



南開大學
Nankai University

网络空间安全学院
网络安全技术实验报告

实验二：基于 **RSA** 算法自动分配密钥
的加密聊天程序

学院：网安学院

年级：2021 级

班级：信息安全一班

学号：2111408

姓名：周钰宸

手机号：13212168838

2024 年 4 月 16 日

目录

1 实验目标	3
2 实验原理	3
2.1 DES	3
2.2 RSA	4
2.2.1 公钥密码体系 (Public-Key Cryptography)	4
2.2.2 RSA	4
2.3 TCP	5
2.4 异步 IO (Asynchronous IO)	6
2.4.1 同步、异步、阻塞、非阻塞	6
3 实验内容	8
3.1 规范习惯	8
3.2 DES	8
3.3 RSA	9
3.3.1 初始化	9
3.3.2 加密与解密	9
3.4 TCP	9
3.5 Asynchronous IO	11
3.6 HeadquarterAndAgent	12
3.6.1 特工介绍	12
3.6.2 Special Operations	12
3.6.3 其它功能	13
4 实验步骤	13
4.1 实验模块搭建	13
4.2 DES	18
4.3 RSA	19
4.3.1 辅助函数	19
4.3.2 初始化	24
4.3.3 加密与解密	27
4.4 TCP	28
4.4.1 Server	28
4.4.2 Client	30
4.5 AIO	33
4.5.1 Single 系列类实现	33
4.5.2 SilentGuardian 优化	38
4.6 AgentTheme	40

5 实验结果	40
5.1 实验结果展示	40
5.1.1 RSA 算法正确性验证	40
5.1.2 基于 RSA 的 DES 密钥共享方案	41
5.1.3 AIO 优化	43
6 实验遇到的问题及其解决方法	44
6.1 RSA	44
6.1.1 整数溢出问题	44
6.1.2 Debug 困难	44
6.1.3 私钥 (d, n) 生成	46
6.2 DES	46
6.2.1 DES 随机密钥	46
6.3 AIO	47
6.3.1 fork 灵异事件再临	47
7 实验改善方向	48
7.1 DES	48
7.1.1 安全性增强	48
7.2 RSA	48
7.2.1 安全性增强	48
7.3 AIO	48
7.3.1 指示文字输入与输出	48
7.3.2 原生 AIO 优化	49
7.3.3 资源释放问题	49
7.4 多进程通信	49
7.4.1 AIO 实现	49
8 实验结论	49

1 实验目标

DES(Data Encryption Standard) 算法是一种用 **56 位密钥来加密 64 位数据的对称分组加密算法**，该算法流程清晰，已经得到了广泛的应用，算是应用密码学中较为基础的加密算法。

TCP(传输控制协议) 是一种面向链接的、可靠的传输层协议。TCP 协议在网络层 IP 协议的基础上，向应用层用户进程提供**可靠的、全双工的数据流传输**。

不同于对称加密系统，在公钥加密系统中，也叫非对称加密系统，加密和解密使用两把不同的密钥。加密的密钥（公钥）可以向公众公开，但是解密的密钥（私钥）必须是保密的，只有解密方知道。

RSA 加密算法是一种典型的公钥加密算法。RSA 算法的可靠性建立在分解大整数的困难性上。**只要其密钥的长度足够长，用 RSA 加密的信息的安全性就可以保证**。本次实验的目的如下：

1. 加深对 RSA 算法基本工作原理的理解。
2. 掌握基于 RSA 算法的保密通信系统的基本设计方法。
3. 掌握在 Linux 操作系统实现 RSA 算法的基本编程方法。
4. 了解 Linux 操作系统异步 IO 接口的基本工作原理。

最终达到的要求为：

1. 要求在 Linux 操作系统中完成基于 RSA 算法的自动分配密钥加密聊天程序的编写。
2. 应用程序保持第三章“基于 DES 加密的 TCP 通信”中示例程序的全部功能，并在此基础上进行扩展，**实现密钥自动生成，并基于 RSA 算法进行密钥共享**。
3. 要求程序实现全双工通信，并且加密过程对用户完全透明。
4. **使用 select 模型或者异步 IO 模型**对“基于 DES 加密的 TCP 通信”一章中 socket 通讯部分代码进行优化。

本次实验，我按照上述实验要求，依然围绕我的**特工主题**。并在上一次实验的基础上进一步完善，完成了本次实验基于 RSA 算法自动分配密钥的加密聊天程序。并最终使用异步 IO 模型对 TCP 聊天通信进行了优化。

2 实验原理

2.1 DES

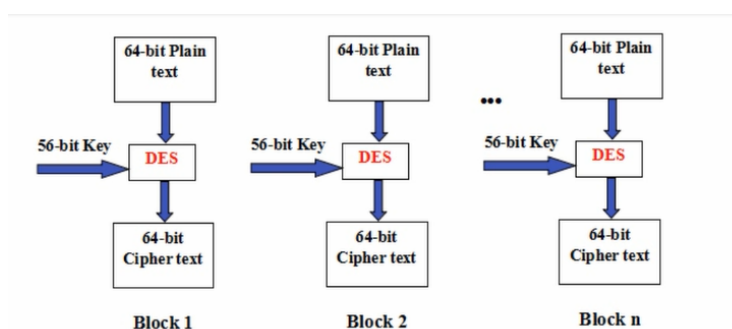


图 2.1: DES 对称加密算法

DES 算法的全部细节都是公开的，其安全性完全依赖于密钥的保密。算法包括初始置换、逆初始置换、16 轮迭代以及子密钥生成算法。上次实验中已经详细阐述过其流程这里不再赘述。

2.2 RSA

2.2.1 公钥密码体系 (Public-Key Cryptography)

1. 在公钥密码体系中，每个参与者有一对密钥：公钥 (public key) 和私钥 (private key)。
2. 公钥是公开的，可以安全地分享给任何人。而私钥必须保密，只有密钥的持有者知道。
3. 加密和解密使用不同的密钥：公钥用于加密信息，私钥用于解密。
4. 这种体系的优点是加密密钥 (公钥) 可以公开，而不会危及解密密钥 (私钥) 的安全。

2.2.2 RSA

1. RSA 算法是一种基于非对称密钥的加密算法，由 Ron Rivest、Adi Shamir 和 Leonard Adleman 在 1977 年提出。

2. 密钥生成步骤：

- 选择两个大的随机素数 p 和 q 。
- 计算 $n = p \times q$ ，其中 n 作为模数用于公钥和私钥。
- 计算欧拉函数 $\phi(n) = (p - 1) \times (q - 1)$ 。
- 选择一个整数 e ，使得 $1 < e < \phi(n)$ 并且 e 与 $\phi(n)$ 互质， e 成为公钥的一部分。
- 计算 e 的模 $\phi(n)$ 的逆 d ： $e \cdot d \equiv 1 \pmod{\phi(n)}$ ， d 成为私钥的一部分。

3. 加密和解密过程：

- 给定消息 m ，计算加密消息 c 为 $c = m^e \pmod{n}$ 。
 - 使用私钥 d 来解密，计算 $m = c^d \pmod{n}$ 。
4. RSA 算法的安全性基于大数分解的困难性。增加 p 和 q 的大小可以提高安全性。

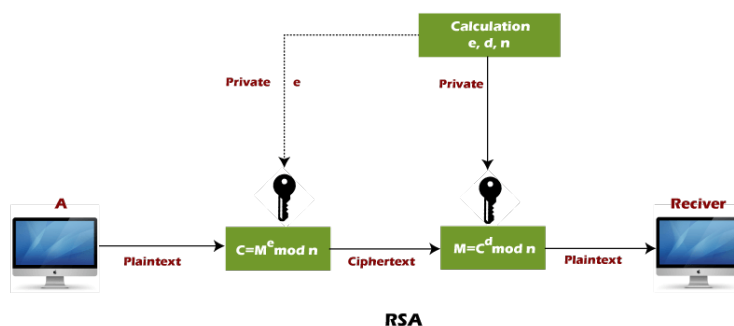


图 2.2: RSA 非对称加密算法

图2.2所示，可以看到完整的 RSA 流程。本次实验将会遵循上述流程对 RSA 算法进行复现。并将其应用到 TCP 的通信之中。

2.3 TCP

TCP 是一种传输控制协议，它是一种端对端协议。使用 TCP 通讯的两台主机必须连接在一起，每一端都要为通话做好准备。

套接字 (Socket) 是一个中间软件抽象层，作为一组接口存在。本次实验就是基于 TCP 协议的流式套接字 (SOCK_STREAM)，提供面向连接的、可靠的数据传输服务，可保证无差错、无重复且按发送顺序提交给接收方。

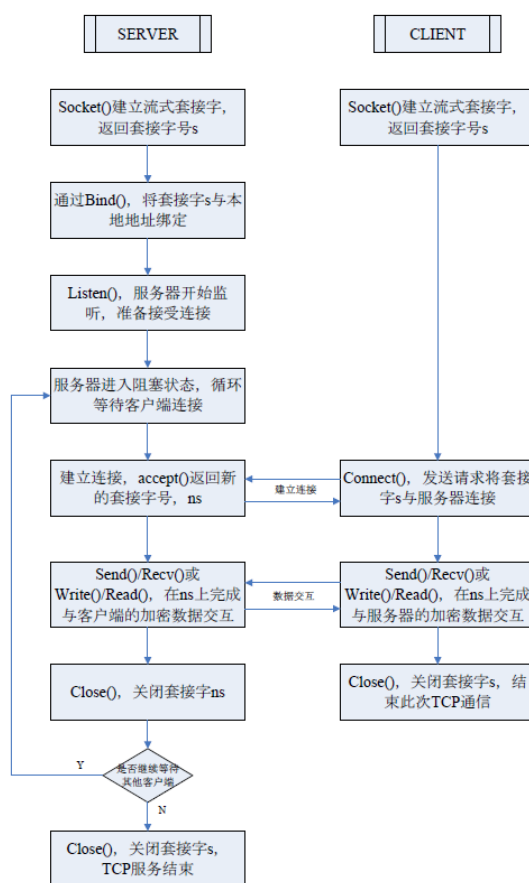


图 2.3: TCP 通信流程图

如2.3所示，具体 Socket 编程步骤如上图所示：

- **服务器端：** socket()->bind()->listen()->accept()->send()<->recv()
- **客户端：** socket()->connect()->send()<->recv()

2.4 异步 IO (Asynchronous IO)

2.4.1 同步、异步、阻塞、非阻塞

	Blocking	Non-blocking
Synchronous	Read/write	Read/write (O_NONBLOCK)
Asynchronous	i/O multiplexing (select/poll)	AIO

图 2.4: 同步、异步、阻塞、非阻塞区别

如2.4所示，很明确地指出了同步、异步、阻塞、非阻塞的划分。

一个 IO 操作其实分成了两个步骤：发起 IO 请求和实际的 IO 操作。

同步 IO 和异步 IO 的区别就在于第二个步骤是否阻塞，如果实际的 IO 读写阻塞请求进程，那么就是同步 IO。

阻塞 IO 和非阻塞 IO 的区别在于第一步，发起 IO 请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞 IO，如果不阻塞，那么就是非阻塞 IO。

所以，IO 操作可以分为 3 类：同步阻塞（即早期的 IO 操作）、同步非阻塞（NIO）、异步（AIO）。

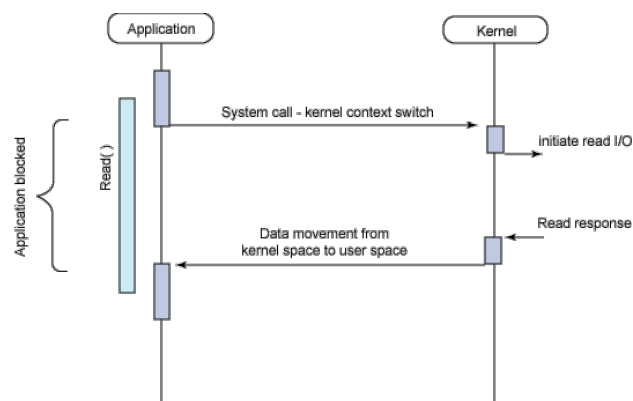


图 2.5: 同步阻塞

- **同步阻塞**：如图2.5所示。在此种方式下，用户进程在发起一个 IO 操作以后，必须等待 IO 操作的完成，只有当真正完成了 IO 操作以后，用户进程才能运行。JAVA 传统的 IO 模型属于此种方式。

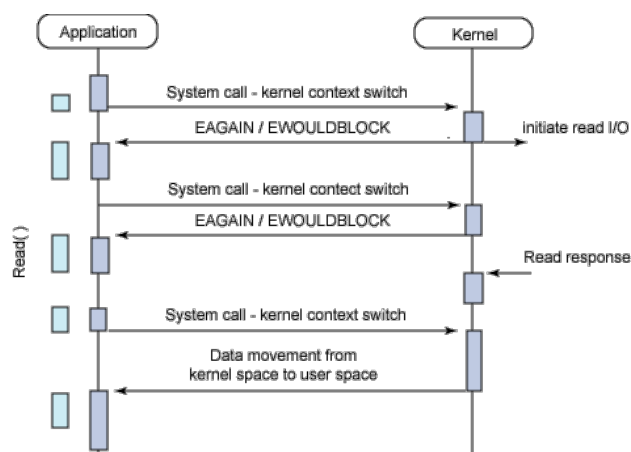


图 2.6: 同步非阻塞

- **同步非阻塞**: 如图2.6所示。在此方式下，用户进程发起一个 IO 操作以后边可返回做其它事情，但是用户进程需要时不时的询问 IO 操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的 CPU 资源浪费。其中目前 JAVA 的 NIO 就属于同步非阻塞 IO。

