Project Overview

For this project, we are implementing asymmetric key encryption, digital signature verification, and setting up an encrypted communication between client and server over Linux operating system. For asymmetric key data encryption and digital signature verification, we used built-in "OpenSSL" command-line tool suite. For encrypted communication between client and server, we used JAVA programming languages to implement Socket communication. We encrypted the communication through two methods: symmetric key encryption & asymmetric key encryption.

Asymmetric Key Encryption & Digital Signature Verification

1) In this section, first, we implemented asymmetric key encryption using built-in "OpenSSL" command tool suite.

    1. First, we generated a 1024-bit RSA private key using below command:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl genrsa -out private.pem 1024
```

    2. We accessed the generated "private.pem" private key file, the value of the private key is displayed below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDFlBcza643b9l42ocnxYCt8hUNNKbfXmTGZ0W+FNySsdmcvCkf
yjFfnKydVGyjHHF/a2wvRGQgGtrjOpPWRJWv1bonQwp4jEFWgw0sS3ZzZDF0Nd8L
pnkmQjDNQdFgxkrc+mRN2xhegeo+F5ihR6Ya5MGKDD5T+vlCvTRxjxTLOwIDAQAB
AoGAUQlseqqvyg0WJXt789QN80pXWBNAXl1Y+Adjk5L65TANETwboe5MUerw4Za+
+CeyZZCpzrk+V+yr2ocsG7YFrl+++SR82SxTNpUkXnvxQsnM3U/HSPi72p9V86kG
Ht/1hNqpMxjnJ5QD5c5h+UdDMUu0xyyEDUfB09o0BKUzOAECQQD2Dfe4A7O/Vl0b
adIoE9jsqwhjcusxOrsO8p2r1CjZXsFq+jcsfLJJOTWCqoPrEIX9FT5cr1H6ijHS
5Pxlarx/AkEAzZCFJfDZIdQW4x1f7PzTMxv91bshETxn5MLhk5w6nEVbZ/dd+fBk
V/kEJBgZT9FiyDmGfhSNeF28iLYsY6WDRQJAOfYEWpaUqwffGuwMk2BJg0auzS5u
9MPO+LByL93kWYAWs+qJQLOHE/SkBEnClYbo6Tst24t5Fjmjjvhd9E32ZQJAYqmN
roMlxB3GimVL6DQErlAuCcpR+bLTbrG2kSF35A9J6uGIYnDlyG+FdEl4xJ2L2uv5
SERXDkF3Puqo7X7APQJBAJSiQnbjHmum5kmEtPFP3B0sHaI2AVsNGErdnoMv1UFy
cMM6xdERRYuvSMqUTJL8BmKDhnM+dBo052CGzXzpANs=
-----END RSA PRIVATE KEY-----
```

    3. Then, we generated the corresponding public key from the previous private key using command below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl rsa -in private.pem -out publi
c.pem -outform PEM -pubout
```

    4. We accessed the "public.pem" public key file, the value of the public key is displayed below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDFlBcza643b9l42ocnxYCt8hUN
NKbfXmTGZ0W+FNySsdmcvCkfyjFfnKydVGyjHHF/a2wvRGQgGtrjOpPWRJWv1bon
Qwp4jEFWgw0sS3ZzZDF0Nd8LpnkmQjDNQdFgxkrc+mRN2xhegeo+F5ihR6Ya5MGK
DD5T+vlCvTRxjxTLOwIDAQAB
-----END PUBLIC KEY-----
```

5. Next, we created a plaintext file for encryption, which named “plaintext.txt” as below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ nano plaintext.txt
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ cat plaintext.txt
Hello World!
This file is intended for encryption project!
Thanks!
```

6. Now, we encrypt the plaintext file using RSA public key and store it in a file named “encryptedfile.ssl” using command below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl rsautl -encrypt -inkey public.
pem -pubin -in plaintext.txt -out encryptedfile.ssl
```

