Table of Contents

Introduction 2
Setting-up of the Stations involved simulation requirements etc. 2
Network Connection 2
Socket programming 2
Diagram for TCP Socket Calls for Connection 3
All cryptosystem and Hashing strategies implemented 4
RSA Cryptography 4
SHA1 hashing 4
SEED Encryption 4
Discussion on the execution (steps) of your program 5
program explanation (on all methods used), overall program structure, data input/output, analysis results 10
input: keygen 10
AES Encryption 11
AES Decryption 11
SHA1 hashing 12
Generate session key 12
Seed Encryption 13
Remake session key 14
Questions 14
Conclusion 15
Reference 15

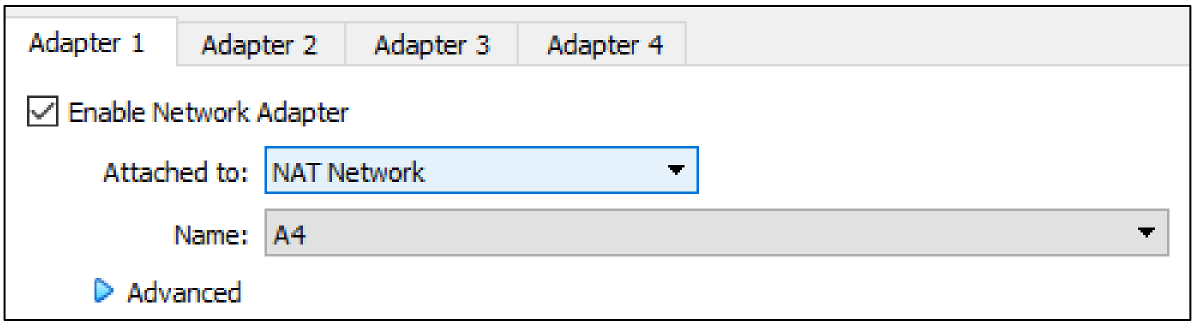
Introduction

This assignment covers socket programming, SEED encryption and RSA algorithm. Socket programming is being use for TCP connection on linux and SEED encryption is being used to encrypt the message send from client and server. SEED is a 128-bit symmetric key block cipher that has been developed by KISA (Korea Information Security Agency) and a group of experts since 1998. The input/output block size of SEED is 128-bit, and the key length is also 128-bit. SEED has the 16-round Feistel structure.

Setting-up of the Stations involved simulation requirements etc.

Network Connection

The simulation is performed under oracle virtual box where 2 virtual machine is being used which operates under Ubuntu OS and Oracle Linux. The network used for both to connects with each other is NAT Network.



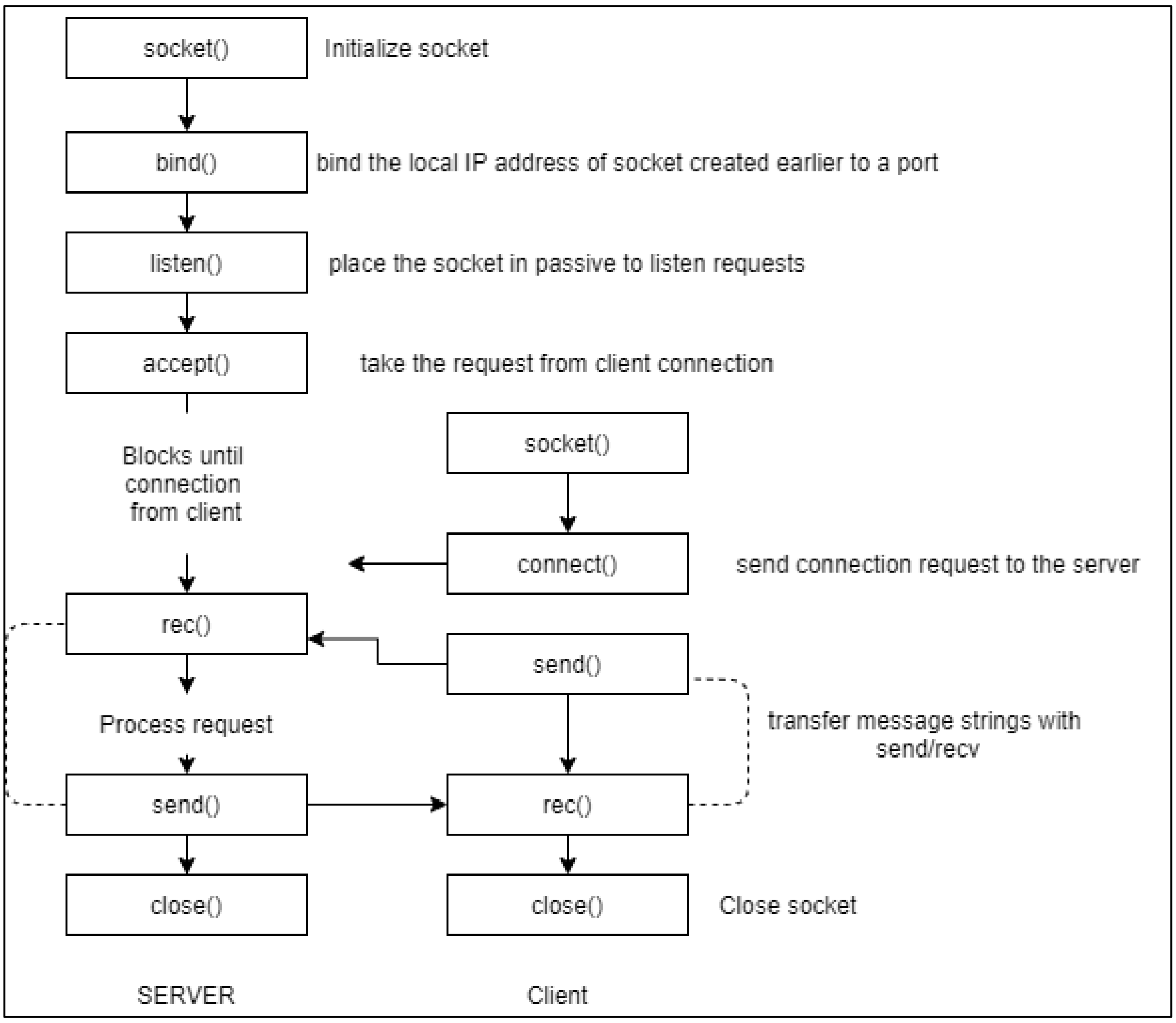
☑ Enable Network Adapter
Attached to: NAT Network
Name: A4

NAT Network stands for Network Address Translation where it translates the IP addresses of computers in a local network to a single address.

Socket programming

In this assignment socket programming is use to manage socket to create a data communication process between a client and server. Essentially, this program will be alike instant messaging program that communicate with TCP connection by using sockets. The client will relate to the server through an ip address of the server and port selected by server. The server will be able to listen for up to 5 requests at a time. After pending connection request is being accepted by the server, then the message will be sent back and forward through a buffer. Either of client or server decided to stop, at the end will shut down the socket then terminate the program. (Oumghar, 2021)

Diagram for TCP Socket Calls for Connection


socket()
Initialize socket
bind()
bind the local IP address of socket created earlier to a port
listen()
place the socket in passive to listen requests
accept()
take the request from client connection
Blocks until connection from client
socket()
connect()
send connection request to the server
rec()
send()
transfer message strings with send/recv
Process request
send()
rec()
close()
close()
Close socket
SERVER
Client

The diagram above is a process of how-to TCP connection is being processed.

All cryptosystem and Hashing strategies implemented

There are only 3 cryptosystem and hashing strategies is being implemented. Which is RSA algorithm, SHA1 hashing and finally SEED encryption.

RSA Cryptography

This cryptography consists of some operation, which include Keys, Encryption Schemes and Signature Schemes. With that being said, The RSA operation that used in the program is Keys operation and also Encryption Schemes.

Key operation is being used to generate public key and private key, this two keys is being used for encryption and decryption of the Session key for IDEA encryption. Besides, the encryption and decryption process are exposed through RSAES.

SHA1 hashing

SHA is the Secure Hash Standard and specified in FIPS 180-4. The security level of SHA-1 has been reduced to approximately 2^60. It is a symmetrical hash which mean it doesn't meant to be decrypted. In this assignment, SHA1 hashing is hashed using a pipeline with HashFilter() provided by crypto++ library.