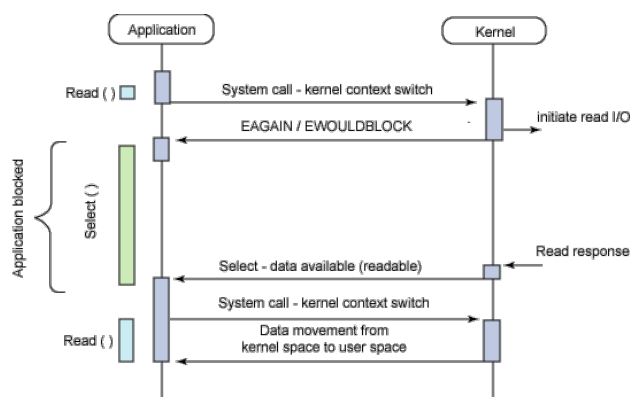


图 2.7: 异步阻塞

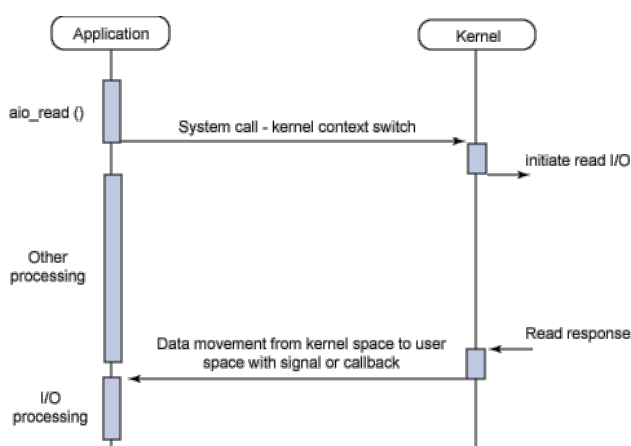


图 2.8: 异步非阻塞

- **异步**: 如图2.7和2.8所示。此种方式下是指应用发起一个 IO 操作以后，不等待内核 IO 操作的完成，等内核完成 IO 操作以后会通知应用程序。

每次 Linux 系统调用就会在内核和用户之态之间进行一次上下文切换，需要消耗系统资源，使用异步 IO 模型，整个过程中都不需要进行上下文切换，而真正进行数据操作的回调函数也都是在 IO 状态变化的时刻由内核自动调用。可见，这个模型可以消除无谓进程上下文切换所需的资源，进而大大提升系统的执行效率。因此本次实验通过异步 IO 模型最后对 TCP 通信进行了一定程度的性能优化。

3 实验内容

首先我说明本次实验的主题——特工电影。

本次实验我继续沿用上次的特工主题。在此基础上实现了全双工通信，使用 RSA 算法自动分配密钥的 DES 的加密 TCP 聊天程序。并且还提供了一个性能优化的异步 IO 版本。具体而言会介绍本次实验的主要工作。

3.1 规范习惯

由于本次实验我依旧在 Linux 平台进行编程。并依然严格要求自己，采用了非常规范的工程编程习惯。极为清晰的框架增强了我的代码的可读性和可维护性。这一点也会在具体实验步骤中展开描述。

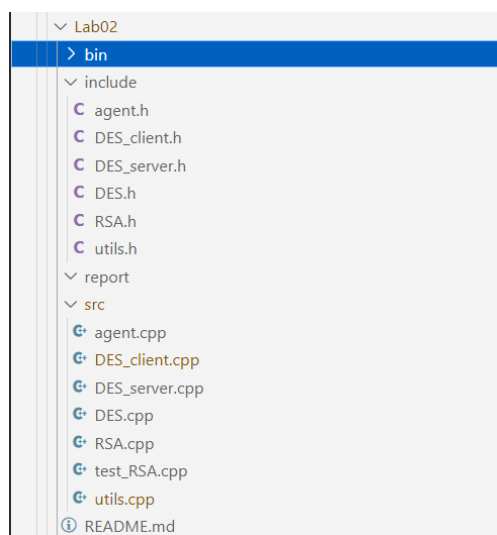


图 3.9: 本次实验规范编程框架

3.2 DES

在 DES 的实现中，为了尽可能优化代码的时间复杂度，本次实验的 DES 代码全部采用了 C++ 底层的位运算实现。并且代码逻辑模块清晰，可读性较强。具体的模块实现基本和上从实验保持一致，这里就不再赘述。本次实验新添加的部分会在实验步骤部分详细阐述。

1. Padding 和 Unpadding
2. KeyGeneration
3. Encryption
4. Decryption

3.3 RSA

RSA 部分是本次实验的重点。同样为了尽可能优化代码的时间复杂度，我通过多个辅助函数接口，高效计算并同时避免溢出完成了实现。这里会简略介绍我的 RSA 模块。具体实现详见实验步骤部分。

3.3.1 初始化

1. 通过调用写好的素数检测算法（暂时只实现了 Miller Rabin）进行大素数生成，而后通过一些给定参数对 RSA 的 p, q, n, ϕ 进行初始化。
2. 通过拓展欧几里得算法生成符合条件的 e ，生成 RSA 公钥 (e, n) 。
3. 通过拓展欧几里得算法生成符合条件的 d ，即与 e 相对于 ϕ_n 互为逆元。生成 RSA 私钥 (d, n) 。

3.3.2 加密与解密

首先通过高效并避免溢出地实现了模乘和模幂算法，然后在加密和解密中调用模幂算法。

1. 加密： $c = m^e \bmod n$ 。
2. 解密： $m = c^d \bmod n$ 。

流程参照2.2中所示。其中加密过程对用户完全透明。

3.4 TCP

本次实验的 TCP 通讯依旧直接使用流式套接字 socket 接口。不过在这基础的 TCP 通信上使用 DES 加密，并通过 RSA 自动分配密钥。具体流程为：

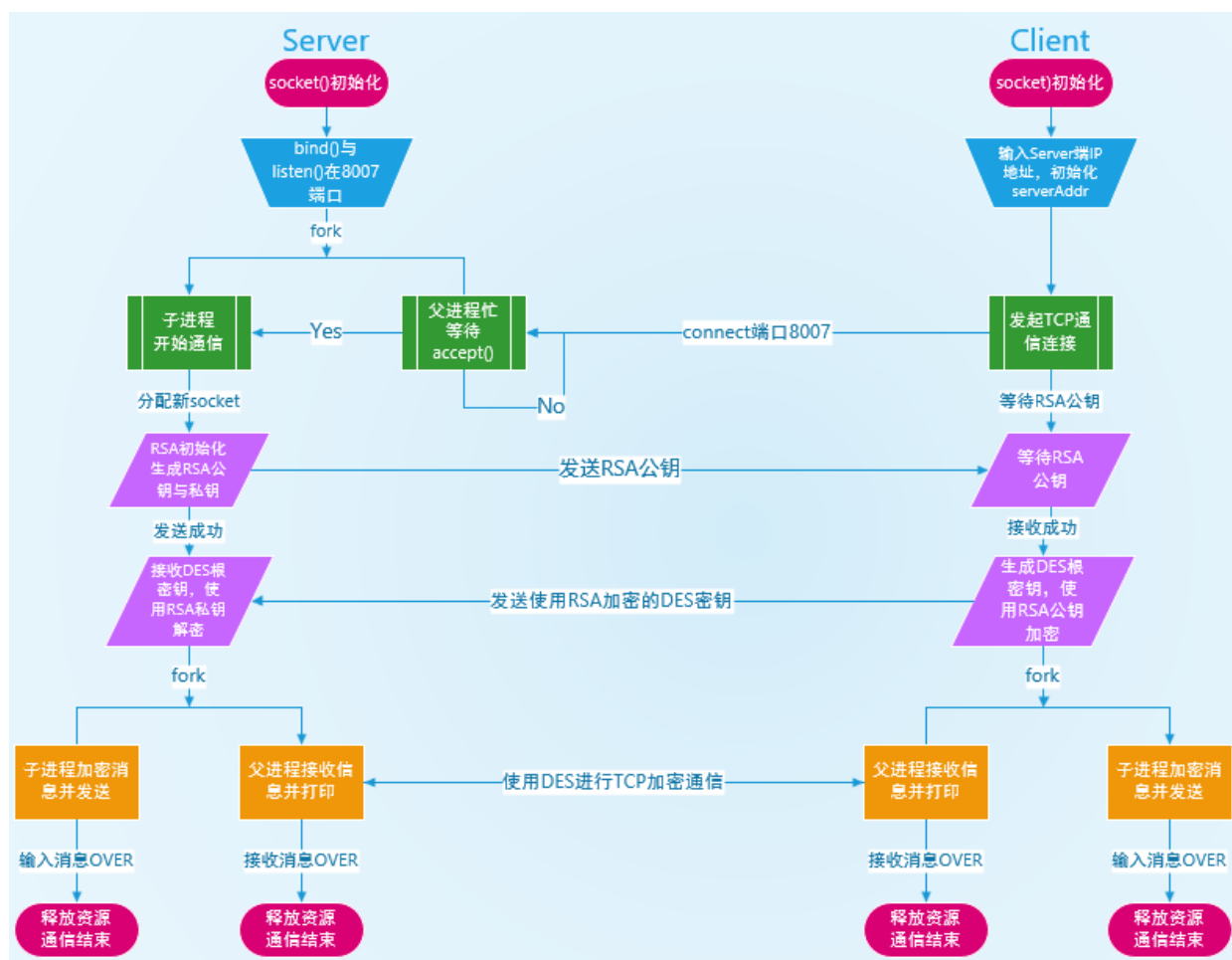


图 3.10: 基于 RSA 自动分配密钥的通信流程

如图3.10所示。在上次实验的基础上，进行改进后的加密通信流程 (这里只介绍 Silent Guardian):

1. 服务器和客户端初始化初始化网络环境，服务器端绑定 socket 并监听，客户端请求连接。
2. 服务器端进行 RSA 初始化，根据提示选择加密的一些参数，包括是否使用默认公钥，进行高级加密等。然后生成 RSA 的公钥和私钥。
3. 服务器端不通过加密直接将 RSA 公钥发送给客户端。客户端接收后利用其设置自己的 RSA 参数。
4. 客户端进行 DES 初始化，随机生成 DES 根密钥和 16 轮密钥后，利用 RSA 公钥加密 DES 根密钥并发送给服务器端。
5. 服务器端接收加密后的 DES 根密钥，通过 RSA 私钥进行解密，解密后利用 DES 根密钥生成自己的 16 轮密钥，理论上应该和客户端相同。
6. 双方通过共享的 DES 进行加密通信，双方都可以发送和接收消息。实现了全双工通信。
7. 服务器端或者客户端通过输入大写的 OVER 退出并通知另一端，本次通讯结束。
8. 服务器端实现了一次登录多次通讯，即可以继续等待下一位客户端的并进行 accept。

上述过程中实现了全面的错误处理，对于各种连接过程的异常情况都进行了处理。

3.5 Asynchronous IO

本次实验在实现了基于 RSA 的自动分配 DES 密钥后，进一步想到了通过 Linux 的异步 IO 机制对程序效率进行优化。准确说是对 DES 双全工加密通信部分实现了效率优化。优化后的 TCP 通信聊天流程为：

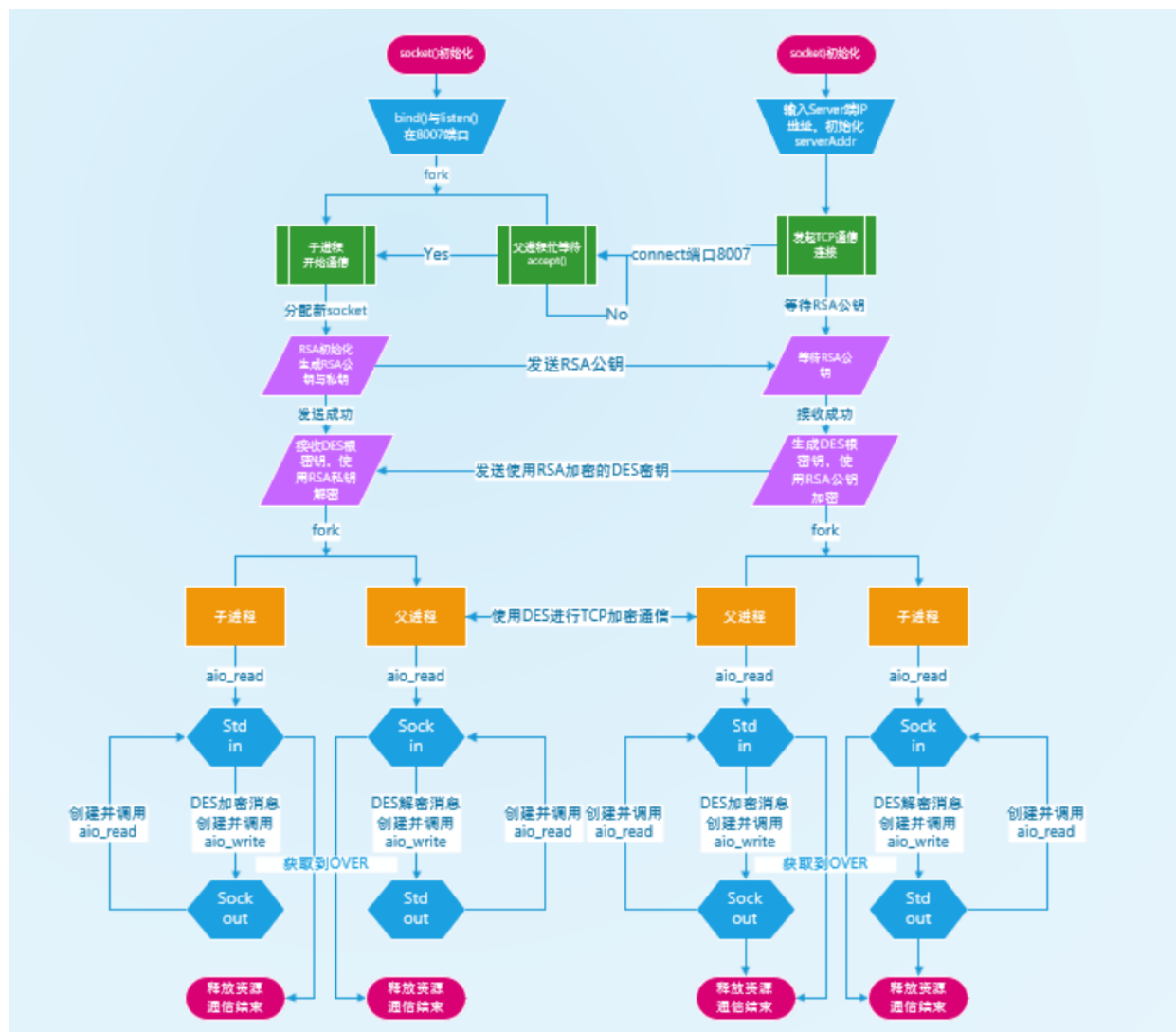


图 3.11: 使用 AIO 优化的基于 RSA 自动分配密钥的通信流程

图3.11展示了完整的优化后的程序流程。具体而言我参考了实验指导书，将实现分为一下就几个模块：

- **StdinSingle**：监控从键盘的 stdin 输入。在回调函数中创建 SockoutSingle，和它互为一组。
- **StdoutSingle**：控制从键盘的 stdout 中输出。在回调函数中创建 SockinSingle，和它互为一组。
- **SockinSingle**：监控从 socket 的 in 输入。在回调函数中创建 StdoutSingle，和它互为一组。
- **SockoutSingle**：控制从 socket 的 out 输出。在回调函数中创建 StdinSingle，和它互为一组。

