南开大学
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

密码学大作业设计实验报告

2023 年 1 月 8 日

目录

一、 设计要求

设计一个保密通信的协议，利用 RSA 公钥密码算法，为双方分配一个 AES 算法的会话密钥，然后利用 AES 加密算法和分配的会话密钥，加解密传送的信息。

假设条件：通讯双方为 A 和 B，假设发方拥有自己的 RSA 公钥 PK_A 和私钥 SK_A ，同时收方拥有自己的 RSA 公钥 PK_B 和私钥 SK_B ，同时收发双方已经通过某种方式知道了双方的公钥。

二、 保密通信协议设计

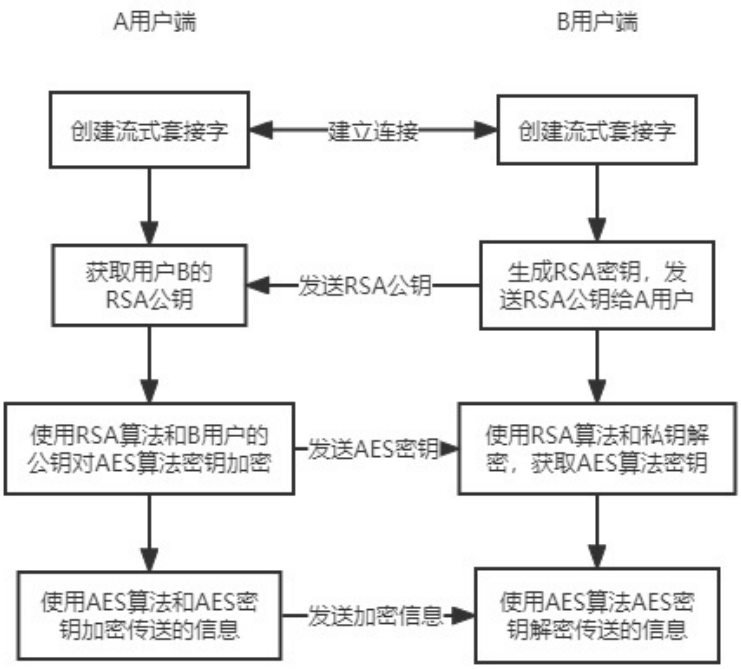

图 1: 密钥分发和加密传输流程图

图1说明从用户端 A 向用户端 B 发送加密信息的过程。在实际编程中，实现了用户端 A 和用户端 B 的相互通信，既可以从用户端 A 向用户端 B 发送加密信息，也可以从用户端 B 向用户端 A 发送加密信息。

- **建立连接**

在建立连接的过程中，使用流式套接字、TCP 协议。将 B 用户端作为服务器端，等待接受连接；A 用户端作为客户端，发送连接请求。在实际通信的过程中，使用多线程，A 和 B 用户端存在专门发送信息的线程，也存在专门接收信息的线程，从而实现 A、B 用户端的相互通信。

- **RSA 密钥初始化**

A、B 用户端默认存储一个自己的 RSA 密钥，并且存储对方的 RSA 公钥。RSA 密钥使用自己编程的 RSA 生成工具生成，双方都可以重新生成 RSA 密钥，并将更新的 RSA 公钥发送给对方。

- **AES 密钥的传输**

双方共同使用一个协商好的 AES 密钥，双方都可以对这个 AES 密钥进行修改。修改 AES 密钥时，想要修改密钥的一方生成一个新的 AES 密钥，使用对方的 RSA 公钥对其进行加密，将加密后的 AES 密钥发送给对方；对方使用自己的 RSA 私钥对信息进行解密，更新获得的 AES 密钥。

- **加密信息的传输**

在传输信息的过程中，发送方通过共享的 AES 密钥采用 CBC 模式对信息进行分组加密，接收方收到信息后，使用共享的 AES 密钥对信息进行解密。传输的信息既可以是一个字符串，也可以是一个文件。

(一) Socket 建立连接

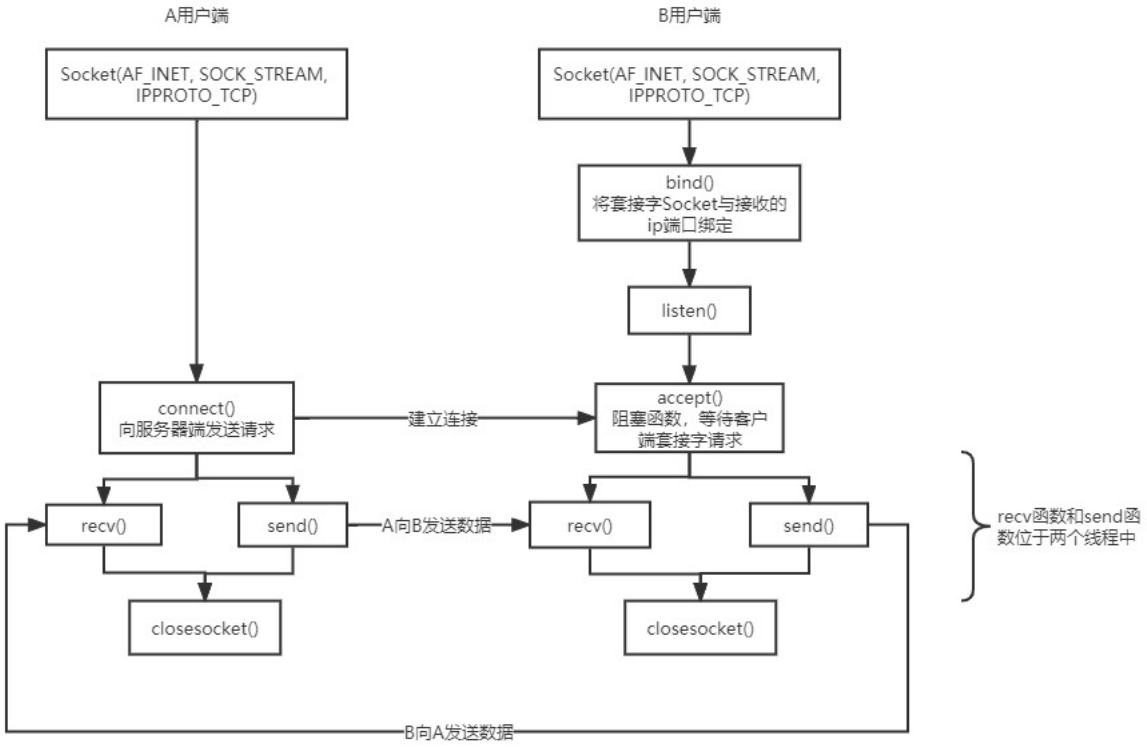

图 2: Socket 建立连接通信流程图

通讯双方 A 和 B 通过流式套接字建立连接，A 端作为客户端发送请求，B 端作为服务器端接受请求。如果服务器进程接受请求，则 AB 用户之间建立了一条通信连接，之后便可以通过 recv 函数和 send 函数分别实现在已建立的套接字上接收和发送数据，实现同一台主机下数据之间的网络传输。