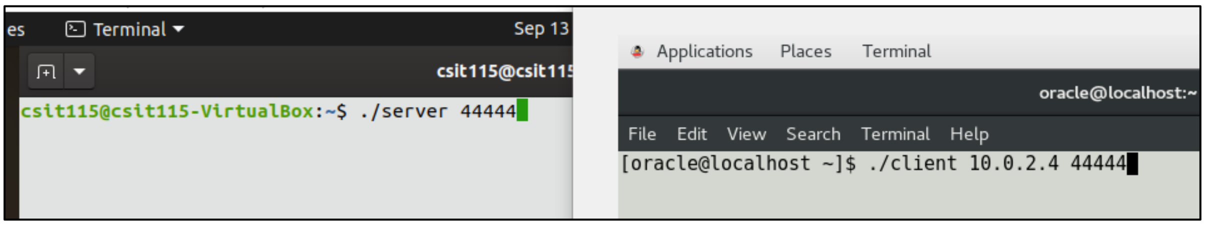
SEED Encryption

SEED is a national industry association standard that is extensively utilised in South Korea for wired and wireless electronic commerce and banking services.

Since 1998, KISA (Korea Information Security Agency) and a group of specialists have been developing SEED, a 128-bit symmetric key block cypher. SEED has a 128-bit input/output block size, as well as a 128-bit key length. SEED features a Feistel structure with 16 rounds. A 128-bit input is split into two 64-bit blocks, with the right 64-bit block being used as an input to the round function with a 64-bit subkey created via key scheduling.

SEED is simple to implement in a variety of software and hardware, and it works well in computing environments with limited resources, such as mobile devices and smart cards.

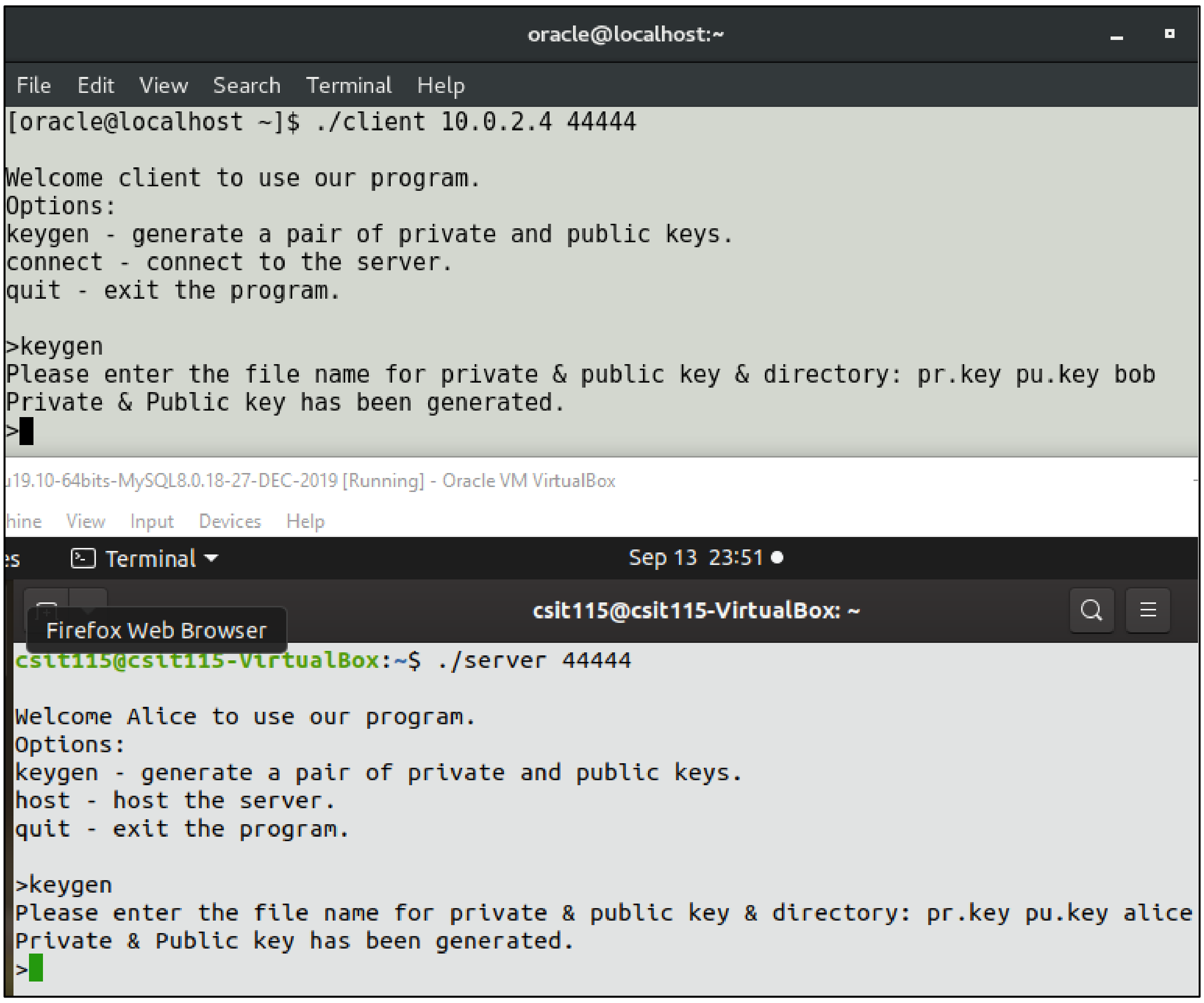
Discussion on the execution (steps) of your program



```
csit115@csit115-VirtualBox:~$ ./server 44444
```

```
[oracle@localhost ~]$ ./client 10.0.2.4 44444
```

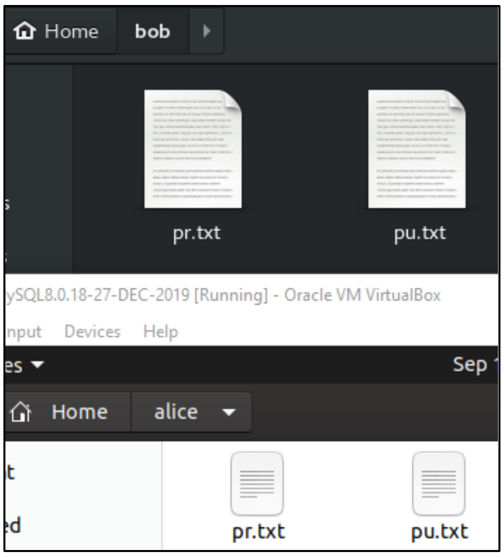
Firstly, the server is required to enter a port number to bind with a socket for the client to connects. After server is opened, then client will have to enter server's IP address and port number to get connected with the server.


oracle@localhost:~
File Edit View Search Terminal Help
19.10-64bits-MySQL8.0.18-27-DEC-2019 [Running] - Oracle VM VirtualBox
hine View Input Devices Help
Terminal
Sep 13 23:51
Firefox Web Browser
csit115@csit115-VirtualBox: ~