7. We convert the “encryptedfile.ssl” file into binary form and displayed it in the terminal as below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ xxd -b encryptedfile.ssl
0000000: 00011001 00110100 01110110 10100011 00011111 10110011  .4v...
0000006: 11010001 00000110 01011110 11000101 10010000 00000001  ..^...
000000c: 01110111 00101010 00110010 10000011 01001101 01010111  w*2.MW
0000012: 01110111 01110110 00111111 11110011 00110001 01100011  wv?.1c
0000018: 10100000 00111111 01011100 01000100 11001001 00001011  .?\D..
000001e: 11101100 01110100 01111110 10010101 01100011 00000110  .t~.c.
0000024: 10111000 10100100 11001000 00010010 01000011 00100101  ....C%
000002a: 10101000 11110101 11110110 10111111 11010101 10101000  ......
0000030: 00011110 00100110 00111011 10101011 01011000 00000010  .&;.X.
0000036: 01111111 11000101 00111001 11001010 01010111 11111101  ..9.W.
000003c: 10110011 11001100 00011111 01000110 01010001 00100101  ...FQ%
0000042: 11000011 00001110 00101011 11000110 00010001 00010110  ..+...
0000048: 01010011 10101100 11111101 11111110 10010010 10011101  S.....
000004e: 10101100 11111010 10101011 11001000 11000101 01011110  .....^
0000054: 10011110 11110100 01110101 10010001 11100000 10010001  ..u...
000005a: 00011000 00100101 00110001 00101111 10000000 10001001  .%1/..
0000060: 01010000 10111100 11101110 01000000 10101100 10001101  P..@..
0000066: 10011100 00011010 00010110 11001110 10000111 11101110  ......
000006c: 00011010 11111010 00101111 01001100 11001001 11100101  ../L..
0000072: 10100011 10110100 01101001 01000100 11101010 11010010  ..iD..
0000078: 01011000 00010100 11010010 11011110 10010101 10101110  X.....
000007e: 11011100 10001101                                      ..
```

8. Finally, we decrypt the “encryptedfile.ssl” file using 1024-bit RSA private key generated previously, and store it in a file named “decryptedfile.txt” using command below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl rsautl -decrypt -inkey private
.pem -in encryptedfile.ssl -out decryptedfile.txt
```

9. Lastly, we accessed the “decryptedfile.txt” and displayed it in the terminal. As displayed below, it matches the original “plaintext.txt” file.

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ cat decryptedfile.txt
Hello World!
This file is intended for encryption project!
Thanks!
```

2) Secondly, we continue to implement digital signature using the same “OpenSSL” command-line tool suite.

- First, we need to compute the hash value of our original file “plaintext.txt” using command below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl dgst -sha1 plaintext.txt
SHA1(plaintext.txt)= 44185f760a2c2317c10f826d13cc3ca9bc76c87f
```

- Then, we computed the hash value of the decrypted file “decryptedfile.txt”, and we can see that the hash value is the same as the original. It means the message is the same as the original.

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl dgst -sha1 plaintext.txt
SHA1(plaintext.txt)= 44185f760a2c2317c10f826d13cc3ca9bc76c87f
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl dgst -sha1 decryptedfile.
txt
SHA1(decryptedfile.txt)= 44185f760a2c2317c10f826d13cc3ca9bc76c87f
```

- In addition to comparing hash value manually, OpenSSL provides an option to compute the hash value for message and sign it with private key. The signed hash value is stored in “sign_ID.bin”. Command as below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl dgst -sha1 -sign private.pem -
out sign_ID.bin plaintext.txt
```

- We then also computed the hash value of the decrypted file “decryptedfile.txt”, and we used public key to decrypt “sign_ID.bin” to get the hash value of original file. After comparison, result shows that the digital signatures of the two files are the same. Therefore, “Verified OK” message is printed as below:

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~$ openssl dgst -sha1 -verify public.pem
-signature sign_ID.bin decryptedfile.txt
Verified OK
```

Encrypted Client-Server Communication

In this section, we used JAVA programming language to implement encrypted client-server communication. We implemented two encryption method, which are 1) Symmetric-Key Encryption; 2) Asymmetric-Key Encryption. We will explain the JAVA codes in detail within this report. (Complete JAVA code will be submitted in separate files; therefore, it will not be included in this report.)

1) Symmetric-Key Encryption

For symmetric-key encryption, we used Tink cryptographic library API powered by Google, which supports multiple programming languages and can be run on many platforms. Therefore, Google Tink requires users to setup system environment to use it.

- **Encryption of the communication between the Client and the Server using AES-256 technique (GF/Counter Mode)**

