南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

密码学实验

公钥密码算法 RSA

学号：2013018

姓名：许健

年级：2020 级

专业：信息安全

2022 年 12 月 29 日

目录

一、 实验目的

通过实际编程了解公钥密码算法 RSA 的加密和解密过程，加深对公钥密码算法的了解和使用。

二、 实验原理

序列密码和分组密码算法都要求通信双方通过交换密钥实现使用同一个密钥，这在密钥的管理、发布和安全性方面存在很多问题，而公钥密码算法解决了这个问题。

公钥密码算法是指一个加密系统的加密密钥和解密密钥是不同的，或者说不能用其中一个推导出另一个。在公钥密码算法的两个密钥中，一个是用于加密的密钥，它是可以公开的，称为公钥；另一个是用于解密的密钥，是保密的，称为私钥。公钥密码算法解决了对称密码体制中密钥管理的难题，并提供了对信息发送人的身份进行验证的手段，是现代密码学最重要的发明。

RSA 密码体制是目前为止最成功的公钥密码算法，它是在 1977 年由 Rivest、Shamir 和 Adleman 提出的第一个比较完善的公钥密码算法。它的安全性是建立在“大数分解和素性检测”这个数论难题的基础上，即将两个大素数相乘在计算上容易实现，而将该乘积分解为两个大素数因子的计算量相当大。虽然它的安全性还未能得到理论证明，但经过 40 多年的密码分析和攻击，迄今仍然被实践证明是安全的。

RSA 算法描述如下：

1. 公钥。选择两个不同的大素数 p 和 q，n 是二者的乘积，即 n = pq，使 $\varphi(n) = (p-1)(q-1)$。$\varphi(n)$ 为欧拉函数。随机选取正整数 e，使其满足 $(e, \varphi(n)) = 1$，即 e 和 $\varphi(n)$ 互素，则将（n，e）作为公钥。

2. 私钥。求出正数 d，使其满足 $e \times d \equiv 1 \bmod \varphi(n)$，则将（n，d）作为私钥。

3. 加密算法。对于明文 m，由 $c \equiv m^e \bmod n$，得到密文 c。

4. 解密算法。对于密文 c，由 $m \equiv c^d \bmod n$，得到明文 m。

如果攻击者获得了 n、e 和密文 c，为了破解密文必须计算出私钥 d，为此需要先分解 n。当 n 的长度为 1024 比特时，在目前还是安全的，但从因式分解技术的发展来看，1024 比特并不能保证长期的安全性。为了保证安全性，要求在一般的商业应用中使用 1024 比特的长度，在更高级别的使用场合，要求使用 2048 比特长度。

三、 RSA 算法实践

(一) 手工演算

为了加深对 RSA 算法的了解，根据已知参数：p = 3，q = 11，m = 2，手工计算公钥和私钥，并对明文 m 进行加密，然后对密文进行解密。

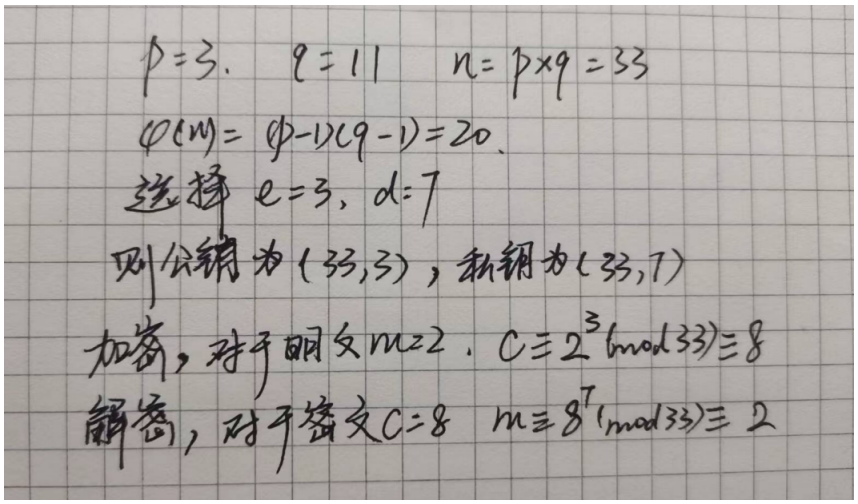

图 1: 手工计算

(二) 使用 RSATool 加密文字

使用 RSATool 之前需要初始化，我尝试使用该工具初始化 1024 位的 n，发现生成速度很快，程序运行效率很高。这里我们选择的 n 的位数为 256bits，使用的 e 为 0x10001，素数 P、Q 均已随机初始化，私钥 D 的值也已经被计算出来。接下来我们将用此 RSA 公钥进行加密，RSA 私钥进行解密。