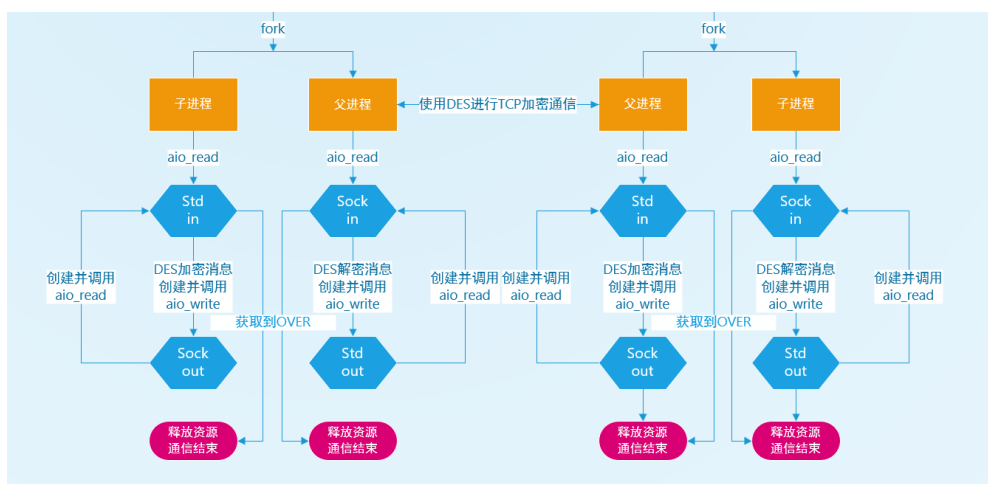


图 3.12: 重点优化的 AIO 的 Workflow

图3.12展示了重点使用 AIO 优化的 Workflow，可以看到分为以下两组：

- StdinSingle \longleftrightarrow SockoutSingle
- SockinSingle \longleftrightarrow StdoutSingle

除此之外还通过了信号量等控制资源的使用，具体详见实验步骤部分。

3.6 HeadquarterAndAgent

由于我的主题的特殊性，本次实验我依然延续之前的特工和总部的内容。并一定程度上进行了扩展。

3.6.1 特工介绍

本次实验选择的特工人物还是沿用上次的五名特工。他们分别是：詹姆斯·邦德 (James Bond)，杰森·伯恩 (Jason Bourne)，黑寡妇 (Black Widow)，伊森·亨特 (Ethan Hunt)，强尼·英格力 (Johnny English)。

3.6.2 Special Operations

- **Silent Guardian Operation:** 基于上次实验的基本 DES 加密进行改进。使用 RSA 自动分配 DES 密钥，DES 加密的 TCP 通信。
- **Phantom Hook Operation:** 不使用任何密码算法包括 RSA 和 DES 的纯 TCP 通信。
- **Silent Guardian Operation AIO:** 不再使用 fork，而是 Linux 的 AIO 机制实现。同样是 RSA 自动分配 DES 密钥，DES 加密的 TCP 通信。
- **Phantom Hook Operation AIO:** 同样采用 Linux AIO 机制。
- **Abort and Destroy**

3.6.3 其它功能

除此之外，本次实验依旧沿用上次实验中一些主题功能设定，包含：身份验证，错误处理，总部多负载通信，虚假身份检测，代号匹配等。这里就不再重复赘述。

4 实验步骤

4.1 实验模块搭建

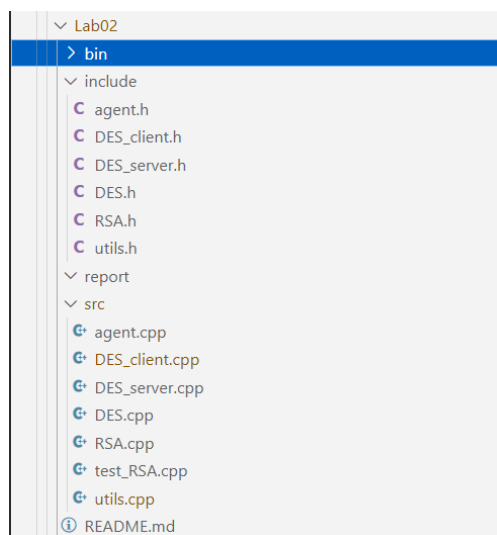


图 4.13: 本次实验模块讲解

如图4.13所示，本次实验搭建了一个相对非常完善且规整的代码框架，具有较高的可读性和可维护性。具体而言：

1. **bin:** 存在最终的可执行文件即 DES_Server.exe 和 DES_Client.exe。还有一个 test_RSA.cp, 是用来测试 RSA 的一些算法逻辑。

- 服务器端：DES_Server.exe
- 客户端：DES_Client.exe
- GoogleTEST: test_RSA.exe

2. **include:** 包含了一系列进行常量声明的头文件。

- agent.h: 包含定义的 Agent 类，用于后续操作存储 Agent 对象。会在 Server 端也就是总部端通过 vector 进行存储，方便多特工同时通信时进行管理（但暂未实现多特工同时通信）。
- DES.h: 定义了将要使用的 **DES 算法的工具类 DESUtils**。其中包含了用于加解密和密钥生成的各个常量，以及一些工具函数。都通过了相对解耦的接口实现，增强了代码的可维护性和鲁棒性。
- RSA.h: 这是本次重点的实现部分。和 DES 的实现类似，通过一个类 RSAUtils 集成，其中包含了 RSA 非对称加密的一些关键常量，接口以及函数。

- (1) 辅助函数:

- mulMod: 模乘。
- powMon: 模幂。
- EulerTotientFunction: 欧拉函数（实际上最后也可以不用）。
- extendedGCD: 拓展欧几里得算法。
- modInverse: 求模逆元。
- MillerRabin: 素数检测。
- primeTest: 素数检测算法。使用不同的方法（暂时只开发了 MillerRabin 算法）进行素数检测。
- generateRadomPrime: 生成随机素数，用于生成大素数 p 和 q。

(2) 关键常量:

- enum 定义的三种素性检测算法。暂时只实现了 Miller Rabin。
- p, q, n, phi_n. 均采用 `uint_64` 存储。因此要注意溢出问题。这一点后面会反复强调。
- RSA 公钥对: `pair<uint64_t, uint64_t> publicKey`
- RSA 私钥对: `pair<uint64_t, uint64_t> privateKey`

(3) 核心函数:

- `int init(int rounds = 100, bool defaultKey = true, bool highSecurity = false):` 初始化函数。完成各种关键常量的生成。同时包含公钥和私钥的生成。
- 加密: `void encrypt(uint64_t &plainText, uint64_t &cipherText)`
- 解密: `void decrypt(uint64_t &cipherText, uint64_t &plainText)`

具体声明代码如下:

```

1  #include <stdint>
2  #include <utility>
3  #include <iostream>
4  #include <random>
5  using namespace std;
6  #ifndef SUCCESS
7  #define SUCCESS 1
8  #endif
9
10 #ifndef FAILURE
11 #define FAILURE 0
12 #endif
13 class RSAUtils
14 {
15 public:
16     RSAUtils() {};
17     ~RSAUtils() {};
18     int init(int rounds = 100, bool defaultKey = true, bool highSecurity = false);
19     pair<uint64_t, uint64_t> getPublicKey();
20     pair<uint64_t, uint64_t> getPrivateKey();
21     void setPublicKey(uint64_t e, uint64_t n);
22     void setPrivateKey(uint64_t d, uint64_t n);

```

```

23     void encrypt(uint64_t &plainText, uint64_t &cipherText);
24     void decrypt(uint64_t &cipherText, uint64_t &plainText);
25
26 private:
27     // enum for prime test approach
28     enum
29     {
30         MILLER_RABIN = 0,
31         EUCLIDEAN = 1,
32         FERMAT = 2
33     };
34     // Basic APIs: Modular arithmetic and so on
35     uint64_t mulMod(uint64_t a, uint64_t b, uint64_t n);
36     uint64_t powMod(uint64_t base, uint64_t pow, uint64_t n);
37     uint64_t EulerTotientFunction(uint64_t n);
38     // Greatest common divisor using Extended Euclidean Algorithm
39     uint64_t extendedGCD(uint64_t a, uint64_t b, int64_t &x, int64_t &y);
40     uint64_t modInverse(uint64_t a, uint64_t n);
41     // Prime test algorithms
42     bool MillerRabin(uint64_t n);
43     /* Continuing to implement Fermat and Euclidan for Prime Test */
44     bool primeTest(uint64_t n, int approach, int rounds = 100);
45     // Generate random prime number
46     uint64_t generateRamdomPrime(int bits, int rounds = 100);
47     int generatePublicKey(uint64_t n, int rounds = 100);
48     int generatePrivateKey(uint64_t n);
49     int generateKeyPair(uint64_t n, int rounds = 100);
50
51     /**
52      * Advanced Standard for better security
53      */
54     // Find Prime with large factor for better security
55     int findPrimeWithLargeFactor(int bits, int rounds);
56     /**
57      * Prime number p & q
58      * n = p * q
59      * phi = (p - 1) * (q - 1)
60      * e: 1 < e < phi, gcd(e, phi) = 1
61      * d: e * d mod phi = 1
62      * Note: Although NIST and ANSI X9 has acquired the RSA key length of 1024 bits
63      *        at least,
64      *        this implementation is for fun only, so the key length is 64 bits
65      *        stroing in uint64_t.
66      */
67     uint64_t p;
68     uint64_t q;
69     uint64_t n;
70     uint64_t phi_n;
71     uint64_t e;

```



```

70     uint64_t d;
71     /**
72      * Public and Private Key
73      * publicKey: (e, n)
74      * privateKey: (d, n)
75      */
76     pair<uint64_t, uint64_t> publicKey;
77     pair<uint64_t, uint64_t> privateKey;
78 };

```

- **utils.h**: 主要定义了一些辅助的工具函数和工具常量。其中和上次实验保持一致的我就不再赘述具体功能。我会重点介绍本次实验进行修改和新增的函数:

- string timeNow()
- void signalHandler(int sig)
- ssize_t TotalRecv(int s, void *buf, size_t len, int flags)
- void PhantomHook(int role, Agent agent): 用于不加密的公开信道交流。
- void SilentGuardian(int role, Agent agent, DESUtils des, RSAUtils rsa): 本次实验重点进行改进的部分。通过 RSA 算法实现了 DES 密钥的自动分配, 加密过程依然使用 DES 算法。
- extern char strStdinBuffer_AIO[BUFFERSIZE], strEncryptedBuffer_AIO 和 strDecryptedBuffer_AIO: 它们三个主要用于 AIO 异步优化使用的缓冲区。
- class StdinSingle: 用于监控 Stdin 输入, 并保存了一系列重要的上下文和异步 IO 的关键信息。这里以 StdinSingle 为例, 介绍一些重点使用的成员变量和函数。
 - * int m_role: 用于区分一些提示信息, 判断该监控对象是由服务器端 (M16) 还是客户端 (AGENT) 创建的。
 - * m_nSocket: 用于存储通信的 socket。
 - * aiocb *m_pReq: 指向 aiocb 结构, 即 Asynchronous I/O control block 的指针。aiocb 是 POSIX 异步 I/O 的核心数据结构, 它包含了一个异步 I/O 操作的所有信息, 例如文件描述符、缓冲区的位置、缓冲区的大小等。m_pReq 被用来存储一个异步读或写操作的信息。
 - * sem_t &m_bStop: 引用到 sem_t 类型的变量。sem_t 是 POSIX 信号量的数据类型, 它可以用来同步多个线程或进程的执行。m_bStop 这里被用来控制何时停止一个操作或线程。详见具体实现部分。
 - * StdinSingle(int role, Agent agent, DESUtils des, sem_t &bStop): 构造函数。
 - * static void StdinReadCompletionHandler(sigval_t sigval): 非常重要的回调函数, aio 系列操作返回后会调用该函数进行进一步处理。
- class StdoutSingle: 用于显示 Stdout 输出。
- class SockinSingle: 用于监控 Socket 的 in 输入。
- class SockoutSingle: 用于发送 Socket 的 out 输出。
- void PhantomHook_AIO(int role, Agent agent): 异步 AIO 改进版的非加密通信。暂未实现。

- void SilentGuardian_AIO(int role, Agent agent, DESUtils des, RSAUtils rsa): **异步 AIO 改进版的基于 RSA 自动分配密钥的 DES 加密。**

这里只展示部分新增加的代码：

```

1  /* Other hpp files */
2  #include <aio.h> // For async I/O
3  #include "agent.h"
4  #include "DES.h"
5  #include "RSA.h"
6
7  /* Other functions and variables */
8  void SilentGuardian(int role, Agent agent, DESUtils des, RSAUtils rsa); // Special
    Operation: Communication between Agent and Headquarter with DES encryption to
    protect the secret
9
    // DES key
    transfered
    by RSA
    encryption
10 void SilentGuardian_AIO(int role, Agent agent, DESUtils des, RSAUtils rsa); //
    Special Operation: Communication between Agent and Headquarter with DES
    encryption to protect the secret
11
12 extern char strStdinBuffer_AIO[BUFFERSIZE];
13 extern char strEncryptedBuffer_AIO[BUFFERSIZE];
14 extern char strDecryptedBuffer_AIO[BUFFERSIZE];
15 /* Class Implementation for asynchronous I/O
16  * storing all the context and other essential information for the asynchronous I/O
17  * Working pair:
18  * StdinSingle <—> SockoutSingle
19  * SockinSingle <—> StdoutSingle
20  */
21 class StdinSingle
22 {
23     /**
24      * Specifically for monitoring the input from stdin
25      */
26 public:
27     int m_role;
28     int m_nSocket;
29     aiocb *m_pReq;
30     sem_t &m_bStop;
31     string m_agentCodeName;
32     DESUtils m_des;
33     StdinSingle(int role, Agent agent, DESUtils des, sem_t &bStop);
34     ~StdinSingle();
35     static void StdinReadCompletionHandler(sigval_t signal);
36 };
37 /* StdoutSingle, SockinSingle和SockoutSingle定义类似, 不再展示 */

```

- DES_Server.h: TCP 通信的服务器端即 Headquarter 端。这里只提一些本次实验修改的地方。不再使用硬编码 DES 密钥保证安全, rsa 对象。

```

1 // Hadcoded DES key for default, discard for now due to unsafety
2 char *defaultKey = "YCZhouNB";
3
4 /* RSA parameters */
5 RSAUtils rsa

```

- DES_Client.h: TCP 通信的客户端即 Agent 端。修改的地方和服务端一样, 不再赘述。

3. src: 包含了一系列对 include 中头文件中声明的函数进行的实现的 cpp 文件。详细实现见下面具体部分, 这里也只对一些本次实验重点的进行介绍。和上次实验保持一致我就不再重复赘述。