- **数据包**

数据在传输的过程中，具有不同的类型。可能是 string 类型的 RSA 密钥；也可能是 BigInt 类型的加密 AES 密钥；也可能是经过 AES 加密的信息。所以需要构建一个数据包类，标识不

同类型的数据信息。当传输加密的 AES 密钥时，type = Packet::AES；当传输加密的字符串时，type = Packet::String。

数据包格式

```
class Packet {
private:
        int type;//类型
        BigInt bigint1;//存储公钥密码和AES加密密钥
        BigInt bigint2;
        int len;//数据长度
        Abyte Data[1024];//存储AES加密的数据
public:
        enum { Null,RSA,AES,String, FileName, FileContent };}
```

- **多线程实现**

当建立连接之后，主线程使用 while 循环，不断发送信息；使用 CreateThread 函数新创建一个子线程，使用 while 循环不断接收信息，并使用 AES 密钥，将接收到的信息解密出来。

(二) RSA 算法原理

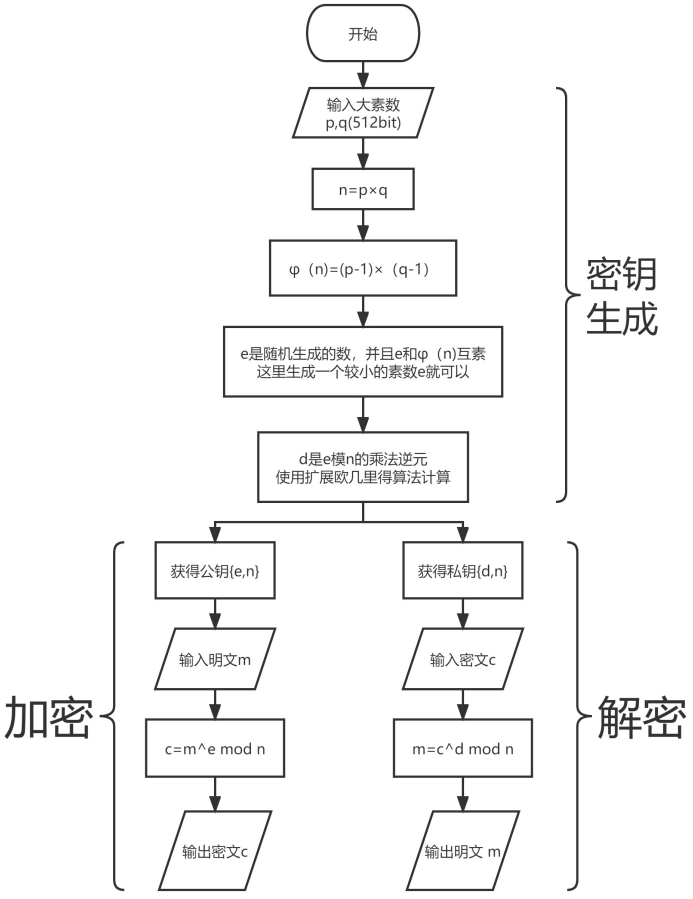

图 3: RSA 算法流程图

AES 是一种对称加密算法，加密和解密使用的是同一个密钥，在密钥从加密者传输到解密者的过程中，密钥一旦暴露，整个加密过程就毫无意义了。所以，在传输 AES 密钥时，使用 RSA 这种非对称加密算法进行加密，把 AES 密钥安全的传递到解密者手里，解决 AES 密钥分发问题。

RSA 算法实现分为三个部分：密钥生成；明文加密；密文解密。

1. 大整数类

为了提高 RSA 算法的安全性，使用 1024bit 的密钥，需要设计一个 BigInt 存储 1024bit 的密钥并对它进行运算。

在大整数类中，使用无符号整数类型的数组存储大整数，数组大小为 128，即可以存储最长的数据长度为 4096bit, 对加减乘除等运算进行函数重载, 在重载的过程中，尽量使用位运算，以加快运算速度。

如：进行大整数乘法时，对两个乘数的大整数数组进行遍历，乘法结果的大整数数组中的第 i+j 个数等于两个乘数的大整数数组中的第 i 个数乘第 j 个数加上第 i 个数加上第 (j-1) 个数乘法结果的溢出。

乘法重载

```
typedef unsigned long long uint64;
BigInt BigInt::operator * (const BigInt& var) const {
    BigInt res = 0;
    for (int i = 0; i < UINT_DIGIT; i++) {
        if (digit[i]) {
            unsigned int overflow = 0;
            for (int j = 0; i + j < UINT_DIGIT; j++) {
                long tmp = (uint64)(digit[i]) * (uint64)(var.digit[j]) + (
                    uint64)(res.digit[i + j]) + (uint64)(overflow);
                res.digit[i + j] = tmp & 0xffffffff;
                overflow = tmp >> 32;
}}}
    return res;}
```

2. 密钥生成

(1). 首先，随机生成两个 512bit 的大素数 p 和 q。

素数生成原理：首先随机选取一个大的奇数，然后用素性检验算法检验这一奇数是否为素数；如果不是，将这个大的奇数加 2（或减 2），重复这一过程，直到找到素数为止。实验中使用 Miller-Robin 这一素性检测算法，算法基于费马小定理，二次探测定理进行检测，是一种概率素数测试法。

Miller-Robin 算法原理：若 n 为大于 2 的素数，则有 $n-1=2^k*q$，$k>0$，q 为奇数，且 $1<a<(n-1)$。则对于以下两个条件:

(a). $a^q \mod n = 1$;

(b). $a^{2^{j-1}}*q \mod n = n-1$，j 是 $[1,k]$ 区间内的正整数;

若满足二者条件之一未必为素数；若二者都不满足，必为合数。

在实际编程过程中，为了提高检测素数正确性的概率，进行二十轮检测，每一轮随机生成一个 a，计算 $a^q \mod == 1$ 和 $a^{2^{j-1}} * q \mod n == n-1$ 是否成立。

素性检测

```
bool isPrime(BigInt num) {
    BigInt m = num - 1;
    int k = 0;
    while (m.getBit(0) == 0)//把n-1写成2的k次方*t的形式
    {   k++;
        m >>= 1;//t向右移动一位，相当于t/2
    }
    for (int i = 0; i <= 20; i++)//进行20轮测试，增加可靠性
    {
        BigInt a = CreateOddNum(3);//选取底数a，1<=a<=n-1
        BigInt x = PowMod(a, m, num);//计算a^q mod n
        BigInt y;
        //计算是否存在j使a ^(2^(j-1))*q mod n == n-1成立
        for (int j = 0; j < k; j++)
        {   y = x * x % num;
            if (y == 1 && x != 1 && x != (num - 1))
                return false;
            x = y;
        }
        if (y != 1)
            return false;
    }
    return true;
}
```