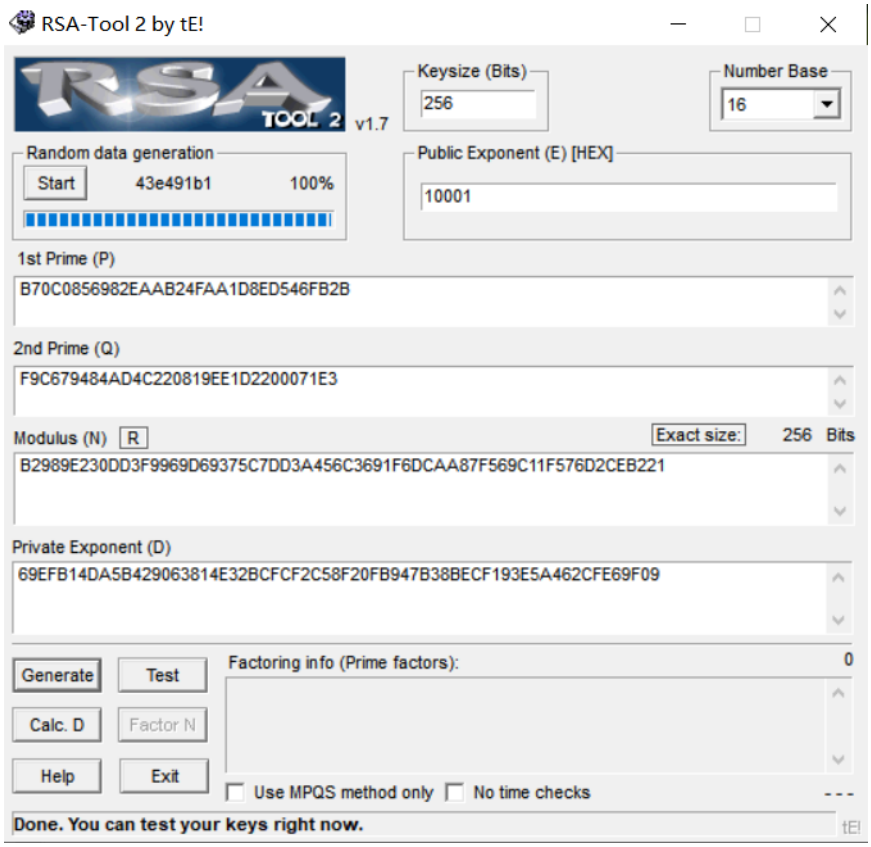

图 2: 初始化 RSATool

使用的加密文字为“测试 RSA 加密”，我们的工具将其加密为十六进制的数字串。再次执行解密操作，可以看到“测试 RSA 加密”这几个字被正确解密。

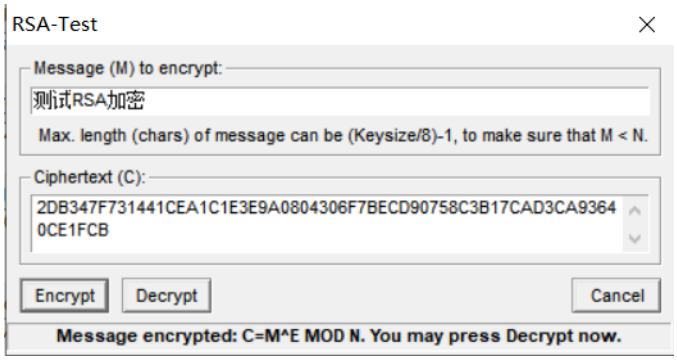

图 3: RSATool 加密文字

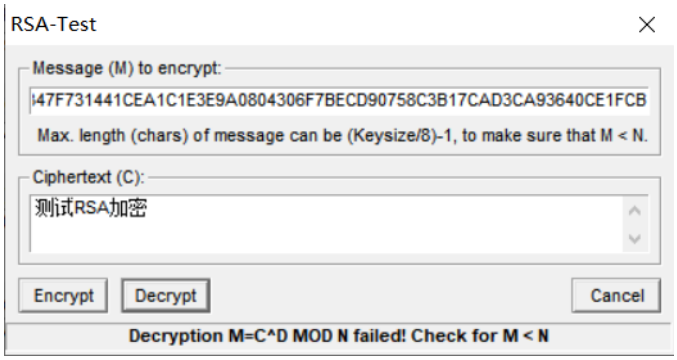

图 4: RSATool 解密文字

四、 大整数类 BigInt

由于 C++ 并没有对应的大整数类，为了实现 1024 位的 RSA 算法，我们需要自己实现一个大整数类。我将其命名为 BigInt，它支持任意位数的整数加、减、乘、绝对值除、取模等一系列基本操作。同时它支持 RSA 需要的大数幂模运算，以及扩展欧几里得算法求乘法逆元操作。