- agent.cpp
- DES.cpp:
- utils.cpp: 其中提供了很多功能函数, 例如显示时间等, 这些都与上次实验保持一致。本次实验新增的是对 SilentGuardian 即 DES 加密算法新加入了 RSA 的自动密钥分配。以及 AIO 的各种成员函数实现, 并最终应用到 SilentGuardian_AIO 中。
- DES_Server.cpp: 几乎没有改变, 和上次实验保持一致。
- DES_Client.cpp: 本次实验在客户端的主函数中实现了客户端接收 RSA 公钥的部分。之后才进入 SilentGuardian 的调用中。
- RSA.cpp: 也是本次实验的重点实现之一。详见后面的 RSA 部分。
- test_RSA.cpp: 通过 GoogleTest 调用的辅助类, 帮助 debug。

4.2 DES

这次的 DES 只在一个方面进行了功能上的添加, 那便是添加了自动生成随机 DES 密钥的功能。准确来说是随机根密钥。

生成时候保证密钥范围在 0 到 64 位之间, 同时最高位不能为 1, 这是为了 RSA 部分操作涉及带符号数 int64_t, 避免转换过程中有 unsigned 出现, 出现正负数的混杂导致错误。

```

1 /* DES.h 中 */
2 // Root Key for DES
3 uint64_t rootKey;
4 /* DES.cpp 中 */
5 void DESUtils::generateRandomRootKey()
6 {
7     /**
8      * @brief Generate a random root key for DES
9      *      The root key is a 64-bit random number
10     */
11     random_device rd;
12     mt19937_64 gen(rd());
13     uniform_int_distribution<uint64_t> dis(0, 0xffffffffffffffff);
14     do

```

```

15     {
16         rootKey = dis(gen);
17     } while (std::to_string(rootKey)[0] == '1');
18
19     return;
20 }
21 uint64_t DESUtils::getRootKey()
22 {
23     /**
24      * @brief Get the root key for DES
25      * @return: the root key
26      */
27     return rootKey;
28 }

```

4.3 RSA

4.3.1 辅助函数

1. **模乘运算 mulMod**。模乘运算我的实现最开始出现了一些问题，最终呈现的如下实现版本**同时**做到了在 $\log b$ 的时间复杂度内做到了高性能优化，同时也做到了避免溢出。具体实现而言：

- 使用**按位进行的运算**进行性能优化。
- 使用移位和加法替代乘法运算；
- 使用大于 `uint64_t` 的大整数 `uint128_t` 避免乘法溢出。

```

1     uint64_t RSAUtils::mulMod(uint64_t a, uint64_t b, uint64_t n)
2     {
3         /**
4          * Calculate (a * b)(mod n) in log(b), while the same time avoiding overflow.
5          * By following:
6          * (1) (a * b)(mod n) = ((a mod n) * (b mod n))(mod n)
7          * (2) Multiplication—>Shifting and Adding
8          * (3) Avoiding overflow—>Use __uint128_t to store the result of multiplication
9          * @param a: var1
10         * @param b: var2
11         * @param n: mod
12         * @return res: (a * b)(mod n)
13         */
14         __uint128_t res = 0;
15         __uint128_t a_mod = a % n;
16         __uint128_t b_mod = b % n; // This line is not strictly necessary but shows how
17         // to use __uint128_t
18         __uint128_t n_mod = n; // Use __uint128_t for modulus to avoid any casting
19         // issues in calculations
20
21         while (b_mod > 0)
22         {

```

```

21     if (b_mod & 1)
22     {
23         res = (res + a_mod) % n_mod;
24     }
25     a_mod = (2 * a_mod) % n_mod;
26     b_mod >>= 1;
27 }
28 return (uint64_t)res; // Cast back to uint64_t to return the expected type
29 }

```

2. 模幂运算 **powMod**。同样通过调用写好的接口 **mulMod** 按位实现幂运算。具体实现时也进行了性能优化，使得运算能够将时间复杂度降到 $\log(pow)$ 。其中 pow 为模幂运算的指数。

```

1  uint64_t RSAUtils::powMod(uint64_t base, uint64_t pow, uint64_t n)
2  {
3      /**
4       * Calculate (base ^ pow)(mod n) in log(pow), while the same time avoiding
5       * overflow.
6       * @param base: base
7       * @param pow: exponent
8       * @param n: mod
9       * @return res: (base ^ pow)(mod n)
10      */
11      uint64_t res = 1;
12      while (pow > 0)
13      {
14          if (pow & 1)
15              // To avoid overflow, we use mulMod instead of res = res * base % n;
16              res = mulMod(res, base, n);
17          base = mulMod(base, base, n);
18          pow >>= 1;
19      }
20      return res;
21 }

```

3. 拓展欧几里得算法 **extendedGCD**。使用递归算法实现求解最大公因数 (**greatest Common Divisor**) 和辅助求逆元。这里需要注意的就是由于运算过程可能出现负号的 **int64_t**，所以使用 **static_cast<int64_t>** 进行稳点转换。

```

1  uint64_t RSAUtils::extendedGCD(uint64_t a, uint64_t b, int64_t &x, int64_t &y)
2  {
3      /**
4       * Calculate the greatest common divisor of a and b using Extended Euclidean
5       * Algorithm.
6       * @param a: var1
7       * @param b: var2
8       * @param x: a-1(mod b)
9       * @param y: b-1(mod a)

```

```

9      * @return gcd: greatest common divisor of a and b
10     */
11     if (b == 0)
12     {
13         x = 1;
14         y = 0;
15         return a;
16     }
17     uint64_t gcd = extendedGCD(b, a % b, x, y);
18     int64_t temp = x;
19     x = y;
20     y = temp - static_cast<int64_t>(a / b) * y;
21     return gcd;
22 }

```

4. **求逆元 modInverse**。通过调用写好的函数接口 extendedGCD 拓展欧几里得算法求解逆元。需要注意的就是判断 gcd 的结果，若不为 1 则没有逆元；同时还有确保最后返回的值 uint64_t 是一个正数。未来会在求解 RSA 私钥时候用到。

```

1  uint64_t RSAUtils::modInverse(uint64_t a, uint64_t n)
2  {
3      /**
4       * Calculate the modular multiplicative inverse of a mod n.
5       * @param a: var1
6       * @param n: mod
7       * @return modular multiplicative inverse of a mod n—— $a^{-1}(\text{mod } n)$ 
8       */
9      int64_t x, y;
10     uint64_t gcd = extendedGCD(a, n, x, y);
11     // ax+ny=gcd(a,n)
12     // x =  $a^{-1}(\text{mod } n)$ 
13     // y =  $n^{-1}(\text{mod } a)$ 
14     if (gcd != 1)
15         // a and n are not coprime, so the modular multiplicative inverse does not exist.
16         return 0;
17     else
18         // To make it a positive number
19         return (x + n) % n;
20 }

```

5. **MillerRabin 素性检测**。通过调用写好的 powMod 接口函数和随机数实现素性检测。具体原理这里不再赘述。需要注意的就是这里是只返回一次判断的结果。

```

1  bool RSAUtils::MillerRabin(uint64_t n)
2  {
3      /**
4       * Test whether n is a prime number using Miller-Rabin Algorithm.

```

```

5      * @param n: number to be tested
6      * @return true: prime number
7      *          false: composite number
8      */
9      if (n == 2 || n == 3)
10         return true;
11      if (n < 2 || n % 2 == 0)
12         return false;
13      random_device rd;
14      mt19937_64 gen(rd());
15      uniform_int_distribution<uint64_t> dis(1, n - 1);
16      uint64_t a = dis(gen);
17      uint64_t temp = n - 1;
18      uint64_t k = 0;
19      while (temp % 2 == 0)
20      {
21         k++;
22         temp /= 2;
23      }
24      uint64_t m = temp;
25      uint64_t b = powMod(a, m, n);
26      if (powMod(b, 1, n) % n == 1)
27         return true;
28      for (uint64_t i = 0; i < k; i++)
29      {
30         if (powMod(b, 1, n) == n - 1)
31             return true;
32         else
33             b = powMod(b, 2, n);
34     }
35     return false;
36 }

```

6. primeTest 素性检测。通过参数 approach 指定使用的方法，暂时只实现了 MillerRabin。

rounds 参数因为素性检测使用 MillerRabin 检测素性，而 MillerRabin 作为概率算法，有一定几率失败。因此设定轮数值来增大算法的可靠性。

```

1 bool RSAUtils::primeTest(uint64_t n, int approach, int rounds)
2 {
3     /**
4      * Test whether n is a prime number using different approaches.
5      * @param n: number to be tested
6      * @param approach: test approach
7      *                  0: Euclidean Algorithm
8      *                  1: Fermat Algorithm
9      *                  2: Miller-Rabin Algorithm
10     * @param rounds: test rounds
11     * @return true: prime number

```

```

12     *           false: composite number
13     */
14     if (n == 2 || n == 3)
15         return true;
16     if (n < 2 || n % 2 == 0)
17         return false;
18     switch (approach)
19     {
20     case MILLER_RABIN:
21         /**
22          * Miller-Rabin Algorithm
23          * Based on the fact that Miller-Rabin Algorithm is a probabilistic algorithm,
24          * A odd composite number n has at least 75% probability to be detected as a
25          * composite number
26          * As a result, we repeat the test for rounds times to improve the accuracy.
27          */
28         while (rounds--)
29         {
30             if (!MillerRabin(n))
31                 return false;
32         }
33         return true;
34         break;
35     case EUCLIDEAN:
36         // continue to implement
37         return true;
38         break;
39     case FERMAT:
40         // continue to implement
41         return true;
42         break;
43     default:
44         return false;
45         break;
46     }
47     return false;
48 }

```

7. **generateRandomPrime 生成指定位数的大素数。**该函数通过参数 bit 指定要生成的素数位数，rounds 参数同样用于 MillerRabin 素数检测。该函数未来会用在生成 p 和 q 等地方。

```

1 uint64_t RSAUtils::generateRandomPrime(int bits, int rounds)
2 {
3     /**
4      * Generate a random prime number with the specified bits.
5      * @param bits: bits of the prime number
6      * @param rounds: test rounds
7      * @return prime: random prime number
8      */

```



```

9     random_device rd;
10    mt19937_64 gen(rd());
11    uniform_int_distribution<uint64_t> dis(0, (((uint64_t)1) << (bits - 2)) - 1);
12    uint64_t base;
13    do
14    {
15        base = (uint64_t)1 << (bits - 1);
16        base += dis(gen);
17        base |= 1; // Make sure the number is odd
18
19    } while (!primeTest(base, MILLER_RABIN, rounds));
20    return base;
21 }

```

8. 除此之外，辅助函数还包括一些为了保证规范性和安全性，设计的**公访私的接口**访问公钥和私钥等成员变量。

4.3.2 初始化

1. **generatePublicKey 生成公钥**。这里主要也是通过随机数生成器去寻找和 **phi_n** 互素的公钥参数 **d**。如果最终成功找到了则返回 SUCCESS，否则比如 rounds 轮数耗尽，则返回 FAILURE。

```

1  int RSAUtils::generatePublicKey(uint64_t n, int rounds)
2  {
3      /**
4       * Generate public key paramter (e, n) for RSA encryption.
5       * @param n: number
6       * @param rounds: test rounds
7       * @return SUCCESS or FAILURE
8       */
9      random_device rd;
10     mt19937_64 gen(rd());
11     uniform_int_distribution<uint64_t> dis(2, phi_n - 1);
12     int64_t x, y;
13     do
14     {
15         e = dis(gen);
16         rounds--;
17     } while ((extendedGCD(e, phi_n, x, y) != 1) && (rounds > 0));
18     if (rounds == 0)
19         return FAILURE;
20     // Assign the public key
21     publicKey.first = e;
22     publicKey.second = n;
23     return SUCCESS;
24 }

```

2. **generatePrivateKey 生成私钥**。这里是一个较大的改动，没有采用和实验参考书中一样遍历二

元一次方程的可行解的做法。而是直接通过已知的关系即 $ed \equiv 1 \pmod{\phi(n)}$ 求逆元。

具体做法就是直接调用 `modInverse` 接口，并判断一下是否逆元根本不存在。

```

1  int RSAUtils::generatePrivateKey(uint64_t n)
2  {
3      /**
4       * Generate private key paramter (d, n) for RSA decryption.
5       * @param n: number
6       * @return SUCCESS or FAILURE
7       */
8      d = modInverse(e, phi_n);
9      if (d == 0)
10     {
11         return FAILURE; // If d = 0, resprsenting the case where no inverse exists
12     }
13     privateKey.first = d;
14     privateKey.second = n;
15     return SUCCESS;
16 }
```

3. **generateKeyPair 生成公钥和私钥对。**这里直接调用生成公钥的接口函数 `generatePublicKey` 和生成私钥的接口函数 `generatePrivateKey`。唯一注意的就是记住先生成公钥才方便根据公钥计算私钥。

```

1  int RSAUtils::generateKeyPair(uint64_t n, int rounds)
2  {
3      /**
4       * Generate RSA key pair both for encryption and decryption.
5       * @param n: number
6       */
7      // Generate public key first , so the private key can be generated based on the
       public key.
8      if (generatePublicKey(n, rounds) == FAILURE)
9          return FAILURE;
10     if (generatePrivateKey(n) == FAILURE)
11         return FAILURE;
12     return SUCCESS;
13 }
```

4. **init 初始化函数。**这里来到 `RSAUtils` 的类对象初始化函数。该函数我本来设计是具有抵御一些常见攻击的能力，和通过一些措施来提高 `RSA` 安全性。不过出于时间原因暂时没有实现，未来有待完善。

其余参数 `rounds` 用于底层调用的 `MillerRabin` 和公钥生成的轮数，都是因为这两个算法本质上是概率算法，有一定几率失败。

这里值得注意的有：

- 生成足够大的素数 `p` 和 `q`：通过接口函数 `generateRandomPrime` 进行生成。其中需要同时

避免溢出，即要求它们的乘积也就是 n 不能大于 64 位。另外还需要让二者越大越好。因此这选择了两个 32 位的数字，它们同时在 32 位时候，乘积与 64 位最接近，并且不会轻易溢出。