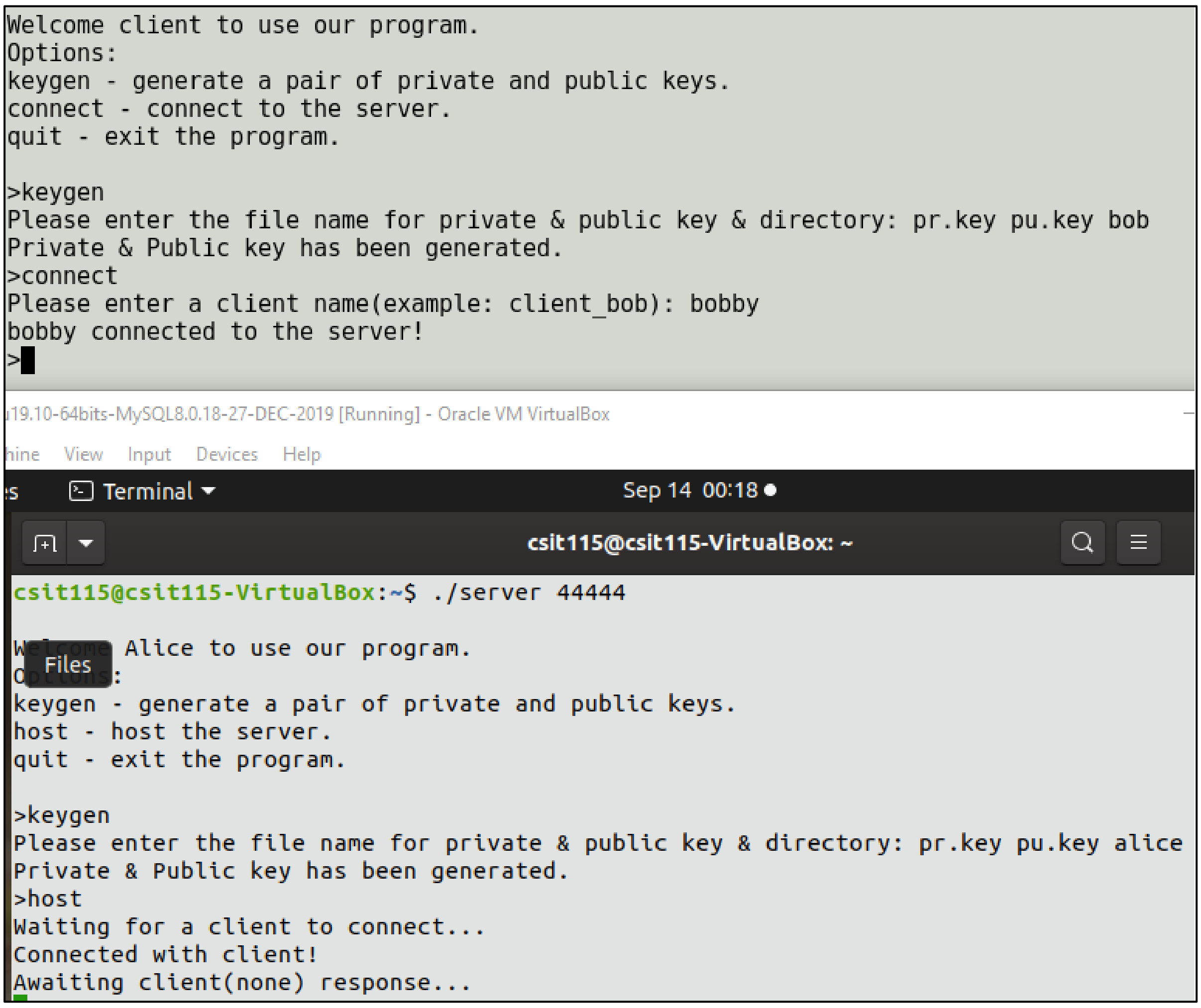
```
[oracle@localhost ~]$ ./client 10.0.2.4 44444

Welcome client to use our program.
Options:
keygen - generate a pair of private and public keys.
connect - connect to the server.
quit - exit the program.

>keygen
Please enter the file name for private & public key & directory: pr.key pu.key bob
Private & Public key has been generated.
>
```

```
csit115@csit115-VirtualBox:~$ ./server 44444

Welcome Alice to use our program.
Options:
keygen - generate a pair of private and public keys.
host - host the server.
quit - exit the program.

>keygen
Please enter the file name for private & public key & directory: pr.key pu.key alice
Private & Public key has been generated.
>
```

Then, Alice and Bob run **KeyGen** to generate a pair of theirs private and public keys with all required parameters based on RSA requirements included, the keys will be store in directory Alice and Bob respectively.



The keys are generated at Alice and Bob's directory as shown in the image above.

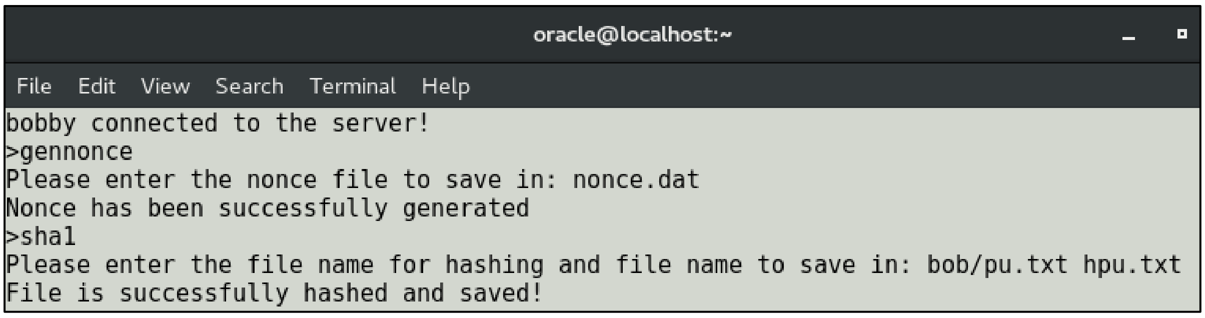


```
Welcome client to use our program.
Options:
keygen - generate a pair of private and public keys.
connect - connect to the server.
quit - exit the program.

>keygen
Please enter the file name for private & public key & directory: pr.key pu.key bob
Private & Public key has been generated.
>connect
Please enter a client name(example: client_bob): bobby
bobby connected to the server!
>
```

19.10-64bits-MySQL8.0.18-27-DEC-2019 [Running] - Oracle VM VirtualBox
hine View Input Devices Help
Terminal
Sep 14 00:18
csit115@csit115-VirtualBox: ~
Files

```
csit115@csit115-VirtualBox:~$ ./server 44444

Welcome Alice to use our program.
Options:
keygen - generate a pair of private and public keys.
host - host the server.
quit - exit the program.

>keygen
Please enter the file name for private & public key & directory: pr.key pu.key alice
Private & Public key has been generated.
>host
Waiting for a client to connect...
Connected with client!
Awaiting client(none) response...
```

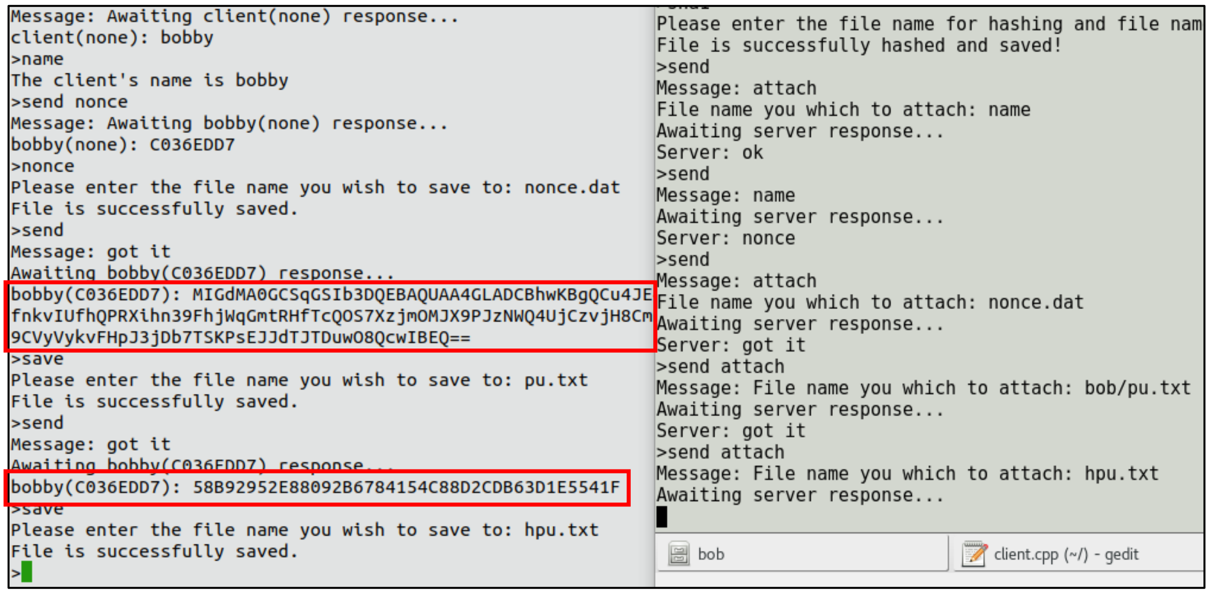
After Alice executes Host(Server), they it will display "waiting for a client to connect…". After Bob execute client program. The program will ask user to enter client name then it will be connected to the server.



```
bobby connected to the server!
>gennonce
Please enter the nonce file to save in: nonce.dat
Nonce has been successfully generated
>sha1
Please enter the file name for hashing and file name to save in: bob/pu.txt hpu.txt
File is successfully hashed and saved!
```

After bob entered client name “bobby”, now bob generates a random 4 byte nonce and hash his public key using **SHA-1**.