(一) 数据存储

BingInt 类采用 vector 向量存储大整数，vector 中每个元素的类型为 unsigned long，存储 32 位无符号整数。对于第 0 号元素，存储大整数的第 0 到 31 位，第 1 号元素存储大整数的第 32 到 63 位，以此类推。每个 BigInt 类实例，包含 bool 值 isNegative 用来表示正负。

同时在 BigInt 内部定义了一个结构体 bit，支持的操作包括 size（返回大整数的位数）、at（返回大整数第 i 位 bit），方便进行大整数的位运算。

数据存储

```
class BigInt {
public:
    typedef unsigned long base_t;
    typedef vector<base_t> data_t;
    static BigInt Zero;
```

```
    static BigInt One;
    static BigInt Two;
    friend class bit;
    friend class Rsa;

private:
    data_t data;//存储数据
    bool isNegative;//区分正负
    ......

    class bit {
    public:
        size_t size();
        bool at(size_t i);
        bit(const BigInt& a);
    private:
        data_t _bitvec;
        size_t _size;
    };
};
```

(二) 基本运算操作

大整数类支持的基本运算操作包括赋值、加、减、乘、除、取模，以及大小比较运算。同时重载了 « 运算符，使其支持大整数类的输出。我们封装这些运算操作，RSA 加密时直接使用即可，不用关注内部实现细节。

基本运算操作

```
//重载运算符
friend BigInt operator + (const BigInt& a, const BigInt& b);
friend BigInt operator - (const BigInt& a, const BigInt& b);
friend BigInt operator * (const BigInt& a, const BigInt& b);
friend BigInt operator / (const BigInt& a, const BigInt& b);
friend BigInt operator % (const BigInt& a, const BigInt& b);
friend bool operator < (const BigInt& a, const BigInt& b);
friend bool operator <= (const BigInt& a, const BigInt& b);
friend bool operator == (const BigInt& a, const BigInt& b);
friend bool operator != (const BigInt& a, const BigInt& b);

friend BigInt operator + (const BigInt& a, const long b);
friend BigInt operator - (const BigInt& a, const long b);
friend BigInt operator * (const BigInt& a, const long b);
friend BigInt operator / (const BigInt& a, const long b);
friend BigInt operator % (const BigInt& a, const long b);
friend bool operator < (const BigInt& a, const long b);
friend bool operator <= (const BigInt& a, const  long b);
friend bool operator == (const BigInt& a, const long b);
friend bool operator != (const BigInt& a, const long b);
```

```
friend ostream& operator << (ostream& out, const BigInt& a);
friend BigInt operator <<(const BigInt& a, const unsigned int n);

//构造函数
BigInt();//无参构造函数
BigInt(const string& num);//根据字符串初始化
BigInt(const long n);//根据long型整数初始化
BigInt(const BigInt& a, bool isNegative);//使用另一个大整数初始化
```

(三) 大数幂模运算

大整数类同样支持模幂运算，函数 moden 接收参数：指数 exp、模数 p，返回计算结果。由于计算产生的中间结果可能很大，如果直接计算内存占用大，且效率不高。因此我们对于每一步的运算结果都会取模，边乘边约简，提高了运算的效率。

moden 函数计算了 *this(调用该函数的对象) 在模数 p 意义下的 exp 次幂。模数幂运算是一种数学运算，它计算了当一个整数指数被模数整除后的余数。它经常用于加密学中，以有效地计算大指数的结果。该函数首先使用 exp 的值初始化一个名为 tmp 的 bit 对象，然后使用值 1 初始化一个名为 d 的 BigInt 对象。然后，函数进入一个循环，该循环从 tmp 的最高有效位开始，逐次向下遍历到最低有效位。在每次循环迭代中，它根据以下规则更新 d 的值：