- 计算 $\phi_n = (p - 1) * (q - 1)$
- 判断是否使用默认公钥 $e=65537(2^{16} + 1)$ 。该公钥不仅由于其极低的 Hamming Weight 使得安全性好，同时又计算上效率高因此广泛使用。使用该公钥能够很好地提供安全性和性能之间的平衡。
- 调用 generateKeyPair 生成公钥和私钥对。

```

1  int RSAUtils::init(int rounds, bool defaultKey, bool highSecurity)
2  {
3      /**
4       * @brief Initialize the RSAUtils
5       * @details We need to follow the steps below:
6       * 1. Generate two random prime numbers p and q
7       * 2. Calculate n = p * q
8       * 3. Calculate phi = (p - 1) * (q - 1)
9       * 4. Generate public key (e, n)
10      * 5. Generate private key (d, n)
11      * @note If there is a need to enhance the security, we need to follow some
          standards:
12      * 1. The length of p and q should be the nearly the same
13      * 2. Large prime factors for (p-1) and (q-1)
14      * 3. Small Greatest Common Divisor(GCD) for (p-1) and (q-1)
15      * 4. Size of d when  $e < n$  and  $d > n^{0.25}$ 
16      * 5. Defending against Timing Attacks by
17      *     - Implementing constant-time algorithms for operations such as modular
          exponentiation, ensuring that execution time does not depend on the values
          of the inputs.
18      *     - Introducing random delays in the algorithm to obscure the execution time.
19      *     - Randomizing operations, such as multiplying the plaintext by a random
          number before encryption, to make the timing analysis more challenging.
20      * @param rounds: test rounds
21      * @param defaultKey: whether to use the default key as  $e=65537(2^{16}+1)$ .
22      *                     It not only provides a good balance between security(low
          Hamming weight) and efficiency(computationally efficient)
23      *                     but also is widely used in practice.
24      * @param highSecurity: whether to use the high security standard
25      * @return SUCCESS or FAILURE
26      */
27      if (highSecurity)
28      {
29          // Advanced Standard for better security
30          // Continuing to implement.....
31          if (defaultKey)
32              e = 65537;
33      }

```

```

34     else
35     {
36         // Basic Standard with high efficiency
37         // First, generate two random prime numbers p and q
38         do
39         {
40             p = generateRandomPrime(32, rounds);
41             q = generateRandomPrime(32, rounds);
42             // In order to avoid overflow, we need to make sure p * q <= 2^64
43             // At the same time, to make both of them as large as possible
44         } while ((rounds--)&& (p * q > 0xffffffffffffffff));
45         n = p * q;
46         phi_n = (p - 1) * (q - 1);
47         // Generate RSA key pair
48         if (defaultKey)
49         {
50             // Use the default key as e=65537(2^16+1)
51             e = 65537;
52             // Assign the public key
53             this->e = e;
54             publicKey.first = e;
55             publicKey.second = n;
56             // Generate private key based on the public key
57             if (generatePrivateKey(n) == FAILURE)
58                 return FAILURE;
59         }
60         else
61         {
62             // Generate RSA key pair
63             if (generateKeyPair(n) == FAILURE)
64                 return FAILURE;
65         }
66     }
67
68     return SUCCESS;
69 }

```

4.3.3 加密与解密

RSA 的加密和解密反倒是简单。主要是通过模幂运算 `powMod`。由于我的底层的模幂运算已经很大程度上在性能上进行了优化，同时避免了溢出。因此这里就可以放心地直接使用了。

```

1 void RSAUtils::encrypt(uint64_t &plaintext, uint64_t &ciphertext)
2 {
3     /**
4      * Encrypt the plaintext using RSA encryption algorithm.
5      * According to the RSA encryption algorithm:
6      *  $C = M^e \pmod n$  using the public key (e, n)

```

```

7      * @param plaintext: plaintext
8      * @param ciphertext: ciphertext
9      */
10     ciphertext = powMod(plaintext, e, n);
11     return;
12 }
13 void RSAUtils::decrypt(uint64_t &plaintext, uint64_t &ciphertext)
14 {
15     /**
16      * Decrypt the ciphertext using RSA decryption algorithm.
17      * According to the RSA decryption algorithm:
18      *  $M = C^d \pmod{n}$  using the private key (d, n)
19      * @param plaintext: plaintext
20      * @param ciphertext: ciphertext
21      */
22     plaintext = powMod(ciphertext, d, n);
23     return;
24 }

```

4.4 TCP

TCP 部分和上次实验保持一致，这里只重点提一下实现 RSA 密钥自动分配的实现过程。具体流程可参照图3.10。

4.4.1 Server

Server 端实现基于 RSA 的 DES 密钥的自动分配全部在 utils.cpp 的 SilentGuardian 的 M16 部分。

主要做的事情有如下：

1. 通知此时在 DES_Client.cpp 的 main 函数中等待指令的特工，启动 SilentGuardian。通过直接发送“Silent Guardian”的 string 串实现。
2. **RSA 初始化**：按照指示输入轮数，是否使用默认 RSA 公钥以积极是否采用更高级的安全性措施。之后直接调用 RSAUtils 的 init 函数初始化生成公钥和私钥。
3. **发送 RSA 公钥**：获取 RSA 公钥后，通过“(e, n)”的格式将 RSA 公钥发送给特工。
4. **等待并接收解密 DES 私钥，DES 密钥生成**：通过 recv 函数阻塞当前进程直到特工发回 DES 的根密钥。然后使用 RSA 的私钥 decrypt，解出 DES 私钥并进行 16 轮的密钥生成。

下面的代码在 utils.cpp 的 SilentGuardian 函数中：

```

1  /* blablabla 初始化以及Agent部分的省略，直接来看M16端 */
2      else if (role == M16)
3      {
4          // This part is in the headquarter's side
5          string str_time = timeNow();
6          bool control_flag = false;

```

```

7      cout << "<Headquarter::System @ " + str_time + " # Message>:Silent Guardian
      is activated.." << endl;
8      // Notify the agent to also activate the Silent Guardian
9      send(socket, "Silent Guardian", 100, 0);
10     while (true)
11     {
12         // Version 2.0: RSA algorithm is used to transmit the DES key
13         // Generate the RSA utils
14         cout << "<SecretHideout::System @ " + timeNow() + " # Message>:RSA
            initializaing....." << endl;
15         int rounds;
16         bool defaulyRSAKey;
17         bool ifHighSecurity;
18         cout << "<SecretHideout::System @ " + timeNow() + " # Message>:Please
            enter the number of maxium rounds for RSA:" << endl;
19         cin >> rounds;
20         cout << "<SecretHideout::System @ " + timeNow() + " # Message>:Please
            enter the default RSA key or not(1 for yes, 0 for no):" << endl;
21         cin >> defaulyRSAKey;
22         cout << "<SecretHideout::System @ " + timeNow() + " # Message>:Please
            enter the high security mode or not(1 for yes, 0 for no):" << endl;
23         cin >> ifHighSecurity;
24         if (rsa.init(rounds, defaulyRSAKey, ifHighSecurity) == FAILURE)
25         {
26             cout << "<Headquarter::System @ " + timeNow() + " # Message>:Oops!RSA
                Initialization failed!" << endl;
27             cout << "<Headquarter::System @ " + timeNow() + " # Message>:It is
                probably due to the the related settings.Please try again." <<
                endl;
28             continue;
29         }
30         break;
31     }
32
33     // Send the RSA public key to the agent
34     pair<uint64_t, uint64_t> publicKey = rsa.getPublicKey();
35     // Send the public key to the agent in the format of "(e, n)"
36     string strPublicKey = "(" + to_string(publicKey.first) + ", " +
        to_string(publicKey.second) + ")";
37     send(socket, strPublicKey.c_str(), 100, 0);
38     // Print the public key
39     // cout << "<Headquarter::System @ " + timeNow() + " # Message>:RSA Public
        Key: " + strPublicKey << endl;
40     if (send(socket, strPublicKey.c_str(), 100, 0) != 100)
41     {
42         cout << "<Headquarter::System @ " + timeNow() + " # Message>:There is a
            problem with the communication...25" << endl;
43         cout << "-----Shutdown Communication-----" << endl;
44         return;

```

```

45     }
46     else
47         cout << "<Headquarter::System @ " + timeNow() + " # Message>:Successfully
            sent the RSA Key!" << endl;
48
49     // Wait for the agent to send the DES key back
50     char strSocketBuffer[255];
51     memset(strSocketBuffer, 0, sizeof(strSocketBuffer));
52     cout << "-----Waiting for the DES Key-----" << endl;
53     // Using the TotalRecv function to ensure all data is received
54     if (recv(socket, strSocketBuffer, 255, 0) < 0)
55     {
56         cout << "<Headquarter::System @ " + timeNow() + " # Message>:There is a
            problem with the communication...26" << endl;
57         cout << "-----Shutdown Communication-----" << endl;
58         return;
59     }
60     else
61     {
62         cout << "<Headquarter::System @ " + timeNow() + " # Message>:Successfully
            received the DES Key!" << endl;
63         cout << "-----Decrypting the DES Key-----" << endl;
64         // Decrypt the DES key using the RSA private key
65         uint64_t encryptedKey = 0;
66         memcpy(&encryptedKey, strSocketBuffer, sizeof(encryptedKey));
67         // Print the encrypted DES key
68         cout << "<Headquarter::System @ " + timeNow() + " # Message>:Encrypted
            DES Key: " + to_string(encryptedKey) << endl;
69         uint64_t decryptedKey = 0;
70         rsa.decrypt(decryptedKey, encryptedKey);
71         // Print the decrypted DES key
72         cout << "<Headquarter::System @ " + timeNow() + " # Message>:Decrypted
            DES Key: " + to_string(decryptedKey) << endl;
73         cout << "-----Generating DES Key-----" << endl;
74         des.genKey(decryptedKey);
75         cout << "<Headquarter::System @ " + timeNow() + " # Message>:DES Key
            Generated." << endl;
76     }
77     cout << "-----Everything Ready-----" << endl;
78     cout << "-----Communication Start-----" << endl;

```

4.4.2 Client

Client 端为了实现 RSA 密钥的自动分配,分为在 DES_Client.cpp 和 utils.cpp 的 SilentGuardian 的 AGENT 部分两端。

在 DES_Client.cpp, 其中在被敌人监听后,一旦接收到”Silent Guardian” 字符串便正式开始。而后通过 recv 阻塞当前进程,直到等待服务器端发来以”(e, n)” 为格式的 RSA 公钥后进行读取。再

读取并赋值公钥后，正式调用 SilentGuardian 函数进行 utils.cpp。

```

1  /* blablabla */
2      else if (strcmp(strSocketBuffer, "Silent Guardian") == 0)
3      {
4          memset(strSocketBuffer, 0, BUFFERSIZE);
5          // Activate the Silent Guardian
6          Agent agent(agentName, agentCodeName, agentSocket);
7          cout << "<SecretHideout::System @ " + timeNow() + " # Message>:All
            channels have been compromised!" << endl;
8          cout << "<SecretHideout::System @ " + timeNow() + " # Message>:Silent
            Guardian is activated.." << endl;
9          cout << "-----Waiting for RSA Key-----" << endl;
10
11         if (recv(agentSocket, strSocketBuffer, BUFFERSIZE, 0) < 0)
12         {
13             cout << "<SecretHideout::System @ " + timeNow() + " #
                Message>:Error in receiving RSA Public Key!" << endl;
14             cout << "<SecretHideout::System @ " + timeNow() + " #
                Message>:Please report this issue to the headquarter
                immediately!" << endl;
15             close(agentSocket);
16             exit(0);
17         }
18         else
19             cout << "<SecretHideout::System @ " + timeNow() + " #
                Message>:Successfully received the RSA Public Key!" << endl;
20         // The RSA Public Key should be in the format of "(e, n)"
21         // Extract the RSA Public Key
22         uint64_t e, n;
23         sscanf(strSocketBuffer, "(%lu, %lu)", &e, &n);
24         // Print the RSA Public Key
25         // cout << "<SecretHideout::System @ " + timeNow() + " # Message>:RSA
            Public Key: (" << e << ", " << n << ")" << endl;
26         rsa.setPublicKey(e, n);
27         // Activate the Silent Guardian
28         SilentGuardian(AGENT, agent, des, rsa);
29         // SilentGuardian_AIO(AGENT, agent, des, rsa);
30         close(agentSocket);
31         exit(0);
32     }
33 /* blablabla */

```

剩下的工作在 utils.cpp 的 Client 一端实现，具体而言：

1. 生成 DES 根密钥。在获取到了 RSA 公钥后，通过 des 的 generateRandomRootKey 实现 DES 根密钥的随机生成。这里注意生成的 DES 根钥大小不能超过 rsa 的公钥参数 e，这一点会在结尾实验问题处解释。
2. des.genKey(des.getRootKey()): 生成 16 轮 DES 密钥。

3. 加密 DES 根密钥并发送给服务器端。使用 rsa.encrypt 进行加密后，通过 send 函数发送。

```

1  /* blablabla */
2  if (role == AGENT)
3  {
4      // This part is in the agent's side
5      string str_time = timeNow();
6      bool control_flag = false;
7
8      // First, the agent will use the public RSA key to encrypt the DES key
9      // Then send the encrypted DES key to the headquarter
10     cout << "-----Generating DES Key-----" << endl;
11     // Generating the DES key until the DES key is smaller than the RSA public key
12     while (true)
13     {
14         des.generateRandomRootKey();
15         if (des.getRootKey() >= rsa.getPublicKey().second)
16             continue;
17         break;
18     }
19
20     // Due to the hard-coded written of the Default DES key, we discard the
21     // hard-coded key for now
22     des.genKey(des.getRootKey());
23     cout << "<SecretHideout::System @ " + str_time + " # Message>:DES Key
24     Generated." << endl;
25     cout << "-----RSA encrypting the DES Key-----" << endl;
26     // Encrypt the DES key using the RSA public key
27     uint64_t encryptedKey = 0;
28     // In order not to be noticed by the enemy, we use this random name to hide
29     // the DES key
30     uint64_t sdjsadixcxzdssf = des.getRootKey();
31     // Print the DES key
32     cout << "<SecretHideout::System @ " + str_time + " # Message>:DES Key: " +
33     to_string(sdjsadixcxzdssf) << endl;
34     rsa.encrypt(sdjsadixcxzdssf, encryptedKey);
35     // Print the encrypted DES key
36     cout << "<SecretHideout::System @ " + str_time + " # Message>:Encrypted DES
37     Key: " + to_string(encryptedKey) << endl;
38     if (send(socket, &encryptedKey, sizeof(encryptedKey), 0) !=
39         sizeof(encryptedKey))
40     {
41         cout << "<SecretHideout::System @ " + str_time + " # Message>:There is a
42         problem with the communication...21" << endl;
43         cout << "-----Shutdown Communication-----" << endl;
44         return;
45     }
46     else
47         cout << "<SecretHideout::System @ " + str_time + " #

```

```

41         Message>:Successfully sent the DES Key!" << endl;
42         cout << "-----Everything Ready-----" << endl;
43         cout << "-----Communication Start-----" << endl;
44     /* blablabla */