(2). 计算 $n = p * q$ 和 n 的欧拉函数 $\Phi(n) = (p-1) * (q-1)$，随机选择一个整数 e，使 $1 < e < \Phi(n)$，且 e 与 $\Phi(n)$ 互质；

(3). 计算 e 对于 $\Phi(n)$ 的乘法逆元 d，使得 $e * d \mod \Phi(n) = 1$。

乘法逆元的生成：使用扩展欧几里得算法，在算法实现过程中，使用递归算法，当递归到某个时候，使 a 等于 0，认为递归结束。（实际上就是将辗转相除法中产生的式子逐个倒回去，得到 $ax + by = \gcd(a, b)$ 的整数解 x 和 y）。

扩展欧几里得算法

```
void ExtendEuclid(BigInt a, BigInt b, BigInt& x, BigInt& y, const BigInt& m)
    {
    if (a == 0) {
        x = 0, y = 1;
        return;
    }
    BigInt c = b / a, d = b % a;
    ExtendEuclid(d, a, y, x, m);
    x = (x + m - (c * y) % m) % m;}
```

(4). 将 n 和 e 封装成公钥，n 和 d 封装成私钥

3. 明文加密

在获取公钥 e，n 之后，加密过程就是对明文 m 进行以下运算：

$$c = m^e \mod n$$

4. 密文解密

在获取私钥 d，n 之后，解密过程就是对密文 c 进行以下运算：

$$m = c^d \mod n$$

(三) AES 算法原理

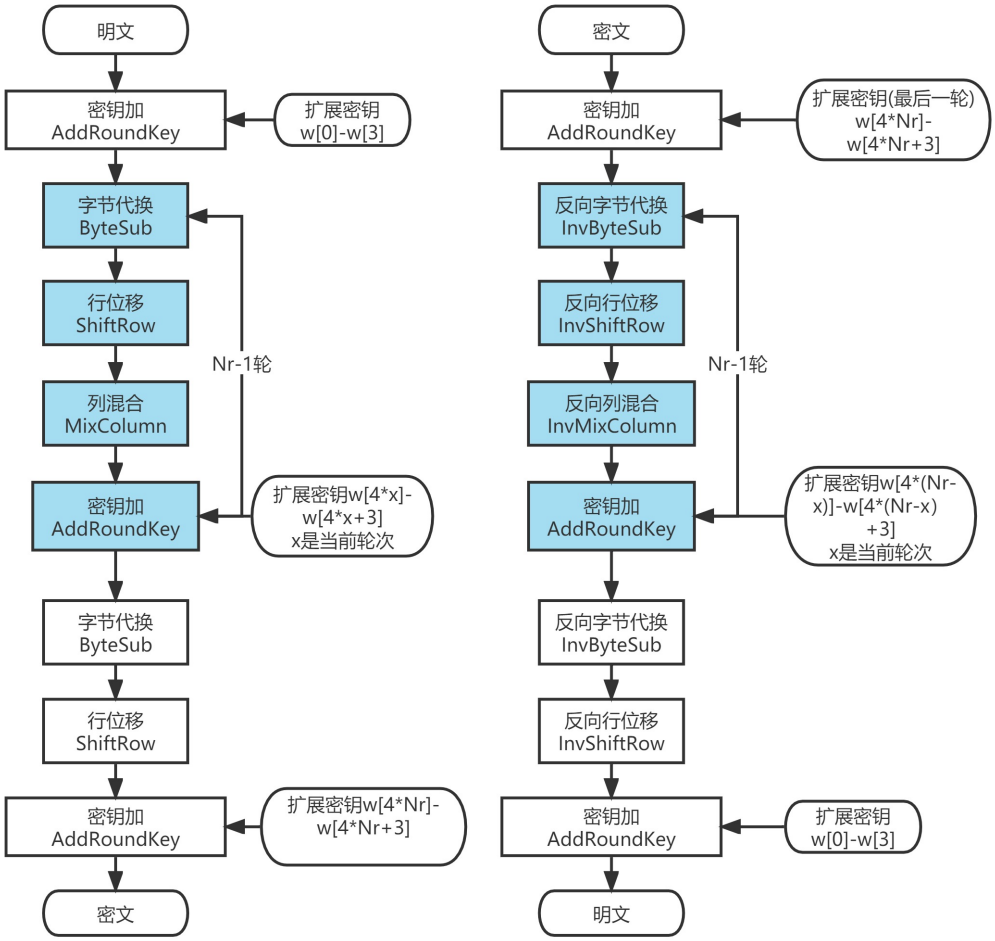

图 4: AES 算法流程图

AES 算法是一个对称分组密码算法。数据分组长度必须是 128 bits，使用的密钥长度为 128bit，192bit 或 256 bits。对于三种不同密钥长度的 AES 算法，分别称为“AES-128”、“AES-192”、“AES-256”。

本次实验对数据加密采用 AES-128 标准，CBC 加密模式。密钥长度为 128 位（16 字节）；明文分组长度也为 128 位（16 字节），如果最后一段或者密钥长度不够 16 个字节，就需要用 Padding 来把这段数据填满 16 个字节，然后分别对每段数据进行加密。

1. CBC 加密模式

CBC 加密模式会将明文分组和前一组的密文分组进行异或 (XOR) 运算，然后再进行加密，第一组明文和一个任意初始化向量 IV 进行异或。CBC 加密模式中，每个密文块都依赖与它前边的所有明文块。优点是即使是同样的原文生成的密文也不一样；缺点是串行处理数据，使得加密速度比较慢。