1. d 被平方后对 p 取模。
2. 如果 tmp 的当前位被设置（即等于 1），则 d 被乘以 *this 后对 p 取模。
3. 最后，函数返回 d 的值。

因此，总的来说，该函数计算了 *this 的 exp 次幂在模数 p 意义下的值，使用了一种高效的算法，该算法利用了将一个数平方后对 p 取模的方法比将其乘以自己的方法要快的事实。

大整数幂运算

```
BigInt BigInt::moden(const BigInt& exp, const BigInt& p)const
{
    BigInt::bit tmp(exp);
    BigInt d(1);

    //逐步运算，计算中间结果
    for (int i = tmp.size() - 1; i >= 0; --i)
    {
        d = (d * d) % p;
        if (tmp.at(i))
            d = (d * (*this)) % p;
    }
    return d;
}
```

(四) 扩展欧几里得算法求乘法逆元

extendEuclid 函数计算乘法逆元的值，利用了扩展欧几里得算法。这是一种求两个整数的最大公约数（GCD）的算法，同时找到整数 x 和 y，使得 ax + by = gcd(a, b)。在这种情况下，

*this 是调用该函数的对象，而 m 是另一个整数。

函数首先检查 m 是否为负数，然后初始化三个名为 a、b 和 t 的 BigInt 数组，它们将在计算过程中用于存储中间值。然后，函数根据扩展欧几里得算法的标准形式初始化 a 和 b 的值。具体地，a[0] 和 b[0] 分别被初始化为 1 和 0，而 a[1] 和 b[1] 分别被初始化为 0 和 1。a[2] 被初始化为 m，b[2] 被初始化为 *this。然后，函数进入一个无限循环，直到 b[2] 的值为 1 时终止。在每次循环迭代中，函数根据扩展欧几里得算法的标准形式更新 a 和 b 的值，使用名为 t 的临时数组来存储中间值。

最后，如果 b[2] 的值变为 1，函数检查 b[1] 是否为负数。如果是，则将 m 加到 b[1] 上，并返回结果对 m 取模的值。否则，直接返回 b[1]。

扩展欧几里得算法求乘法逆元

```
BigInt BigInt::extendEuclid(const BigInt& m) {
    assert(m.isNegative == false);
    BigInt a[3], b[3], t[3];
    a[0] = 1; a[1] = 0; a[2] = m;
    b[0] = 0; b[1] = 1; b[2] = *this;
    if (b[2] == BigInt::Zero || b[2] == BigInt::One) {
        return b[2];
    }

    while (true) {
        if (b[2] == BigInt::One) {
            if (b[1].isNegative == true)
                b[1] = (b[1] % m + m) % m;
            return b[1];
        }

        BigInt q = a[2] / b[2];
        for (int i = 0; i < 3; ++i) {
            t[i] = a[i] - q * b[i];
            a[i] = b[i];
            b[i] = t[i];
        }
    }
}
```

至此，实验的准备工作已经完成，我们的大整数类支持基本运算，以及专门用于 RSA 的大数模幂运算、扩展欧几里得算法求乘法逆元。

五、 实验内容

(一) 大素数生成

createPrime 函数用来生成一个长度为 n 的大素数，一共需要三步：

1. 生成一个 n 位的奇数（createOddNum 函数）
2. 使用 RabinMiller 算法检验 k 轮，判断是否为素数（rabinMiller 函数）

3. 如果是素数，则退出；否则将当前奇数加 2，跳回第二步

```
BigInt createPrime(unsigned int n, int it_cout);//生成长度为 n 的素数
BigInt createOddNum(unsigned int n);//生成长度为 n 的奇数
bool rabinMiller(const BigInt& a, const unsigned int k);//判断素数
```

1. 随机数生成原理

createOddNum 函数生成一个给定长度 n 的奇数、随机的大整数。参数 n 指定了大整数的长度，单位为位，函数会生成一个大约有 n/4 个十六进制数位的大整数。

函数首先通过除以 4 将 n 转换为十六进制数位的数量。然后，它初始化了一个名为 *hex_table* 的十六进制数位数组，该数组用于将随机大整数生成为十六进制数位的字符串。

如果 n 不为 0，函数使用 *hex_table* 数组和一个 ostringstream 对象生成一个随机的十六进制数位字符串。然后，它使用 rand() 函数生成一个介于 0 和 15 之间的随机数 k，并通过重复生成直到 k 为奇数来确保 k 为奇数。最后，它将 k 的值附加到字符串末尾，并返回使用该字符串初始化的 BigInt 对象。如果 n 为 0，函数只是返回使用值 0 初始化的 BigInt 对象。