```

4.5 AIO

接下来我将会介绍本次实验我的拓展探索部分，我通过 POSIX 的 Linux 异步 IO 实现了性能优化。具体流程如3.11和3.12所示。

4.5.1 Single 系列类实现

1. **构造函数：**构造函数这里以 StdinSingle 的为例，说明其主要工作流程，其余的 Single 系列类都和它类似。

- 对 m_bStop, m_role, m_des, m_nSocket 和 m_agentCodeName 等成员变量进行赋值初始化。
- 对 m_pReq 这个最关键的 AIO 控制块进行初始化。
- 通过设置作为 AIO 控制块的 m_pReq 的一些结构体中的变量，进行赋值，保存一些重要的设置信息。其中包含：
 - **文件描述符：**设置为 STDIN_FILENO，这是标准输入的文件描述符，表示异步操作将从标准输入读取数据。
 - **缓冲区指定：**指定了一个缓冲区 strStdinBuffer_AIO，用于存储从标准输入读取的数据。
 - **读取的字节数：**设置异步读取操作要从标准输入中读取的字节数，BUFFERSIZE 是这个操作指定要读取的最大字节数。
 - **文件操作的起始偏移量：**对于标准输入，通常偏移量设置为 0，因为标准输入流不支持随机访问。
 - **配置操作完成后的通知方式：**SIGEV_THREAD 表示完成后将启动一个新线程来处理通知。这也是因为 POSIX 的异步 IO 机制本质就是多线程。
 - **指定回调函数：**指定了一个函数 StdinReadCompletionHandler，这个函数将在异步读取操作完成后被调用。
 - **设置新线程属性为默认，**并将当前对象的指针作为额外数据传递给处理函数，这样在处理函数中可以访问当前对象的成员。

```

1 StdinSingle::StdinSingle(int role, Agent agent, DESUtils des, sem_t &bStop) :
   m_bStop(bStop), m_role(role), m_des(des)
2 {
3     /**
4     * Constructor for the StdinSingle class
5     * @param role: AGENT for agent, M16 for headquarter
6     * @param agent: Agent class object storing the essential information
7     * @param des: DESUtils class object used for encryption and decryption
8     * @param bStop: semaphore to control the communication

```

```

9      */
10     this->m_nSocket = agent.getAgentSocket();
11     this->m_agentCodeName = agent.getAgentCodeName();
12     bzero(&strStdinBuffer_AIO, sizeof(strStdinBuffer_AIO));
13     this->m_pReq = new aiocb;
14     bzero((char *)this->m_pReq, sizeof(struct aiocb));
15     /* Assign the aio control block */
16
17     this->m_pReq->aio_fildes = STDIN_FILENO;
18                                     // Set the file descriptor
19     to STDIN_FILENO
20     this->m_pReq->aio_buf = strStdinBuffer_AIO;
21                                     // Set the buffer to store the
22     input from stdin
23     this->m_pReq->aio_nbytes = BUFFERSIZE;
24                                     // Set the number of
25     bytes to read from stdin
26     this->m_pReq->aio_offset = 0;
27                                     // Set the
28     offset to 0
29     this->m_pReq->aio_sigevent.sigev_notify = SIGEV_THREAD;
30                                     // Set the notification method to
31     SIGEV_THREAD
32     this->m_pReq->aio_sigevent._sigev_un._sigev_thread._function =
33     StdinReadCompletionHandler; // Set the completion handler function to
34     StdinReadCompletionHandler
35     this->m_pReq->aio_sigevent._sigev_un._sigev_thread._attribute = NULL;
36                                     // Set the thread attribute to NULL
37     this->m_pReq->aio_sigevent.sigev_value.sival_ptr = this;
38                                     // Set the thread attribute to NULL
39 }

```

整个设置流程是为使得标准输入的读取可以异步进行，当读取操作完成时，一个新线程将被创建并调用指定的处理函数，而主程序可以继续执行其他任务而不被阻塞。

上述构造函数对于 StdoutSingle, SockinSingle 以及 SockoutSingle 实现均类似。

区别在于对于 StdoutSingle 中将文件描述符设置为 STDOUT_FILENO，而 socket 的两个都设为 this->m_nSocket。另外处理函数都设置为各自的回调函数。

- 析构函数：这里也是由于比较类似，只展示 StdinSingle 的析构函数实现。

```

1 StdinSingle::~StdinSingle()
2 {
3     /**
4      * Destructor for the StdinSingle class
5      */
6     delete this->m_pReq;
7     bzero(&strStdinBuffer_AIO, sizeof(strStdinBuffer_AIO));
8 }

```

3. 回调处理函数：回调处理函数是 AIO 实现的重点。这里和实验参考书相互补，我这里重点展示 StdinSingle 和 SockoutSingle 之外的另一组，也就是 SockinSingle 和 StdoutSingle。前两者实现和实验参考书基本一致。

其主要分为以下几个部分：

- 回调函数返回后首先检查 `aio_error(pThis->m_pReq)` 是否为 0，若为 0 则正常。若不为 0，通过向另一端发送“COMMUNICATION ERROR”后结束通信。
- 获取 `aio_return(pThis->m_pReq)` 级读取 socket 到的数据长度。
- 通过 `this->m_des` 的 DES 密钥进行解密。
- 接下来是非常关键的一步，然后通过 StdoutSingle 的构造函数创建一个文件描述符对象，然后调用其 `aio_write` 让 Stdout 进行打印显示。
- 如果接收到的消息解密后为“OVER”或者“COMMUNICATION ERROR”就结束通信。通过 `sem_post(&pThis->m_bStop)` 设置信号量，通知主进程结束，并自行退出。

```

1 void SockinSingle::SockinReadCompletionHandler(sigval_t signal)
2 {
3     /**
4      * Completion handler for the SockinSingle class
5      * @param sigval: signal value, containing the result of the asynchronous I/O
6      * @return void
7      */
8     SockinSingle *pThis = (SockinSingle *)signal.sival_ptr;
9     if (aio_error(pThis->m_pReq) == 0)
10    {
11        // There is no error in the asynchronous I/O
12        int nLength = aio_return(pThis->m_pReq);
13        pThis->m_des.decrypt(strEncryptedBuffer_AIO, strDecryptedBuffer_AIO);
14        StdoutSingle *pStdout = new StdoutSingle(
15            pThis->m_role,
16            Agent("Doesn't matter", pThis->m_agentCodeName, pThis->m_nSocket),
17            pThis->m_des,
18            pThis->m_bStop);
19        aio_write(pStdout->m_pReq);
20        // Check if the message is "OVER"
21        if (memcmp("OVER", strDecryptedBuffer_AIO, 4) == 0)
22        {
23            // If the message received is "OVER"
24            // It means the other end is done communicating
25            if (pThis->m_role == AGENT)
26            {
27                cout << "<SecretHideout::System @ " + timeNow() + " # Message>:The
28                    Headquarter is off the line." << endl;
29                cout << "<SecretHideout::System @ " + timeNow() + " #
30                    Message>:Communication is OVER." << endl;
31            }
32            else if (pThis->m_role == M16)
33            {

```

```

32         cout << "<Headquarter::System @ " + timeNow() + " # Message>:Agent
33             OUT." << endl;
34         cout << "<Headquarter::System @ " + timeNow() + " #
35             Message>:Communication is OVER." << endl;
36     }
37     cout << "-----Communication OVER-----" << endl;
38     sem_post(&pThis->m_bStop);
39     delete pThis;
40     exit(0);
41 }
42 // Check if the message is "COMMUNICATION ERROR"
43 else if (memcmp("COMMUNICATION ERROR", strDecryptedBuffer_AIO, 19) == 0)
44 {
45     // If the message the pther received is "COMMUNICATION ERROR"
46     // It means there is a problem with the communication
47     if (pThis->m_role == AGENT)
48     {
49         cout << "<SecretHideout::System @ " + timeNow() + " # Message>:There
50             is a problem with the communication...7" << endl;
51     }
52     else if (pThis->m_role == M16)
53     {
54         cout << "<Headquarter::System @ " + timeNow() + " # Message>:There is
55             a problem with the communication...8" << endl;
56     }
57     cout << "-----Shutdown Communication-----" << endl;
58     sem_post(&pThis->m_bStop);
59     delete pThis;
60     exit(0);
61 }
62 }
63 else
64 {
65     // Communication ERROR
66     // Shut it down
67     if (pThis->m_role == AGENT)
68     {
69         cout << "<SecretHideout::System @ " + timeNow() + " # Message>: There is
70             a problem with the communication...9" << endl;
71     }
72     else if (pThis->m_role == M16)
73     {
74         cout << "<Headquarter::System @ " + timeNow() + " # Message>: There is a
75             problem with the communication...10" << endl;
76     }
77     cout << "-----Shutdown Communication-----" << endl;
78     char *hint = "COMMUNICATION ERROR";
79     bzero(strEncryptedBuffer_AIO, sizeof(strEncryptedBuffer_AIO));
80     pThis->m_des.encrypt(hint, strEncryptedBuffer_AIO);
81     send(pThis->m_nSocket, strEncryptedBuffer_AIO,
82         sizeof(strEncryptedBuffer_AIO), 0);
83     sem_post(&pThis->m_bStop);
84     delete pThis;
85     exit(0);
86 }

```

```

74     delete pThis;
75     return;
76 }

```

下面是 Stdout 的回调处理函数 **StdoutWriteCompletionHandler**。具体过程也类似。值得注意的是这里也是在打印显示数据后，构造并驱动类 **SockinSingle** 继续工作。以此实现循环往复的永动机。调用类 **SockinSingle** 中指定的回调函数 **SockinReadCompletionHandler()**。如此循环，直到用户输入“OVER”命令。该永动机才正式退出。

```

1  void StdoutSingle::StdoutWriteCompletionHandler(signal_t signal)
2  {
3      /**
4       * Completion handler for the StdoutSingle class
5       * @param signal: signal value, containing the result of the asynchronous I/O
6       * @return void
7       */
8      StdoutSingle *pThis = (StdoutSingle *)signal.sival_ptr;
9      if (aio_error(pThis->m_pReq) == 0)
10     {
11         // There is no error in the asynchronous I/O
12         int nLength = aio_return(pThis->m_pReq);
13         // Initialize a SockinSingle Object to move back in
14         SockinSingle *pSockin = new SockinSingle(
15             pThis->m_role,
16             Agent("Doesn't matter", pThis->m_agentCodeName, pThis->m_nSocket),
17             pThis->m_des,
18             pThis->m_bStop);
19         // Start the operation on Sockin
20         // Motivate the SockinSingle to continue working
21         aio_read(pSockin->m_pReq);
22     }
23     else
24     {
25         // Communication ERROR
26         // Shut it down
27         if (pThis->m_role == AGENT)
28             cout << "<SecretHideout::System @ " + timeNow() + " # Message>: There is
                a problem with the communication...3" << endl;
29         else if (pThis->m_role == M16)
30             cout << "<Headquarter::System @ " + timeNow() + " # Message>: There is a
                problem with the communication...4" << endl;
31
32         cout << "-----Shutdown Communication-----" << endl;
33         char *hint = "COMMUNICATION ERROR";
34         bzero(strEncryptedBuffer_AIO, sizeof(strEncryptedBuffer_AIO));
35         pThis->m_des.encrypt(hint, strEncryptedBuffer_AIO);
36         send(pThis->m_nSocket, strEncryptedBuffer_AIO,
                sizeof(strEncryptedBuffer_AIO), 0);

```

```

37     sem_post(&pThis->m_bStop);
38     delete pThis;
39     exit(0);
40 }
41 delete pThis;
42 return;
43 }

```

4.5.2 SilentGuardian 优化

这里主要通过创建 **AIO 控制块 sem_t** 并初始化。然后 **fork** 进行进程分支，而后：

- **父进程：** SockinSingle 调用 **aio_read** 从 socket 读入。实现消息的读取。
- **子进程：** StdinSingle 调用 **aio_read** 从键盘 Stdin 读入。实现消息的输入。

两个进程都通过 **sem_wait(&bStop)** 和 **sem_destroy(&bStop)** 前者阻塞调用线程,直到由 &bStop 指向的信号量大于零。它会将信号量的值减少 1; 后者释放与该信号量相关联的任何资源。

```

1 void SilentGuardian_AIO(int role, Agent agent, DESUtils des, RSAUtils rsa)
2 {
3     /* blablabla 初始化 */
4     if (role == AGENT)
5     {
6         /* blablabla 基于RSA的DES密钥交换省略 */
7         pid_t nPid;
8         nPid = fork();
9         sem_t bStop;
10        sem_init(&bStop, 0, 0);
11        REALSTART1:
12        if (nPid != 0)
13        {
14
15            /**
16             * Parent process
17             * In charge of the following:
18             * 1. Receive message from the headquarter
19             * 2. Decrypt the message
20             * 3. Output the message
21             */
22            char strSocketBuffer[255];
23            char encryedtext[255];
24            char decryptedtext[255];
25            // Register the signal handler
26            signal(SIGTERM, signalHandler);
27            if (control_flag)
28            {
29                SockinSingle *pSockin = new SockinSingle(AGENT, agent, des, bStop);
30                aio_read(pSockin->m_pReq);

```

```
31         // Wait until the semaphore value > 0
32         sem_wait(&bStop);
33         // Release the semaphore
34         sem_destroy(&bStop);
35         // Kill the child process
36         kill(nPid, SIGTERM);
37         return;
38     }
39     control_flag = true;
40     goto REALSTART1;
41 }
42 else
43 {
44     /**
45     * Child process
46     * In charge of the following:
47     * 1. Get the message from the agent
48     * 2. Encrypt the message
49     * 3. Send the message to the headquarter
50     */
51     char strStdinBuffer[255];
52     char encryedtext[255];
53     char decryptedtext[255];
54     // Register the signal handler
55     signal(SIGTERM, signalHandler);
56     if (control_flag)
57     {
58         StdinSingle *pStdin = new StdinSingle(AGENT, agent, des, bStop);
59         aio_read(pStdin->m_pReq);
60         // Wait until the semaphore value > 0
61         sem_wait(&bStop);
62         // Release the semaphore
63         sem_destroy(&bStop);
64     }
65     control_flag = true;
66     goto REALSTART1;
67 }
68 }
69 else if(role == M16)
70 {
71     /* blablabla 基于RSA的DES密钥交换省略 */
72     /* 和AGENT部分控制流几乎类似 */
73 }
74 }
```


4.6 AgentTheme

这里 Agent 和 M16 的部分和之前保持一致，并没有添加符合主题的的新功能。基于 RSA 自动分配密钥的 DES 加密也是在之前设计的秘密行动 SilentGuardian 之中。因此这部分不再赘述。

