



## Assignment 4

LEE JUN LIN  
6664222

18-06-2021

—  
Cryptography and Secure Applications

—  
MOHAMMAD FAIZAL ALIAS

---

Table of Contents

Introduction ..... 3

RSA ..... 4

    Introduction..... 4

IDEA ..... 5

    Introduction..... 5

SHA-1 ..... 6

    Introduction..... 6

Crypto++ ..... 7

    Introduction..... 7

Implementation ..... 8

    Requirements ..... 8

    Summary..... 9

    Instructions to run..... 21

Appendix ..... 24

# Introduction

This assignment consists of 1 part. This assignment is recommended to be implemented with Crypto++ library in C++. The total weightage of this assignment is 10%.

# RSA

## Introduction

RSA is the work of Ron Rivest, Adi Shamir, and Leonard Adleman. The work is based on the Integer Factorization Problem. The system includes RSAES (encryption scheme), and RSASS (signature scheme).

In this assignment, we are required to use both RSASS and RSAES. The assignment requires us to create a public key on the client side and send it to the server side along with the digest using SHA1 and RSASS.

After server has confirmed on the originality using the digest appended to the public key, RSAES will be used to encrypt a session key that only the server and client will only be able to see. Server will send the encrypted session key and the digest of the session key using SHA1 and no RSASS will be used here.

# IDEA

## Introduction

IDEA is the International Data Encryption Standard cipher that was created by Massey and Lai. It is a 64-bit cipher that uses 126-bit keys and a 64-bit initialization vector. In this assignment, IDEA will be used alongside CFB mode to encrypt and decrypt messages sent to and from the server and the client.

The server will be preparing the IDEA key and send it over the network to the client, once the client has completed the verification process, the handshake process is completed, and the key will now be used alongside CFB mode.

# SHA-1

## Introduction

SHA is the Secure Hash Standard and specified in FIPS 180-4. The standard provides SHA-1 but it is now considered insecure. SHA-1 will be used with RSASS when the client sends over the public key, and when the server verifies the client's public key. Since the public key is a key that you want everyone to know, no encryption will be done on it.

SHA-1 will also be used another once when the server sends the session key to client. The server generates the IDEA session key and then encrypts it using the client's public key in RSAES and appends a digest of the key for validation.

# Crypto++

## Introduction

Crypto++ is an open-source library for C++ to implement secure algorithms for encryption and decryption process. It uses pipelining for all of its operations which saves down on unnecessary lines of declaration, and smoothen the programmer's thinking process while also increasing efficiency. Necessary transformation and filtration to the data can also be done.

Crypto++ has RSA, IDEA and SHA-1 included. Therefore, it is recommended to use in our implementation in our C++ program to save us time to write this report. Compiling the program requires the traditional way.

# Implementation

## Requirements

The program created should act like a chat program. An assumption made on the chat program is the chat program will only allow a party to send only once after the other party has sent their message. A party is unable to send more than one messages at a time before the other party sends a message. The server and client program must be in different folders as an added requirement.

The program, both the server and the client, must go through a handshake process using TCP or UDP sockets. TCP sockets are connection oriented while UDP sockets are connection-less. This means that if we are using UDP sockets instead of TCP sockets, the connection-less approach will result in packets arriving at different times, resulting in a shuffled order of packets arriving. For our chat program, it is highly advisable for us to use TCP sockets instead.

When the program first starts up, the server will prompt for the port number to bind to. A port not used by other services is best, while the client will prompt for the server's IP address and the server's bound port number.

After the client has connected to the socket of the server, it is time for the handshake process to begin. The handshake process begins with the client generating a public key using RSAES and signing the public key with RSASS and SHA-1. The public key concatenated with the signature of the public key is then sent to the server.

Upon the public key with its signature arriving on the server side, the server will then verify the authenticity of the public key by checking against its signature. Upon success, the server will prepare the session key used for communicating between the client and server. This session key will only be used if the client is connected to the server. The server then encrypts the session key using client's public key and hashes the session key without encrypting it using SHA-1. The server then sends the encrypted session key and the digest to client for verification.

Once the client receives the session key packet, only the client can decrypt and see the session key. The client then creates a digest and compares it against the one appended to the encrypted session key. Once verification is completed, the key will now be used in CFB mode for encryption and decryption of messages between the server and client. This marks the end of the handshake process.

Crypto++ library was used alongside other standard libraries that come with linux and C++.



# Summary

## Server

The server program begins with initializing 3 integer variables that will store our server's file descriptor number, the socket that the client will be connected to, and a variable to hold the status of the functions such as read and send. These functions return the number of bytes that are sent or read. Then, address is a struct type that specifies the information of the connection.

```
37 int main()
38 {
39     int fd_socket, new_socket, status;
40     struct sockaddr_in addr_info;
41     int options = 1;
42     int addrlen = sizeof(addr_info);
43     int PORT;
44     char buffer[2049] = {0}; // initialize 2049 size buffer
45     string received = "";
46     string message;
47
48
49     //get port number
50     do
51     {
52         cout << "Enter port number to host" << endl;
53         cin >> PORT;
54     } while (PORT == NULL || PORT < 0 || PORT > 65535);
55
56
57     // Creating socket file descriptor
58     if ((fd_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0)
59     {
60         perror("socket failed");
61         exit(EXIT_FAILURE);
62     }
63
64     // Forcefully attaching socket to the port
65     if (setsockopt(fd_socket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &options, sizeof(options)))
66     {
67         perror("setsockopt");
68         exit(EXIT_FAILURE);
69     }
70
71     addr_info.sin_family = AF_INET;
72     addr_info.sin_addr.s_addr = INADDR_ANY;
73     addr_info.sin_port = htons(PORT);
```

Then, the server will prompt the user to enter the port number which the server will bind to. Then the server will try to create a file descriptor socket using SOCK\_STREAM or also known as TCP, and a IPv4 type IP address will be passed to it later. Socket options are also specified to be reusable for address and port.