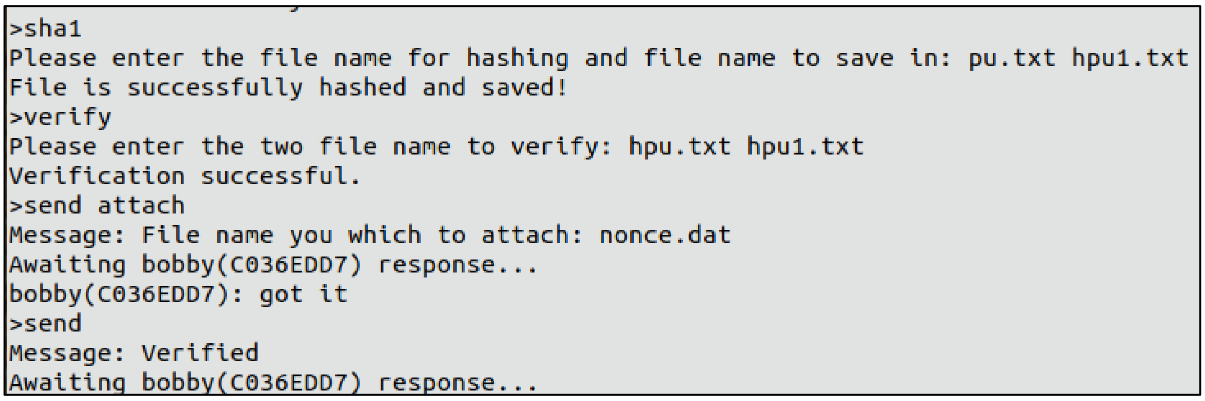
Server Client



```
Message: Awaiting client(none) response...
client(none): bobby
>name
The client's name is bobby
>send nonce
Message: Awaiting bobby(none) response...
bobby(none): C036EDD7
>nonce
Please enter the file name you wish to save to: nonce.dat
File is successfully saved.
>send
Message: got it
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQCu4JE
fnkvIUfhQPRXihn39FhjWqGmtRHfTcQOS7XzjmOMJX9PJzNWQ4UjCzvjH8Cm
9CVyVykvFHpJ3jDb7TSKPsEJJdTJTDuwO8QcwIBEQ==
>save
Please enter the file name you wish to save to: pu.txt
File is successfully saved.
>send
Message: got it
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): 58B92952E88092B6784154C88D2CDB63D1E5541F
>save
Please enter the file name you wish to save to: hpu.txt
File is successfully saved.
>
```

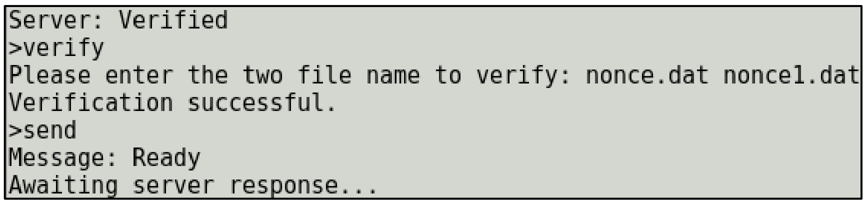
```
Please enter the file name for hashing and file nam
File is successfully hashed and saved!
>send
Message: attach
File name you which to attach: name
Awaiting server response...
Server: ok
>send
Message: name
Awaiting server response...
Server: nonce
>send
Message: attach
File name you which to attach: nonce.dat
Awaiting server response...
Server: got it
>send attach
Message: File name you which to attach: bob/pu.txt
Awaiting server response...
Server: got it
>send attach
Message: File name you which to attach: hpu.txt
Awaiting server response...
```

This is an image of data transmission of Bob sending his public key, client name, nonce, SHA-1 hashed public key. Besides, Alice also has Bob’s client name and nonce displayed.



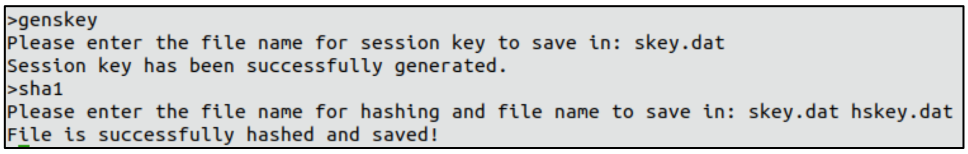
```
>sha1
Please enter the file name for hashing and file name to save in: pu.txt hpu1.txt
File is successfully hashed and saved!
>verify
Please enter the two file name to verify: hpu.txt hpu1.txt
Verification successful.
>send attach
Message: File name you which to attach: nonce.dat
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): got it
>send
Message: Verified
Awaiting bobby(C036EDD7) response...
```

Alice (server) now SHA1 hash the bob’s public key then perform an integrity check, then send earlier received nonce back to Bob with the “**Verified**” message.



```
Server: Verified
>verify
Please enter the two file name to verify: nonce.dat nonce1.dat
Verification successful.
>send
Message: Ready
Awaiting server response...
```

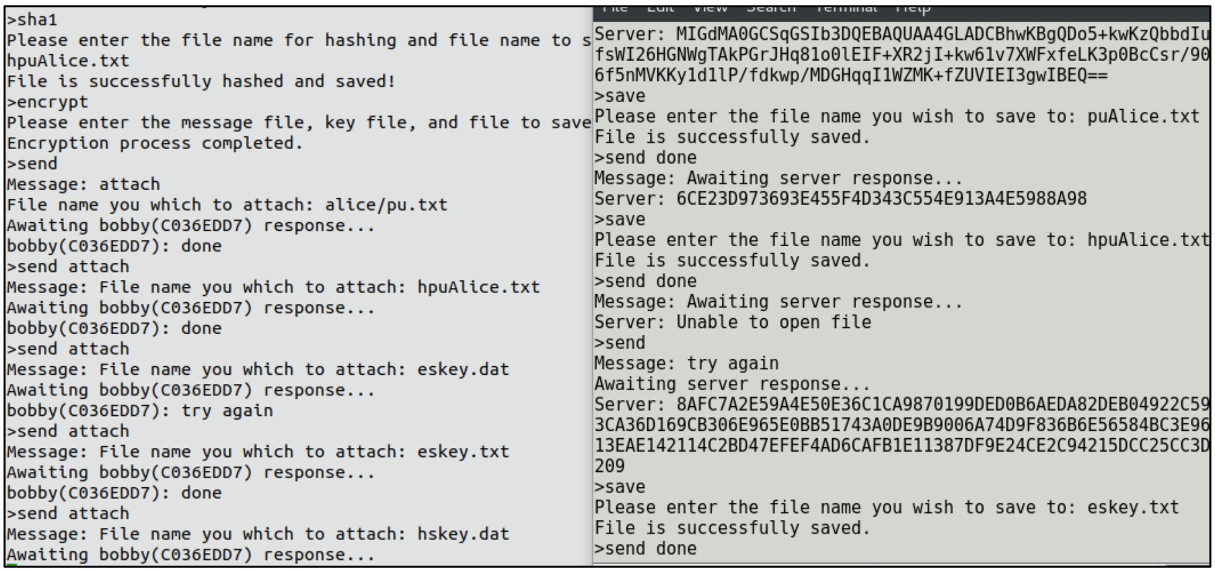
Bob (client) received message “Verified” now he match the nonce with his owns then send the message “Ready”.



```
>genskey
Please enter the file name for session key to save in: skey.dat
Session key has been successfully generated.
>sha1
Please enter the file name for hashing and file name to save in: skey.dat hskey.dat
File is successfully hashed and saved!
```

After Alice (server) received “Ready” message, then generates 128 bit SEED session key and hash it with SHA-1.