总的来说，这个函数使用 rand() 函数和十六进制数位字符串生成了一个大约有 n/4 个十六进制数位的随机、奇数大整数。

随机数生成

```
BigInt Rsa::createOddNum(unsigned int n) {
    n = n / 4;
    unsigned char hex_table[] = { '0','1','2','3','4','5','6','7','8','9','A'
        ,'B','C','D','E','F' };
    if (n) {
        ostringstream oss;
        for (size_t i = 0; i < n - 1; ++i) {
            oss << hex_table[rand() % 16];
        }
        int k;
        do {
            k = rand() % 16;
        } while (k % 2 == 0);
        oss << hex_table[k];
        string str(oss.str());
        return BigInt(str);
    }
    else
        return BigInt::Zero;
}
```

2. RabinMiller 素性检测

rabinMiller 函数使用 Rabin-Miller 素数测试来确定给定的大整数 n 是否是质数。参数 k 指定要执行的测试迭代次数。

函数首先检查 n 是否等于 0 或 2。如果 n 等于 0，函数返回 false；如果 n 等于 2，函数返回 true。然后，函数使用 bit 对象生成 n-1 的位表示，并使用该值初始化一个大整数 n_1。它还检查 n_1 的最低有效位（LSB）是否为 1。如果是，函数返回 false。如果 n_1 的 LSB 不是 1，函数进入一个循环，执行 k 次 Rabin-Miller 测试。对于每次迭代，函数生成一个介于 2 和 n_1 之间的随机大整数 a，并使用值 1 初始化大整数 d。然后，它进入另一个循环，从最高有效位开始迭代 n_1 的位。

对于 n_1 的每一位，函数按如下方式更新 d 的值：

1. 如果当前位是 0，d 在模 n 意义下取平方。
2. 如果当前位是 1，d 在模 n 意义下取平方，然后在模 n 意义下乘以 a。

如果 d 在循环过程中变为 1，但它变为 1 之前的值不是 1 或 n_1，函数返回 false。如果循环完成时 d 没有变为 1，函数检查 d 是否等于 1。如果不是，函数返回 false。如果内部循环在所有 k 次迭代中完成，函数返回 true。

总的来说，这个函数使用 Rabin-Miller 素数测试来确定给定的大整数 n 是否是质数。它执行测试 k 次，如果 n 在每次测试中都通过，则返回 true，如果在任何一次测试中失败，则返回 false。

RabinMiller 素性检测

```
bool Rsa::rabinMiller(const BigInt& n, const unsigned int k) {
    assert(n != BigInt::Zero);
    if (n == BigInt::Two)
        return true;

    BigInt n_1 = n - 1;
    BigInt::bit b(n_1);
    if (b.at(0) == 1)
        return false;
    for (size_t t = 0; t < k; ++t) {
        BigInt a = createRandom(n_1);//创建一个小于n-1的随机数
        BigInt d(BigInt::One);
        for (int i = b.size() - 1; i >= 0; --i) {
            BigInt x = d;
            d = (d * d) % n;
            if (d == BigInt::One && x != BigInt::One && x != n_1) {
                return false;
            }
            if (b.at(i)) {
                assert(d != BigInt::Zero);
                d = (a * d) % n;
            }
        }
        if (d != BigInt::One) {
            return false;
        }
    }
    return true;
}
```