Next, the socket will be forcefully attached to the port, however the port number has not been passed until the latter step. Then, the IP type (IPv4 or IPv6), the IP address of the to-be client (which we specify allow all IP), and the port number.

```

75 // Forcefully attaching socket to the port
76 if (bind(fd_socket, (struct sockaddr *)&addr_info, sizeof(addr_info))<0)
77 {
78     perror("bind failed");
79     exit(EXIT_FAILURE);
80 }
81
82 //listen for connection from client
83 cout << "\nlistening for client to connect" << endl;
84 if (listen(fd_socket, 3) < 0)
85 {
86     perror("Error at setting connection to passive mode");
87     exit(EXIT_FAILURE);
88 }

```

Then, the previous server file descriptor socket and the address struct will bind together and the port listens to any incoming requests. The program then sets the server file descriptor socket to accept at most 3 connections at a time.

```

91 //accept connection
92 if ((new_socket = accept(fd_socket, (struct sockaddr *) &addr_info, (socklen_t*) &addrlen))<0)
93 {
94     perror("Error at accepting connection");
95     exit(EXIT_FAILURE);
96 }
97
98
99 cout << "\nConnection accepted." << endl;
100 cout << "Socket setup done. Waiting for client to send PUkey\n" << endl;
101

```

Next, the client will attempt to connect to the client here and the server will accept the connection. The server will now expect a public key from the client.

```

101
102 //receive PUkey with hash
103 status = read(new_socket, buffer, 804);
104 cout << "\nPUkey received from client. Performing validation.\n" << endl;
105 received = "";
106 for(int i = 0; i < 804; i++)
107 {
108     received += buffer[i];
109 }
110
111
112 string outputHex, PUhash, PUkey;
113 //get PUkey
114 //get 3072 bits PUkey
115 StringSource PUump(received, false, new StringSink(PUkey));
116 PUump.Pump(420);
117
118 //get hash of PUkey
119 PUump.Attach(new StringSink(PUhash));
120 PUump.PumpAll();
121
122 //display hex of PUkey
123 outputHex = "";
124 StringSource(PUkey, true, new HexEncoder(new StringSink(outputHex)));
125 cout << "\nClient's public key in hex: " << outputHex << endl;
126
127
128 //display hex of hash of PUkey
129 outputHex = "";
130 StringSource(PUhash, true, new HexEncoder(new StringSink(outputHex)));
131 cout << "\nClient's PUkey hash is: " << outputHex << endl;
132

```

After the client has sent the PUkey, the server will separate the packet into the PUkey alone and the digest of the PUkey. The server will display out the PUkey on the screen in hex, the same goes for the digest.

```

133 //confirm the PUkey authenticity
134 RSA::PublicKey publicKey;
135 StringSource getPUkey(PUkey, true);
136 publicKey.Load(getPUkey);
137
138
139 //create verifier object
140 RSASSA<PSS, SHA1>::Verifier verifier(publicKey);
141 //PSS is probabilistic signature scheme
142
143 //verify the signature
144 bool result = verifier.VerifyMessage((const byte*) PUkey.c_str(), PUkey.length(),
145 (const byte*) PHash.c_str(), PHash.size());
146

```

Next, the PUkey will be verified using a verifier object created using the client's PUkey with SHA-1. Then, the verifier object will return a boolean value when we try to verify the authenticity of the PUkey. Once the PUkey is verified, the program will generate IDEA session key and IV and send both along with their digests using SHA-1.

```

147 if (result == true)
148 {
149     cout << "\nVerification is successful. The signature is genuine.\n"
150     << "Server will now prepare IDEA session key for client.\n" << endl;
151
152
153     //encryptor object using client's public key
154     //this will be used to encrypt the IDEA session key
155     RSAES_PKCS1v15_Encryptor encryptor(publicKey);
156
157
158     //SHA1 hasher
159     SHA1 hasher;
160
161
162     //generate IDEA key
163     //start prepare IDEA key and send it
164     string ivHash, keyHash, PUSessionKey, PUiv;
165     SecByteBlock sessionKey(SESSIONKEYLEN); //create a key of size 16 bytes
166     byte iv[IVBLOCKLEN]; //create iv of size 8 bytes
167     AutoSeededRandomPool prng;
168     string sessionKeyStr;
169     prng.GenerateBlock(sessionKey, sessionKey.size());
170     prng.GenerateBlock(iv, sizeof(iv));
171     StringSource(reinterpret_cast<const char*>(&sessionKey[0]), true, new StringSink(sessionKeyStr));
172
173
174     //sign the key and iv
175     StringSource(sessionKeyStr, true, new HashFilter(hasher, new StringSink(keyHash)));
176     StringSource(reinterpret_cast<char*>(&iv[0]), true, new HashFilter(hasher, new StringSink(ivHash)));
177
178
179     //encrypt the key and iv
180     StringSource(sessionKeyStr, true,
181     | new PK_EncryptorFilter(prng, encryptor, new StringSink(PUSessionKey))
182     | );
183
184
185     StringSource(reinterpret_cast<char*>(&iv[0]), true,
186     | new PK_EncryptorFilter(prng, encryptor, new StringSink(PUiv))
187     | );
188

```

Once the PUkey is verified, the IDEA key and IV will be generated, and an encryptor object using the client's PUkey is created. Then, a SHA-1 hasher object will be created for creating the digests of the IV, and the program proceeds to sign and encrypt using the encryptor object and the hasher object into string variables.

```

190 //display session key
191 outputHex = "";
192 StringSource(sessionKeyStr, true, new HexEncoder(new StringSink(outputHex)));
193 cout << "Generated IDEA session key: " << outputHex << endl;
194 outputHex = "";
195 StringSource(PUSessionKey, true, new HexEncoder(new StringSink(outputHex)));
196 cout << "IDEA session key encrypted with client's PUkey: " << outputHex << endl;
197 outputHex = "";
198 StringSource(keyHash, true, new HexEncoder(new StringSink(outputHex)));
199 cout << "IDEA session key hashed with SHA1: " << outputHex << endl;
200
201 //iv display
202 outputHex = "";
203 StringSource(reinterpret_cast<char*>(&iv[0]), true, new HexEncoder(new StringSink(outputHex)));
204 cout << "\nIV generated: " << outputHex << endl;
205 outputHex = "";
206 StringSource(PUIv, true, new HexEncoder(new StringSink(outputHex)));
207 cout << "\nIV encrypted with client's PUkey: " << outputHex << endl;
208 outputHex = "";
209 StringSource(ivHash, true, new HexEncoder(new StringSink(outputHex)));
210 cout << "\nIV hash using SHA1: " << outputHex << endl;
211 outputHex = "";
212