5 实验结果

5.1 实验结果展示

由于在上次实验中已经完整展示过完整的 TCP 流程，这里本次只展示增加的新部分。

5.1.1 RSA 算法正确性验证

我通过如下的程序 test_RSA.cpp，利用 GoogleTest 库进行测试，并在其中输出一些公钥和私钥，以及每一步的加密结果，以此验证正确性。

验证代码如下：

```
1 #include <gtest/gtest.h>
2 #include "RSA.h"
3
4 int main(int argc, char **argv)
5 {
6     testing::InitGoogleTest(&argc, argv);
7     return RUN_ALL_TESTS();
8 }
9 TEST(RSAUtilsTests, Test1)
10 {
11     RSAUtils rsa;
12     rsa.init(10000);
13     cout << "Public key:(" << rsa.getPublicKey().first << ", " <<
        rsa.getPublicKey().second << ")" << endl;
14     cout << "Private key:(" << rsa.getPrivateKey().first << ", " <<
        rsa.getPrivateKey().second << ")" << endl;
15     uint64_t plaintext = 123456789;
16     uint64_t encryptedtext = 0;
17     uint64_t decryptedtext = 0;
18     rsa.encrypt(plaintext, encryptedtext);
19     cout << "Encrypted text: " << encryptedtext << endl;
20     rsa.decrypt(decryptedtext, encryptedtext);
21     cout << "Decrypted text: " << decryptedtext << endl;
22     EXPECT_EQ(plaintext, decryptedtext);
23 }
24 TEST(RSAUtilsTests, Test2)
25 {
26     RSAUtils rsa;
27     rsa.init(10000);
28     cout << "Public key:(" << rsa.getPublicKey().first << ", " <<
        rsa.getPublicKey().second << ")" << endl;
```

```

29     cout << "Private key:(" << rsa.getPrivateKey().first << ", " <<
        rsa.getPrivateKey().second << ")" << endl;
30     uint64_t plaintext = 987654321;
31     uint64_t encryptedtext = 0;
32     uint64_t decryptedtext = 0;
33     rsa.encrypt(plaintext, encryptedtext);
34     cout << "Encrypted text: " << encryptedtext << endl;
35     rsa.decrypt(decryptedtext, encryptedtext);
36     cout << "Decrypted text: " << decryptedtext << endl;
37     EXPECT_EQ(plaintext, decryptedtext);
38 }

```

验证结果如下：



```

G++ test_RSA.cpp M X
NetworkSecurity2024 > Labs > Lab02 > src > C++ test_RSA.cpp > TEST(RSAUtilsTests, Test2)
1 #include <gtest/gtest.h>
2 #include "RSA.h"
3
4 int main(int argc, char **argv)
5 {
6     testing::InitGoogleTest(&argc, argv);
7     return RUN_ALL_TESTS();
8 }
9 TEST(RSAUtilsTests, Test1)
10
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from RSAUtilsTests
[ RUN ] RSAUtilsTests.Test1
Public key:(65537, 8490795087134282821)
Private key:(1977173562043745405, 8490795087134282821)
Encrypted text: 2795120230378424601
Decrypted text: 123456789
[ OK ] RSAUtilsTests.Test1 (920 ms)
[ RUN ] RSAUtilsTests.Test2
Public key:(65537, 775673085778987461)
Private key:(6582564954312470273, 775673085778987461)
Encrypted text: 7525008624823862565
Decrypted text: 987654321
[ OK ] RSAUtilsTests.Test2 (914 ms)
[-----] 2 tests from RSAUtilsTests (1834 ms total)
[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (1834 ms total)
[ PASSED ] 2 tests.
"/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-r0d3mtr.zo1" 1>"/tmp/Microsoft-MIE
ngine-out-dlpe2lft.y00"
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes$

```

图 5.14: RSA 验证结果

如图5.14所示，可以看到 **RSA 全部通过了测试**，具体每一步的加密结果也可以进行验证。由此证明了 **RSA 算法的正确性**。

5.1.2 基于 RSA 的 DES 密钥共享方案

接下来展示关于基于 RSA 的 DES 密钥自动分配，在密钥分配之前就是和上次实验一样。服务器和客户端各种初始化，然后客户端输入 IP 地址（这里选择默认），连接成功。然后敌人潜入，由服务器端（总部）发布命令采用“Silent Guardian”行动，二者都进入这个状态。

```

NetworkSecurity2024 > Labs > Lab02 > src > DES_server.cpp > main()
53 int main()
229 cin.clear();
230 cin.ignore(numeric_limits<streamsize>::max(), '\n');

erwinzhou@erwinzhou-virtual-machine:~/Lab/codes/NetworkSecurity2024/Labs/Lab02/bin$ ./DES_client.exe
-----Secret Hideout-----
<SecretHideout::System @ Tue Apr 16 10:40:05 2024 # Message>:Stop there mate! Who are you?
Well, I am:James Bond
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Welcome back, Mr.007!
-----Agent:007-----
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:The mission is almost ready to launch !
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Please wait for a moment, Mr.007!
-----Creating Socket-----
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Successfully creating socket!
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Agent, please input the IP adress of the Headquarter in M16:
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:1:Use the Default IP Address
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:2:Input the IP Address Manually
1
-----Connecting to the Headquarter-----
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Successfully connected to the Headquarter!
<SecretHideout::System @ Tue Apr 16 10:40:12 2024 # Warning>:The enemy is potentially in earshot. We are under surveillance, be careful!
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:All channels have been compromised!
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:Silent Guardian is activated..
-----Waiting for RSA Key-----
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:RSA initializaing.....
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:Please enter the number of maxium rounds for RSA:
100
<SecretHideout::System @ Tue Apr 16 10:40:14 2024 # Message>:Please enter the default RSA key or not(1 for yes, 0 for no):
0
<SecretHideout::System @ Tue Apr 16 10:40:15 2024 # Message>:Please enter the high security mode or not(1 for yes, 0 for no):
0
<Headquarter::System @ Tue Apr 16 10:40:02 2024 # Message>:Successfully creating socket!
-----Binding Socket-----
<Headquarter::System @ Tue Apr 16 10:40:02 2024 # Message>:Successfully binding socket!
-----Listening for Agents-----
<Headquarter::System @ Tue Apr 16 10:40:02 2024 # Message>:Waiting for agents to come in...
<Headquarter::System @ Tue Apr 16 10:40:09 2024 # Message>:We got a agent from 0.0.0.0, port 0, socket 4
<Headquarter::System @ Tue Apr 16 10:40:09 2024 # Message>:Agent code name:007.
<Headquarter::System @ Tue Apr 16 10:40:12 2024 # WARNING>:The enemy is potentially in earshot. Be careful!
<Headquarter::System @ Tue Apr 16 10:40:12 2024 # Message>:Please give us further instructions!
1:Let's GO DARK! Silent Guardian, ACTIVATE!
2:Forget about their existence.
3:Terminate the connection! ASAP!
1
<Headquarter::System @ Tue Apr 16 10:40:13 2024 # Message>:Yes, sir!
<Headquarter::System @ Tue Apr 16 10:40:13 2024 # Message>:Silent Guardian is activated..
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:RSA initializaing.....
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:Please enter the number of maxium rounds for RSA:
100
<SecretHideout::System @ Tue Apr 16 10:40:14 2024 # Message>:Please enter the default RSA key or not(1 for yes, 0 for no):
0
<SecretHideout::System @ Tue Apr 16 10:40:15 2024 # Message>:Please enter the high security mode or not(1 for yes, 0 for no):
0

```

图 5.15: RSA 初始化输入

图5.15展示了 RSA 初始化时候，要求输入默认轮数，是否使用默认密钥，是否使用高级加密等。

```

<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:2:Input the IP Address Manually
1
-----Connecting to the Headquarter-----
<SecretHideout::System @ Tue Apr 16 10:40:09 2024 # Message>:Successfully connected to the Headquarter!
<SecretHideout::System @ Tue Apr 16 10:40:12 2024 # Warning>:The enemy is potentially in earshot. We are under surveillance, be careful!
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:All channels have been compromised!
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:Silent Guardian is activated..
-----Waiting for RSA Key-----
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:RSA initializaing.....
<SecretHideout::System @ Tue Apr 16 10:40:13 2024 # Message>:Please enter the number of maxium rounds for RSA:
100
<SecretHideout::System @ Tue Apr 16 10:40:14 2024 # Message>:Please enter the default RSA key or not(1 for yes, 0 for no):
0
<SecretHideout::System @ Tue Apr 16 10:40:15 2024 # Message>:Please enter the high security mode or not(1 for yes, 0 for no):
0
<Headquarter::System @ Tue Apr 16 10:40:57 2024 # Message>:Successfully sent the RSA Key!
-----Waiting for the DES Key-----
<Headquarter::System @ Tue Apr 16 10:40:57 2024 # Message>:Successfully received the DES Key!
-----Decrypting the DES Key-----
<Headquarter::System @ Tue Apr 16 10:40:57 2024 # Message>:DES Key Generated.
-----Everything Ready-----
-----Communication Start-----
特工你好
<Headquarter::System @ Tue Apr 16 10:41:01 2024 # Message>:Message Sent.
<Headquarter::007 @ Tue Apr 16 10:41:05 2024 # Message>: 总部你好
<Headquarter::System @ Tue Apr 16 10:41:09 2024 # Message>:Message Sent.
<SecretHideout::Headquarter @ Tue Apr 16 10:41:01 2024 # Message>: 特工你好
总部你好
<SecretHideout::Headquarter @ Tue Apr 16 10:41:09 2024 # Message>: 不错不错
我准备好接受任务了，随时准备着!
<Headquarter::System @ Tue Apr 16 10:41:26 2024 # Message>:Message Sent.

```

图 5.16: RSA 自动分配密钥完成

```

<SecretHideout::Headquarter @ Tue Apr 16 10:41:01 2024 # Message>: 特工
你好
总部你好
<Headquarter::System @ Tue Apr 16 10:41:05 2024 # Message>:Message Sent.

<SecretHideout::Headquarter @ Tue Apr 16 10:41:09 2024 # Message>: 不错
不错
我准备好接受任务了, 随时准备着!
<Headquarter::System @ Tue Apr 16 10:41:26 2024 # Message>:Message Sent.

<SecretHideout::Headquarter @ Tue Apr 16 10:42:50 2024 # Message>: OVER
<SecretHideout::System @ Tue Apr 16 10:42:50 2024 # Message>:The Headqua
rter is off the line.
<SecretHideout::System @ Tue Apr 16 10:42:50 2024 # Message>:Communicati
on is OVER.
-----Communication OVER-----
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/L
abs/Lab02/bin$

```

图 5.17: RSA 聊天结束

图5.16和图5.17可以看到如下内容:

- **RSA 自动分配:** 可以看到提示文字中显示了加密过程的日志信息。服务器端生成 RSA 公钥发送, 然后客户端接收到 RSA 公钥, 生成 DES 私钥并发送, 最后服务器端接收到 DES 公钥。正式开始聊天。
- **正常聊天:** 可以看到通信双方可以正常进行聊天, 发送的内容经过加密和解密都是一致的。这也侧面证明了 DES 密钥的分配成功, 证明了 RSA 的加解密成功。
- **聊天结束:** 可以看到在服务器端输入一个 OVER 后聊天正式结束, 客户端接收到该消息后自动结束, 服务器端忙等待下一个特工接入连接。

5.1.3 AIO 优化

最后这里展示一下我的 Asynchronous AIO 优化结果:

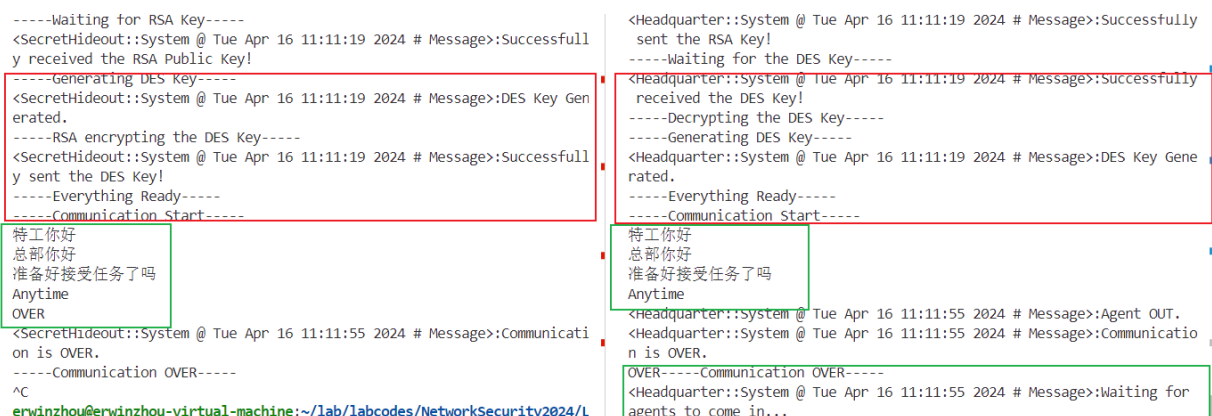
```

erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/L
abs/Lab02/bin$ ./DES_client_AIO.exe
-----Secret Hideout-----
<SecretHideout::System @ Tue Apr 16 11:11:05 2024 # Message>:Stop there
mate! Who are you?
Well, I am:James Bond
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:Welcome bac
k, Mr.007!
-----Agent:007-----
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:The mission
is almost ready to launch !
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:Please wait
for a moment, Mr.007!
-----Creating Socket-----
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:Successfull
y creating socket!
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:Agent, plea
se input the IP adress of the Headquarter in M16:
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:1:Use the D
efault IP Address
<SecretHideout::System @ Tue Apr 16 11:11:13 2024 # Message>:2:Input the
IP Address Manually
1

```

图 5.18: AIO 优化的基于 RSA 的 DES 密钥自动分配聊天程序启动

图5.18可以看到使用了这次是运行的 Asynchronous IO 优化后的服务器端和客户端可执行程序。启动过程初始化等都是是一样的。



```
-----Waiting for RSA Key-----
<SecretHideout::System @ Tue Apr 16 11:11:19 2024 # Message>:Successfully received the RSA Public Key!
-----Generating DES Key-----
<SecretHideout::System @ Tue Apr 16 11:11:19 2024 # Message>:DES Key Generated.
-----RSA encrypting the DES Key-----
<SecretHideout::System @ Tue Apr 16 11:11:19 2024 # Message>:Successfully sent the DES Key!
-----Everything Ready-----
-----Communication Start-----
特工你好
总部你好
准备好接受任务了吗
Anytime
OVER
<SecretHideout::System @ Tue Apr 16 11:11:55 2024 # Message>:Communication is OVER.
-----Communication OVER-----
^C
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/NetworkSecurity2024/L

<Headquarter::System @ Tue Apr 16 11:11:19 2024 # Message>:Successfully sent the RSA Key!
-----Waiting for the DES Key-----
<Headquarter::System @ Tue Apr 16 11:11:19 2024 # Message>:Successfully received the DES Key!
-----Decrypting the DES Key-----
-----Generating DES Key-----
<Headquarter::System @ Tue Apr 16 11:11:19 2024 # Message>:DES Key Generated.
-----Everything Ready-----
-----Communication Start-----
特工你好
总部你好
准备好接受任务了吗
Anytime
<Headquarter::System @ Tue Apr 16 11:11:55 2024 # Message>:Agent OUT.
<Headquarter::System @ Tue Apr 16 11:11:55 2024 # Message>:Communication is OVER.
OVER-----Communication OVER-----
<Headquarter::System @ Tue Apr 16 11:11:55 2024 # Message>:Waiting for agents to come in...
```

