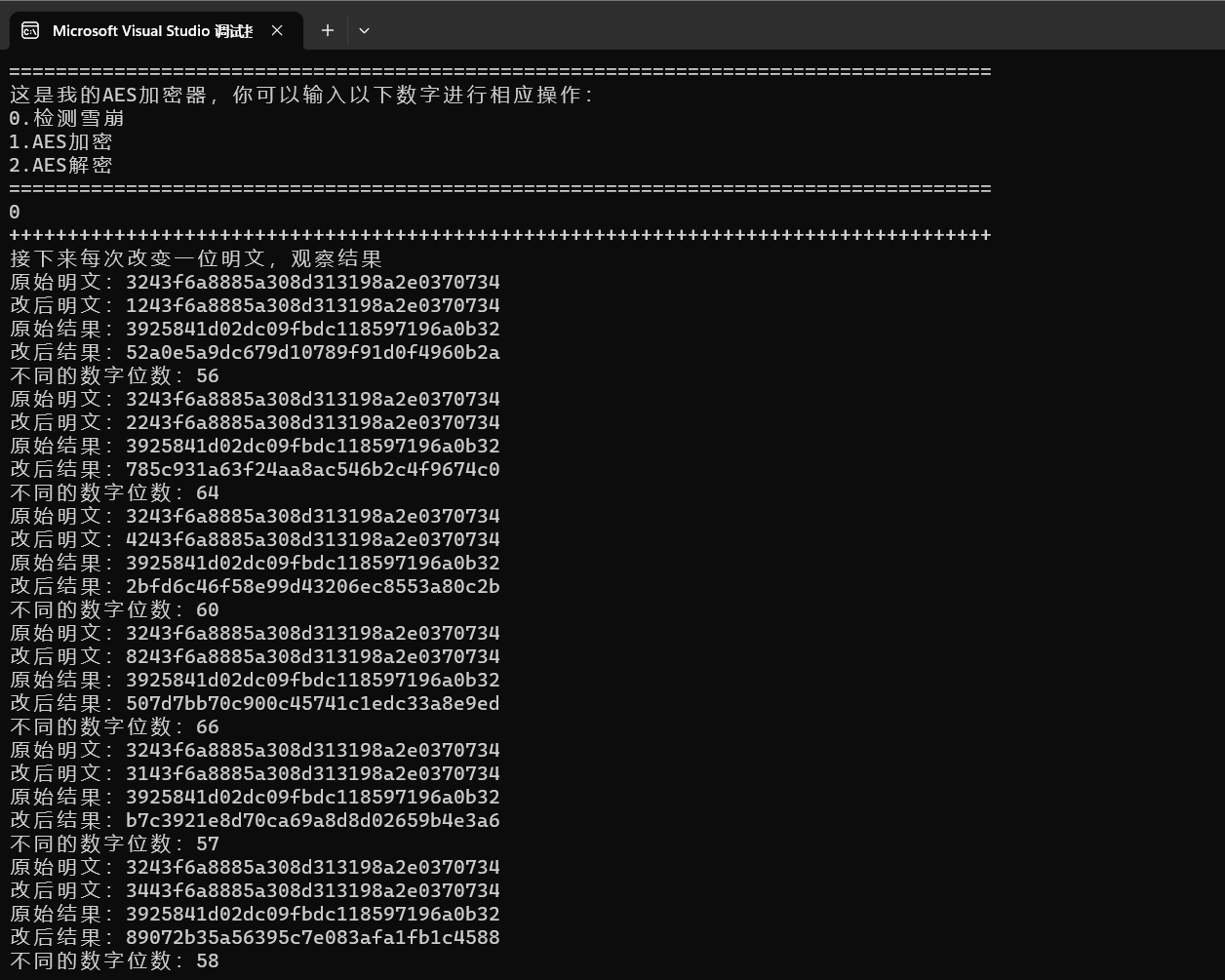
**可以看到解密结果与正确结果一样**

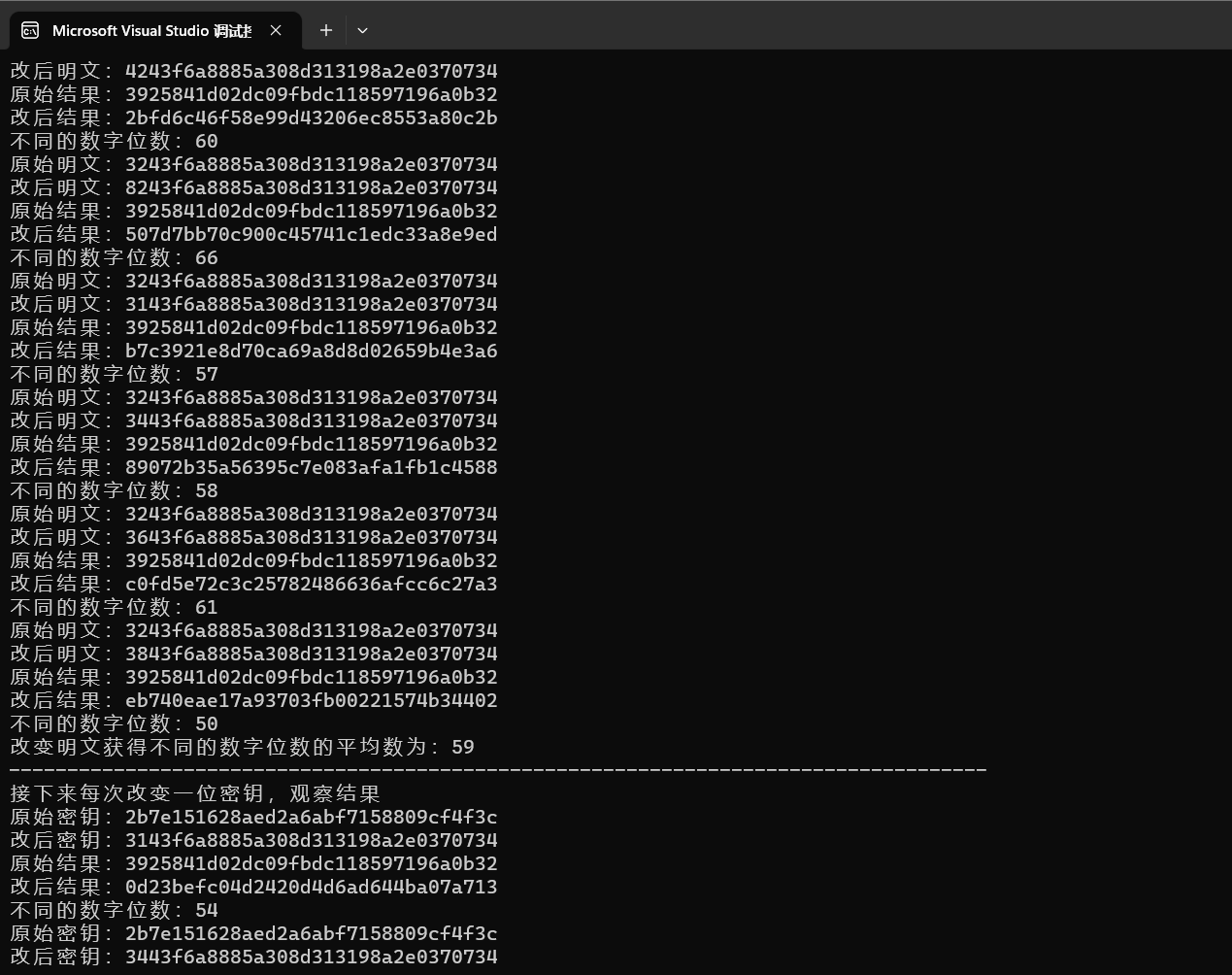
雪崩效应

* 在输入为0时进行雪崩效应
* 先固定密钥，每次改变一个明文，进行八次，观察更改的结果的不同位数
* 然后固定明文，每次改变一个密钥，进行八次，观察更改的结果的不同位数*最后的雪崩效应的输出结果如下：*
* 改变明文：
* 