```

Next, the server will output the hex for the session key and IV. The hex displayed are their binary form, encrypted form, and hashed form.

```

215 //send IDEA key concatenate with hash encrypted using PUkey and SHA1
216 message = PUSessionKey + keyHash;
217 send(new_socket, message.c_str(), message.length(), 0);
218 cout << "\nIDEA session key sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;
219
220 //send iv concatenate with hash encrypted using PUkey and SHA1
221 message = PUIv + ivHash;
222
223
224 //this is for future debugging purpose
225 //cout << PUIv.length() << endl << ivHash.length() << endl;
226
227
228 send(new_socket, (char*) message.c_str(), message.length(), 0);
229 cout << "\nIDEA IV sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;
230
231
232 //this is for future debugging purpose
233 //cout << message.length() << endl;
234

```

Then, the encrypted session key and the digest of the raw session key is sent before the encrypted IV and the digest of the raw IV is sent.

```

229 send(new_socket, (char*) message.c_str(), message.length(), 0);
230 cout << "\nIDEA IV sent encrypted with PUkey, concatenated with SHA1 hash\n" << endl;
231
232
233 //this is for future debugging purpose
234 //cout << message.length() << endl;
235
236
237 cout << "\nIDEA key and IV sent with their respective hashes\n" << endl;
238 cout << "\nServer will now use session key to communicate with client\n" << endl;
239
240
241 CFB_Mode<IDEA>::Encryption encryptIDEA;
242 CFB_Mode<IDEA>::Decryption decryptIDEA;
243 encryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
244 decryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
245 cout << "\nChat program starting up...\nPress CTRL + C to terminate or type in '\\\\0.'" << endl;
246
247 cin.ignore(numeric_limits<streamsize>::max(), '\n');

```

Once the session key and IV has been verified by the client, the chat program will start now and signal the end of the handshake process. The encryptor and decryptor object is created.

```

247 while(1)
248 {
249     string temp = "", k, output = "";
250
251     //this block will read as many times as needed
252     //this block is still flawed as if the sender sent 2049
253     //bytes then it will be trapped here
254     //but since the message will be sent in an encrypted manner,
255     //the ciphertext length is not likely to be an odd number
256     do
257     {
258         bzero(buffer, 2049); //clear buffer
259         status = read(new_socket, buffer, 2049);
260         if(status < 0)
261         {
262             cout << "\nError while trying to read." << endl;
263         }
264         else if (status == 0)
265         {
266             break;
267         }
268         else
269         {
270             temp += buffer;
271         }
272     } while (status == 2049);
273
274     StringSource(temp, true,
275         new StreamTransformationFilter(decryptIDEA, new StringSink(output))
276     ); //decrypt the message using session key
277     if (output != "\\0")
278     {
279         cout << "\nClient: " << output << endl;
280     }
281     else
282     {
283         cout << "Client closed connection." << endl;
284         close(new_socket);
285         close(server_fd);
286         break;
287     }
288     //clear output string and buffer
289     bzero(buffer, 2049);
290     output = temp = "";
291 }

```

In an infinite loop, the server will first receive messages from the client, read into a char array buffer and will attempt to read once again if found out that the number of bytes read is 2049 bytes. If the number of bytes sent is lesser, then the program will not read again. The program will read and append to a temp variable before it is passed into the decryptor to get the plaintext and outputted. The program will check if the client signalled to terminate the chat. Then, the server will clear the buffer to read and other string variables.

```

293 //sending message
294 cout << ">";
295 getline(cin, message);
296
297 StringSource(message, true,
298     new StreamTransformationFilter(encryptIDEA, new StringSink(temp))
299 );//encrypt the message using session key
300
301 do
302 {
303     if (temp.length() > 2048)
304     {
305         k = temp.substr(0, 2047);//get first 2048 bytes
306         temp = temp.substr(2048);//subtract off first 2048 bytes
307     }
308     else if (temp.length() == 2048) //this will send twice for when 2049 bytes read by receipient
309     {
310         k = temp;
311         temp = "";
312         status = send(new_socket, k.c_str(), k.length(), 0);
313     }
314     else
315     {
316         k = temp;
317         temp = "";
318     }
319     status = send(new_socket, k.c_str(), k.length(), 0);
320 } while (temp.length() > 0);
321 if (message == "\\0")
322 {
323     close(new_socket);
324     close(server_fd);
325     cout << "Connection closed" << endl;
326     break;
327 }
328 else if(status < 0)
329 {
330     cout << "\nError while trying to send." << endl;
331 }
332 temp = message = "";
333 }

```

Next, the server will reply to the client's message by getting the entire line of input. Then, the program will split up the input into 2048 bytes each. Sends 2049 bytes max to the client. The program will send as many times as needed in case the message is longer.

```

334 }
335 else
336 {
337     cout << "\nVerification is unsuccessful. Program will terminate and close the connection.\n"
338     << endl;
339     close(new_socket);
340     close(server_fd);
341 }
342 return 0;
343 }

```

If the PUkey from earlier is not verified, the program will close the connection.