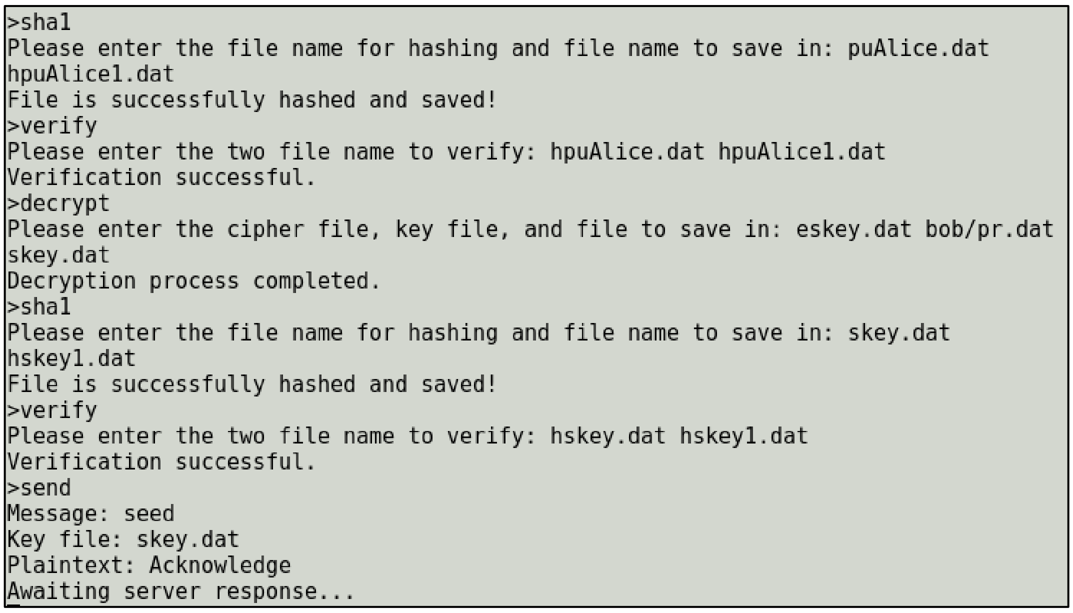
Server Client



```
>sha1
Please enter the file name for hashing and file name to s
hpuAlice.txt
File is successfully hashed and saved!
>encrypt
Please enter the message file, key file, and file to save
Encryption process completed.
>send
Message: attach
File name you which to attach: alice/pu.txt
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): done
>send attach
Message: File name you which to attach: hpuAlice.txt
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): done
>send attach
Message: File name you which to attach: eskey.dat
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): try again
>send attach
Message: File name you which to attach: eskey.txt
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): done
>send attach
Message: File name you which to attach: hskey.dat
Awaiting bobby(C036EDD7) response...
```

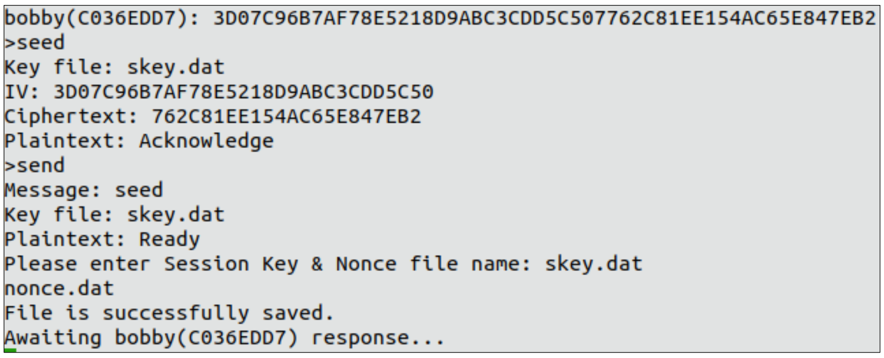
```
Server: MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQDo5+kwKzQbbdIu
fsWI26HGNWgTAkPGrJHq81o0lEIF+XR2jI+kw61v7XWFxfeLK3p0BcCsr/90
6f5nMVKKy1d1lP/fdkwp/MDGHqqI1WZMK+fZUVIEI3gwIBEQ==
>save
Please enter the file name you wish to save to: puAlice.txt
File is successfully saved.
>send done
Message: Awaiting server response...
Server: 6CE23D973693E455F4D343C554E913A4E5988A98
>save
Please enter the file name you wish to save to: hpuAlice.txt
File is successfully saved.
>send done
Message: Awaiting server response...
Server: Unable to open file
>send
Message: try again
Awaiting server response...
Server: 8AFC7A2E59A4E50E36C1CA9870199DED0B6AEDA82DEB04922C59
3CA36D169CB306E965E0BB51743A0DE9B9006A74D9F836B6E56584BC3E96
13EAE142114C2BD47EFEF4AD6CAFB1E11387DF9E24CE2C94215DCC25CC3D
209
>save
Please enter the file name you wish to save to: eskey.txt
File is successfully saved.
>send done
```

This diagram shows server transmitting over Alice’s public key, SHA-1 hashed Alice’s public key, encrypted 128 bit SEED session key with Bob’s public key and SHA-1 hashed session key.



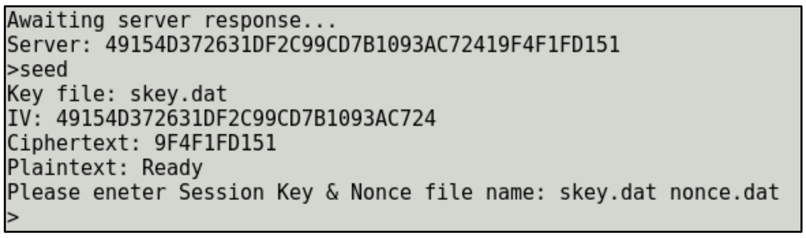
```
>sha1
Please enter the file name for hashing and file name to save in: puAlice.dat
hpuAlice1.dat
File is successfully hashed and saved!
>verify
Please enter the two file name to verify: hpuAlice.dat hpuAlice1.dat
Verification successful.
>decrypt
Please enter the cipher file, key file, and file to save in: eskey.dat bob/pr.dat
skey.dat
Decryption process completed.
>sha1
Please enter the file name for hashing and file name to save in: skey.dat
hskey1.dat
File is successfully hashed and saved!
>verify
Please enter the two file name to verify: hskey.dat hskey1.dat
Verification successful.
>send
Message: seed
Key file: skey.dat
Plaintext: Acknowledge
Awaiting server response...
```

After bob received all the component from server, then bob perform an integrity check on Alice’s public key. After that, Bob decrypts the SEED encrypted session key with his private key and perform another integrity check on it. Lastly, Bob sends SEED encrypted “Acknowledge” message with session key from server.



```
bobby(C036EDD7): 3D07C96B7AF78E5218D9ABC3CDD5C507762C81EE154AC65E847EB2
>seed
Key file: skey.dat
IV: 3D07C96B7AF78E5218D9ABC3CDD5C50
Ciphertext: 762C81EE154AC65E847EB2
Plaintext: Acknowledge
>send
Message: seed
Key file: skey.dat
Plaintext: Ready
Please enter Session Key & Nonce file name: skey.dat
nonce.dat
File is successfully saved.
Awaiting bobby(C036EDD7) response...
```

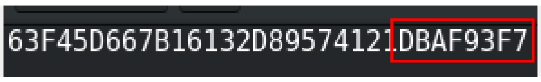
Alice (server) received the message “Acknowledge” after decrypting the ciphertext with session key, then send the encrypted message “Ready” to Bob (client). Also the system detected “Ready” sent, therefore the remake session key process has been conducted.



```
Awaiting server response...
Server: 49154D372631DF2C99CD7B1093AC72419F4F1FD151
>seed
Key file: skey.dat
IV: 49154D372631DF2C99CD7B1093AC724
Ciphertext: 9F4F1FD151
Plaintext: Ready
Please eneter Session Key & Nonce file name: skey.dat nonce.dat
>
```

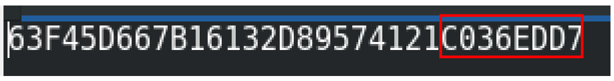
After “Ready” were received by Bob (client), then the program automatically conduct the session remake process.

Before



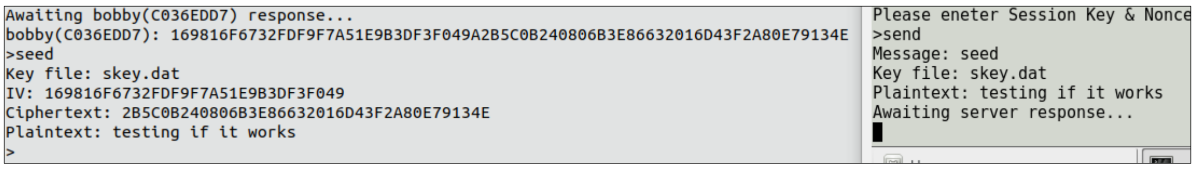
```
63F45D667B16132D89574121DBAF93F7
```

After



```
63F45D667B16132D89574121C036EDD7
```

Last 4 bytes of session key changed to nonce which is generated randomly by client.



```
Awaiting bobby(C036EDD7) response...
bobby(C036EDD7): 169816F6732FDF9F7A51E9B3DF3F049A2B5C0B240806B3E86632016D43F2A80E79134E
>seed
Key file: skey.dat
IV: 169816F6732FDF9F7A51E9B3DF3F049
Ciphertext: 2B5C0B240806B3E86632016D43F2A80E79134E
Plaintext: testing if it works
>
```

```
Please eneter Session Key & Nonce
>send
Message: seed
Key file: skey.dat
Plaintext: testing if it works
Awaiting server response...
```

A message sent from client to server with the new session key to demonstrate the session key is match for both sides.