```
//Generating key materials with AES_256 using GF/Counter Mode
KeysetHandle keysetHandle = KeysetHandle.generateNew(AeadKeyTemplates.AES256_GCM);
```

- **Before the demonstration, we need to set up CLASSPATH first**
  a. export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-i386
  b. export PATH=${PATH}:${JAVA_HOME}/bin
  c. export CLASSPATH=.:/usr/lib/jvm/java-1.7.0-openjdk-i386/lib:/home/Desktop/Jar/json-20180813.jar:/home/Desktop/Jar/protobuf-java-3.7.0.jar:/home/Desktop/Jar/tink-1.2.2.jar

- **Imported packages**

```
import java.net.*;
import java.io.*;
import java.util.*;

//Base Classes for Tink Crypto Library
import com.google.crypto.tink.*;
import com.google.crypto.tink.aead.AeadFactory;
import com.google.crypto.tink.config.TinkConfig;
import com.google.crypto.tink.aead.AeadFactory;
import com.google.crypto.tink.aead.AeadKeyTemplates;
import com.google.crypto.tink.CleartextKeysetHandle;
import com.google.crypto.tink.JsonKeysetWriter;
```

- **Key is generated by the Server when connection is initialized and stored into my_keyset.json file**

```
TinkConfig.register();

//Generating key materials with AES_256 using GF/Counter Mode
KeysetHandle keysetHandle = KeysetHandle.generateNew(AeadKeyTemplates.AES256_GCM);

//Write to a file
String mySecretKeyset = "my_keyset.json";

CleartextKeysetHandle.write(keysetHandle, JsonKeysetWriter.withFile(new File(mySecretKeyset)));

//Reading the keyset from .json file
keysetHandle = CleartextKeysetHandle.read(JsonKeysetReader.withFile(new File(mySecretKeyset)));


//Getting the Primitive from input which uses for encryption
Aead aead = AeadFactory.getPrimitive(keysetHandle);
```

- **Reading the key from my_keyset.json file, and uses the key for decryption**

my_keyset.json - Notepad

File Edit Format View Help

```
{
    "primaryKeyId": 2050489839,
    "key": [{
        "keyData": {
            "typeUrl": "type.googleapis.com/google.crypto.tink.AesGcmKey",
            "keyMaterialType": "SYMMETRIC",
            "value": "GiD98xUlV0G+7rWGb3qoAJjIokjgYP+unMRfCFA1LzJ4xA=="
        },
        "outputPrefixType": "TINK",
        "keyId": 2050489839,
        "status": "ENABLED"
    }]
}
```

- **Encrypt the message with generated symmetric key and sent out the encrypted message**

```
OutputStream outToClient = server.getOutputStream();
DataOutputStream out = new DataOutputStream(outToClient);
Scanner myObj = new Scanner(System.in);
System.out.println("Enter a message");
String line = myObj.nextLine();
//encrypting plaintext
byte[] cipherText = aead.encrypt(line.getBytes(), null);
out.writeUTF(byte2hex(cipherText));
//out.writeUTF(line);
```

- **"byte2hex" method**
  - ✓ The purpose of using this method is to convert binary value (value after encryption) to HEX value so that server side can read the message as String

```
public static String byte2hex(byte buf[])
{
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < buf.length; i++)
    {
        String hex = Integer.toHexString(buf[i] & 0xFF);
        if (hex.length() == 1)
        {
            hex = '0' + hex;
        }
        sb.append(hex.toUpperCase());
    }
    return sb.toString();
}
```

- **Receive encrypted the message and decrypted message with symmetric key**

```
DataInputStream in = new DataInputStream(client.getInputStream());
String input = in.readUTF();
// decrypted text
byte[] decryptedText = aead.decrypt(parseHexStr2Byte(input), null);
String output = new String(decryptedText);
//Output in command line
System.out.println("Cipher text is: " + input);
System.out.println("Plain text is: " + output);
```

- **"parseHexStr2Byte" method**
  - ✓ The purpose of using this method is to convert HEX String value back to binary form so that the decryption method could understand the cipher to decrypt the message

