南開大學
Nankai University

计算机与网络安全学院

网路安全技术实验报告

基于 RSA 算法分配密钥的聊天程序

姓名：杨鑫

学号：2011028

专业：信息安全

2023 年 5 月 2 日

目录

1 实验要求

设计一个保密通信的聊天程序，具体要求为：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。

2 前置知识

2.1 对称加密和非对称加密

- 对称加密：加密和解密都是用同一个密钥的算法，称作对称加密。
  1. 甲方选择某一种加密规则，对信息进行加密；
  2. 乙方使用同一种规则，对信息进行解密。
- 非对称加密：加密和解密需要不同的密钥。
  1. 乙方生成两把密钥 (公钥和密钥)。公钥是公开的，任何人都可以获得，私钥则是保密的。
  2. 甲方获取乙方的公钥，然后用它对信息加密。
  3. 乙方得到加密后的信息，用私钥解密。

  在本次实验种使用对称加密 AES 算法用作会话通信时信息的加密，用非对称加密 RSA 算法用作刚开始 AES 密钥的交换工作，这是因为非对称加密算法消耗的工作量太大了，因此只用来交换密钥即可。

2.2 RSA 公钥加密

2.2.1 RSA 原理

公开密钥加密 (public-key cryptography), 也称为非对称加密，是密码学的一种算法，它需要两个密钥，一个是公开密钥，另一个是私有密钥，一个用作加密的时候，另一个则用作解密。

- 明文：需要加密的内容，称为明文。
- 密文：使用密钥把明文加密后的内容。只能用相应的另一个密钥才能解密得到原来的明文。在 RSA 算法种只能用私钥才能解密用公钥加密得到的密文。

密钥计算方法:

- 选择两个大素数 p 和 q(典型值为 1024 位，本次实验中也是设置的 1024 位)；
- 计算 n=p×q 和 z=(p-1)×(q-1)；
- 选择一个与 z 互质的数，令其为 d；
- 找到一个 e 使满足 exd= 1 (mod z)；
- 公开密钥为 (e，m)，私有密钥为 (d，m)。

加密方法:

- 将明文看成比特串，将明文划分成 k 位的块 P 即可，这里 k 是满足 2*k<n 的最大整数。

- 对每个数据块 P，计算 C= P(mod n),C 即为 P 的密文。

$$C = P^e (mod\, n)$$

解密方法:
对每个密文块 C，计算 P=Cd(mod n),P 即为明文:

$$P = c^d (mod\, n)$$

2.2.2 RSA 的安全性

从 RSA 的原理中我们知道 (d*e) mod ((p-1)*(q-1)) = 1，可以推导出 d e-1 (mod((p-1)(q-1))) 或 de 1 (mod((p-1)(q-1))) 由此我们可以看出。密码破解的实质问题是：从 p、q 的数值，去求出 (p-1) 和 (q-1)。换句话说，只要求出 p 和 q 的值，我们就能求出 d 的值而得到私钥。

当 p 和 q 是一个大素数的时候，从它们的积 p*q 去分解因子 p 和 q，这是一个公认的数学难题。比如当 p*q 大到 1024 位时，迄今为止还没有人能够利用任何计算工具去完成分解因子的任务。因此本次实验也采用的 1024 位 RSA 来保证本次实验通信协议的安全性。

RSA 算法缺点:

- RSA 算法的加密长度为 2048 位，因此对于服务端的消耗是比较大的，所以计算的速度也会比较慢，效率相对较低。
- RSA 算法比起其他的算法而言，它的安全性并不算非常的高，容易被攻击，所以它的防御能力并不高。
- RSA 算法在运行的过程之中，内容使用比较多，这也是其效率低下、消耗高的原因之一。

2.2.3 RSA 在本次实验作用

基于以上 RSA 的缺点，因此本次实验中 RSA 采用的是 1024 位密钥，且仅用作密钥交换部分加密，用来保证密钥交换的时候足够安全，能够使得服务器和客户端获得相同的会话密钥同时减少服务器开销，提高程序性能。

2.3 AES 分组密码

2.3.1 AES 原理

AES 为分组密码，分组密码也就是将明文分成一组一组的，每组长度相等，每次加密一组数据，直到加密完整个明文。在 AES 标准规范中，分组长度只能是 128 位，也就是说，每个分组为 16 个字节（每个字节 8 位）。密钥的长度可以使用 128 位、192 位或 256 位。密钥的长度不同，推荐加密轮数也不同。如图2.1所示

AES	密钥长度（32位比特字）	分组长度(32位比特字)	加密轮数
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

图 2.1: AES 不同长度的轮数

AES 的加密公式为 C=E(K,P)，在加密函数 E 中，会执行一个轮函数，并且执行 10 次这个论函数，这个轮函数的前 9 次执行的操作是一样的，只有第 10 次有所不同，也就是说一个明文分组会被加密 10 轮。AES 的核心就是实现一轮中的所有操作。AES 加密的具体流程如下图2.2所示：

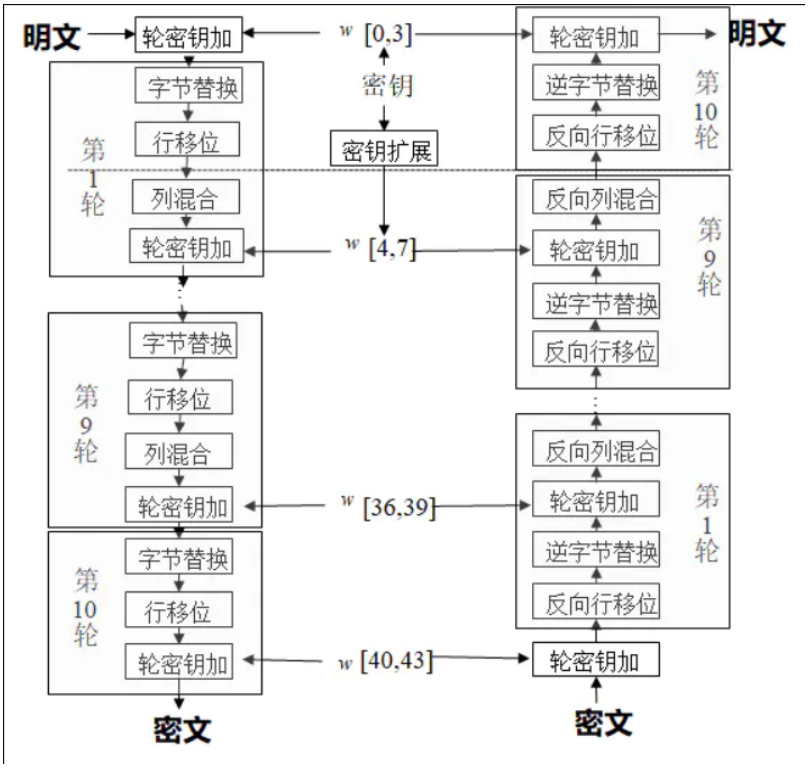

图 2.2: AES 加密流程

具体的字节代换、行移位、列混合、轮密钥加以及密钥扩展部分参考本人密码学第三次实验中 AES 算法实现，在本次大作业中不再赘述。

2.3.2 4 种模式以及特点

- 电码本（ECB）模式：电码本模式是最简单的运行模式，它一次对一个长为 64bit 的明文分组进行加密，而且每次加密秘钥都相同。如果明文消息长于 64bit，则将其分为 64bit 长的分组，如果少于 64bit，则需要填充。电码本模式最大的特性是若同一明文分组在消息中重复出现，则产生的密文分组也相同。
- 密码分组链接（CBC）模式：一次对一个明文分组加密，每次加密使用同一个密钥，加密算法的输入是当前明文分组与上一次密文分组的异或，一次加密算法的输入不会显示出与这一次明文分组

之间的固定关系。在产生第一个密文分组时，需要一个初始向量 IV 与第一个明文分组异或，IV 对于收发双方都是已知的，并且 IV 可以以明文形式发送。本次实验中也是采用的这个方式。

- 密码反馈（CFB）模式：CFB 模式可以将 DES 转换为流密码，流密码不需要对消息进行填充，而且运行是实时的，流密码具有密文和明文一样长的性质。
- 输出反馈（OFB）模式：输出反馈模式和密码反馈模式类似，不同之处在于，输出反馈模式是将加密算法的输出反馈到移位寄存器，而密码反馈模式是将密文单元反馈到移位寄存器。

2.3.3 AES 的安全性

AES 是一种典型的对称加密/解密算法，使用加密函数和密钥来完成对明文的加密，然后使用相同的密钥和对应的函数来完成解密。AES 的优点在于效率非常高，相比 RSA 要高得多。AES 密钥的编排存在一定的脆弱性。由于密钥编排的原因，AES_192 和 AES_256 的线性置换层的不完全扩散性比 AES_128 更为显著，导致针对前两种的攻击进展速度更快。利用密钥编排原理建立密钥之间存在的固有差分关系，然后利用固有差分关系建立不可能的差分路径，利用密钥、明文以及密文之间的固有的关系及猜测相应密钥来计算需要用到的相应差分关系，最后利用这些差分关系来恢复初始密钥。