## Client

The chat program is an exact version of the server except for the handshake process.

```
39 int main()
40 {
41     int sock = 0;
42     sockaddr_in addr_info;
43     string message, received = "";
44     char buffer[2049] = {0};
45     int PORT;
46     string SERVERADDR;
```

The client program starts with fewer variables. The main difference is that the socket variable count is lesser. In the server counterpart, we have two of them because we need one to set up the port and another to accept the connection from client. On the client counterpart, the client will only need to set up the connection only.

```
51     do
52     {
53         cout << "Enter server port number" << endl;
54         cin >> PORT;
55         if (cin.bad() || !cin.good())
56         {
57             cin.clear();
58             cin.ignore(numeric_limits<streamsize>::max(), '\n');
59             PORT = -1;
60             cout << "Port number invalid" << endl;
61         }
62         cin.ignore(numeric_limits<streamsize>::max(), '\n');
63     } while (PORT == -1);
64
65
66     do
67     {
68         cout << "Enter server address" << endl;
69         getline(cin, SERVERADDR);
70         if (SERVERADDR == "")
71         {
72             cout << "Server address must not be empty" << endl;
73         }
74     } while (SERVERADDR == "");
```

Next, the program will prompt for the port number and the address of the server.

```
85     addr_info.sin_family = AF_INET; //IPv4
86     addr_info.sin_port = htons(PORT); //set port number
87
88     // Convert IPv4 and IPv6 addresses from text to binary form
89     if(inet_pton(AF_INET, (char*) SERVERADDR.c_str(), &addr_info.sin_addr)<=0)
90     {
91         printf("\nInvalid address/ Address not supported \n");
92         return -1;
93     }
```

The program will then create the file descriptor socket, and the address information is passed to serv\_addr. Then, the IP address will be converted from string to binary form of 4 bytes and is passed to sin\_addr using inet\_pton.

```

95 //connecting
96 cout << "\nAttempting connection" << endl;
97 int count = 0, status = 0;
98 do
99 {
100     //attempt connect 4 times
101     status = connect(sock, (struct sockaddr *)&addr_info, sizeof(addr_info));
102     count++;
103     if (status == 0)
104     {
105         break;
106     }
107 } while (count < 3);

```

Next, the program will attempt to connect to the server for a total amount of 4 times. The server should be accepting the connection on the first try.

```

109 if (status == 0)
110 {
111     cout << "\nConnected to server" << endl;
112
113     //generate RSA keys of size 3072 bits
114     AutoSeededRandomPool prng; //rng
115     InvertibleRSAFunction parameters; //object of n p q etc.
116     parameters.GenerateRandomWithKeySize(prng, 3072); //generate privatekey
117     RSA::PrivateKey privateKey(parameters); //assign
118     RSA::PublicKey publicKey(parameters); //assign
119
120
121     //get public key string format
122     string PUStr, PUhash, outputHex;
123     StringSink transferPU(PUStr);
124     ByteQueue sink;
125     publicKey.Save(transferPU);
126
127
128     //convert PUkey to hex format
129     outputHex = "";
130     StringSource(PUStr, true, new HexEncoder(new StringSink(outputHex)));
131     cout << "\nPublic key in hex: " << outputHex << endl;
132     outputHex = "";
133
134     //signing the PUkey using SHA1
135     RSASSA<PSS, SHA1>::Signer signer(privateKey);
136     StringSource(PUStr, true, new SignerFilter(prng, signer, new StringSink(PUhash)));
137
138
139     //convert to hex format
140     outputHex = "";
141     StringSource(PUhash, true, new HexEncoder(new StringSink(outputHex)));
142     cout << "\nPublic key hash in hex: " << outputHex << endl;
143     outputHex = "";
144

```

Next, once the connection is accepted by the server, the program will generate a pair PUkey and PRkey using RSA, and then signing it with RSASS with SHA-1. Then, the hex format for the PUkey and digest is shown on the screen.

```

146     cout << "\nSending public key concatenated with hash of public key" << endl;
147     //PUStr send over with Hshal(PUStr) to server
148     message = PUStr + PUhash;
149     int status = send(sock, message.c_str(), message.length(), 0);
150     if (status == -1)
151     {
152         cout << "\nSending unsuccessful\n" << endl;
153         close(sock);
154         return 1;
155     }
156     else
157     {
158         cout << "\nPublic key sent to server\n" << endl;
159     }

```

The program will then send the PUkey along with the digest of the PUkey. The program then waits for the session key from the server.



```

161 //getting the session key and iv and hashes from server
162 //server will send the key and key hash
163 //then send the iv and iv hash
164 RSAES_PKCS1v15_Decryptor decryptor(privateKey); //decryptor for the IDEA session key and IV
165
166
167 string PUSessionKey, sessionKeyStr, keyHash, ivStr, ivHash, PUivStr;
168 bool result1, result2; //holds the result for the hash verification
169 SHA1 hasher; //hasher object to verify the hash later
170
171
172 //get the session key and store it temporarily
173 read(sock, buffer, 404);
174 received = "";
175 for(int i = 0; i < 404; i++)
176 {
177     received += buffer[i];
178 }
179
180 cout << "\nReceived session key from server\n" << endl;
181 StringSource getSessionKey(received, false, new StringSink(PUSessionKey));
182 getSessionKey.Pump(384);
183 getSessionKey.Attach(new StringSink(keyHash)); //key key hash
184 getSessionKey.PumpAll();
185
186
187 //output encrypted session key in hex
188 StringSource(PUSessionKey, true, new HexEncoder(new StringSink(outputHex)));
189 cout << "\nSession key encrypted with PUkey: " << outputHex << endl;
190 outputHex = "";
191
192
193 //decrypt the idea session key and store into sessionKeyStr
194 StringSource(PUSessionKey, true,
195     new PK_DecryptorFilter(prng, decryptor, new StringSink(sessionKeyStr))
196 );

```

Once the session key is delivered, the program will then conduct verification process. A decryptor object will be created and the PRkey is passed to the decryptor object. Then, the decryptor object will decrypt the session key and store it in a variable.

```

199 //output key in hex
200 StringSource(sessionKeyStr, true, new HexEncoder(new StringSink(outputHex)));
201 cout << "\nIDEA session key in hex: " << outputHex << endl;
202 outputHex = "";
203 StringSource(keyHash, true, new HexEncoder(new StringSink(outputHex)));
204 cout << "\nIDEA session key hash in hex: " << outputHex << endl;
205 outputHex = "";
206
207
208 //verify the hash of sessionkey and output the result to result1
209 //tells hashverificationfilter to throw exception on error and
210 //the hash is concatenated at the back
211 const int verificationFlags = HashVerificationFilter::HASH_AT_END;
212
213
214
215 //verify the session key
216 //hash dunno why put at back got issue but not infront.
217 //putting hash in front for now until further solutions found
218 StringSource(keyHash + sessionKeyStr, true, new HashVerificationFilter(hasher,
219 |   new ArraySink((byte*) &result1, sizeof(result1)))
220 );
221
222
223 //get the iv and iv hash and store it
224 read(sock, buffer, 404);
225 received = "";
226 for(int i = 0; i < 404; i++)
227 {
228 |   received += buffer[i];
229 }
230
231
232 StringSource getIV(received, false, new StringSink(PUivStr));
233 getIV.Pump(384);
234 getIV.Attach(new StringSink(ivHash));
235 getIV.PumpAll();
236
237 StringSource(PUivStr, true, new PK_DecryptorFilter(prng, decryptor, new StringSink(ivStr)));
238
239 //verify the hash of iv and output the result to result2
240 StringSource(ivHash + ivStr, true, new HashVerificationFilter(hasher,
241 |   new ArraySink((byte*) &result2, sizeof(result2)))
242 );
243
244