```
public static byte[] parseHexStr2Byte(String hexStr)
{
    if (hexStr.length() < 1)
        return null;
    byte[] result = new byte[hexStr.length() / 2];
    for (int i = 0; i < hexStr.length() / 2; i++)
    {
        int high = Integer.parseInt(hexStr.substring(i * 2, i * 2 + 1), 16);
        int low = Integer.parseInt(hexStr.substring(i * 2 + 1, i * 2 + 2),
                16);
        result[i] = (byte) (high * 16 + low);
    }
    return result;
}
```

- **Demonstration**
  - ✓ Compile Server.java code

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ javac Server.java
```

  - ✓ Compile Client.java code

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ javac Client.java
```

  - ✓ Create communication between server and client through port 8000

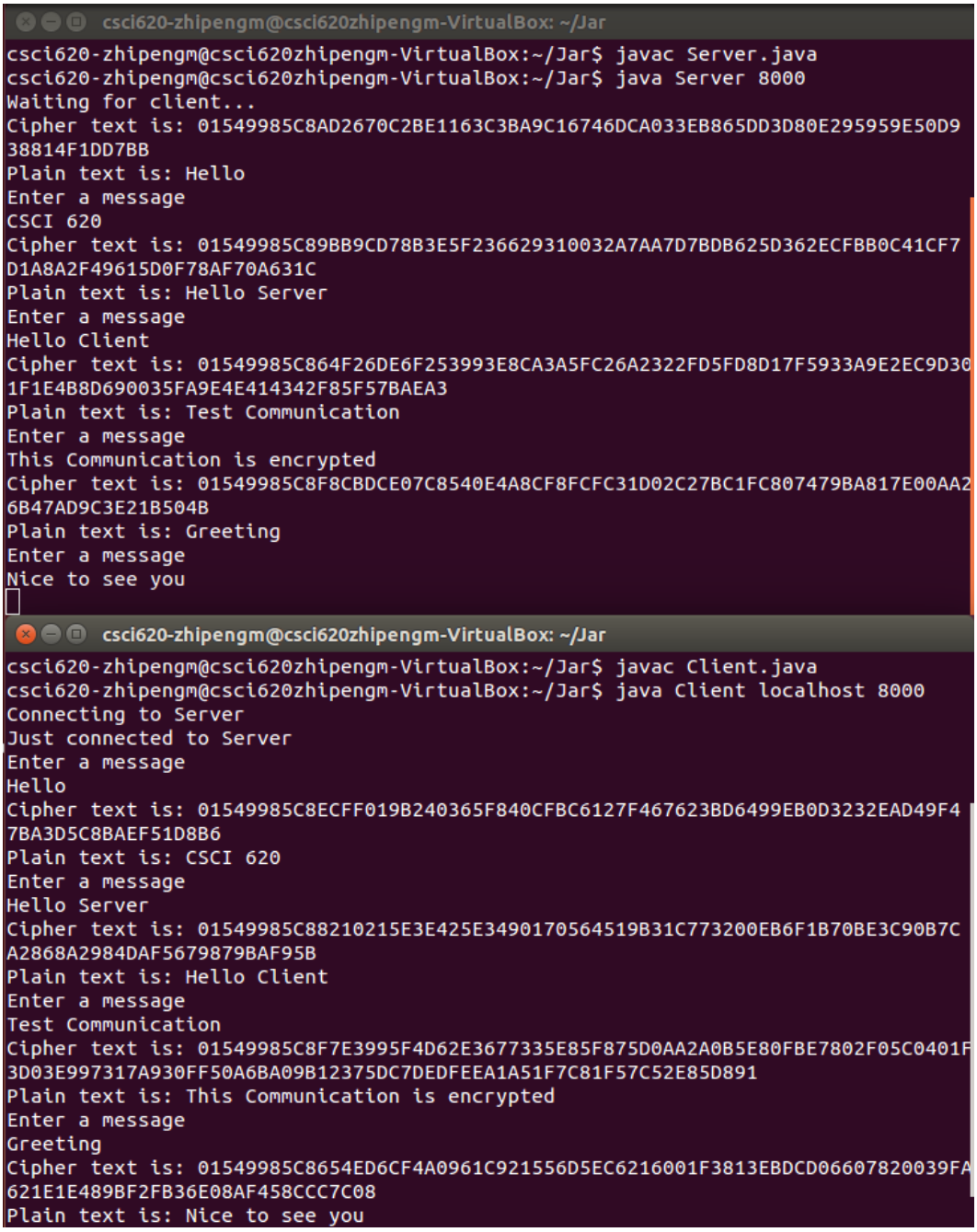
$java Server 8000

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ java Server 8000
Waiting for client...
```