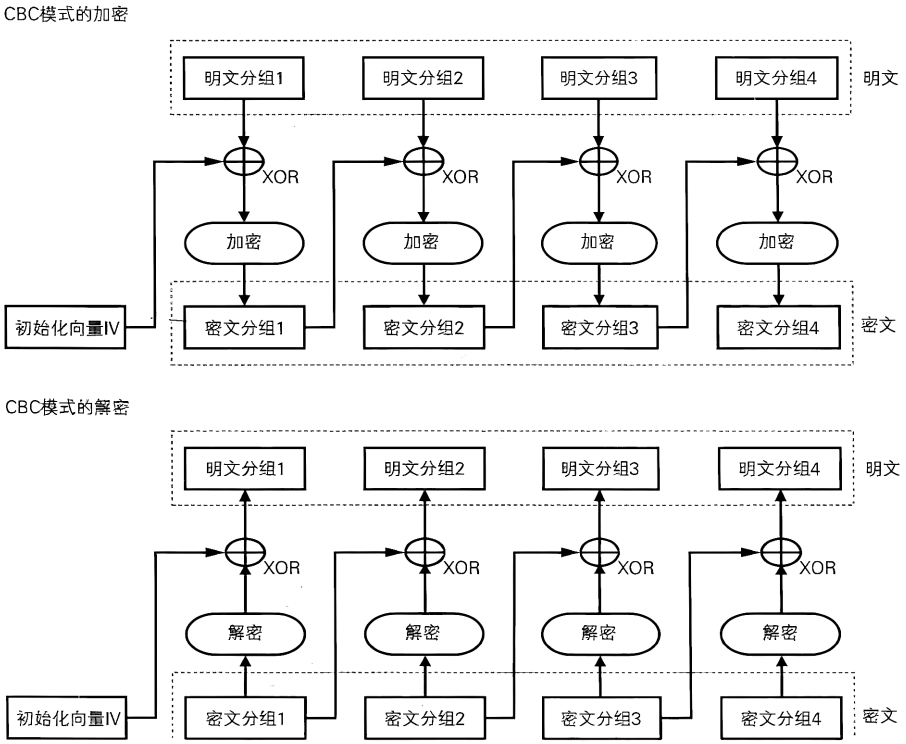

图 5: CBC 加密解密原理图

在编程过程中，在对分组信息进行 AES 加密之前，需要对分组明文进行进一步处理，将明文和上一组加密获得的密文异或，获得实际加密的明文；在解密过程中，解密获得的信息和上一组的密文异或，才得到真正的明文。

CBC 加密模式

```
if (t == 0)
    for (int i = 0; i < Nb * 4; i++)
        P_temp[i] = P_temp[i].to_ulong() ^ AES_IV[i].to_ulong();
else
    for (int i = 0; i < Nb * 4; i++)
        P_temp[i] = P_temp[i].to_ulong() ^ pkt->getDataNum(16 * (t - 1) + i).
            to_ulong();

Encryption(P_temp, result, w);
```

2. AES 算法中使用的数据结构

宏定义 Abyte 和 word 两种类型，typedef bitset<8> byte；typedef bitset<32> word；定义 Abyte 为 8bit 数据；word 为 32bit 数据（因为一个字等于四个字节）。

3. AES 加密算法

在整体的加密算法中，进行了 Nk 轮加密，每一轮加密进行的运算相同，都进行了字节代换、行位移、列混合和密钥加（密钥加使用的密钥是通过密钥扩展得到的）；结尾轮和前面各轮稍有不同，省略了列混合这一步。此外，在第一轮开始之前，需要对输入明文和输入密钥进行密钥加，获得初始状态矩阵，在开始循环轮加密。

AES 加密函数

```
void Encryption(const byte *in, byte *out, word *w) {
        for (int i = 0; i < Nb; ++i)
                Roundkey[i] = w[i];
        AddRoundKey(state, Roundkey);//密钥加
       //进行Nk-1轮循环
        for (int round = 1; round < Nr; ++round)
        {    ByteSub(state);//字节代换
                ShiftRow(state);//行位移
                MixColumn(state);//列混合
                for (int i = 0; i < Nb; ++i)
                        Roundkey[i] = w[4 * round + i];
                AddRoundKey(state, Roundkey);//密钥加
        }
        //最后一轮，没有列混合
        ByteSub(state);
        ShiftRow(state);
        for (int i = 0; i < Nb; ++i)
                Roundkey[i] = w[4 * Nr + i];
        AddRoundKey(state, Roundkey);
}
```

(1).ByteSub()——字节代换函数实现

首先，将字节看作 GF(2^8) 上的元素，映射到自己的乘法逆元，规定 00 映射到自身 00。其次，对之前的结果做如下的 (GF(2) 上、可逆的) 仿射变换。

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

图 6: 放射变换

为了简化计算过程，提高程序效率，将字节代换对各种可能字节的变换结果制成一个表，称它为 AES 的字节代替表 (或 S 盒)。

所以，在实际实验中，计算出字节对应的 int 值，从 AES 的字节代替表中直接获得替换后的字节即可，不需要进行仿射变换的计算。

字节代换函数

```
void ByteSub(byte **state) {
    for (int i = 0;i < 4;i++) {
        for (int j = 0;j < Nb;j++) {
            int a = 0;int k = 1;
                for (int t = 0;t < 8;t++) {//按位计算字节对应int值，共8bit
                                a += state[i][j][t] * k;
                                k *= 2;}
                state[i][j] = AESReplaceTable[a];
}}}
```

(2).ShiftRow()——行位移函数实现 行位移是将状态矩阵的各行进行循环移位，不同状态行的位移量不同。在 AES-128 标准中，行位移量如下图??所示:

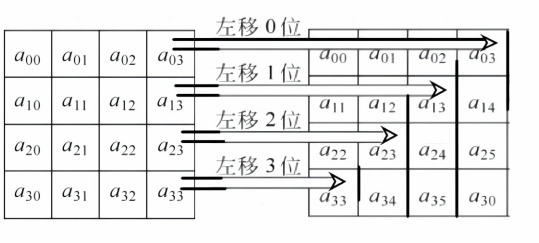

图 7: 行位移量

行位移函数

```
void ShiftRow(byte** state) {
    for (int i = 0;i < 4;i++) {
        byte temp[Nb];//取出状态矩阵的一行
        for (int j = 0;j < Nb;j++)
                temp[j]= state[i][j];
        int Shiftnum = ShiftNum[(Nb - 4) / 2][i];//根据行数和Nb值获得位移量
        for (int j = 0;j < Nb;j++) {
                state[i][j] = temp[(j+Shiftnum )%Nb];
}}}
```

(3).MixColumn()——列混合函数实现

在列混合的变换中，将状态矩阵的每列看作 GF(2^8) 上的多项式，与一个固定的多项式 c(x) 进行模 x^4+1 的乘法。

$$c(x) = 03x^3 + 01x^2 + 01x + 02$$

上面的列混合运算也可以写为矩阵乘法，$sj'(x) = c(x) \bigotimes s(x)$。

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

图 8: 矩阵乘法

列混合函数

```
void MixColumn(byte** state){
    byte temp[4];
    for (int i = 0; i < Nb; ++i){
        for (int j = 0; j < 4; ++j)//获取一个字(4个字节)
           temp[j] =state[j][i];
        for (int j = 0;j < 4;j++) {
           state[j][i] = GFMUL(c_MixColumns[0][(4 - j) % 4], temp[0]);
           for (int t = 1;t < 4;t++)
                state[j][i]^= GFMUL(c_MixColumns[0][(4+t-j)%4], temp[t]);
}}}
```