2.3.4 AES 在本次实验作用

在本次实验中采用 AES_128 进行通信信息的加密。采用 RSA 进行 AES 密钥的分配后，然后使用 AES 会话密钥对通信数据进行加密。

2.4 MD5 哈希算法

2.4.1 MD5 算法原理

Hash 函数是将任意长的数字串转换成一个较短的定长输出数字串的函数，输出的结果称为 Hash 值。Hash 函数具有快速性、单向性、无碰撞性等特点。Hash 函数可用于数字签名、消息的完整性检验、消息的来源认证检测等。现在常用的 Hash 算法有 MD5、SHA-1 等。下面从 MD5 入手来介绍 Hash 算法的实现机制。MD5 算法的主要步骤如下：

第一步填充：如果输入信息的长度 (bit) 对 512 求余的结果不等于 448，就需要填充使得对 512 求余的结果等于 448。填充的方法是填充一个 1 和 n 个 0。填充完后，信息的长度就为 N*512+448(bit)；

第二步记录信息长度：用 64 位来存储填充前信息长度。这 64 位加在第一步结果的后面，这样信息长度就变为 N*512+448+64=(N+1)*512 位。

第三步装入标准的幻数(四个整数)：标准的幻数(物理顺序)是(A=(01234567) 16 ,B=(89ABCDEF) 16 ,C=(FEDCBA98) 16 ,D=(76543210) 16)。如果在程序中定义应该是(A=0X67452301L,B=0XEFCDAB89L, C=0X98BADCFEL，D=0X10325476L)。

第四步四轮循环运算：循环的次数是分组的个数（N+1），将每一 512 字节细分为 16 个小组，每个小组 64 位 (8 个字节)，然后通过四个线性函数 F、G、H、I 定义的四种操作进行四轮运算，每轮循环后，将 A，B，C，D 分别加上 a，b，c，d，然后进入下一循环。

MD5 的安全性：普遍认为 MD5 是很安全，因为暴力破解的时间是一般人无法接受的。实际上如果把用户的密码 MD5 处理后再存储到数据库，其实是很不安全的。因为用户的密码是比较短的，而且很多用户的密码都使用生日，手机号码，身份证号码，电话号码等等。或者使用常用的一些吉利的数字，或者某个英文单词。流程图如图2.3所示

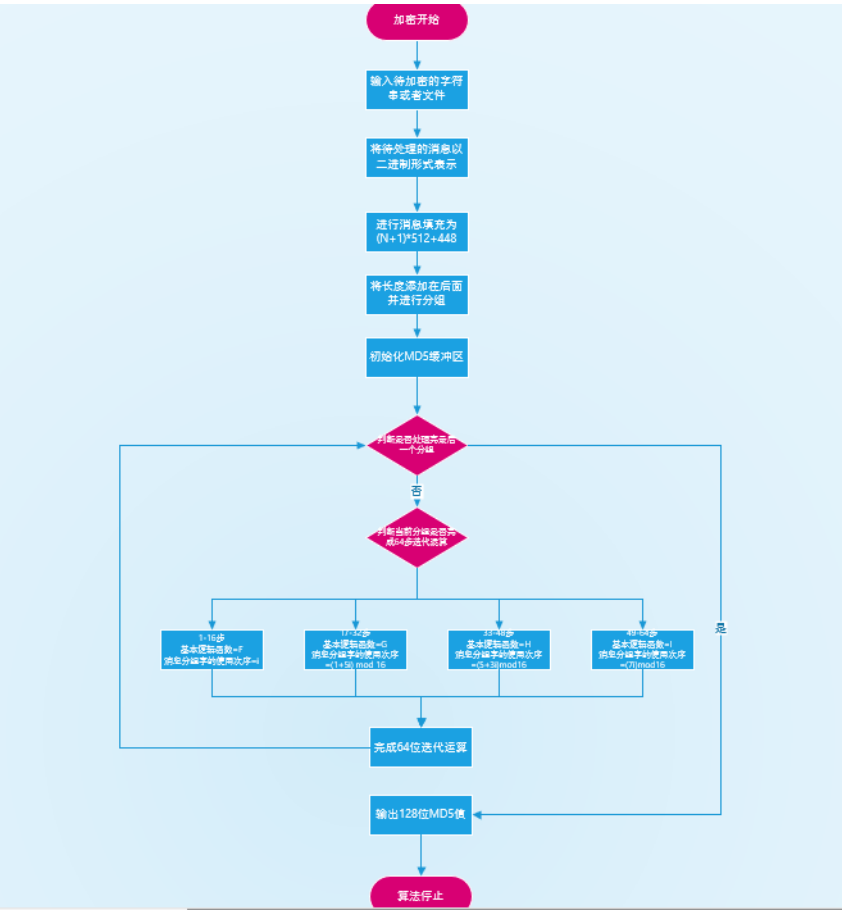

图 2.3: MD5 流程图

2.4.2 MD5 在本次实验作用

本次实验中使用 MD5 只要用于生成 AES 会话密码以及生成随机数。

3 实验实现

3.1 协议设计

本次实验的要求是设计一个保密通信协议，其利用 RSA 公钥密算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。因此本次设计的主要思路如下：

1. 通过编写客户端程序和服务端程序，利用 Windows 下的 socket 套接字来实现同一台或者不同主机上客户端与服务器进程之间数据的网路传输，用来实现密钥的分发和数据的加密传输。

2. 密钥的传输基于 RSA 算法实现，首先还需要对 RSA 算法进行优化使其生成速度加快，然后由服务器根据 RSA 算法生成公钥和私钥。

3. 通信的开始是由客户端发起的，首秀客户端发送一个请求建立连接的报文且用 MD5 生成一个 128 位的随机数 1。

4. 然后服务器端回复客户端，向客户端发送自己的公钥以及使用 MD5 生成的一个 128 位的随机数 2。

5. 客户端在收到服务器的公钥以及随机数 2 后，首先利用 MD5 生成随机数 3(预主密钥)，然后通过服务器的公钥进行 RSA 加密，然后发送给服务器。

6. 服务器在接收到客户端发送的消息后使用私钥进行解密，得到随机数 3(预主密钥)，然后客户端和服务端可以利用手上的三个随机数通过 MD5 加密得到一个相同的会话密钥。

7. 服务端发送经过会话密钥加密后的字符串"ok"，客户端接收并解密进行验证用来验证会话密钥的一致性。

8. 用此会话密钥用作 AES 的加密密钥，同时用得到的预主密钥作为刚开始的初始向量进行 AES 的 CBC 模式加密接下来的通信内容。

以上的流程可以具体用以下流程图显示，如图3.4所示

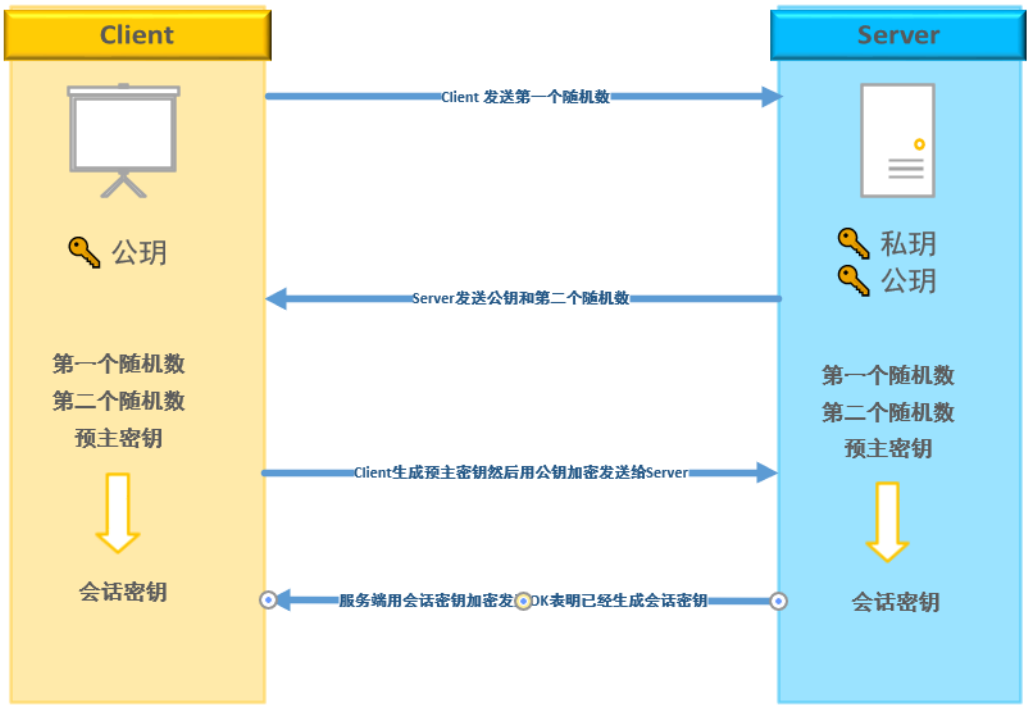

图 3.4: 本次协议设计的流程图

3.2 设计原理

3.2.1 建立 socket 套接字连接

建立 socket 连接原理：

在网络中，要全局的标识一个参与通信的进程，需要三元组：协议，IP 地址以及端口号。要描述两个应用进程之间的端到端的通信关联需要五元组：协议，信源主机 IP，信源应用进程端口，信宿主机 IP，信宿应用进程端口。为了实现两个应用进程的通信连接，提出了套接字的概念。套接字可以理解为通信连接的一端，将两个套接字连接在一起，可以实现不同进程之间的通信。