3. 程序框图

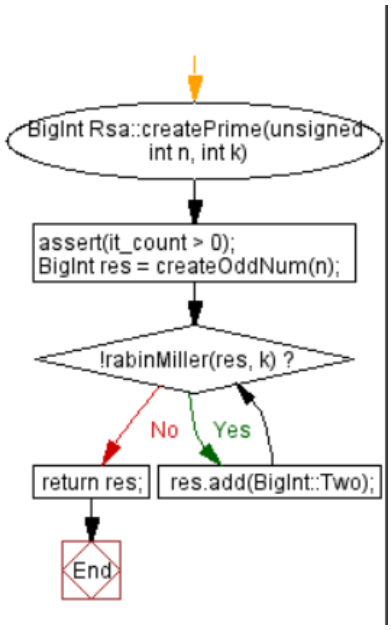

图 5: 生成素数程序框图

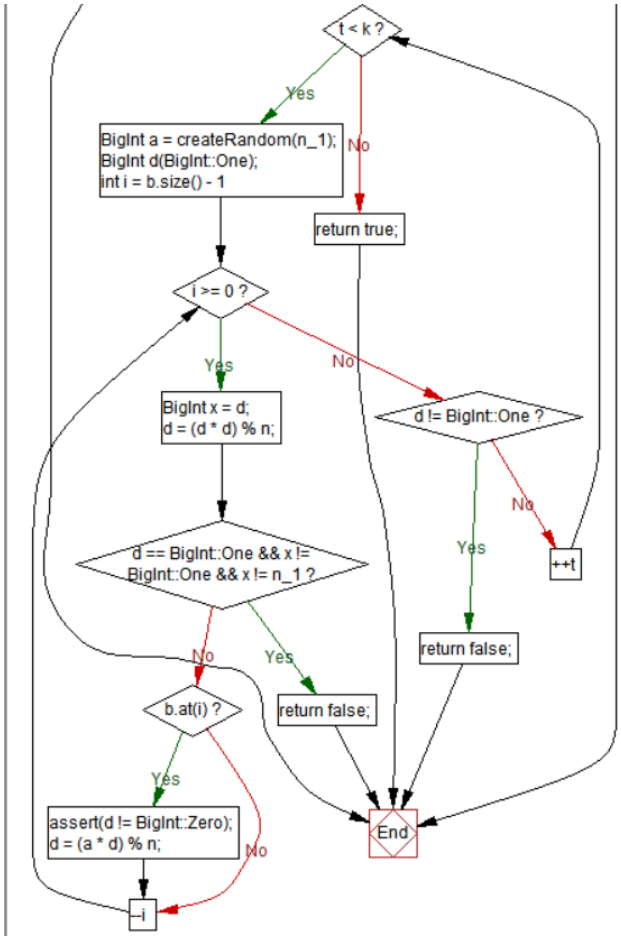

图 6: RabinMiller 素性检测程序框图

(二) RSA 初始化

init 函数初始化 RSA 密钥对，createExp 函数生成 RSA 公钥和私钥。init 函数首先使用 srand 函数设置随机数种子。然后，它调用 createPrime 函数生成两个大质数 p 和 q。然后，它将 p 和 q 相乘并赋值给变量 N, 并计算 (p-1) * (q-1) 的值并赋值给变量 ou。最后, 它调用 createExp 函数生成 RSA 公钥和私钥。

createExp 函数首先将公钥 e 设置为 65537(0x10001)。然后,它调用 BigInt 类的 extendEuclid 函数计算私钥 d。extendEuclid 函数使用扩展欧几里得算法求出 d，使得 d * e 1 (mod ou)。

总的来说，这两个函数使用随机数生成两个大质数 p 和 q，并使用扩展欧几里得算法生成 RSA 公钥和私钥。

RSA 初始化

```
void Rsa::init(unsigned int n) {
    srand(time(NULL));
    p = createPrime(n, 10);
    q = createPrime(n, 10);
    N = p * q;
    ou = (p - 1) * (q - 1);
    createExp(ou);
}
void Rsa::createExp(const BigInt& ou){
    e = 65537;
    d = e.extendEuclid(ou);
}
```

(三) RSA 加密和解密

RSA 加密和解密实现均调用大整数类的模幂运算来实现。加密函数使用 RSA 公钥和模数 N 进行加密。它调用 BigInt 类的 moden 函数，计算 m 的 e 次幂模 N 的值。解密函数使用 RSA 私钥和模数 N 进行解密。它调用 BigInt 类的 moden 函数，计算 c 的 d 次幂模 N 的值。

RSA 初始化

```
BigInt Rsa::encryption(const BigInt& m) {
    return m.moden(e, N);
}
BigInt Rsa::decryption(const BigInt& c)
{
    return c.moden(d, N);
}
```