(4).AddRoundKey()——密钥加函数实现

将状态矩阵与轮密钥进行按位异或，轮密钥的长度等于分组长度 Nb。

密钥加函数

```
void AddRoundKey(byte** state, word *w){
    for (int t = 0; t < Nb; t++){
        //将word中的轮密钥，反序放在byte[4]数组中
        //因为word存储小端序，word的24到31位应该和第一个byte异或
        byte temp[4];
        for (int i = 3;i >=0;i--)
                for (int j = 0;j<8;j++)
                     temp[3-i][j] = w[t][8*i+j];
        for (int i = 0;i < 4;i++)
                state[i][t] = temp[i] ^ state[i][t];
}}
```

4. AES 解密算法

解密算法实际上是对加密算法的逆运算，所以对于每一个加密部件都求出它的逆部件，就可以实现解密过程。

InvByteSub()——逆字节代换函数：求出 AES 的字节代替表的逆表，对于输入的字节获取逆表中对应的字节。

InvShiftRow()——反向行位移函数：加密是左移 ShiftNum 表中对应的值，解密就是右移 ShiftNum 表中对应的值。

InvMixColumn()——反向列混合函数：同样是每一列与一个多项式进行 GF(2^8) 上的乘法运算，只不过这个多项式 c(x) 变为：

$$c(x) = 0bx^3 + 0dx^2 + 09x + 0e$$

因为解密是对加密的逆运算，所以从最后一轮开始解密，输入的扩展密钥是从最后一轮开始的。

5. 密钥编排

在 AES 算法中,用户输入 Nk 个字长度的密钥,通过密钥扩展算法,将密钥扩展成 Nb*(Nr+1) 个字，存放在数组 word w[Nb*(Nr+1)] 中。

密钥扩展算法的具体实现：

(1). 扩展密钥的前 Nk 个字是输入的密钥 key，GetWord() 函数可以将四个 Abyte 类型转换成一个 word 类型；

(2).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；

(3). 特别地，当 i/Nb 为整数时，需要将前一个字 w[i-1] 经过一系列变换，包括位置变换 RotWord()；字代换 SubWord() 以及和轮常量 Rcon 进行异或。

密钥扩展函数

```
void KeyExpansion(const byte* Key, word* W){
    // W[]的前Nk个字是输入的key
    for (int i = 0;i < Nk;i++)
        W[i] = Get_word(Key[4 * i], Key[4 * i + 1], Key[4 * i + 2], Key[4 * i
            + 3]);
    for (int i = Nk;i < Nb * (Nr + 1);i++) {
        word temp = W[i - 1]; //前一个word
        if (i % Nk == 0) //如果i是Nk的倍数
            temp = SubWord(RotWord(temp)) ^ Rcon[i / Nk - 1];
        else if (Nk > 6 && i % Nk == 4)
            temp = SubWord(temp);
        W[i] = W[i - Nk] ^ temp;
}}
```

RotWord()——位置变换函数：接受一个字 [a0, a1, a2, a3] 作为输入，循环左移一个字节后输出 [a1, a2, a3, a0]

位置变换函数

```
word RotWord(word w){
    word w_high = w << 8;//获取w的后8位，左移
    word w_low = w >> 24;//获取w的前24位，右移
    word result = w_high | w_low;//按位或，获得新的word
    return result;
}
```

SubWord()——字代换函数：接受一个字 [a0, a1, a2, a3] 作为输入，对每一个字节计算出对应的 int 值，从 AES 的字节代替表中直接获得替换后的字节，获得一个新字节。

Rcon[10]——轮常数：对轮常数的定义为：Rcon[i] =(Rc[i]，00，00，00)；Rc[i] 表示在有限域 GF(2^8) 中 x^{i-1} 的值。实际上就是对 Rcon[i-1] 在 GF(2^8) 上乘以 2，获得下一个 Rcon[i]。

三、 具体实现功能说明和性能测试

在 A 和 B 两个用户端都实现了 RSA 密钥生成和传输、AES 密钥自定义和传输、加解密传输信息和加解密传输文件四个功能。

(一) RSA 密钥生成和传输

在 A 和 B 用户端各自默认存储一对 RSA 密钥和对方的 RSA 公钥，所以程序运行后，可以不生成 RSA 密钥，而使用程序默认的 RSA 密钥；也可以重新生成 RSA 密钥，并将生成的公钥发送给对方，让对方更新存储的 RSA 公钥。

默认的 RSA 密钥

```
//用户端A
BigInt A_e = "10001";
BigInt A_d = "486D6A4EA4166C757B710E92851488C04559726FA5866C5A08D3DD07
5DF5F0A96EB43F16BDC2E969812A8457BD12E32A12CF2AAAFEDF2ADFE3827FC91D956D
008BCB422C15F1854DCD1E7BD74EF98680D043A3390C3BB3F16B60D6457E91B7DDCA40
4656734898EB8A808996DCF4938CD52AB2566597E9D26E4C923DFF6A006D";
BigInt A_n = "86F9AA341FDCF2168D23CEE753BAD4169C6BC3D2148ED6A5F04DF684
343078C71B0E4610CAF747C5AA7725EBC9A7C9C2808D258E6F861FB7A89AC215A8CB7F
3382F5308B7511E65CC7538A49B6A95D970C56394D9FB784C0E9607E4303B9DDE035B7
F04C70345A09097F51269924F7D1F871A8849EBCE1A6EE8D5774015704E1";
//用户端B
BigInt B_e = "10001";
BigInt B_d = "6AEA02F6F9E139F255EA91A9D6F6A1C7F2004A4F07CE54A63E78A95E
5D6A28D3AB47D55B0B13B84B0CA903C607E8B326E80DCDCEB857B91AC08C90DEFD13A7
BDE96F962AEBAD8DD16417BF1A149516A50F3E16117ABF02B19D99B1A6CDF1A37213F6
3286856E17375D6FF4F9531A89C6A2920BC2187D4184AB84C386EABE9A2D";
BigInt B_n = "934141168AAAC5F93B2B6616CDDAD966AD71AD5E8329A90CE42C7B48
D591F7D2451C17C09A82500F4354FB07705DE35274C31CEBB3A8BE35AB7430B833E0CC
CD2CF56564F33D5FE219036C94F87255AB9D9826C65B84190BDE9F0C22120CBA9FD32B
547EA56D8806294C6E5D365177A20CACEF3591EDB2C3E7B69AB8AE0546C1";
```

因为生成大素数的过程具有随机性，经实际测量，本实验程序生成一个 512bit 大素数平均需要 5min-10min 的时间，所以建议手动输入大素数，在实验数据文件中，给出提前生成的 15 个 512bit 大素数，随机选取其中两个大素数作为 RSA 密钥生成的大素数。