在本程序中，首先通过 socket 函数分别为客户端和服务器创建套接字，然后定义家族协议 (本次实验采用 TCP 协议)、主机地址和主机端口等，得到 socket 嵌套字，通过 bind 函数在服务器端将套接字地址与所创建的套接字号连接起来，设定套接字为监听状态，准备接收由客户端进程发出的连接请求。客户端提出与服务器建立连接的请求，如果服务器进程接受请求，则服务器进程与客户机进城之间便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。

建立 socket 连接的流程图：

本次实验建立 socket 连接的流程图如图3.5所示

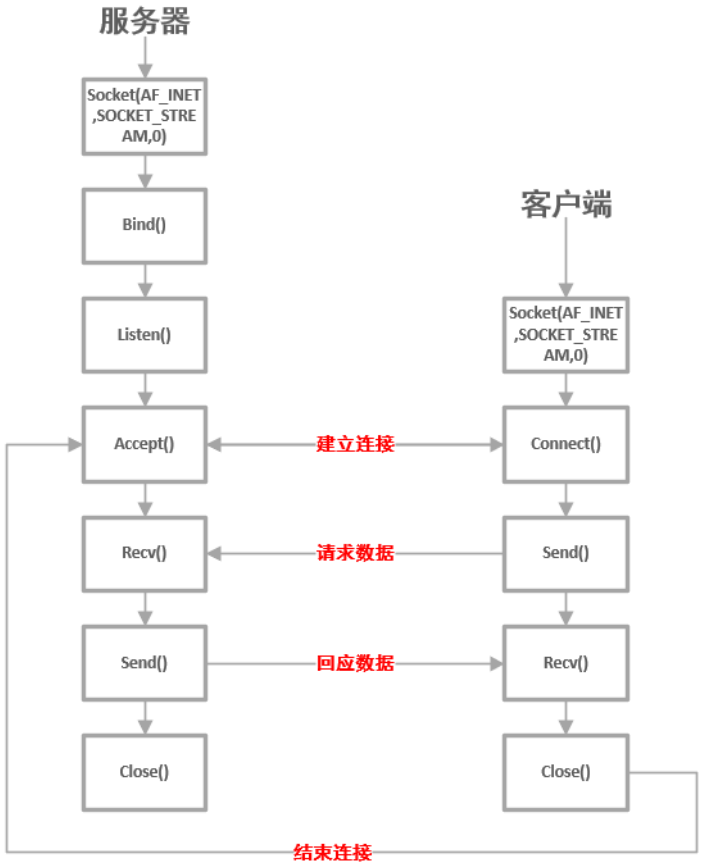

图 3.5: socket 连接流程图

定义消息结构体：

同时本次实验为了能够准确的知道消息的长度，这里定义了一个消息结构体，用来作为消息发送的协议。协议的结构体如下：

```
/*
用结构体来定义协议
*/
struct chatBody {
    int m_length;  //消息的长度
    char content[MESSAGELEN]; //接受的消息内容
```

```
};
```

建立 socket 连接的代码：

根据以上流程图，其中服务端用来建立 socket 连接代码如下：

服务端建立 socket 主要代码

```
//初始化WSA
WORD ver = MAKEWORD(2, 2);
WSADATA dat;
if (WSAStartup(ver, &dat) != 0)return -1;

/*server socket相关*/
SOCKET sev_fd;  //监听sev_fd，新的连接cli_fd
struct sockaddr_in sev_addr;  //服务器IP信息

sev_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sev_fd == INVALID_SOCKET) {
  perror("[ ERR ] Socket error!");  //分配失败退出
  exit(1);
}

memset(&sev_addr, 0, sizeof(sev_addr)); //先用0填充
sev_addr.sin_family = AF_INET;  //地址类型
sev_addr.sin_port = htons(MYPORT); //端口号
sev_addr.sin_addr.s_addr = inet_addr(IP);     //本地IP 127.0.0.1

//进行绑定
if (bind(sev_fd, (struct sockaddr*)&sev_addr, sizeof(struct sockaddr)) ==
    SOCKET_ERROR) {
  perror("[ ERRO ] Bind error!");
  exit(1);
}

//进行监听
if (listen(sev_fd, BACKLOG) == SOCKET_ERROR) {
  perror("[ ERRO ] Listen error!");
  exit(1);
}
//循环接收连接
DWORD ThreadID;
struct sockaddr_in cli_addr;
int sin_size = sizeof(struct sockaddr_in);
while (1) {
  //进行接受
  if ((m_Client = accept(sev_fd, (struct sockaddr*)&cli_addr, &sin_size)) ==
      INVALID_SOCKET) {
    perror("[ ERRO ] Accept error!\n");
    continue;
  }
```

```
    closesocket(m_Client);
}
    //关闭socket
    closesocket(sev_fd);
    WSACleanup();
```

客户端用来建立 socket 连接的主要代码如下：

客户端建立 socket 主要代码

```
WORD sockVersion = MAKEWORD(2, 2);
WSADATA data;
if (WSAStartup(sockVersion, &data) != 0)
{
  return 0;
}
m_Server = -1;
int ret = -1;
m_Server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (m_Server == -1) {
  perror("[ ERRO ] Socket error!\n");
  closesocket(m_Server);
  exit(1);
}

if (m_Server == INVALID_SOCKET)
{
  perror("[ ERRO ]Invalid socket!\n");
  return 0;
}

struct sockaddr_in serAddr;
memset(&serAddr, 0, sizeof(serAddr)); //先用0填充
serAddr.sin_family = AF_INET; //设置tcp协议族
serAddr.sin_port = htons(MYPORT); //设置端口号
serAddr.sin_addr.s_addr = inet_addr(IP); //设置ip地址
if (connect(m_Server, (sockaddr*)&serAddr, sizeof(serAddr)) == SOCKET_ERROR)
{  //连接失败
  perror("[ ERRO ] Connect error!\n");
  closesocket(m_Server);
  WSACleanup();
  return 0;
}
DWORD ThreadID;
closesocket(m_Server);

WSACleanup();
```

3.2.2 RSA 算法实现密钥加密分发

在对称加密体系中，由于对称加密的加密和解密使用的是同一个密钥，所以对称加密的安全性就不仅仅取决于加密算法本身的强度，更取决于密钥是否被安全的保管，因此加密者如何把密钥安全的传递到解密者手里，就成了对称加密面临的关键问题。比如，我们客户端肯定不能直接存储对称加密的密钥，因为被反编译之后，密钥就泄露了，数据安全性就得不到保障，所以实际中我们一般都是客户端向服务端请求对称加密的密钥，而且密钥还得用非对称加密算法加密后再传输。由于 RSA 算法是非对称加密算法，因此 RSA 算法便可以解决密钥的分发传输问题。

在本次实验中我们基于第四次实验中 RSA 算法的实现，在此基础上进行了 RSA 算法的优化，使得其能够在 2s 以内生成 512 位的 RSA，在 10s 以内能够生成 1024 位的 RSA(在 release 模式下更快，1024 位仅需 1s 左右就行生成)，从而提高我们程序的效率。