```

The same goes for IV when it arrives with its digest. It will be decrypted first before compared to its digest. After the raw binary form of the IV, IV digest, session key, session key digest is obtained, the program will now verify the integrity of the session key and the IV. The verification result is outputted into two boolean variables. One boolean variable will hold the verification result for the session key and the other will hold the verification result for the IV.

```

254     if (result1 == true && result2 == true)
255     {
256
257         cout << "\nBoth IDEA session key and IV are verified and genuine.\n"
258         << "Switching over to IDEA session key to communicate with server\n" << endl;
259
260
261         SecByteBlock sessionKey(SESSIONKEYLEN); //allocate 16 byte for IDEA key
262         byte iv[IVBLOCKLEN]; //allocate 8 bytes for iv
263         //assign session key
264         sessionKey.Assign(reinterpret_cast<const byte*>(&sessionKeyStr[0]), sessionKeyStr.size());
265         for(int i = 0; i < ivStr.length(); i++)
266         {
267             iv[i] = ivStr[i];
268         }
269
270         CFB_Mode<IDEA>::Encryption encryptIDEA;
271         CFB_Mode<IDEA>::Decryption decryptIDEA;
272         encryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
273         decryptIDEA.SetKeyWithIV(sessionKey, sessionKey.size(), iv);
274         cout << "\nChat program starting up...\nPress CTRL + C to terminate or type in '\\\\0.'" << endl;
275
276
277 > while(1) -
360     }
361     else
362     {
363         cout << "\nVerification status:"
364         << "\nIDEA session key: " << (result1 ? "Verified" : "Failed")
365         << "\nIV: " << (result2 ? "Verified" : "Failed") << endl;
366         close(sock);
367     }
368 }
369 else
370 {
371     cout << "\nConnection failed. Is the server up?" << endl;
372 }

```

Once the results are true, then the program will start using the session key. The decryptor and encryptor objects will be created and the session key and IV will be passed to it. With this, the handshake process is now completed.

```

277     while(1)
278     {
279         string temp = "", k, output = "";
280
281         message = "";
282         //sending message
283         cout << ">";
284         getline(cin, message);
285
286         StringSource(message, true,
287             new StreamTransformationFilter(encryptIDEA, new StringSink(temp))
288         ); //encrypt the message using session key
289         do
290         {
291             if (temp.length() > 2048)
292             {
293                 k = temp.substr(0, 2047); //get first 2048 bytes
294                 temp = temp.substr(2048); //subtract off first 2048 bytes
295             }
296             else if (temp.length() == 2048) //this will send twice for when 2049 bytes read by receipient
297             {
298                 k = temp;
299                 temp = "";
300                 status = send(sock, k.c_str(), k.length(), 0);
301             }
302             else
303             {
304                 k = temp;
305                 temp = "";
306             }
307             status = send(sock, k.c_str(), k.length(), 0);
308         } while (temp.length() > 0);
309         if (message == "\\0")
310         {
311             close(sock);
312             cout << "Connection closed" << endl;
313             break;
314         }
315         else if(status < 0)
316         {
317             cout << "\nError while trying to send." << endl;
318         }
319         temp = "";
320         message = "";

```

Now, the program enters chat mode. The chat mode is the same with the server's with a small change. In the requirement, the client is supposed to send a message first. Therefore for client's chat mode, it starts with sending a message to server.

```

312 do//this block will read as many times as needed
313 {
314     bzero(buffer, 2049);//clear buffer
315     status = read(sock, buffer, 2049);
316     if(status < 0)
317     {
318         cout << "\nError while trying to read." << endl;
319     }
320     else if (status == 0)
321     {
322         break;
323     }
324     else
325     {
326         temp += buffer;
327     }
328 } while (status == 2049);
329
330 StringSource(temp, true,
331             new StreamTransformationFilter(decryptIDEA, new StringSink(output))
332 );//decrypt the message using session key
333
334 if (output != "\\0")
335 {
336     cout << "\nServer: " << output << endl;
337 }
338 else
339 {
340     cout << "Server closed connection." << endl;
341     close(sock);
342     break;
343 }
344 //clear output string and buffer
345 bzero(buffer, 2049);
346 output = temp = "";
347
348 }

```

Finally, here we can see that the reading part is exactly like the server's. The program will end if CTRL + C was pressed or \\0 was typed into the chat. If the user presses CTRL + C, the other party's connection will not close automatically, but not for the alternative.

## Instructions to run.

By using 2 command line window or 2 machines, navigate to the folder that contains the folder “serverfolder” and “clientfolder”. After doing so, copy and paste this into the command line and gdb will compile the program using Crypto++ library:

```
Server: g++ -g3 -ggdb -O0 -Wall -Wextra -Wno-unused -o server server.cpp -lcryptopp
```

```
Client: g++ -g3 -ggdb -O0 -Wall -Wextra -Wno-unused -o client client.cpp -lcryptopp
```

After compilation is done, type ./server in one of the command line or machines and ./client in the other to start the program up. Once it is started up, the server will prompt you for the port number to bind to. After inputting the port number, the server will be up and running. Remember to do port forwarding or NAT forwarding for your router if you haven't done so.

The client will then prompt you for the port number and the public IPv4 address of the server. You can find the server's public IPv4 address by going to [www.whatismyip.com](http://www.whatismyip.com) and it will show you your IPv4 for the server. Once you inputted the IP address in, you may now start chatting.

Below shows the signing and verification processes of the program.

Below shows the MD5 digest generation process using 3 different file types.