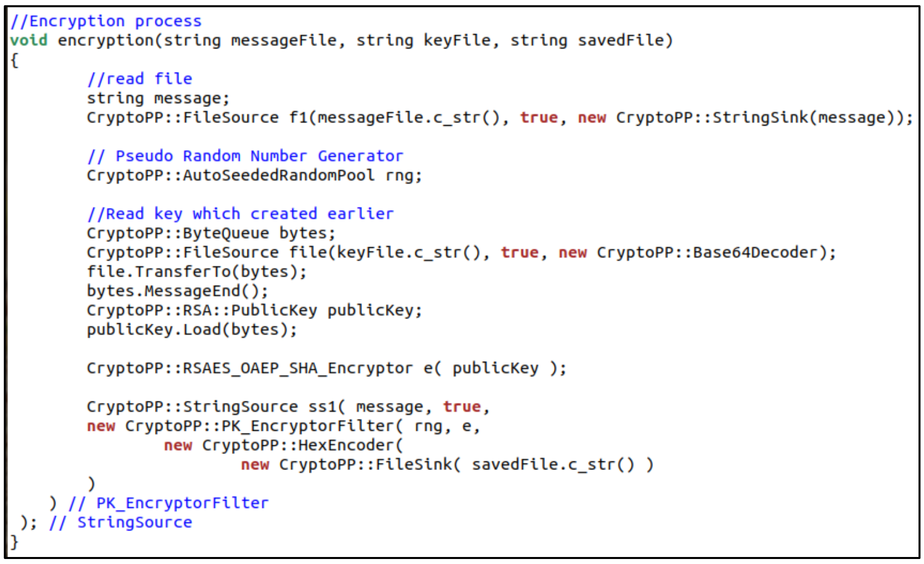
program explanation (on all methods used), overall program structure, data input/output, analysis results

input: keygen

```
void keyGen(string prFileName, string puFileName, string dir)
{
    // InvertibleRSAFunction is used directly only because the private key
    // won't actually be used to perform any cryptographic operation;
    // otherwise, an appropriate typedef'ed type from rsa.h would have been used.
    CryptoPP::AutoSeededRandomPool rng;
    CryptoPP::InvertibleRSAFunction privkey;
    privkey.Initialize(rng, 1024);

    // With the current version of Crypto++, MessageEnd() needs to be called
    // explicitly because Base64Encoder doesn't flush its buffer on destruction.
    string pr = dir + "/" + prFileName;
    CryptoPP::Base64Encoder privkeysink(new CryptoPP::FileSink(pr.c_str()));
    privkey.DEREncode(privkeysink);
    privkeysink.MessageEnd();

    // Suppose we want to store the public key separately,
    // possibly because we will be sending the public key to a third party.
    CryptoPP::RSAFunction pubkey(privkey);
    string pu = dir+ "/" + puFileName;
    CryptoPP::Base64Encoder pubkeysink(new CryptoPP::FileSink(pu.c_str()));
    pubkey.DEREncode(pubkeysink);
    pubkeysink.MessageEnd();

}
```

This is the function being use when generating the public and private keys, the only input requires is the file name for the keys to be saved in. It's a reuse code from my 361-cryptography assignment 4.

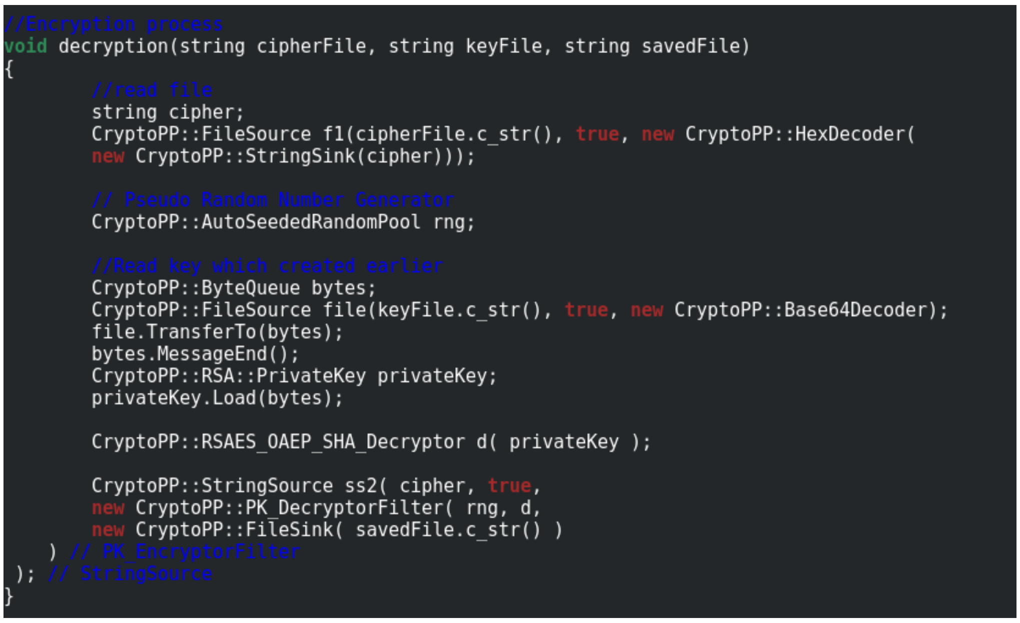
AES Encryption



```
//Encryption process
void encryption(string messageFile, string keyFile, string savedFile)
{
        //read file
        string message;
        CryptoPP::FileSource f1(messageFile.c_str(), true, new CryptoPP::StringSink(message));

        // Pseudo Random Number Generator
        CryptoPP::AutoSeededRandomPool rng;

        //Read key which created earlier
        CryptoPP::ByteQueue bytes;
        CryptoPP::FileSource file(keyFile.c_str(), true, new CryptoPP::Base64Decoder);
        file.TransferTo(bytes);
        bytes.MessageEnd();
        CryptoPP::RSA::PublicKey publicKey;
        publicKey.Load(bytes);

        CryptoPP::RSAES_OAEP_SHA_Encryptor e( publicKey );

        CryptoPP::StringSource ss1( message, true,
        new CryptoPP::PK_EncryptorFilter( rng, e,
                new CryptoPP::HexEncoder(
                        new CryptoPP::FileSink( savedFile.c_str() )
        )
    ) // PK_EncryptorFilter
 ); // StringSource
}
```

This is an encryption algorithm for RSA encryption. It start off by read the message for encryption then perform encrypt process with RSAES Encryptor. Lastly, it write to file with Hex encoded format to prevent data loss.

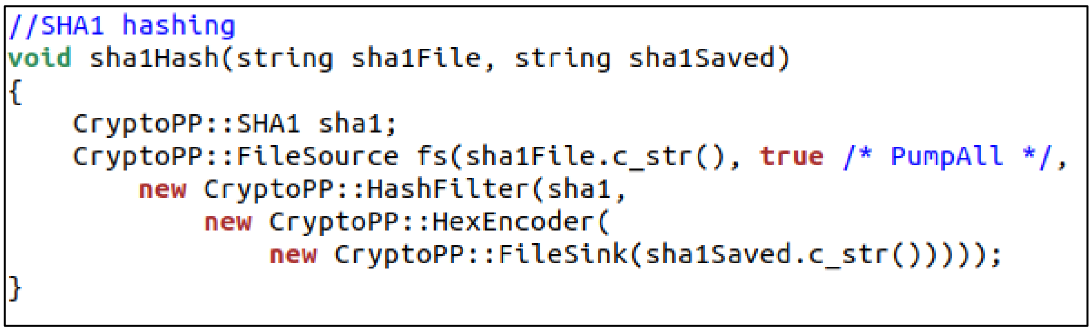
AES Decryption



```
//Encryption process
void decryption(string cipherFile, string keyFile, string savedFile)
{
        //read file
        string cipher;
        CryptoPP::FileSource f1(cipherFile.c_str(), true, new CryptoPP::HexDecoder(
        new CryptoPP::StringSink(cipher)));

        // Pseudo Random Number Generator
        CryptoPP::AutoSeededRandomPool rng;

        //Read key which created earlier
        CryptoPP::ByteQueue bytes;
        CryptoPP::FileSource file(keyFile.c_str(), true, new CryptoPP::Base64Decoder);
        file.TransferTo(bytes);
        bytes.MessageEnd();
        CryptoPP::RSA::PrivateKey privateKey;
        privateKey.Load(bytes);

        CryptoPP::RSAES_OAEP_SHA_Decryptor d( privateKey );

        CryptoPP::StringSource ss2( cipher, true,
        new CryptoPP::PK_DecryptorFilter( rng, d,
        new CryptoPP::FileSink( savedFile.c_str() )
    ) // PK_EncryptorFilter
 ); // StringSource
}
```

This is very similar to the encryption process mention earlier and it also uses RSAES Decryptor for the decryption. Besides, HexDecoder is used at the file reading process due to the data has been HexEncoded when encryption write to file.