优化除法/取模：

最基础的除法算法是竖式除法，商位时用二分的方法，复杂度为 $\mathcal{O}(mn \cdot log_2(B))$（被除数的位数是 n，除数的位数是 m，存储的进制为 B）。经过测试，基于该除法的 RSA 算法，RSA-768 平均耗时为 2000ms，RSA-1024 平均耗时为 4000ms。利用 VS 的 Profiler 分析运行时间，如下图，可以发现，最大的瓶颈在于除法的计算。如图3.6所示

rsa.exe (PID: 11920)	3459 (100.00%)	0 (0.00%)	rsa.exe
[External Code]	3458 (99.97%)	688 (19.89%)	Multiple modules
__scrt_common_main_seh	3444 (99.57%)	0 (0.00%)	rsa.exe
main	3442 (99.51%)	0 (0.00%)	rsa.exe
ycdfwzy::RSA::rsa	3442 (99.51%)	0 (0.00%)	rsa.exe
ycdfwzy::RSA::isPrime	3441 (99.48%)	26 (0.75%)	rsa.exe
ycdfwzy::RSA::MillerRabin	3415 (98.73%)	0 (0.00%)	rsa.exe
ycdfwzy::RSA::qck	3413 (98.67%)	5 (0.14%)	rsa.exe
ycdfwzy::BigInt::operator%	3278 (94.77%)	0 (0.00%)	rsa.exe
ycdfwzy::BigInt::divide	3261 (94.28%)	2370 (68.52%)	rsa.exe
ycdfwzy::RSA::randomPrime	1325 (38.31%)	0 (0.00%)	rsa.exe
ycdfwzy::BigInt::leftShift	498 (14.40%)	46 (1.33%)	rsa.exe
std::vector<int,std::allocator<int> >::_Emplace_re...	405 (11.71%)	38 (1.10%)	rsa.exe
operator new	387 (11.19%)	14 (0.40%)	rsa.exe
operator delete	243 (7.03%)	5 (0.14%)	rsa.exe
std::vector<ycdfwzy::BigInt,std::allocator<ycdfwzy...	194 (5.61%)	11 (0.32%)	rsa.exe
ycdfwzy::BigInt::times	122 (3.53%)	114 (3.30%)	rsa.exe
std::vector<int,std::allocator<int> >::_Clear_and_r...	120 (3.47%)	8 (0.23%)	rsa.exe
std::vector<int,std::allocator<int> >::_Buy_raw	111 (3.21%)	5 (0.14%)	rsa.exe
ycdfwzy::BigInt::BigInt	104 (3.01%)	7 (0.20%)	rsa.exe

图 3.6: 除法运算的 CPU 消耗明显大于其他

为此，我改进了除法的计算。仍然采用竖式除法的思路，仅改良竖式商位时的方法。下面是商位的伪代码：

显然这个算法是正确的，这里的 while 循环最多只会进行 $log_2 B$ 次，实际上达到 $log_2 B$ 次是非常极端的情况，在大多数，可以认为这里只有常数次的循环。这样改良后的除法，复杂度依然是 $\mathcal{O}(mn \cdot log_2(B))$ ，但是常数比原算法小得多。

优化扩展欧几里得：

在计算公私钥时，需要用到扩展欧几里得算法，也就是计算 e 和 $\varphi(n)$ 的扩欧。$\varphi(n)$ 必然是一个大数，而 e 只要与 $\varphi(n)$ 互质，就可以任意选。我们可以检查 100000 以内的所有质数 h，只要 $\varphi(n)$ 不是 h 的倍数，就可以确定 h 与 $\varphi(n)$ 互质（这样的 h 是必然存在的，因为 100000 以内所有质数的积远

远大于 2^{1024}，所以 $\varphi(n)$ 不可能与 100000 以内的所有质数都互质）。我们就选取这样的 h 作为公钥 e，再用扩展欧几里得计算私钥 d。回忆扩展欧几里得的迭代算法，一开始计算 exgcd($\varphi(n)$, e)，第一次迭代计算 exgcd(e, $\varphi(n)$ mod e)，显然 e 和 $\varphi(n)$ mod e 都是 100000 以内的数，之后的迭代都可以在 32 位整型数字下进行运算，也就是说只有一次迭代是涉及到大整数计算的。这可以节省大量的时间。

3.2.3 AES 算法实现会话通信加密

在本程序中采用的是 **AES-128 标准，CBC 加密模式**。在该标准中，密钥长度必须为 128 位，也就是 16 字节；AES 是分组密码，也就是将需要加密的明文分成组，每个分组的长度为 128 位，同样是 16 个字节，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，最后再把每段加密数据拼起来形成最终的密文。

Padding 用来把不满 16 个字节的分组数据填满 16 个字节，有三种模式 PKCS5、PKCS7 和 NOPADDING。PKCS5 是指分组数据缺少几个字节，就在数据的末尾填充几个字节的几，比如缺少 5 个字节，就在末尾填充 5 个字节的 5。PKCS7 是指分组数据缺少几个字节，就在数据的末尾填充几个字节的 0，比如缺少 7 个字节，就在末尾填充 7 个字节的 0。NoPadding 是指不需要填充，也就是说数据的发送方肯定会保证最后一段数据也正好是 16 个字节。解密端需要使用和加密端同样的 Padding 模式，才能准确的识别有效数据和填充数据。一般采用在末尾填充 0 的方式。**本次通信协议中采用的是 PKCS5 填充方式**。

AES 具体的加密流程已经在上述介绍中详细说明了，需要注意的是本次通信协议中 AES 的密钥是经过随机数 1、随机数 2 以及预主密钥 (随机数 3) 三个形成的最后的会话密钥。在经过协议的建立后，发送方发送消息前需要经过 AES 加密信息，然后再进行发送，接收方接收到消息后进行 AES 解密才能得到原来的消息。**同时本次实验中 AES 的代码主要来源于第三次 AES 算法实验，其中需要将 ECB 模式改为 CBC 模式，本次程序中初始向量均设置为 0**。

3.2.4 MD5 算法实现随机数以及会话密钥的生成

MD5 的作用是让大容量信息在用数字签名软件签署私人密钥前被”压缩”成一种保密的格式（就是把一个任意长度的字节串变换成一定长的 16 进制数字串）。

MD5 算法的实现主要依赖于学期第五次实验，在本次实验中主要用来对得到的三个随机数进行压缩生成最后的会话密钥，同时用来生成随机数。

3.2.5 本次主要使用的函数

本次实验使用的主要函数如表1所示

方法名称	返回值类型	参数类型	方法描述
void RSA::rsa(BigInt& n, BigInt& e, BigInt& d, unsigned bitSize)	NULL	BigInt&, BigInt&, BigInt&, Int	根据最后一个 LEN 生成 RSA_LEN 的公密钥
BigInt RSA::encrypt(const BigInt& m, const BigInt& d, const BigInt& N)	BigInt	const BigInt&, const BigInt&, const BigInt&	进行 RSA 加密，根据公钥 (e,n)，加密明文 m
BigInt RSA::decrypt(const BigInt& c, const BigInt& e, const BigInt& N)	BigInt	const BigInt&, const BigInt&, const BigInt&	进行 RSA 解密，根据密钥 (d,n)，解密密文 c
string md5(string message)	string	string	md5 加密，输入字符串，得到 32 位 MD5 值
int encrypt_cbc(string plaintext, char* ciphertext, string key)	int	string, char*, string	AES 加密，CBC 模式，PKCS5 填充，输入明文和密钥，输出密文以及长度
string decrypt_cbc(char* ciphertext, int len, string key)	string	char*, int, string	AES 解密，CBC 模式，输入密文、长度以及密钥，得到解密后的明文
bool agreementBegin()	bool	NULL	本次通信协议的主要函数，用来进行通信协商
void EncrptSend(string content, string key)	void	string, string	根据传入要发送的内容以及密钥，首先用 AES 加密，然后 send 发送
DWORD WINAPI RecvThread(LPVOID args)	DWORD WINAPI	LPVOID	接收消息的线程，接收消息后通过 AES 解密得到接收到的内容
int StartChat()	int	NULL	开启服务函数，程序的界面函数

表 1: 本次程序主要函数

3.2.6 本协议安全性数学保证

本次通信中，假如小蓝和小红互相发送消息，另外小黑作为一个黑客，一直想偷听小蓝和小红之间谈话的内容，如果使用明文传输，即不使用本次通信协议，那么，小黑很容易通信截获小蓝和小红之间的数据包得到他们之间谈话的内容，如图3.7所示

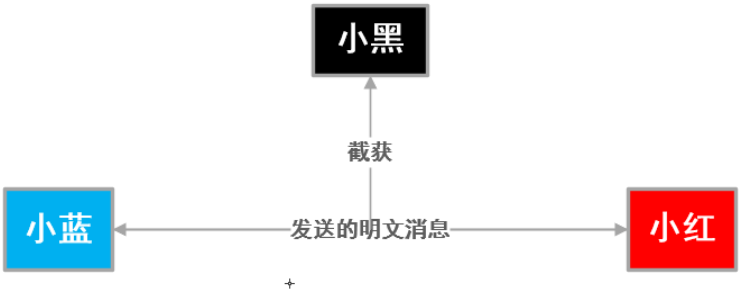

图 3.7: 明文传输小黑很容易得到谈话内容

于是如果小蓝和小红不想让小黑知道它们谈话的内容，就要使用一种只有他们俩知道的加密方式，也就是本次实现的加密通信协议。小蓝和小红各自拥有一个只有自己用的 32 位随机数 (即随机数 1 和随机数 2) 并分享给对方，同时作为服务提供的一方小红会产生 RSA 密钥对，并将自己的公钥发送给小蓝，这时候都是以明文传输的，因此小黑可以监听到随机数 1、随机数 2 以及小红的公钥。接着小蓝将产生另外一个 32 位随机数作为预主密钥即将要生成会话密钥的随机数，然后通过公钥加密后发送给小红，小红收到小蓝发送的加密信息后使用自己的私钥解密，这里小黑因为没有小红的私钥，因此是无法解密小蓝发送的内容的。然后小蓝和小红根据得到的三个随机数，进行相加然后进行 MD5 算法，得到最后的 32 位会话密钥，而小黑因为无法破解预主密钥而无法得到会话密钥，接下来小红和小蓝就可以进行保密通信了。如图3.8所示