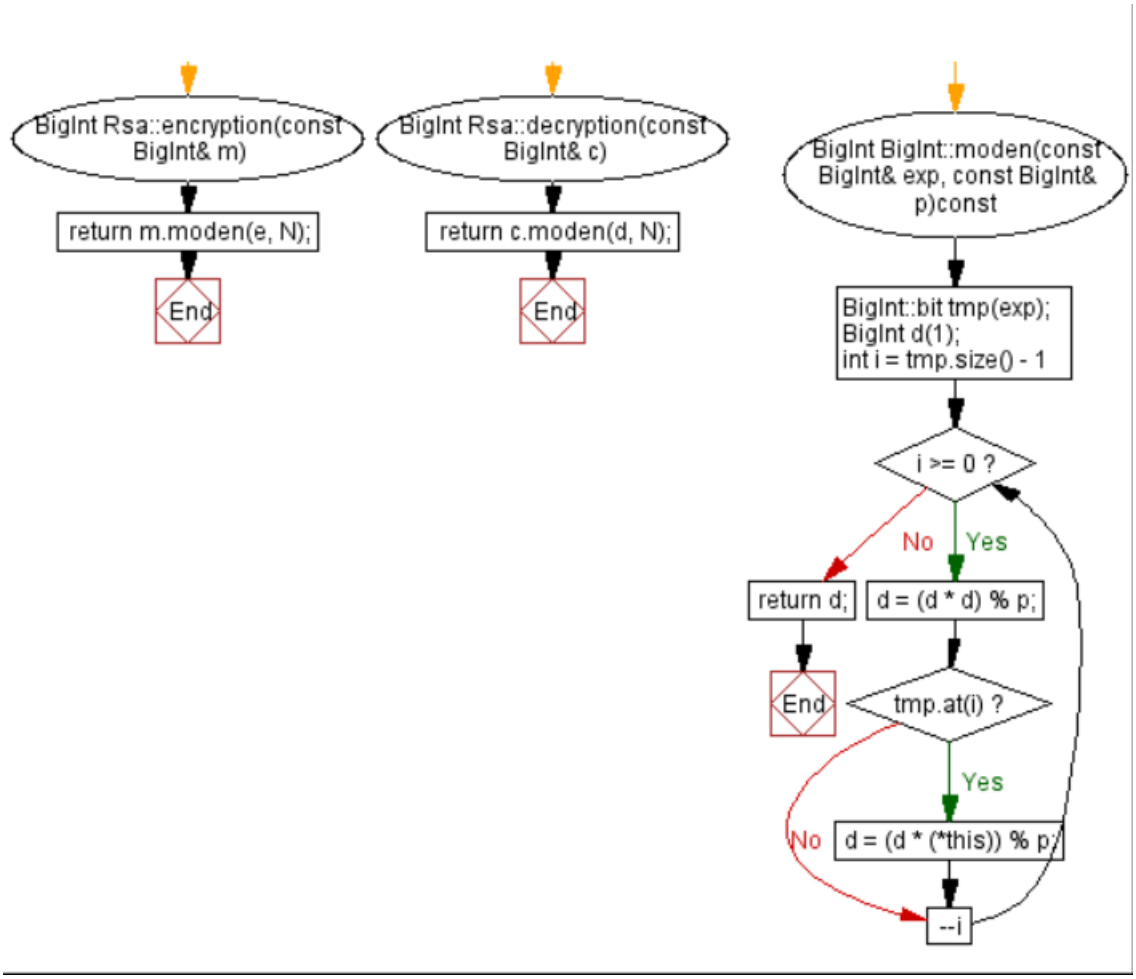

图 7: RSA 加密解密程序框图

六、 程序演示

(一) 256 位 RSA 演示

对于 256 位的 RSA，初始化只需要 1s，图8显示打印出的 RSA 的公钥和私钥，接下来我们使用这对参数进行加解密。

如图9所示，输入 16 进制数据 0000ABCD，被加密成一串十六进制数据；在执行解密函数，输入这一串十六进制数据，可以看到 0000ABCD 被成功解密出来，耗时 62ms。

```
输入位数:256
初始化....
初始化完成.
用时:1s.
==========Welcome to use RSA encoder==========
                e.encrypt 加密
                d.decrypt 解密
                s.setkey 重置
                p.print 显示
                q.quit 退出
input your choice:
>p
N:A1C5B2AA7FD33358A1D101059BFC1C417C5B94FCAD3FFEED9F04EBC330F6BDDB
p:B5A0D9E211D295C37E153881C675C75F
q:E40370602B36B38F21A27FF456B1C705
e:00010001
d:59D16D5092E60D75216AECB95390D79274EB35976C24441289BB3047943F0D69
```

图 8: 256 位 RSA 初始化

```
input your choice:
>e
>输入16进制数据:abcd
明文:0000ABCD
密文:9D1ED1B5A9840D05B2D973F972943316F41E25F35E39A82F0CEC8B37D2BBF571
==========Welcome to use RSA encoder==========
                e.encrypt 加密
                d.decrypt 解密
                s.setkey 重置
                p.print 显示
                q.quit 退出
input your choice:
>d
>输入16进制数据:9D1ED1B5A9840D05B2D973F972943316F41E25F35E39A82F0CEC8B37D2BBF571
用时:62ms.
密文:9D1ED1B5A9840D05B2D973F972943316F41E25F35E39A82F0CEC8B37D2BBF571
明文:0000ABCD
```