图 9: A 用户端生成 1024bitRSA 密钥

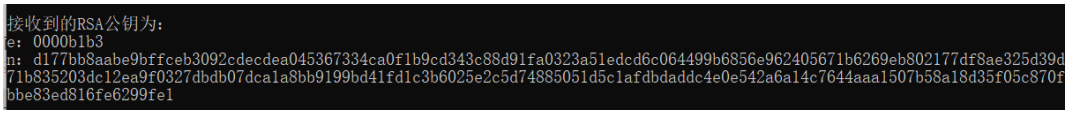

图 10: B 用户端接收 RSA 密钥

经过测量，手动输入两个大素数后，生成一对 RSA 密钥约花费 35ms。

(二) AES 密钥自定义和传输

在 A 和 B 用户端默认存储了一个 AES 密钥“AES_KEY AES_KEY ”。在程序运行后，可以使用这个默认的 AES 密钥，A 和 B 任意一方也可以重新设置 AES 密钥，并将经过 RSA 加密后的 AES 密钥发送给对方，更新 AES 密钥。

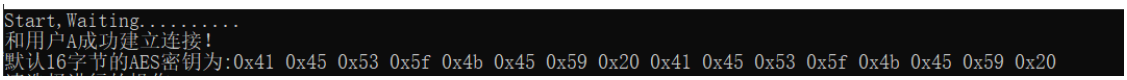

图 11: A 用户端默认存储的 AES 密钥

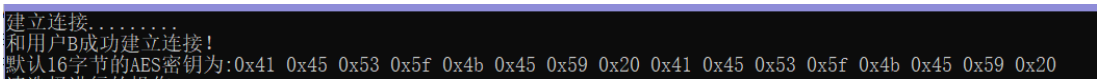

图 12: B 用户端默认存储的 AES 密钥

在 AES 密钥重置的过程中，输入一个字符串，程序会自动将其截断或填充形成 16 字节的 AES 密钥，然后对密钥使用 RSA 公钥进行加密。接收端接收到经 RSA 加密的密钥后，使用 RSA 私钥对其进行解密。

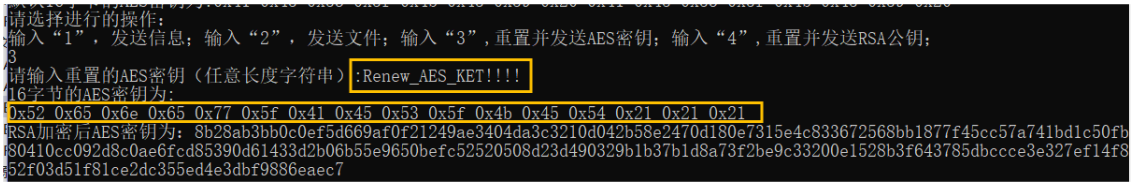

图 13: A 用户端重置并加密 AES 密钥

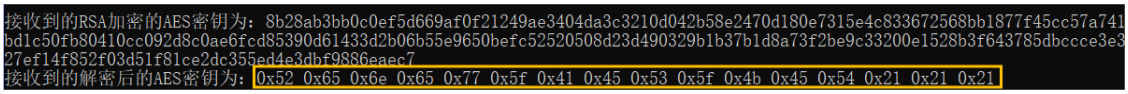

图 14: B 用户端接收并解密 AES 密钥

测量三次，接收端接收 RSA 加密的 AES 密钥并将其解密出来分别花费：2057ms；2018ms；1979ms。则接收端接收 RSA 加密的 AES 密钥并将其解密出平均花费 2018ms。

(三) 加解密传输信息

在发送端输入任意长度字符串，程序对字符串进行分组，并使用 AES 密钥在 CBC 模式下进行分组加密；接收端接收到加密信息后，使用 AES 密钥对其进行解密，获得明文。

```
请选择进行的操作:
输入“1”，发送信息；输入“2”，发送文件；输入“3”,重置并发送AES密钥；输入“4”,重置并发送RSA公钥:
1
请输入任意长度明文：Hello!My name is Geng!This is my homework!
加密结果,第1组:74 65 38 15 b9 ae bb 4b 56 ce a6 eb b6 ad d4 ba
加密结果,第2组:2b b6 3d f0 7b d7 87 f0 c5 ce 6 a7 ba 1d 1c 7f
加密结果,第3组:e8 33 d7 38 33 a7 9a 88 e6 d 5f 53 e4 de d5 1f
```

图 15: A 用户端输入明文字符串并使用 AES 密钥分组加密

```
接收到的密文,第1组:61 7e 69 a 3c f1 ed ef 4d 1c 90 e7 58 b6 0 f7    解密获得明文：Hello,my name is
接收到的密文,第2组:4b da 7 b3 8d 31 a7 e3 f4 5c 60 25 82 50 13 2b    解密获得明文： Geng.This is my
接收到的密文,第3组:d2 59 7b b6 e0 7a 87 53 b3 e0 32 10 73 f9 25 6a    解密获得明文： homework!
```

图 16: B 用户端解密输出获得的明文

经测量，加密发送长度为 16 字节的字符串平均需要 3ms-4ms；解密收到长度为 16 字节的密文也平均需要 3ms-4ms。即使用 AES 算法加密或解密一个分组数据平均需要 3ms-4ms。

(四) 加解密传输文件

首先使用 ifstream 打开文件，将文件流指针 infile 定位到文件流的开始。因为每个数据包可以传输 1024 字节，所以使用 read 函数 (infile.read(buf, 1024);) 每次从文件中读取 1024 字节的内容；并对这 1024 字节分组 AES 加密 (每组 16 字节)，将 1024 字节加密结果放入数据包中，发送到接收端；接收端接收数据包后，使用 AES 解密函数分组解密，并将解密结果使用 write 函数写入文件中 (outfile.write(buf, 1024);)。

读取文件函数

```
ifstream ReadFile()
{
    cout << "请输入要发送的文件名: ";
    cin >> filepath;
    ifstream infile(filepath, ios::in | ios::binary);//以二进制方式打开文件
    if (!infile.is_open()) {
        cout << "文件无法打开!" << endl;
        file_send = false;
    }
    else {
        file_send = true;//计算文件长度
        infile.seekg(0, std::ios_base::end);  //将文件流指针定位到流的末尾
        file_len = infile.tellg();
        original_file_len = file_len;
        totalpacket = file_len / 1024 + 1;
        cout << "文件大小为" << file_len << "Bytes,总共有" << totalpacket <<
            "个数据包" << endl;
        infile.seekg(0, std::ios_base::beg);//将文件流指针重新定位到流的开始
    }
    return infile;
};
```