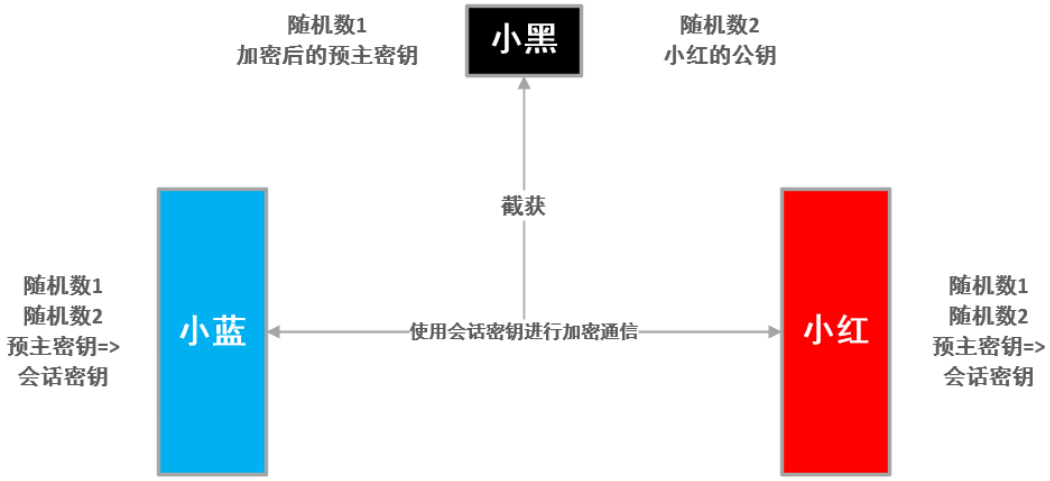

图 3.8: 使用通信协议后小黑无法监听内容

- 本次实验中采用 RSA_1024 进行加密，RSA 的安全性依赖于大整数的因数分解，对于 512 位可能容易被破解，因此采用 1024 位进行加密，进一步保证了通信的安全性。

- 本次实验中使用三个随机数相加然后取 MD5 值，这是因为 MD5 值的特性，随便改一个比特都有可能造成完全不同的结果，因此就算小黑得到了随机数 1 和随机数 2，也无法通过爆破获得会话密钥。
- **在进行消息加密前，所有进行协议连接建立的随机数以及 RSA 密钥对全部销毁，因此只剩下会话密钥作为单次会话的加密，当连接断开后会重新进行生成。**
- 本次实验中使用 AES_128，CBC 模式进行通信消息的加密，密钥位会话密钥，双方的会话密钥都是一样的因此可以保证正确的加解密，同时因为小黑无法获得会话密钥，并且通过 AES 的安全性保证，能够保证单次会话密钥的安全性。
- 随机数的产生是通过 srand 函数，其种子设置为当前时间，然后再用 MD5 值生成，这样可以防止监听者得到随机数的产生规律。

3.3 实验流程

3.3.1 socket 连接

在以上 socket 的建立中已经详细说明，这里不再赘述。

3.3.2 进行通信协议建立

在建立完 TCP 连接后，然后进行通信协议的建立即建立保密通信协议，而这些步骤主要在 **agreementBegin 函数**中进行完成。

客户端发送第一个随机数：

通过 srand 得到随机数，然后进行 MD5 加密得到一个 32 位的随机数 1。

客户端发送第一个随机数

```
//首先客户端发送第一个随机数(由随机数+MD5生成的128位)
srand((int)time(0));
string rand1 = md5(to_string(rand() % 1000));
r1 = BigInt(rand1, 16);
cout << "[ RAND1 ] Your Rand1 is => " << r1 << endl;
send(m_Server, rand1.c_str(), rand1.length(), NULL);
```

服务端接收第一个随机数，发送第二个随机数和公钥：

服务端接收到客户端发来的第一个随机数后，然后通过 RSA 函数生成自己的 RSA 公钥和私钥，然后生成第二个随机数，并保存私钥，将第二个随机数以及公钥发送给客户端。

服务端发送给客户端随机数 2 以及公钥

```
//首先接收客户端第一个随机数
char rand1[RAND_LEN / 4 + 1];
ret = recv(m_Client, rand1, RAND_LEN / 4, 0);
if (ret > 0) {
    rand1[RAND_LEN / 4] = '\0';
    r1 = BigInt(rand1, 16);
    cout << "[ RAND1 ] RecvFrom Rand1 is => " << r1 <<endl;
}
```

```
    //然后服务端生成自己的RSA公钥和私钥
    cout << "[ INFO ] Waiting to generate RSA...." << endl;
    ULONGLONG start = GetTickCount64();
    RSA::rsa(n, e, d, RSA_LEN);
    ULONGLONG finish = GetTickCount64();
    cout <<"[ RSA_1024 ] " << finish - start << " ms to generate RSA numbers" << endl;
    cout << "[ RSA_N ] N: 0x" << n << endl;
    cout << "[ RSA_E ] Public key: 0x" << e << endl;
    cout << "[ RSA_D ] Private key: 0x" << d << endl;

    //然后服务端生成第二个随机数，并将第二个随机数以及公钥(e,N)发送给服务端
    srand((int)time(0));
    string rand2 = md5(to_string(rand() % 1000));
    r2 = BigInt(rand2, 16);
    cout << "[ RAND2 ] Your Rand2 is => " << r2 << endl;
    send(m_Client, rand2.c_str(), rand2.length(), NULL);

    cout << "[ RSA_E ] Your Public Key is => " << e << endl;
    memset(sendBody, 0, sizeof(chatBody));
    sendBody->m_length = e.toString(16).length();
    strcpy(sendBody->content, e.toString(16).c_str());
    send(m_Client,(char*)sendBody, e.toString(16).length() + HEADERLEN, NULL);
    //cout << "sendbody" << sendBody->m_length << sendBody->content << endl;
    cout << "[ RSA_N ] Your Public N is => " << n << endl;
    memset(sendBody, 0, sizeof(chatBody));
    sendBody->m_length = n.toString(16).length();
    strcpy(sendBody->content, n.toString(16).c_str());
    send(m_Client, (char*)sendBody, n.toString(16).length() + HEADERLEN, NULL);
```

这里进行打印在屏幕上只是为了进行实验过程的展示，在实际运行中可以去掉打印。

客户端接收随机数 2 和公钥，生成预主密钥并加密发送：

客户端在接收到服务端发送过来的随机数 2 和公钥后，生成第三个随机数也就是预主密钥，然后通过得到的 RSA 公钥进行加密然后发送给服务端。

客户端生成预主密钥并加密发送

```
    //然后客户端接收第二个随机数
    char rand2[RAND_LEN / 4 + 1];
    ret = recv(m_Server, rand2, RAND_LEN / 4, 0);
    if (ret > 0) {
        rand2[RAND_LEN / 4] = '\0';
        r2 = BigInt(rand2, 16);
        cout << "[ RAND2 ] RecvFrom Rand2 is => " << r2 << endl;
    }

    //然后客户端接收服务端发送来的RSA公钥(e,N)
    ret = recv(m_Server, message, MAXDATALEN, 0);
    if (ret > 0) {
        recvBody = (chatBody*)message;
```

```
        int length = recvBody->m_length;
        string rsa_e = recvBody->content;
        rsa_e = rsa_e.substr(0, length);
        e = BigInt(rsa_e, 16);
        cout << "[ RSA_E ] RecvFrom RSA Public E is => 0x" << rsa_e << endl;
    }

    ret = recv(m_Server, message, MAXDATALEN, 0);
    if (ret > 0) {
        recvBody = (chatBody*)message;
        int length = recvBody->m_length;
        string rsa_n = recvBody->content;
        rsa_n = rsa_n.substr(0, length);
        n = BigInt(rsa_n, 16);
        cout << "[ RSA_E ] RecvFrom RSA Public N is => 0x" << rsa_n << endl;
    }

    //然后客户端生成第三个随机数也就是预主密钥，通过RSA_E进行加密
    srand((int)time(0));
    string rand3 = md5(to_string(rand() % 1000));
    r3 = BigInt(rand3, 16);
    cout << "[ RAND3 ] Your Rand3 is => " << rand3 << endl;
    //进行RSA加密quit
    BigInt c = RSA::encrypt(r3, e, n);
    //cout << "[ INFO ] Encrypt(RSA) Rand3 is => " << endl;
    sendBody->m_length = c.toString(16).length();
    strcpy(sendBody->content, c.toString(16).c_str());
    send(m_Server, (char*)sendBody, c.toString(16).length() + HEADERLEN, NULL);
    Sleep(10);
    if (rand1 == "" || rand2 == "" || rand3 == "") {
        cout << "[ ERRO ] Haven't recvieve one of rands!" << endl;
        return false;
    }
```