Below shows the compilation of the program as well as the program setting up the port and proceeding with handshake process.

```
ken@ken: ~/361/ass4
ken@ken:~/361/ass4$ g++ -g3 -gdb -O0 -Wall -Wextra -Wno-unused -o serverfolder/server serv
erfolder/server.cpp -lcryptopp
serverfolder/server.cpp: In function 'int main()':
serverfolder/server.cpp:54:22: warning: NULL used in arithmetic [-Wpointer-arith]
   54 |         } while (PORT == NULL || PORT < 0 || PORT > 65535);
      |                  ^~~~~
ken@ken:~/361/ass4$ ./serverfolder/server
Enter port number to host
9999

Listening for client to connect

Connection accepted.
Socket setup done. Waiting for client to send Pukey

Pukey received from client. Performing validation.

Client's public key in hex: 308201A030806092A864886F7D0B1010105008302018D003082018802820180A5F0275
00A5F0275B7C08708A528E3A071F980814B81712513E5514130F508E605E092A01EB60027095571636A86F23AD5C3796
34A528235B1C4C5791D09256AD2505F1202D1E079C5A4BACF4DECD0A21072C9275F3830FD8682681360F4F4042
60F440427053D3119789189566127455E0A708CD40C44FC8C58385F71F075CBA937EA50280570AE7772849EBE16882111C
82CE98F5076903CB89058CF845AD6BE2D31A2499CF6FBCA81649C5A85B72D86615F37B08F6D039702B0AF11B086E
AF11B086E56DAF3B076AD0F5075455F9B38911957863EBEE7E5C9A4A69BC014D9000404F962434066ED824C88C2
62A6CF3276B4D863CEC7AF5E0D555687FFBEDB75B8F8EF8992AEC301220F8C647DEA69EB9C9CB05F1AD090A8452
20EB78AE2D760585CE52DC9A360E19041AD61E8FD00028A73EC250355E25B094E9A986A1E6673884E4D28803CC3312D
C0DA 95F5AD572686A59FEF80318EA68CB9E60273D0591904F8A748665DC9214EF110F25D27F46C95853A15E600F87E
909B3F9EAB6E615535E6882AB020111

Client's Pukey hash is: 49D3EE55082B07F5E61A23D10F74928FB024AFCB164E09A3B0E9F0E836F1B4808F9
F0C7C0B73072FE5D017B58250A1E340EE1AC552250AE4EB1FFB601749B1A30FAD548A14AFCFE902148A74FD2640E
2138139374F0E3A030D2A4682E3A5E27280181051650C5EA605A506788636F6839B656D42440D0E77E052267683B3
070E25AF789DA333A0E9705805C8C0719E4371C00508E16E928A2527B1E826E65482D0C809C38F53A4A105E64F
8D386AE7956898805C22963A3789AE4508AE1F880484345333D01908A1695E6043FF8B2B2F2576E01A564E2B8B4
FB3D2B585CF0595122D786140CCDFF184EE97E7F93C3CECF8AFAE0A0B663A268002D3147181C2988337551F3F2
AA3D1C3EA23C762B47AC8B655C3D0A951B69896C23ABA97BAC1EEAF2640C00CEA31B0DBFAC57D3325ECD0AF7E20FC
758CA0C367119F96908922C428B019F47507B720D41D2C6F775CD5E1438CD08BC278A870B0A45A23C548FAC7B2
B677802D233A7080A2E4BF4443A112A3D6323F6964834AED26FF1F62

Verification is successful. The signature is genuine.
Server will now prepare IDEA session key for client.

Generated IDEA session key: 17A3F2C810250A0FB1C1878D2D410661
IDEA session key encrypted with client's Pukey: 349C16970FD1B8CF054920FF3F4602F48BF8439EF895
F103DC6E97647D17E123D0B81F8471B15C5AFB69A751A3C96455190883F6C52B9E1518F3D7EBA29FB990C8F548C
27589EA78A5A50DFFC97D900ABADFB6F87E57A25612FE819EE7717439AB78EF1507BE62D0E6805339955A9A2D
B0E6D95D38CF9B54F40342742C0F5B9177FEFE68816B711A2C5F385D0809898980601E5BE7F226C7F41052A478
ED0E6F14C8410C6642740F5738D6B691AC87128723652B8C419ABE99660FB62111C3948F796E1154893C34181
61667DA0C7B5AEF6535C7FE7F60138201823E8ACD60880F8D73FAB4848338C906093D470CE630E3D4C1847CE03
379C15C913587884A28830AE7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1
78006F34897F83852F650F7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1

Below shows the client sending the first message to server.

Verification is successful. The signature is genuine.
Server will now prepare IDEA session key for client.

Generated IDEA session key: 17A3F2C810250A0FB1C1878D2D410661
IDEA session key encrypted with client's Pukey: 349C16970FD1B8CF054920FF3F4602F48BF8439EF895
F103DC6E97647D17E123D0B81F8471B15C5AFB69A751A3C96455190883F6C52B9E1518F3D7EBA29FB990C8F548C
27589EA78A5A50DFFC97D900ABADFB6F87E57A25612FE819EE7717439AB78EF1507BE62D0E6805339955A9A2D
B0E6D95D38CF9B54F40342742C0F5B9177FEFE68816B711A2C5F385D0809898980601E5BE7F226C7F41052A478
ED0E6F14C8410C6642740F5738D6B691AC87128723652B8C419ABE99660FB62111C3948F796E1154893C34181
61667DA0C7B5AEF6535C7FE7F60138201823E8ACD60880F8D73FAB4848338C906093D470CE630E3D4C1847CE03
379C15C913587884A28830AE7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1
2B096F8582F38652F650F7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1
C37924B04295A0A8B094A86923CEB794BC8B8484725ADF9740F9D376449C0A4127621A7972432E2
758CA0C367119F96908922C428B019F47507B720D41D2C6F775CD5E1438CD08BC278A870B0A45A23C548FAC7B2
IDEA session key hashed with SHA1: DFDA0CB83F0DBF840CFD1B455DA74ADC20408B5

IV generated: C9137448CA843004308201A030806092A864886F7D0B1010105

IV encrypted with client's Pukey: 88AF0AD2AA3CE3A50CF7B8674992D190361D01977841E1FE35D09913A
988AC7B6C41839A0C0D72EB1B617D1F25680E212506F1A44C871E47E9B8C0D9A21A2BA880315846557D09AB08
B48619B5E17CBAD6E0329F7C4C475FE2C0D508C75A52B080C9C7B364DC9678ACAF39F58511B90488F2D0820B
C8FFA7443FD0049AC3A3E1C1B054580756FASD25580B0C2FA8A8C51B87878C4A7E789C9C5F2C510644EFB53443F5
28611627EC72C3B9D1854C8BAE64947911A06629836EEFF12F5A060171D0D9996E6735617D209886380682CE0A325
AA44A533A080AF63ACF279C083F5A50E113F08C279134395C7059CB5E393FE782E8B877E39CFE964A18B0C71FDF
19368403AF4E071BCA7624652E548427600570886F80334CA90241A443ED7F16D776AF2F9C857476DA58050572E1
57AA7BBE7D7E6C7DC9C247F2F8D8747D7F4530065E5F3E9537EEF347EA84266A38FF7FEB9FC6E4384BE21EC47AAB
07AF1D54BB88EF399C985F682960C053083D7893880755CA24A2E2D4A539E2A90

IV hash using SHA1: 70D1CFE391888C5F79822A8BDEDEEA924A9A865F

IDEA session key sent encrypted with Pukey, concatenated with SHA1 hash

IDEA IV sent encrypted with Pukey, concatenated with SHA1 hash

IDEA key and IV sent with their respective hashes

Server will now use session key to communicate with client

Chat program starting up...
Press CTRL + C to terminate or type in \\.

Client: hi server
>hi client

Public key hash in hex: 49D3EE55082B07F5E61A23D10F74928FB024AFCB164E09A3B0E9F0E836F1B4808F9
F0C7C0B73072FE5D017B58250A1E340EE1AC552250AE4EB1FFB601749B1A30FAD548A14AFCFE902148A74FD2640E
2138139374F0E3A030D2A4682E3A5E27280181051650C5EA605A506788636F6839B656D42440D0E77E052267683B3
070E25AF789DA333A0E9705805C8C0719E4371C00508E16E928A2527B1E826E65482D0C809C38F53A4A105E64F
8D386AE7956898805C22963A3789AE4508AE1F880484345333D01908A1695E6043FF8B2B2F2576E01A564E2B8B4
FB3D2B585CF0595122D786140CCDFF184EE97E7F93C3CECF8AFAE0A0B663A268002D3147181C2988337551F3F2
AA3D1C3EA23C762B47AC8B655C3D0A951B69896C23ABA97BAC1EEAF2640C00CEA31B0DBFAC57D3325ECD0AF7E20FC
758CA0C367119F96908922C428B019F47507B720D41D2C6F775CD5E1438CD08BC278A870B0A45A23C548FAC7B2
B677802D233A7080A2E4BF4443A112A3D6323F6964834AED26FF1F62

Sending public key concatenated with hash of public key

Public key sent to server

Received session key from server

Session key encrypted with Pukey: 349C16970FD1B8CF054920FF3F4602F48BF8439EF895F103DC6E97647D
17E123D0B81F8471B15C5AFB69A751A3C96455190883F6C52B9E1518F3D7EBA29FB990C8F548C27589EA78A5A50
DFF9C97D900ABADFB6F87E57A25612FE819EE7717439AB78EF1507BE62D0E6805339955A9A2DDB0E6D95D38CF9B
54F40342742C0F5B9177FEFE68816B711A2C5F385D0809898980601E5BE7F226C7F41052A478ED0E6F14C8410C
6642740F5738D6B691AC87128723652B8C419ABE99660FB62111C3948F796E1154893C3418161667DA0C7B5AEF6535C7FE7F60138201823E8ACD60880F8D73FAB4848338C906093D470CE630E3D4C1847CE03379C15C913587884A28830AE7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA178006F34897F83852F650F7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1
```