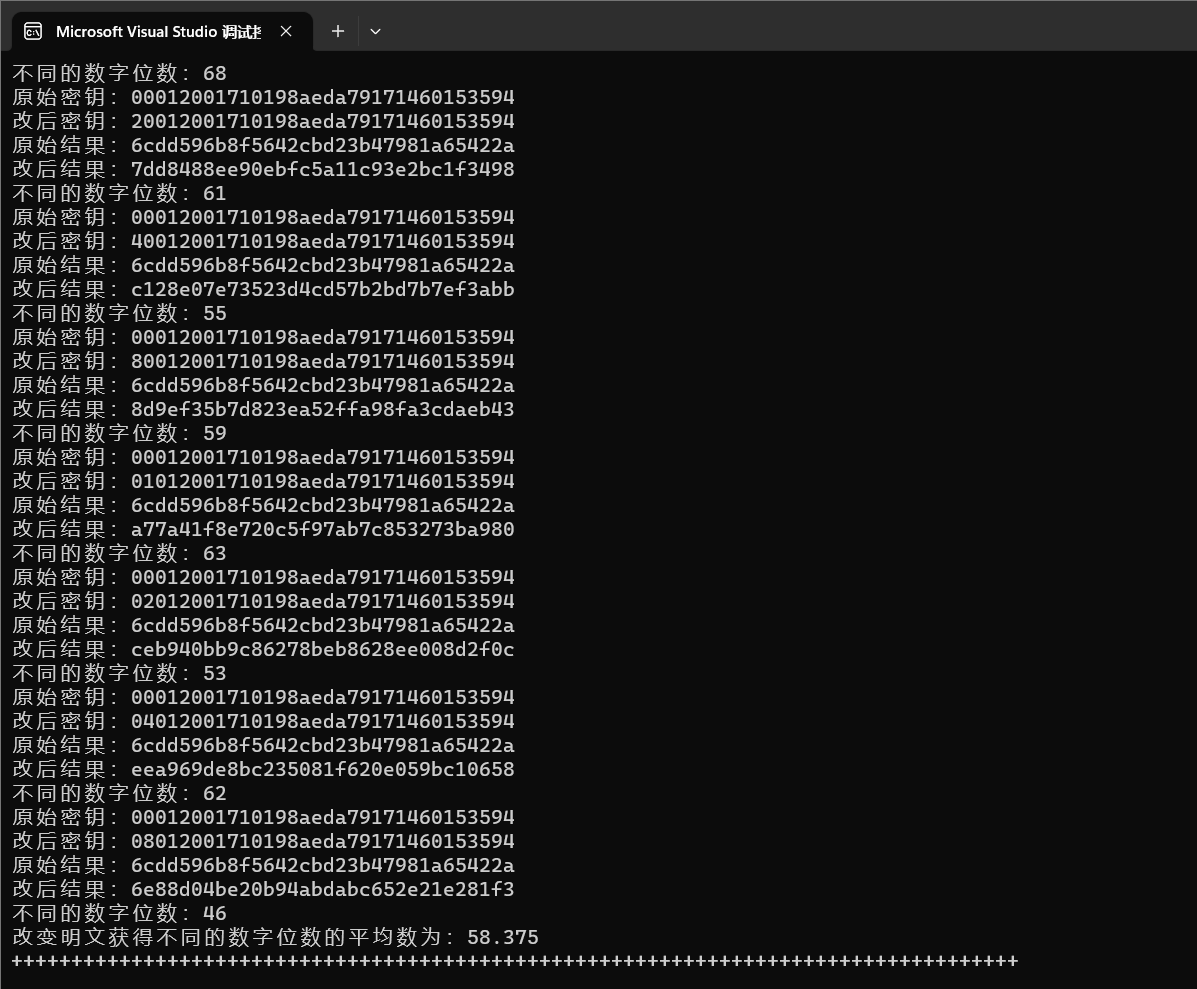
使秘钥不变，每次仅更改一位明文，重复八次，平均每次有57.5位发生变化





对于实验二的测试数据，使秘钥不变，每次仅更改一位明文，重复八次，平均每次有54位发生变化

改变密钥：

使明文不变，每次仅更改一位密钥，重复八次，平均每次有58.375位发生变化



对第二组数据，使明文不变，每次仅更改一位密钥，重复八次，平均每次有59.5位发生变化

***综上可以看出即使小小的改动，加密结果也会发生很大的变化***