这里进行打印在屏幕上只是为了进行实验过程的展示，在实际运行中可以去掉打印。

客户端/服务端生成会话密钥：

在客户端和服务端都得到三个随机数后就可以生成会话密钥了，这里生成会话密钥的过程为：三个随机数相加然后进行取 MD5 值。这里因为客户端/服务端如果中间消息没有被篡改，那么三个随机数应该都是一样的，因此采用这种方法得到的会话密钥都应该是一样的。

生成会话密钥

```
    session_key = md5((r1 + r2 + r3).toString(16));
    cout << "[ SESSION_KEY ] Session Key is => " << session_key << endl;
```

这里进行打印在屏幕上只是为了进行实验过程的展示，在实际运行中可以去掉打印。

服务端使用会话密钥加密发送 OK 进行验证：

在双方计算得到会话密钥后，由于不知道信息是不是半路被截获更改了从而导致会话密钥不一致

的情况，因此服务端利用会话密钥对”ok” 字符串进行 AES 加密后发送给客户端，然后客户端利用会话密钥进行解密，如果得到的解密结果为”ok”，说明会话密钥一致，且中间没有经过篡改，因此可以进行加密通信了，如果没有收到或者解密结果不正确，则退出本次通信 (因为信息被篡改或者会话密钥不一致)。

服务器发送加密后的”ok”

```
string ok = "ok";
EncrptSend(ok, session_key);
```

加密发送函数 EncrptSend

```
void EncrptSend(string content, string key) {
    chatBody* sendBody = new chatBody;
    char ciphertext[MESSAGELEN];
    int len = encrypt_cbc(content, ciphertext, key);
    sendBody->m_length = len;
    strcpy(sendBody->content, ciphertext);
    //通过send将得到的密文发送出去
    send(m_Client, (char*)sendBody, strlen(ciphertext) + HEADERLEN, NULL);
}
```

客户端接收密文并解密对比

```
//接收服务器OK
ret = recv(m_Server, message, MAXDATALEN, 0);
if (ret > 0) {
    recvBody = (chatBody*)message;
    string result = decrypt_cbc(recvBody->content, recvBody->m_length,
        session_key);
    cout << "ok => "<<  result << endl;
    if (result != "ok")
        return false;
}
```

在进行完以上工作后，客户端和服务端就可以正常进行保密通信了!，同时需要注意的在以后的通信中加密发送要使用 EncrptSend 函数，解密要使用 decrypt_cbc 函数，并且密钥为当前的会话密钥。

3.4 实验结果

首先启动服务端，可以发现服务端开启后在等待客户端的连接，如图3.9所示

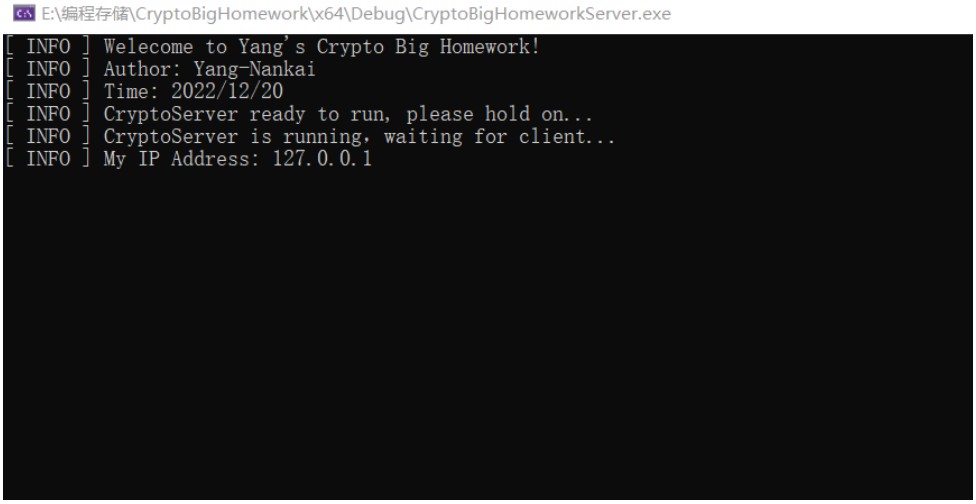

图 3.9: 服务端启动

客户端主动与服务端建立 TCP 连接，连接建立后客户端向服务端发送第一个随机数，如图3.10所示

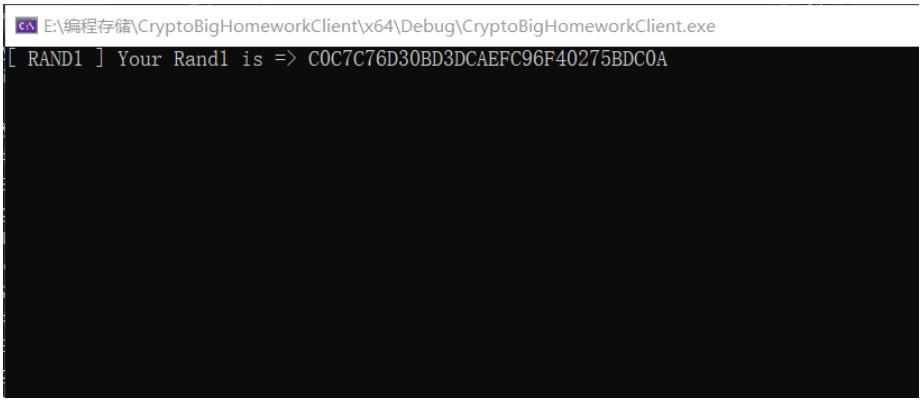

图 3.10: 客户端发送第一个随机数

服务端在接收到第一个随机数后，生成第二个随机数以及生成 RSA 密钥对，并向客户端发送公钥和第二个随机数，如图3.11所示

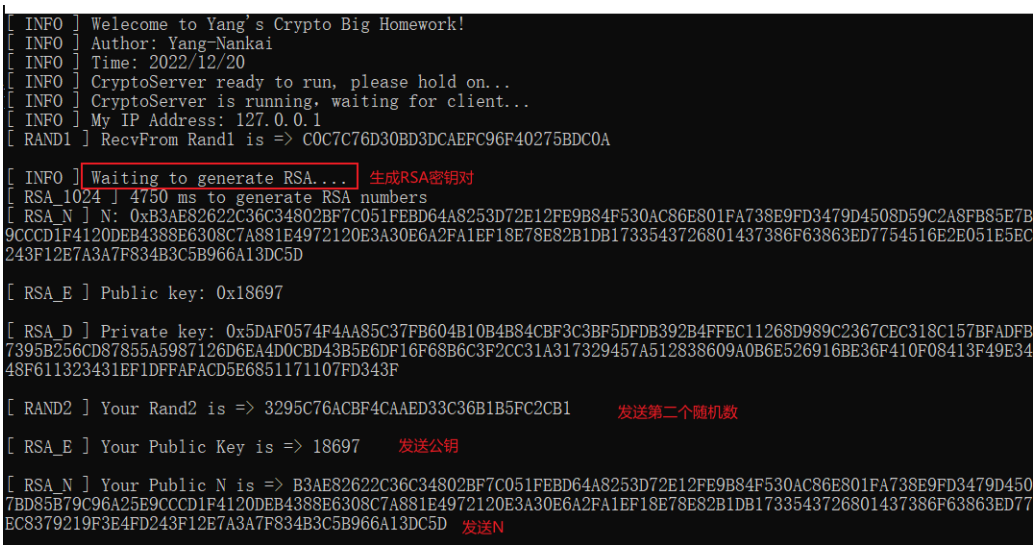

图 3.11: 服务端发送公钥和第二个随机数

客户端在接收到第二个随机数以及公钥对 (e,N) 后，生成预主密钥，并通过公钥进行加密发送，在图中展示的随机数的值，在发送前是进行加密了的，因为加密后不一定能打印出来因此图中就没有展示，如图3.12所示