Below shows the client sending the first message to server.

```
56DAF38076AD04F5D075455F9B38911957863EBEE7E5C9A4A69BC014D9000404F962434066ED824C88C262A6FC372
684D863CE7AF5E0D555687FFBEDB75B8F8EF8992AEC301220F8C647DEA69EB9C9CB05F1AD090A845220EB78AE2
07D65B5CE52DC9A360E19041AD61E8FD00028A73EC250355E25B094E9A986A1E6673884E4D28803CC3312DC0DA
95F5AD572686A59FEF80318EA68CB9E60273D0591904F8A748665DC9214EF110F25D27F46C95853A15E600F87E
909B3F9EAB6E615535E6882AB020111

Public key hash in hex: 49D3EE55082B07F5E61A23D10F74928FB024AFCB164E09A3B0E9F0E836F1B4808F9
F0C7C0B73072FE5D017B58250A1E340EE1AC552250AE4EB1FFB601749B1A30FAD548A14AFCFE902148A74FD2640E
2138139374F0E3A030D2A4682E3A5E27280181051650C5EA605A506788636F6839B656D42440D0E77E052267683B3
070E25AF789DA333A0E9705805C8C0719E4371C00508E16E928A2527B1E826E65482D0C809C38F53A4A105E64F
8D386AE7956898805C22963A3789AE4508AE1F880484345333D01908A1695E6043FF8B2B2F2576E01A564E2B8B4
FB3D2B585CF0595122D786140CCDFF184EE97E7F93C3CECF8AFAE0A0B663A268002D3147181C2988337551F3F2
AA3D1C3EA23C762B47AC8B655C3D0A951B69896C23ABA97BAC1EEAF2640C00CEA31B0DBFAC57D3325ECD0AF7E20FC
758CA0C367119F96908922C428B019F47507B720D41D2C6F775CD5E1438CD08BC278A870B0A45A23C548FAC7B2
B677802D233A7080A2E4BF4443A112A3D6323F6964834AED26FF1F62

Sending public key concatenated with hash of public key

Public key sent to server

Received session key from server

Session key encrypted with Pukey: 349C16970FD1B8CF054920FF3F4602F48BF8439EF895F103DC6E97647D
17E123D0B81F8471B15C5AFB69A751A3C96455190883F6C52B9E1518F3D7EBA29FB990C8F548C27589EA78A5A50
DFF9C97D900ABADFB6F87E57A25612FE819EE7717439AB78EF1507BE62D0E6805339955A9A2DDB0E6D95D38CF9B
54F40342742C0F5B9177FEFE68816B711A2C5F385D0809898980601E5BE7F226C7F41052A478ED0E6F14C8410C
6642740F5738D6B691AC87128723652B8C419ABE99660FB62111C3948F796E1154893C3418161667DA0C7B5AEF6535C7FE7F60138201823E8ACD60880F8D73FAB4848338C906093D470CE630E3D4C1847CE03379C15C913587884A28830AE7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA178006F34897F83852F650F7F40544EB81A5CFB8E8197627B6FEE38C7EC5588FB322A7A56F9635AF74EEA1
```