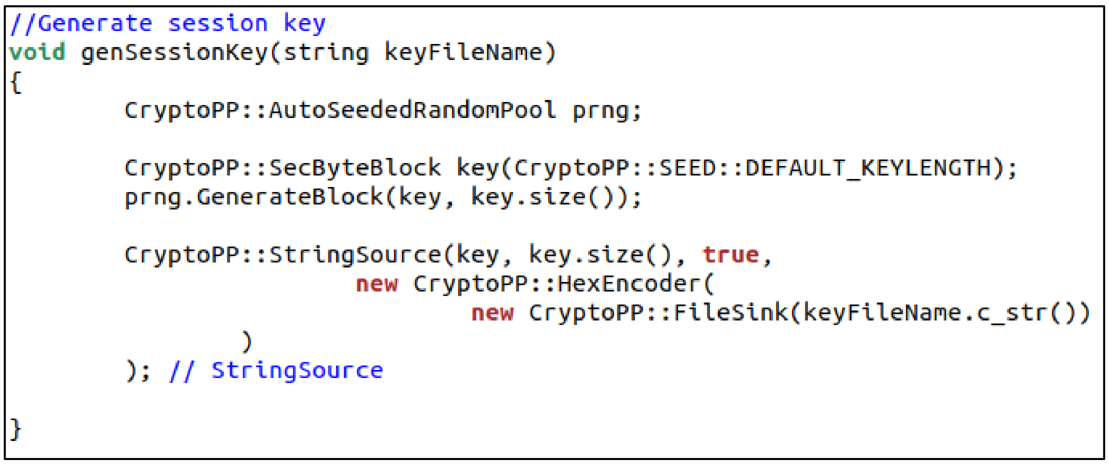
SHA1 hashing



```
//SHA1 hashing
void sha1Hash(string sha1File, string sha1Saved)
{
    CryptoPP::SHA1 sha1;
    CryptoPP::FileSource fs(sha1File.c_str(), true /* PumpAll */,
        new CryptoPP::HashFilter(sha1,
            new CryptoPP::HexEncoder(
                new CryptoPP::FileSink(sha1Saved.c_str()))));
}
```

This function uses CryptoPP library's SHA1 hashing with Hash Filter to hash the message then written to files with Hex encoded values.

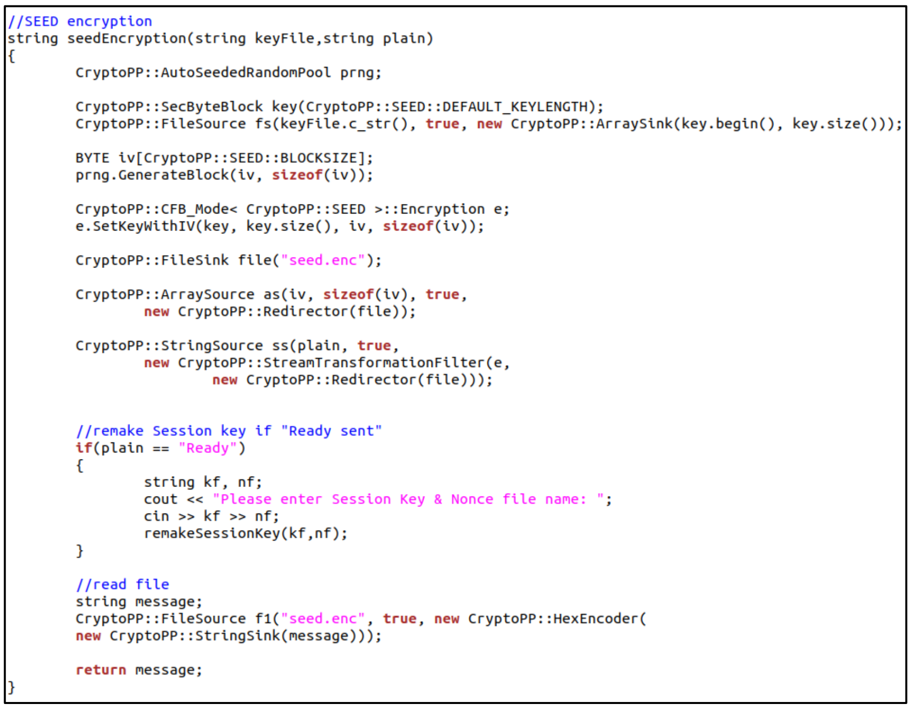
Generate session key



```
//Generate session key
void genSessionKey(string keyFileName)
{
        CryptoPP::AutoSeededRandomPool prng;

        CryptoPP::SecByteBlock key(CryptoPP::SEED::DEFAULT_KEYLENGTH);
        prng.GenerateBlock(key, key.size());

        CryptoPP::StringSource(key, key.size(), true,
                        new CryptoPP::HexEncoder(
                                new CryptoPP::FileSink(keyFileName.c_str())
                )
        ); // StringSource

}
```

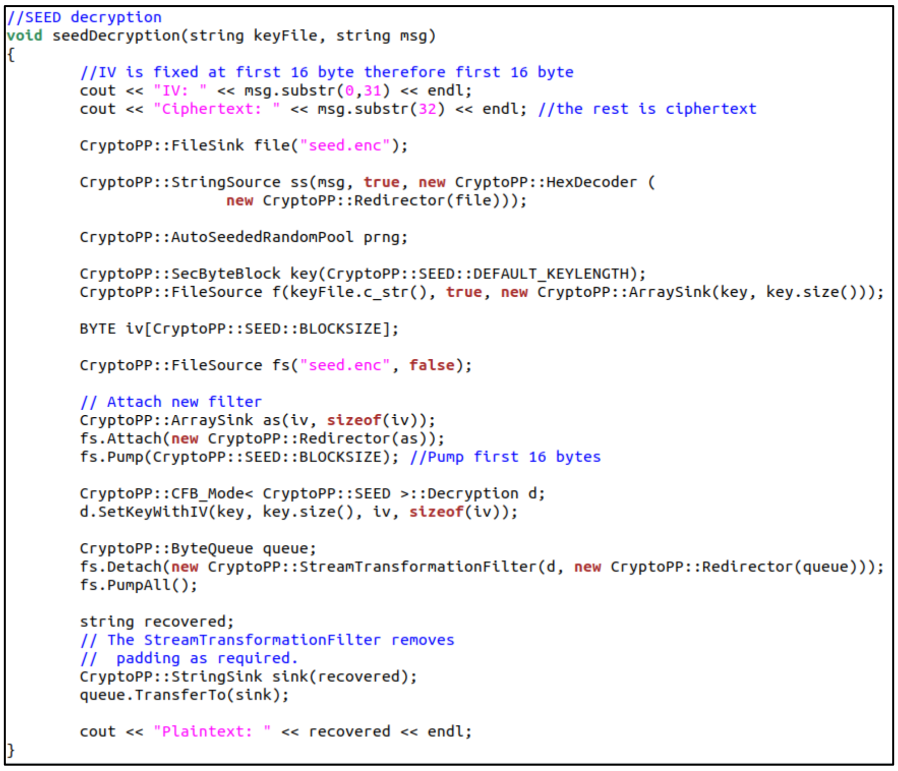
This function is being use for Session Key generate, the require input Is file name that user wish to save into. It uses crypto++ library's SecByteBlock and RNG to random generates a key with a SEED default key size which is 16 then store it into a file.