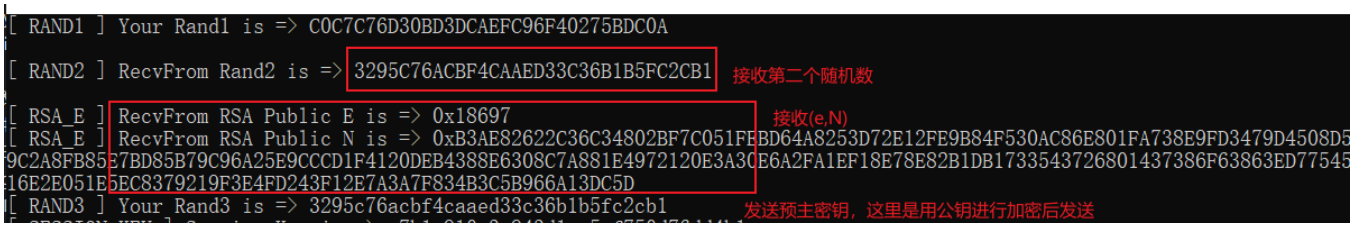

图 3.12: 客户端发送预主密钥

服务端接收到消息后进行解密得到预主密钥，然后生成会话密钥并加密字符串”ok” 进行发送，如图3.13所示

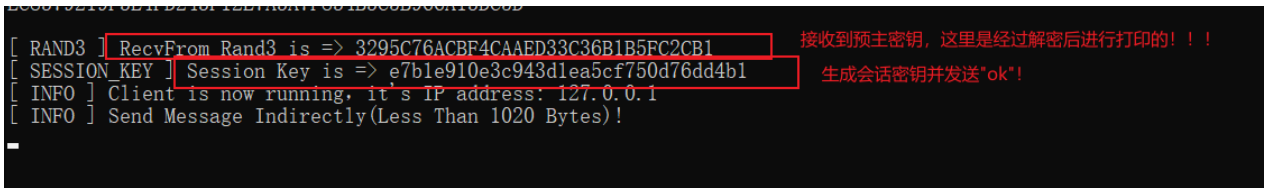

图 3.13: 服务端生成会话密钥并验证

客户端生成会话密钥后接收服务端消息并解密，判断是否是字符串”ok”，如果是就可以进入保密通信了，如图3.14所示

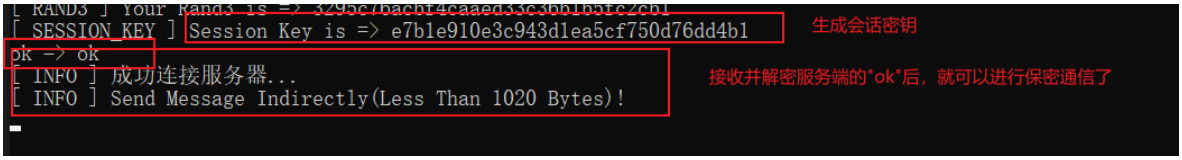

图 3.14: 客户端生成会话密钥并验证

然后客户端可以向服务端发送经过加密后的消息了，如图3.15所示

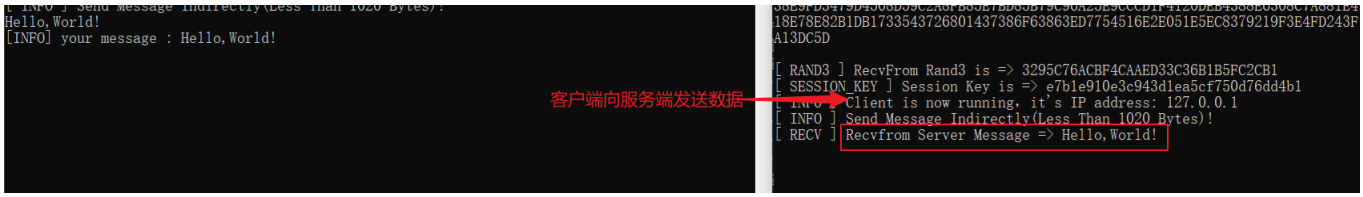

图 3.15: 客户端向服务端发送消息

同时服务端也可以向客户端发送经过加密后的消息了，如图3.16所示

图 3.16: 服务端向客户端发送消息

当客户端退出当前会话后，然后重新与服务端建立新的会话的时候，可以发现两次的会话密钥是完全不同的，如图3.17所示

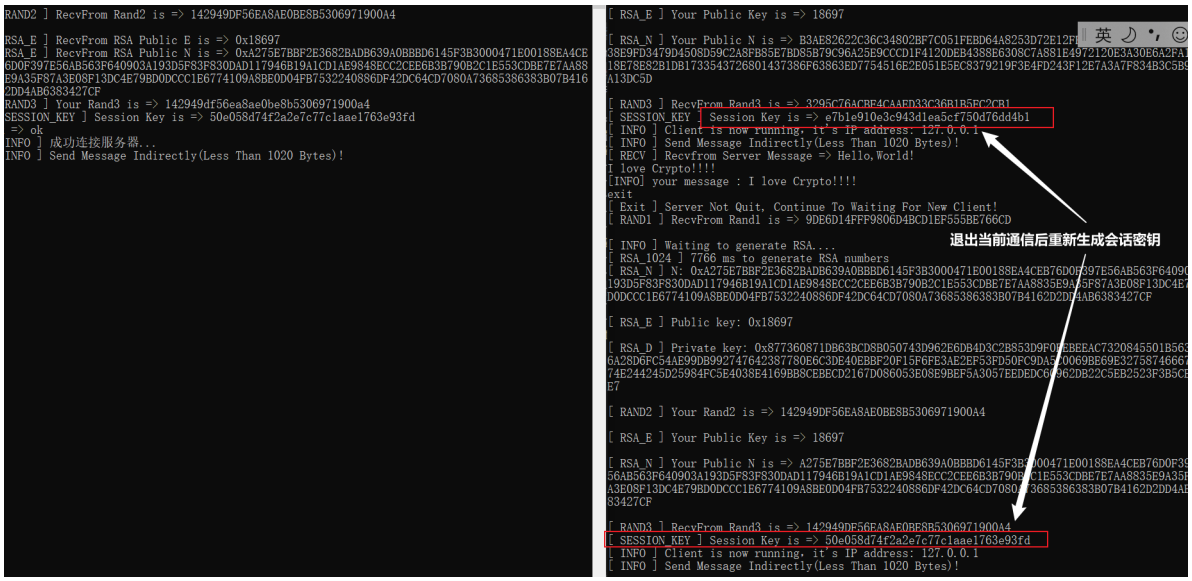

图 3.17: 两次会话的会话密钥不同

这里为了进行验证，使用 wireshark 工具进行抓包，然后验证我们的保密通信协议的正确性：

首先是客户端发送到服务端的 Rand1 是明文传输，如图3.18所示

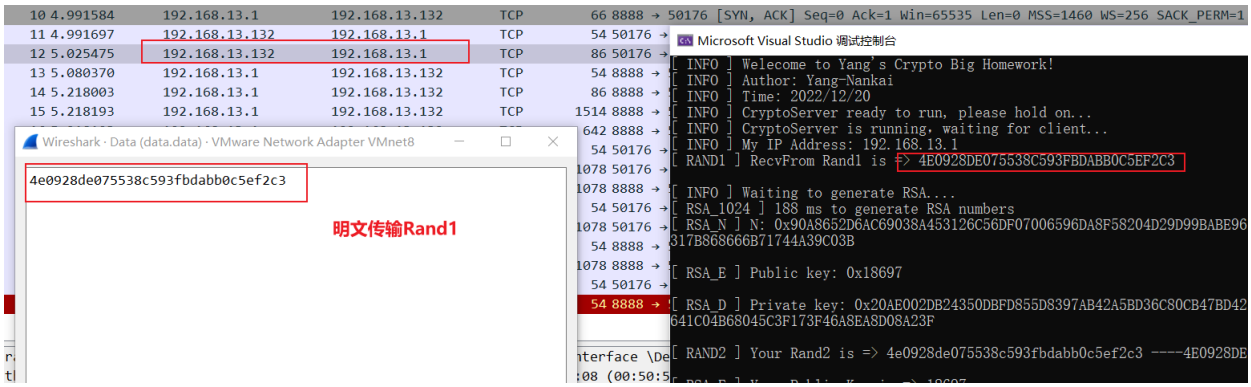

图 3.18: Rand1 明文传输

然后 Rand2 以及公钥对 (e,N) 都是明文传输，如图所示

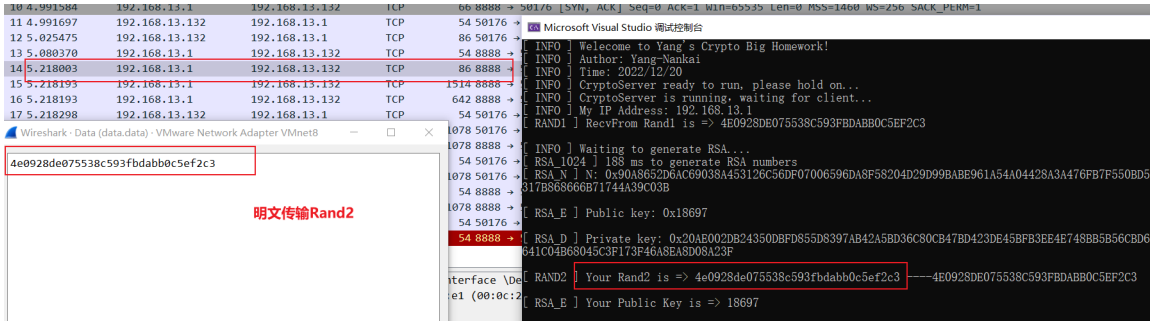

图 3.19: Rand2 明文传输

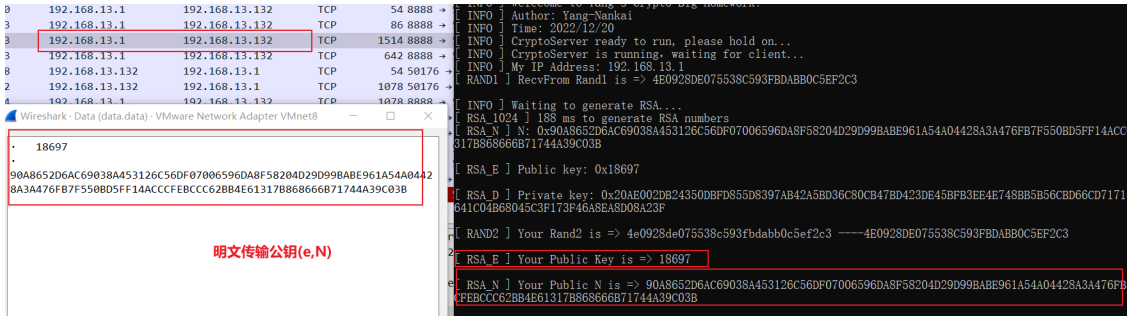

图 3.20: (e,N) 公钥对明文传输

在客户端得到公钥对后便可以加密传输经过 RSA 加密后的预主密钥，如图3.21所示

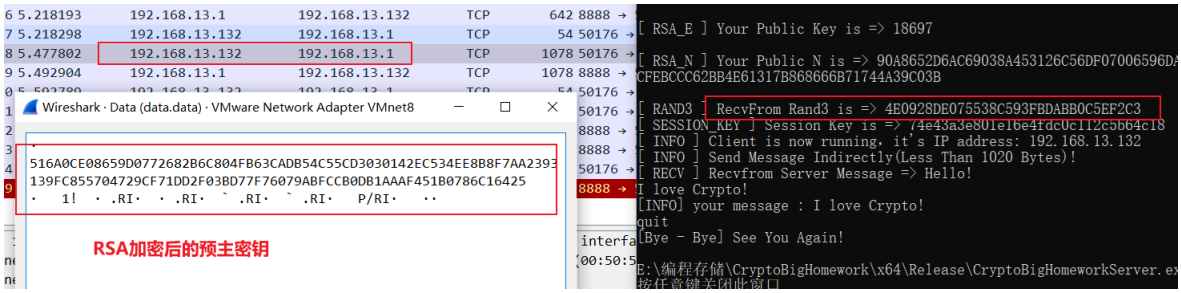