图 9: 256 位 RSA 加解密

(二) 1024 位 RSA 演示

接下来我们初始化 1024 位的 RSA，从图10可以看到，初始化 1024 位的 RSA 需要 32s，从效率上来说还有待提高，openssl 库能够在 3s 以内初始化 1024 位的 RSA。后期想要优化的话，可以考虑优化大整数类的运算，比如使用消耗较小的加减移位运算取代消耗较大的除法运算等等。对于大数幂模运算，可以使用蒙哥马利大数模乘算法。

使用 1024 位的 RSA 执行加解密，如图11所示，加密解密操作没有问题，解密耗时 1636ms。

```
input your choice:
>s
输入位数:1024
初始化....
初始化完成.
用时:32s.
==========Welcome to use RSA encoder==========
              e.encrypt 加密
              d.decrypt 解密
              s.setkey 重置
              p.print 显示
              q.quit 退出
input your choice:
>p
N:5188D33A2EBAB0FD9B58C129B5AD651FAA984C375C2B2437EC78303A9284AC77664284D980FD10BFD70E2E3D603D643A8F3F6B8793AE80D533D373F4CE1867A0B05CB0FB8A8B6A7403AB545AD61A4123E076043135F0FDD0C04A4B7D29D7DCC5377873E8361B92427FF8E2E54227E751AEACEBB69A6B4D51BB5BDB95577050CF
p:F8BB1F43F53721F197FBE729B4E9C88F764AA0CB4A677F1BC21F77CEC4342259880E6A954EE60FBFC5AC75E7D51213C2A68EB539875C6B372839571F60290453
q:53EAD30AA4B4B249B2321E715D984759677DD228177DA74FD7B8A44D5651481897FF18DBBF5FDEF34B61A5F0D877DEDA6809FEAA24D3DEDC85F59AF44A827215
e:00010001
d:12F38BAD4D6945745D245F1993C9EEE45261BDED6C35F6E02CCDAEE01E684A322289B99958BAF9A6AD94F5387CE32825D85FF45D79FD28A2E5BDF388E65293DB5F70FD96F3031506B04EA8FC37B96B4FFD0A5CE51321817A585A581AE655D28EEFB582991FA02FB94A1DB6107D86CD772272309B8D9617E5D768260A671F0669
```

图 10: 1024 位 RSA 初始化

```
>e
>输入16进制数据:2735365738256325364756
明文:00273536573825632536 4756
密文:50CA348C3F69F78A71846A2ADEDCAD69F7C055263D275F6B22D853B00CD1DCF5B758E164E7EBF1EB3507859B3BDCEBC1EB6BA0F4FED6E532C5CB904322E5F7ED459FB6DDB128466BDD7E85F2D9C8694A2A6E95506D2F5B2EEACF92DD3663DB061D59609CEBFB02EB94A1BF63CFB3B23518C2E304D709C08B2E8B760A33E8B858
==========Welcome to use RSA encoder==========
              e.encrypt 加密
              d.decrypt 解密
              s.setkey 重置
              p.print 显示
              q.quit 退出
input your choice:
>d
>输入16进制数据:50CA348C3F69F78A71846A2ADEDCAD69F7C055263D275F6B22D853B00CD1DCF5B758E164E7EBF1EB3507859B3BDCEBC1EB6BA0F4FED6E532C5CB904322E5F7ED459FB6DDB128466BDD7E85F2D9C8694A2A6E95506D2F5B2EEACF92DD3663DB061D596009CEBFB02EB94A1BF63CFB3B23518C2E304D709C08B2E8B760A33E8B858
用时:1636ms.
密文:50CA348C3F69F78A71846A2ADEDCAD69F7C055263D275F6B22D853B00CD1DCF5B758E164E7EBF1EB3507859B3BDCEBC1EB6BA0F4FED6E532C5CB904322E5F7ED459FB6DDB128466BDD7E85F2D9C8694A2A6E95506D2F5B2EEACF92DD3663DB061D59609CEBFB02EB94A1BF63CFB3B23518C2E304D709C08B2E8B760A33E8B858
明文:00273536573825632536 4756
```

图 11: 1024 位 RSA 加解密