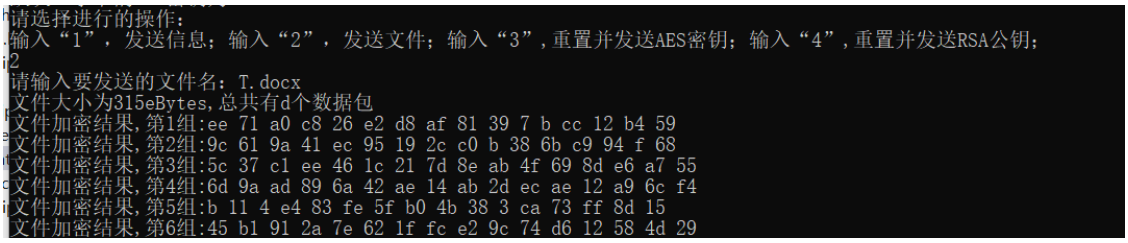

图 17: A 用户端打开指定文件并对其加密发送

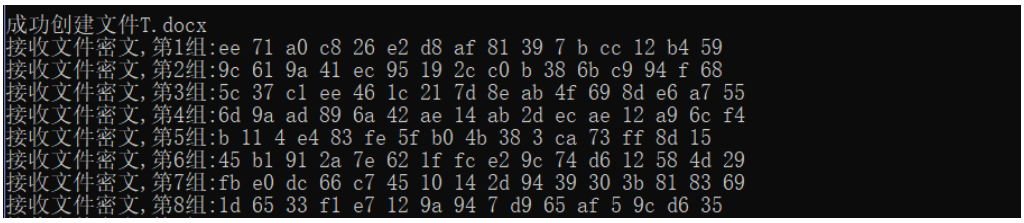

图 18: B 用户端接收密文并对其进行解密获取文件

经实验，成功将用户端 A 文件夹下的一个 docx 文件发送到用户端 B 的文件夹下。发送的 T.docx 文件大小为 12638Bytes, 总需要 13 个数据包发送，每个数据包中最多存放 65 组分组加密数据。

使用 AES 算法加密 T.docx 文件并发送总计花费 3122ms，总计分组加密 781 组数据，平均每组数据加密花费 3.997ms；接收密文并使用 AES 算法解密生成 T.docx 文件总计花费 3325ms，平均每组数据解密花费 4.257ms。

（五） 数据传输安全性测试

使用程序发送数据包，使用 WireShark 进行抓包。

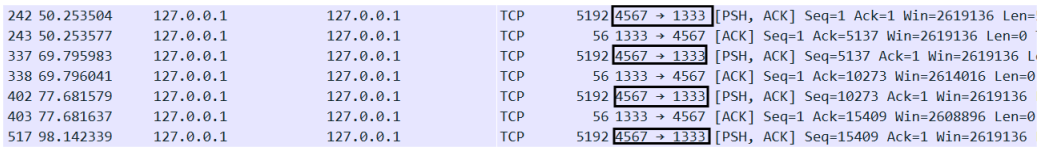

图 19: WireShark 捕获数据包

首先发送 RSA 公钥，在数据包中可以读取出 RSA 公钥密码中的 e 和 n（Windows 存储数据采用小端序；数据包中使用大端序，所以数据包数据顺序与程序中数据字节顺序相反）。但是攻击者获取这个公钥是没有意义的，利用大整数分解的难解性，攻击者很难解析出生成 n 的两个大整数，所以传输公钥不影响 RSA 算法的安全性。

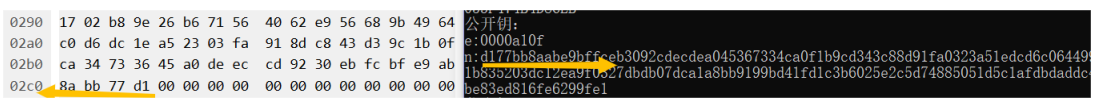

图 20: 发送 RSA 公钥

其次，发送经过 RSA 加密的 AES 密钥。在数据包中可以读取出加密后的 AES 密钥，但攻击者获得这个加密内容在没有 RSA 私钥的情况下，是没有办法解密出 AES 密钥的明文的。

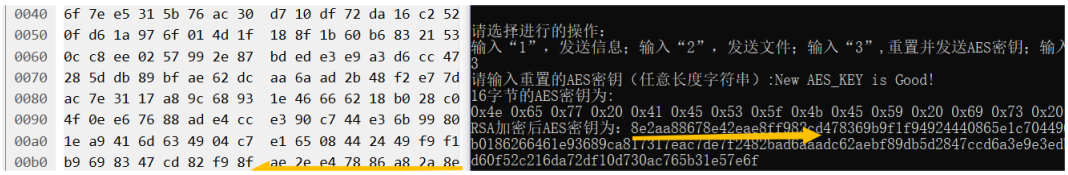

图 21: 发送经 RSA 加密的 AES 密钥

再次，发送经 AES 加密的消息数据。在数据包中读取加密数据密文后，攻击者在没有 AES 密钥的情况下，是没有办法解密出消息的明文的。

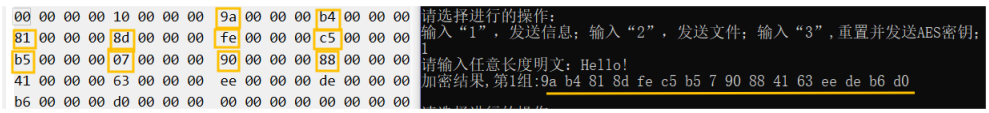

图 22: 发送经 AES 加密的消息

四、 性能优化

在本次实验中，主要实现的程序优化是大整数的运算。

- **版本一**

在初始版本中，使用 string 类型存储大整数并进行加减乘除的运算。乘法重载实现过程中，利用乘法分配率来将 a*b 转化为多个式子相加的形式求解（注意：选取 a,b 中较小的数将其转换成二进制）。例如：$20\times14 = 20\times0b1110 = 20\times2^3\times1 + 20\times2^2\times1 + 20\times2^1\times + 20\times2^0\times0 = 160 + 80 + 40 = 280$。

版本一的乘法重载

```
string Mult(string num, string s)//两个string类型十六进制数相乘
{       Bit small("");//十六进制转换成二进制字符串存储
        string temp_num;
        if (num.length() > s.length()) {
                small.setnum_bit(s);
                temp_num = num;
        }
        else {
                small.setnum_bit(num);
                temp_num = s;
        }
        string result = "";
        for (int i = small.size() - 1;i >= 0;--i)
        {  if (small.at(i))
                        result = Addition(result , temp_num);
                temp_num = Addition(temp_num, temp_num);
        }
        return result;}
```

经测量，计算 512bit 大整数乘 512bit 大整数平均需要花费 50ms；计算 512bit 大整数除以 256bit 大整数和 512bit 大整数对 256bit 大整数求余平均需要花费 30ms。

- **版本二**

为了加快运算速率，又尝试使用 int 类型的容器存储大整数（vector<int> num;），根据大整数的长度，动态决定 vector 容器大小。进行大整数乘法时，对两个乘数的大整数容器进行遍历，将大整数结果存入 uint64 容器中，然后对 uint64 容器进行遍历，将结果转换存入 int 容器中。