Below shows the server responding to client's message.

```
IDEA IV sent encrypted with Pukey, concatenated with SHA1 hash

IDEA key and IV sent with their respective hashes

Server will now use session key to communicate with client

Chat program starting up...
Press CTRL + C to terminate or type in \\.

Client: hi server
>hi client

IDEA session key in hex: 17A3F2C810250A0FB1C1878D2D410661

IDEA session key hash in hex: DFDA0CB83F0DBF840CFD1B455DA74ADC20408B5

Both IDEA session key and IV are verified and genuine.
Switching over to IDEA session key to communicate with server

Chat program starting up...
Press CTRL + C to terminate or type in \\.
>hi server

Server: hi client
>hi client
```



Below shows the client initiating a normal conversation with server.

<pre>IDEA key and IV sent with their respective hashes     printf("message received = %s\n",         (char *)buffer[2048] + 16);  Server will now use session key to communicate with client     printf("Server ready\n");  Chat program starting up... Press CTRL + C to terminate or type in \\0.  Client: hi server &gt;hi client     printf("hi\n");     if (strcmp(buffer, "hi") != 0) {         printf("not a good!\n");     }  Client: How are you? &gt;     printf("clear\n");</pre>	<pre>IDEA session key in hex: 17A572C810258A0F61C1678D2D410061  IDEA session key hash in hex: DFDA0CBB3F0DBF840CFFD1B455DA74ADC2040B85  Both IDEA session key and IV are verified and genuine. Switching over to IDEA session key to communicate with server  Chat program starting up... Press CTRL + C to terminate or type in \\0. &gt;hi server  Server: hi client &gt;How are you? </pre>
--	--

Below shows the client closing the connection using \\0.

<pre>IDEA key and IV sent with their respective hashes     printf("message received = %s\n",         (char *)buffer[2048] + 16);  Server will now use session key to communicate with client     printf("Server ready\n");  Chat program starting up... Press CTRL + C to terminate or type in \\0.  Client: hi server &gt;hi client     printf("hi\n");     if (strcmp(buffer, "hi") != 0) {         printf("not a good!\n");     }  Client: How are you? &gt;Im fine     printf("clear\n");  Client: ok bye &gt;bye     printf("clear\n");     if (strcmp(buffer, "bye") != 0) {         printf("not a good!\n");     } Client closed connection. ken@ken:~/361/ass4\$</pre>	<pre>IDEA session key hash in hex: 6F6A0CBB3F0DBF840CFFD1B455DA74ADC2040B85  Both IDEA session key and IV are verified and genuine. Switching over to IDEA session key to communicate with server  Chat program starting up... Press CTRL + C to terminate or type in \\0. &gt;hi server  Server: hi client &gt;How are you?  Server: Im fine &gt;ok bye  Server: bye &gt;\\0 Connection closed ken@ken:~/361/ass4\$</pre>
--	--

# Appendix

## References

Crypto++, 2021, *RSA Cryptography*, viewed 17 June 2021,  
<[https://www.cryptopp.com/wiki/RSA\\_Cryptography](https://www.cryptopp.com/wiki/RSA_Cryptography)>

Crypto++, 2021, *RSA Encryption Schemes*, viewed 17 June 2021,  
<[https://www.cryptopp.com/wiki/RSA\\_Encryption\\_Schemes](https://www.cryptopp.com/wiki/RSA_Encryption_Schemes)>

Crypto++, 2021, *RSA Signature Schemes*, viewed 17 June 2021,  
<[https://www.cryptopp.com/wiki/RSA\\_Signature\\_Schemes](https://www.cryptopp.com/wiki/RSA_Signature_Schemes)>

Crypto++, 2021, *SHA*, viewed 17 June 2021,  
<https://www.cryptopp.com/wiki/SHA>>

Crypto++, 2021, *IDEA*, viewed 17 June 2021,  
<<https://www.cryptopp.com/wiki/IDEA>>

GeeksforGeeks, 2019, *Socket Programming in C/C++*, *GeeksforGeeks*, viewed 17 June 2021,  
<<https://www.geeksforgeeks.org/socket-programming-cc>>

Think and Learn 2017, *Code the Server Part 2 / Socket Programming / Tutorial No 5*, online video, 19 Aug 2017, viewed 17 June 2021,  
<[https://www.youtube.com/watch?v=Ts8eXOkx8TE&list=PLPyar5G9aNDvs6TdpLcVO43\\_jvxp4emI&index=5&t=515s&ab\\_channel=ThinkandLearnThinkandLearn](https://www.youtube.com/watch?v=Ts8eXOkx8TE&list=PLPyar5G9aNDvs6TdpLcVO43_jvxp4emI&index=5&t=515s&ab_channel=ThinkandLearnThinkandLearn)>

Think and Learn 2017, *Code the Client - Running our Chat Application / Socket Programming / Tutorial No 6*, online video, 19 Aug 2017, viewed 17 June 2021,  
<[https://www.youtube.com/watch?v=DboEGcU6rLI&list=PLPyar5G9aNDvs6TdpLcVO43\\_jvxp4emI&index=6&ab\\_channel=ThinkandLearnThinkandLearn](https://www.youtube.com/watch?v=DboEGcU6rLI&list=PLPyar5G9aNDvs6TdpLcVO43_jvxp4emI&index=6&ab_channel=ThinkandLearnThinkandLearn)>