Seed Encryption



```
//SEED encryption
string seedEncryption(string keyFile,string plain)
{
	CryptoPP::AutoSeededRandomPool prng;

	CryptoPP::SecByteBlock key(CryptoPP::SEED::DEFAULT_KEYLENGTH);
	CryptoPP::FileSource fs(keyFile.c_str(), true, new CryptoPP::ArraySink(key.begin(), key.size()));

	BYTE iv[CryptoPP::SEED::BLOCKSIZE];
	prng.GenerateBlock(iv, sizeof(iv));

	CryptoPP::CFB_Mode< CryptoPP::SEED >::Encryption e;
	e.SetKeyWithIV(key, key.size(), iv, sizeof(iv));

	CryptoPP::FileSink file("seed.enc");

	CryptoPP::ArraySource as(iv, sizeof(iv), true,
		new CryptoPP::Redirector(file));

	CryptoPP::StringSource ss(plain, true,
		new CryptoPP::StreamTransformationFilter(e,
			new CryptoPP::Redirector(file)));


	//remake Session key if "Ready sent"
	if(plain == "Ready")
	{
		string kf, nf;
		cout << "Please enter Session Key & Nonce file name: ";
		cin >> kf >> nf;
		remakeSessionKey(kf,nf);
	}

	//read file
	string message;
	CryptoPP::FileSource f1("seed.enc", true, new CryptoPP::HexEncoder(
	new CryptoPP::StringSink(message)));

	return message;
}
```

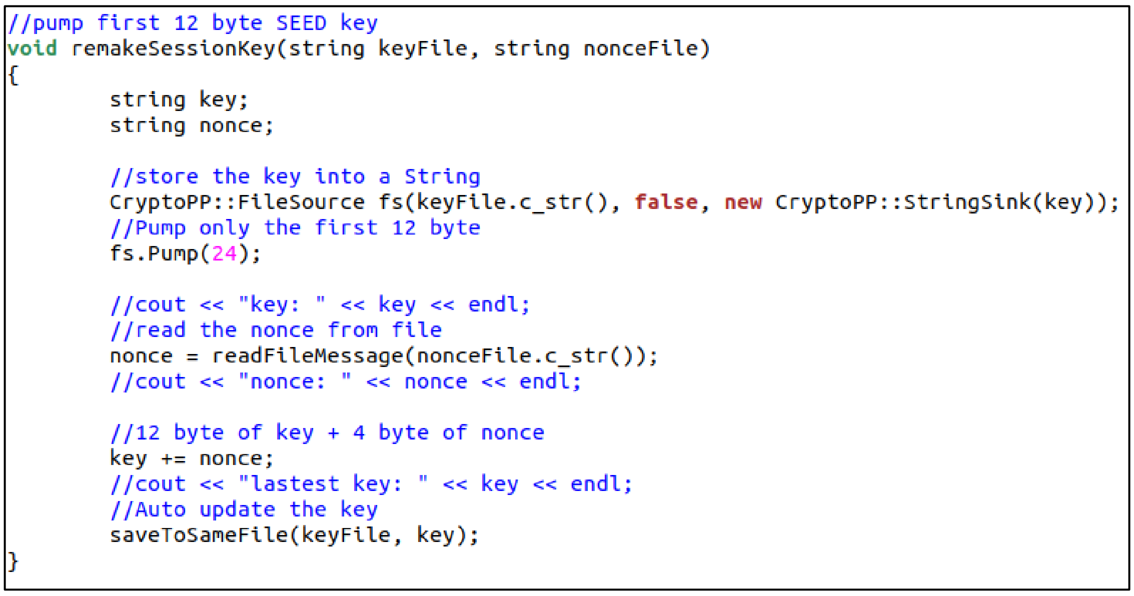
This is the function that perform SEED encryption, the required input by the user is session key's file name and the plaintext that wanted to send by the user. The explanation of this function is detail covered in the demo video.



```
//SEED decryption
void seedDecryption(string keyFile, string msg)
{
	//IV is fixed at first 16 byte therefore first 16 byte
	cout << "IV: " << msg.substr(0,31) << endl;
	cout << "Ciphertext: " << msg.substr(32) << endl; //the rest is ciphertext

	CryptoPP::FileSink file("seed.enc");

	CryptoPP::StringSource ss(msg, true, new CryptoPP::HexDecoder (
				new CryptoPP::Redirector(file)));

	CryptoPP::AutoSeededRandomPool prng;

	CryptoPP::SecByteBlock key(CryptoPP::SEED::DEFAULT_KEYLENGTH);
	CryptoPP::FileSource f(keyFile.c_str(), true, new CryptoPP::ArraySink(key, key.size()));

	BYTE iv[CryptoPP::SEED::BLOCKSIZE];

	CryptoPP::FileSource fs("seed.enc", false);

	// Attach new filter
	CryptoPP::ArraySink as(iv, sizeof(iv));
	fs.Attach(new CryptoPP::Redirector(as));
	fs.Pump(CryptoPP::SEED::BLOCKSIZE); //Pump first 16 bytes

	CryptoPP::CFB_Mode< CryptoPP::SEED >::Decryption d;
	d.SetKeyWithIV(key, key.size(), iv, sizeof(iv));

	CryptoPP::ByteQueue queue;
	fs.Detach(new CryptoPP::StreamTransformationFilter(d, new CryptoPP::Redirector(queue)));
	fs.PumpAll();

	string recovered;
	// The StreamTransformationFilter removes
	//  padding as required.
	CryptoPP::StringSink sink(recovered);
	queue.TransferTo(sink);

	cout << "Plaintext: " << recovered << endl;
}
```

This is exactly same as above function except this is a decryption process. Covered in the video as well.

Remake session key



```
//pump first 12 byte SEED key
void remakeSessionKey(string keyFile, string nonceFile)
{
	string key;
	string nonce;

	//store the key into a String
	CryptoPP::FileSource fs(keyFile.c_str(), false, new CryptoPP::StringSink(key));
	//Pump only the first 12 byte
	fs.Pump(24);

	//cout << "key: " << key << endl;
	//read the nonce from file
	nonce = readFileMessage(nonceFile.c_str());
	//cout << "nonce: " << nonce << endl;

	//12 byte of key + 4 byte of nonce
	key += nonce;
	//cout << "lastest key: " << key << endl;
	//Auto update the key
	saveToSameFile(keyFile, key);
}
```

The purpose of this function is to remake the session key as only keeping the first 12 byte of the key and using the nonce as the remaining 4 bytes. The functions store the original key in a string with only pumping 12 bytes of the whole session key into it then adding it with the nonce as the remaining byte. Finally, it saves into the same key file as user input.

Questions

1. What happen if acknowledgment messages such as “verified”, acknowledge” and “ready” never arrives or corrupted during transmission?

   Ans: The program will not proceed to the next step if this kind of message never arrives or corrupted during the transmission. The method I implement in the program is using if statement on these messages, if the message = “ready” then it automatically proceeds.

2. What happen if connection is terminated half-way?

   Ans: If the connection being terminated half-way by the user, then on the other user screen will see server/client leave to session and the connection is closed.

3. How to ensure the program continue running and “catch” this error?

   Ans: Setup error handling for the program to prevent program terminate itself once error occur. Instead, the program should output the error message then the rest works just fine.

Conclusion

In this report, I've covered all the use of function, program flow, how socket programming is being use and what hashing/ cryptography strategy is being used. Finally, I did a similar assignment during my 361 cryptographies, therefore lots of reuse of my own code. Since this is the second time doing it, I consider this an not challenging assignment like the first time I did (which is very difficult).

Reference

1. Oumghar, V., 2021. *Client-Server chat in C++ using sockets*. [online] Bits and Pieces of Code. Available at: <https://simpledevcode.wordpress.com/2016/06/16/client-server-chat-in-c-using-sockets/> [Accessed 1 September 2021].

2. Cryptopp.com. 2021. *SHA - Crypto++ Wiki*. [online] Available at: <https://www.cryptopp.com/wiki/SHA> [Accessed 11 September 2021].

3. Cryptopp.com. 2021. *SEED - Crypto++ Wiki*. [online] Available at: <https://www.cryptopp.com/wiki/SEED> [Accessed 11 September 2021].

4. Lee, H. and Yoon, J., 2021. *The SEED Cipher Algorithm and Its Use with IPsec*. [online] Ipa.go.jp. Available at: <https://www.ipa.go.jp/security/rfc/RFC4196EN.html> [Accessed 12 September 2021].