版本二的乘法重载

```
typedef unsigned long long uint64;
BigInteger operator*(const BigInteger& a, const BigInteger& b) {
    int lena = a.num.size();
    int lenb = b.num.size();
    vector<uint64> ansLL;
    for (int i = 0; i < lena; i++)
        for (int j = 0; j < lenb; j++)
            if (i + j >= ansLL.size())
                ansLL.push_back((uint64)a.num[i] * (uint64)b.num[j]);
            else
                ansLL[i + j] += (uint64)a.num[i] * (uint64)b.num[j];
    while (ansLL.back() == 0 && ansLL.size() != 1)
        ansLL.pop_back();
    int len = ansLL.size();
    uint64 carry = 0,temp;
    BigInteger ans;
    ans.num.clear();
    for (int i = 0; i < len; i++) {
        temp = ansLL[i];
        ans.num.push_back((temp + carry) % BigInteger::BASE);
        carry = (temp + carry) / BigInteger::BASE;
    }
    if (carry > 0)
        ans.num.push_back(carry);
    return ans;
}
```

经测量计算 512bit 大整数乘 512bit 大整数平均需要花费 10ms；计算 512bit 大整数除以 256bit 大整数和 512bit 大整数对 256bit 大整数求余平均需要花费 20ms，发现一次乘法、除法和求余的运算速度都明显加快。

- **版本三**

继续进行优化，使用 int 类型的数组存储大整数（unsigned int digit[128];），相比于容器，数组虽然会占据更多存储空间，但是可以加快运算速度；此外，在乘法和除法的过程中，使用位运算来加快运算速度。

经测量计算 512bit 大整数乘 512bit 大整数平均需要花费 8ms；计算 512bit 大整数除以 256bit 大整数和 512bit 大整数对 256bit 大整数求余平均需要花费 5ms，发现除法和求余运算速度大幅度提升。

经过优化，512bit 乘法运算的时间优化为原来的 0.16；512bit 对 265bit 的求余运算的时间优化为原来的 0.17，实现了运算速度的大幅度提升。

五、 附录：程序使用方式说明

- **建立连接**

首先，启动 B 用户端（因为将 B 作为服务器端，需要先启动），选择使用默认端口号，在出现“Start,Waiting.....”字符串后，启动 A 用户端，同样选择使用默认端口号；当两个用户端分别打印“和用户 A/B 成功建立连接!”字符串后，说明建立连接成功。

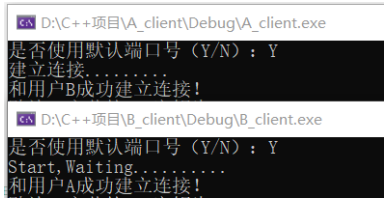

图 23: 建立连接

错误处理：如果 A 用户端打印“和用户 B 成功建立连接!”，但是 B 用户端没有打印“和用户 A 成功建立连接!”字符串，则说明未成功连接，关闭两个用户端程序，按照正确用户端启动顺序重新建立连接。如果仍未成功建立连接，检查程序中的端口号和 IP 地址，程序中默认 A 用户端端口号为：4567，B 用户端端口号为：1333，检查这两个端口号是否被占用，如果被占用，在启动 A 和 B 用户端时选择不使用默认端口号，自己输入重新设定的 A 和 B 用户端端口号，注意 A、B 用户端的目的端口号和源端口号应该相互对应。

- **发送 RSA 公钥**

如果希望重新设置 RSA 密钥，选择功能 4：重置并发送 RSA 公钥。选择这个功能后，会提示选择是否手动输入两个大素数。因为程序生成大素数的速度较慢（生成一个大素数约需要 5min-10min），所以建议使用给出的 BigPrime.txt 文档中已经提前生成好的 512bit 大素数，如果选择程序生成大素数，程序中的大素数生成函数会生成两个 512bit 的大素数。在设置完大素数后，程序中自带的 RSA 密钥生成函数会自动生成一对 RSA 密钥，并将公钥发送给另一个用户端，私钥存储在自己的用户端。

- **发送 AES 密钥**

如果希望重新设置 AES 密钥，选择功能 3：重置并发送 AES 密钥。选择这个功能后，会要求输入一个 AES 密钥字符串，输入任意长度字符串都可以，程序会自动截断或补充字符串至 16 字节。然后程序会将这个 AES 密钥加密，发送给另一个用户端。

- **发送消息**

如果希望和另一个用户端进行通信，选择功能 1：发送信息。选择这个功能后，输入任意长度的消息字符串。发送端对消息分组加密，并且发送端会打印加密结果；接收端收到消息后，对消息分组解密。

错误处理：如果发送端显示成功发送消息，但是接收端没有打印接收消息。这可能是因为接收端命令行界面卡顿，没有及时将接收到的消息打印出来，选择接收端窗口，输入回车，接收消息就可以打印出来了。

- **发送文件**

如果希望向另一个用户端发送加密文件，选择功能 2：发送文件。选择这个功能后，输入文件路径，如果文件路径正确，发送端会读取文件，进行分组加密，并将加密结果存入固定大小的数据包中发送；接收端接收数据包后，解密数据并写入文件中。

错误处理：如果在发送端发送文件之后，在接收端的目录文件下，接收到一个文件名是乱码的文件，可能是输出文件过程中，没有找到生成的目标文件，需要手动将文件重命名，正常打开即可。

六、 附录：程序文件组成

```
├─A_client.....A 用户端
│   │   A_client.exe.....A 用户端可执行文件
│   │   BigPrime.txt.....15 个 512bit 大素数
│   │   T.docx.....用于测试传输文件功能的文档
│   ├── include.....A 用户端源代码文件
│   │       BigInt.h .....大整数类
│   │       AES_data.h.....AES 算法数据结构和 S 盒等
│   │       AES_Cipher.h.....AES 算法加解密函数
│   │       AES_KeyExtend.h.....AES 算法密钥扩展函数
│   │       RSA_Cipher.h.....RSA 算法加解密函数
│   │       Packet.h.....数据传输的数据包类
│   └── src
│           BigInt.cpp
│           AES_Cipher.cpp
│           AES_KeyExtend.cpp
│           RSA_Cipher.cpp
│           A_client.cpp.....A 用户端 main 函数体
│
└─B_client.....B 用户端
    │   B_client.exe.....B 用户端可执行文件
    │   BigPrime.txt.....15 个 512bit 大素数
    │   test.jpg.....用于测试传输文件功能的图片
    ├── include.....B 用户端源代码文件
    │       BigInt.h
    │       AES_data.h
    │       AES_Cipher.h
    │       AES_KeyExtend.h
    │       RSA_Cipher.h
    │       Packet.h
    └── src
            BigInt.cpp
            AES_Cipher.cpp
            AES_KeyExtend.cpp
            RSA_Cipher.cpp
            B_client.cpp.....B 用户端 main 函数体
```