图 5.19: AIO 优化后的具体聊天过程

图5.19可以看到如下的重点：

- **正常的基于 RSA 的 DES 自动密钥分配：**该过程和未经过 AIO 优化的完全一致，因为还没到使用 AIO 优化的地方。可以看到经过生成了 DES 根密钥，发送到服务器端后，服务端也使用 RSA 私钥进行了解密。最终万事俱备，只欠东风！
- **聊天过程：**可以看到特工（客户端）和总部（服务器端）通过 TCP 进行聊天，聊天内容中英文均可以正常显示。双方均可以发送和接收内容。
- **聊天结束：**可以看到在客户端（特工）在 Stdin 输入了一个“OVER”后结束了通信。然后通过 Ctrl+C 信号量控制其结束。总部（服务器端）也是接收到了客户端的结束通信消息，重新回到最早的父进程接收下一位特工的连接。

Phantom Hook 的过程和 SilentGuardian 类似这里就不再重复展示。到此就展示了本次实验实现的全部功能，实验非常成功！

6 实验遇到的问题及其解决方法

6.1 RSA

6.1.1 整数溢出问题

- **遇到的问题：**首先我的 RSA 在刚开始实现时候出现了一些整数溢出问题，具体而言一般发生都在乘法过程中。而最底层进行封装乘法的就是 `mulMod` 函数。
- **解决方案：**最基础的运算分量采用 gcc 支持的更大的数字 `uint128_t` 存储。多注意在每一步通过 `mod` 运算化简。

6.1.2 Debug 困难

- **遇到的问题：**由于本次 RSA 实验中，在很多地方都可能会出现溢出问题。这是由于 RSA 经常在各种地方涉及大整数运算决定的。由于涉及的位置之多，需要一个很好的工具进行 Debug，没有 Visual Studio 只有 Visual Studio Code 的情况下，Debug 变得更加困难。

- 解决方案：通过智慧编程工具的生成样例测试功能和 Linux 的 Google TeST。

首先为了配置 Google Test，通过 apt-get 下载后。然后在 tasks.json 进行命令行设置，添加“-L/usr/lib”指定安装完成的 gtest 库。添加“-lgtest”和“-lrt”命令选项。

```

1      "tasks": [
2      {
3          "args": [
4              "-fdiagnostics-color=always",
5              "-g",
6              "-I",
7              "~/lab/labcodes/NetworkSecurity2024/Labs/Lab02/include",
8              "-L/usr/lib",
9              "-lgtest",
10             "-lrt",
11             "-o",
12             "${fileDirname}/../bin/${fileBasenameNoExtension}.exe"
13         ],

```

然后使用智慧编程工具 Copilot 自动生成测试样例和程序：

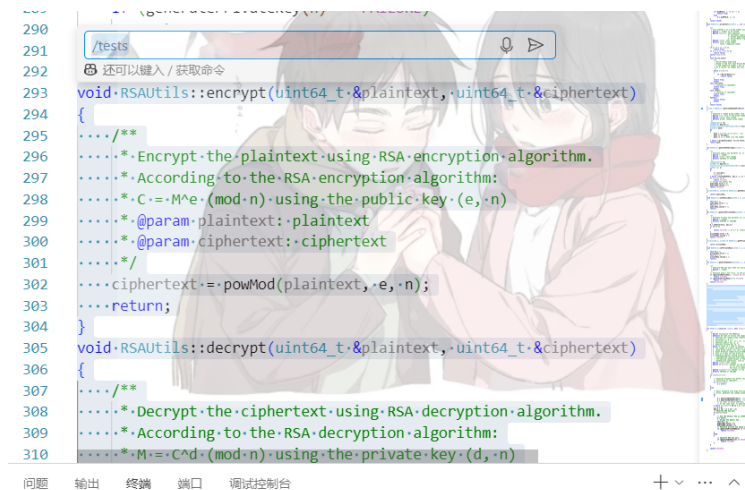


图 6.20: Github Copilot 自动生成测试样例

图6.20展示了使用 Copilot 的样子。会自动在 src 下帮我生成 test_RSA.cpp。

```

1 #include <gtest/gtest.h>
2 #include "RSA.h"
3
4 int main(int argc, char **argv)
5 {
6     testing::InitGoogleTest(&argc, argv);
7     return RUN_ALL_TESTS();
8 }
9 TEST(RSAUtilsTests, Test1)
10 {
11     RSAUtils rsa;

```



```

12     rsa.init();
13     uint64_t plaintext = 123456789;
14     uint64_t encryptedtext = 0;
15     uint64_t decryptedtext = 0;
16     rsa.encrypt(plaintext, encryptedtext);
17     rsa.decrypt(decryptedtext, encryptedtext);
18     EXPECT_EQ(plaintext, decryptedtext);
19 }
20 TEST(RSAUtilsTests, Test2)
21 {
22     RSAUtils rsa;
23     rsa.init();
24     uint64_t plaintext = 987654321;
25     uint64_t encryptedtext = 0;
26     uint64_t decryptedtext = 0;
27     rsa.encrypt(plaintext, encryptedtext);
28     rsa.decrypt(decryptedtext, encryptedtext);
29     EXPECT_EQ(plaintext, decryptedtext);
30 }

```

6.1.3 私钥 (d, n) 生成

- **遇到的问题：**在实验参考书中，生成私钥的方法与我最终选择的方法有些不同。

书中生成 RSA 解密的私钥参数 d 时，核心算法依赖于寻找一个整数 i ，使得 $(i * \phi(n) + 1)$ 可以被 e 整除。这里的 $\phi(n)$ ，其中 n 是 RSA 公钥的一部分（通常是两个大素数的乘积），而 e 是公钥的另一部分。**该算法目前使用的是简单的线性搜索，**

我发现在我选定的 p 和 q 都为 32 位时， n 的大小过大导致这段代码用于这在实际中效率非常低，尤其是对于大数的 RSA 加密。**这样的时间复杂度和响应时间根本不适用于实际应用。**

- **解决方案：**我通过编写辅助函数 `extendedGCD` 拓展欧几里得和 `modInverse` 求逆直接求解 d 。

具体而言就是根据扩展欧几里得算法，求解模线性方程（如 $ax + by = \gcd(a, b)$ ）。对于 RSA 中的私钥 d 的计算，实际上就是在求解 $ed \equiv 1 \pmod{\phi(n)}$ 的 d 值。**扩展欧几里得算法可以高效地求解出这样的 d ，其时间复杂度远低于线性搜索。**

6.2 DES

6.2.1 DES 随机密钥

- **遇到的问题：**本次实验虽然采用 RSA 非对称加密算法，但由于其时间复杂度较高，在实践中并不通过其对每一个消息进行加密，而是只对 DES 密钥进行加密，实现密钥自动分配。

不过 DES 的密钥随机生成实际上并不能完全随机，这是由于 RSA 加密本身的一些要求决定的。我最开始并不知道这些，导致加密和解密结果尽管在解决了大整数溢出问题后仍然不一致。

- **解决方案：**在 RSA 加密算法中，所有的操作都是在模 n 的算术环境中进行的，这里的 n 是两个大素数 p 和 q 的乘积。**对于任何有效的 RSA 操作，消息 m 必须小于 n 。**

这是因为在这种环境下，每个数字都被视为一个介于 0 和 $n - 1$ 之间的等价类。如果加密一个等于或大于 n 的数字，它实际上将会被简化为一个更小的等价类，从而导致信息丢失。

甚至如果消息 m 接近 n 的大小，它在加密过程中实际上首先被简化为 $m \bmod n$ ，可能与原始 m 大不相同，从而在解密时无法恢复原始消息。

因此为了保证信息保全和加解密的一致性。我在生成了 DES 随机密钥后，会通过一个 **while** 循环判断其是否大于 RSA 的公钥参数 n ，若是则重新生成 DES 根密钥。直到满足条件。

```

1      // Generating the DES key until the DES key is smaller than the RSA public key
2      while (true)
3      {
4          des.generateRandomRootKey();
5          if (des.getRootKey() >= rsa.getPublicKey().second)
6              continue;
7          break;
8      }

```

6.3 AIO

6.3.1 fork 灵异事件再临

- **遇到的问题：**上次实验中我就提到过，我的程序在通过 fork 进行进程分支后，程序控制流会忽略阻塞直接运行到结尾再重新回来。这导致第一遍运行时候会提前输出一些文本，这不是我想要的。
- **解决方案：**我虽然没有找到这个 bug 的原因，但是通过标志位 **control_flag** 对控制流进行限制，解决了问题。

```

1      REALSTART1:
2          if (mPid != 0)
3          {
4
5              /**
6               * Parent process
7               * In charge of the following:
8               * 1. Receive message from the headquarter
9               * 2. Decrypt the message
10              * 3. Output the message
11              */
12              char strSocketBuffer[255];
13              char encryptedtext[255];
14              char decryptedtext[255];
15              // Register the signal handler
16              signal(SIGTERM, signalHandler);
17              if (control_flag)
18              {
19                  SockinSingle *pSockin = new SockinSingle(AGENT, agent, des, bStop);
20                  aio_read(pSockin->m_pReq);
21                  // Wait until the semaphore value > 0

```



```
22         sem_wait(&bStop);
23         // Release the semaphore
24         sem_destroy(&bStop);
25         // Kill the child process
26         kill(nPid, SIGTERM);
27         return;
28     }
29     control_flag = true;
30     goto REALSTART1;
31 }
```

7 实验改善方向

7.1 DES

7.1.1 安全性增强

在上次实验参考中就提到过，DES 的安全性在如今时代已经早早落后。不仅是因为其具有 Feistel 网络架构，容易受到差分攻击。同时也是因为其密钥只有 56 位，长度较短，GPU 的广泛使用以及量子计算的问世，让 DES 安全性已经被时代与技术所淘汰，其安全性无法得到保障。

因此我考虑，未来改进将 TCP 加密算法改为使用 **TribbleDES 或者 AES**，以此提高安全性。

7.2 RSA

7.2.1 安全性增强

实验参考中提到：要想确保 RSA 算法的安全性，就必须保证 n 足够大，此外 RSA 的发明者建议 p 和 q 满足下列条件：

- p 和 q 长度应该仅相差几位；
- $(p-1)$ 和 $(q-1)$ 都应该有一个大的质因子；
- $(p-1)$ 和 $(q-1)$ 的最大公因数应该比较小；
- 若 $e < d$ 且 $d > n^{\frac{1}{4}}$ ，则 d 容易被确定。

另外还提到为了防御一种通过记录计算机解密消息所用时间来确定私钥的计时攻击。可以采用的方法还有保证幂运算结果相同你，随即延时以及将密文加以随机数隐藏的。

这些就是我的 RSA 的 highSecurity 提供的更高安全性，有待完善。

7.3 AIO

7.3.1 指示文字输入与输出

从实验结果可以看到，AIO 优化后的 TCP 通信，由于是通过 **aio_write** 进行输出和打印的。因此没有像单纯的多进程双全工通信那种的发言时间等提示。从第三方来看不好辨认发送某条消息的一方究竟是谁。

为了解决这个问题，未来有待完善其输出的内容，使得聊天过程更加清晰。

7.3.2 原生 AIO 优化

由于我采用的是 POSIX 的 Linux 的 AIO。其本质是通过多线程优化。POSIX AIO 是一个用户级实现，它在多个线程中执行正常的阻塞 I/O，因此给出了 I/O 异步的错觉。为了更好地实现真正的 Linux 的 AIO 优化，libaio 内核 AIO。libaio 通过异步 I/O 操作的内核支持，其中 io 请求实际上在内核中排队。这样进一步提高 TCP 性能。

7.3.3 资源释放问题

实际上我的 AIO 优化仍然有一定问题，那就是资源释放问题出于时间原因还没有解决，偶尔会出现问题。

具体而言就是 SilentGuardian_AIO 在服务器端通过"OVER"结束通信后，再使用信号 Ctrl+C 的退出，仍然会偶尔会出现导致端口被阻塞。使得第二次运行服务器端会出现问题。如果是客户端自动退出就没有这个问题。

若真出现上述问题，则需要每次手动修改端口并重新运行，这确实是我的 AIO 性能优化带来的一个缺陷。因此我提供了两套 exe 文件，DES_client.exe 和 DES_client_AIO。其中没有经过 AIO 优化的不存在资源释放问题。

7.4 多进程通信

7.4.1 AIO 实现

最后由于通过了 LinuxAIO 实现了进程的双全工通信，并且实现了异步非阻塞，这样我想到可以对服务器端每次等待下一个特工进入的监听阶段也使用同样的方式。这样以此实现多进程同时通信。可以同时监听多个客户端进程。

8 实验结论

本次实验是本学期网络安全实验课的第二次课程作业，在第一次 DES 加密的 TCP 聊天程序的基础上，我进一步实现了基于 RSA 的自动密钥分配，并拓展实现了 AIO 对 TCP 进行性能优化。我收获颇丰，具体而言：

1. 本次实验我继续维持着规范的编程架构，通过清晰的架构，让我本次代码具有极高的可读性和可维护性，我也会保持这个习惯，用于我今后的其它项目中。并且我还通过研究了 Visual Studio Code 在 Linux 平台下使用 Google Test 进行 Debug 的能力，提升了我解决问题的能力。
2. 本次实验我通过对 RSA 算法原理进行复习后，将全部流程进行了实现。在这期间还遇到了运行性能差，整数溢出等问题，但我凭借耐心一一解决。有的自主创新，并没有完全按照实验参考书的代码和解决思路。这不仅加深了我对非对称加密算法尤其是 RSA 算法有了更加清晰的认识，并且增强了我的信息安全实践能力。我还深入思考，对 RSA 的不安全性，易受攻击等特点尝试进行了升级，都有带拓展。
3. 本次实验在上次在多进程编程基础上，我对自己提出了更高的要求：通过查阅资料并研读实验参考书中代码，我实现了通过 Linux 平台上 POSIX 的 Asynchronous I/O 编程提高了 TCP 的聊天性能。编程过程中，我也遇到了诸如进程释放资源，进程通信等更加实际的多进程问题。极大地加深了我对多进程的理解，强化了我多进程编程的能力。同时最重要的让我能够区分同步

异步，阻塞非阻塞的区别，并实践到了如何调用函数来具体实现。真正做到对 TCP 通信进行优化。

4. 本次实验后我认真思考与总结，总结了许多实验解决问题的思路，找到了很多未来可以进一步改进的方向。我希望我可以在未来对本次实验工作进一步完善。

作为第一次实验的拓展实验，让我对 RSA 加密算法实现密钥自动分配以及 Aysnchronous I/O 都有了深刻的认识。我希望经过本学期的多次实验，能够将网络安全的理论知识学透，更重要的是将每次实验做好，更好地应用于实践！争取发挥更多的创造性，成为一名有趣的信安人。

参考文献