图 3.21: RSA 加密传输预主密钥

然后两边生成会话密钥，服务端使用会话密钥对字符串“ok”进行加密后发送给客户端，可以发现内容是加密后的内容，是不可读的，如图3.22所示

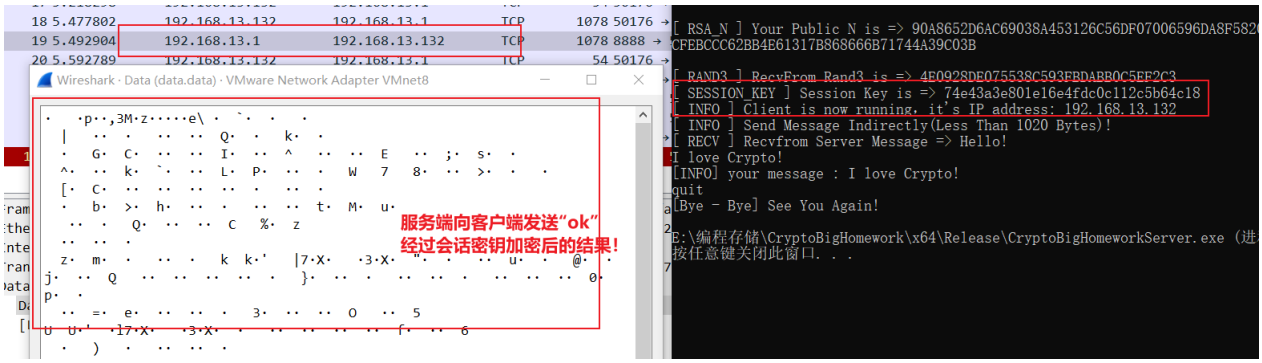

图 3.22: 传输加密后的“ok”

客户端发送消息“Hello!”给服务端，使用会话密钥进行加密，可以发现加密后的内容是不可读的，需要进行解密才能知道原始内容是什么，如图3.23所示

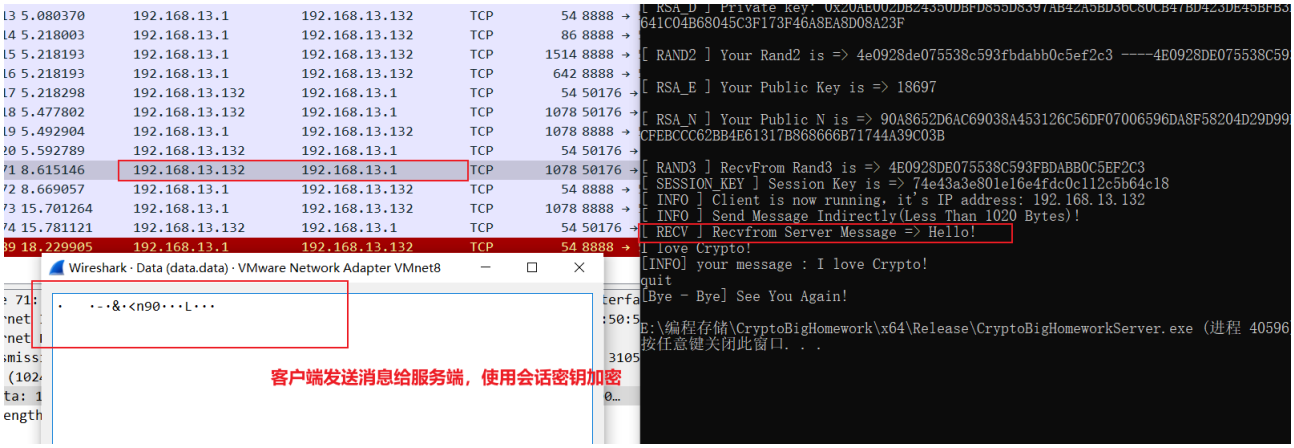

图 3.23: 会话密钥加密通信内容

3.5 实验改进

在本次实验中本人发现以下几点还可以进行改进：

- 解决 TCP 粘连问题，TCP 会将较小的数据包合并发送，因此可能会造成接收方接收错误，如图3.24所示。因此这里需要对这个问题进行一些改进，在本次实验中为了解决这个问题，是将发送的 sendbody 中 content 多余的内容填充 0 处理。
- 未进行消息认证，在本次实验中如果黑客伪造成服务端与客户端进行建立连接，然后再伪造成客户端与服务端建立连接，那么可以通过两个会话密钥进行监听。
- 生成会话密钥的算法比较简单,这里生成会话密钥的算法是根据三个随机数相加后然后采用 MD5 值得到，因此在黑客得到两个随机数的时候，容易经过 MD5 碰撞得到会话密钥 (但实际上因为 MD5 的特性因此也不太简单比较安全)，因此如果需要更加安全的话可以采用 SHA-256 等更长长度的哈希算法。

图 3.24: TCP 粘连问题

3.6 实验总结

通过本次实验，我学会了如何去设计一个保密通信协议并通过编程的方法去实现。在协议设计中需要考虑安全性，高效性以及准确性，通过对实现的 RSA、AES、MD5 等算法进行利用并改进 (对 RSA 算法进行优化使得能够更快生成密钥对和加解密)，然后利用公钥交换的设计思想，最终设计出本次实验的保密通信协议。在代码的编写中也不断遇到困难，通过调试逐渐发现了问题并解决。

4 附录

4.1 仓库

本次实验的全部代码、实验报告、源程序、图片等全部上传至本人的 **Github** 中：Github