$java Client localhost 8000

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ java Client localhost 8000
Connecting to Server
Just connected to Server
Enter a message
```

✓ Results


csci620-zhipengm@csci620zhipengm-VirtualBox: ~/Jar
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ javac Server.java
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ java Server 8000
Waiting for client...
Cipher text is: 01549985C8AD2670C2BE1163C3BA9C16746DCA033EB865DD3D80E295959E50D9
38814F1DD7BB
Plain text is: Hello
Enter a message
CSCI 620
Cipher text is: 01549985C89BB9CD78B3E5F236629310032A7AA7D7BDB625D362ECFBB0C41CF7
D1A8A2F49615D0F78AF70A631C
Plain text is: Hello Server
Enter a message
Hello Client
Cipher text is: 01549985C864F26DE6F253993E8CA3A5FC26A2322FD5FD8D17F5933A9E2EC9D30
1F1E4B8D690035FA9E4E414342F85F57BAEA3
Plain text is: Test Communication
Enter a message
This Communication is encrypted
Cipher text is: 01549985C8F8CBDCE07C8540E4A8CF8FCFC31D02C27BC1FC807479BA817E00AA2
6B47AD9C3E21B504B
Plain text is: Greeting
Enter a message
Nice to see you
csci620-zhipengm@csci620zhipengm-VirtualBox: ~/Jar
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ javac Client.java
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ java Client localhost 8000
Connecting to Server
Just connected to Server
Enter a message
Hello
Cipher text is: 01549985C8ECFF019B240365F840CFBC6127F467623BD6499EB0D3232EAD49F4
7BA3D5C8BAEF51D8B6
Plain text is: CSCI 620
Enter a message
Hello Server
Cipher text is: 01549985C88210215E3E425E3490170564519B31C773200EB6F1B70BE3C90B7C
A2868A2984DAF5679879BAF95B
Plain text is: Hello Client
Enter a message
Test Communication
Cipher text is: 01549985C8F7E3995F4D62E3677335E85F875D0AA2A0B5E80FBE7802F05C0401F
3D03E997317A930FF50A6BA09B12375DC7DEDFEEA1A51F7C81F57C52E85D891
Plain text is: This Communication is encrypted
Enter a message
Greeting
Cipher text is: 01549985C8654ED6CF4A0961C921556D5EC6216001F3813EBDCD06607820039FA
621E1E489BF2FB36E08AF458CCC7C08
Plain text is: Nice to see you

2) Asymmetric-Key Encryption

JAVA provides built-in packages for the generation of RSA public and private key pairs with the package *java.security*. You can use RSA key pairs to achieve asymmetric key cryptography.

One obvious advantage of using asymmetric key encryption in this project is that we do not need to setup system environment for codes to compile and run

- **Imported packages**

```
import java.net.*;
import java.io.*;
import java.util.*;
import javax.crypto.Cipher;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.nio.file.Paths;
import java.security.*;
import java.nio.file.Files;
import java.nio.file.Path;
```

- **Generating a Key Pair**

```
// First step in creating an RSA Key Pair, Initialize the KeyPairGenerator
// with the key size. Use a key size of 1024 or 2048. Currently recommended key
// size for SSL certificates used in e-commerce is 2048 so that is what we use here.
public static KeyPair buildKeyPair() throws NoSuchAlgorithmException {
    final int keySize = 2048;
    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(keySize);
    return keyPairGenerator.genKeyPair();
}
```

- **Saving the Keys in Binary Format**

  We use the two following codes to get public and private key respectively from the previous key pair generation

  - ✓ PublicKey pubKey = keypair.getPublic();
  - ✓ PrivateKey priKey = keypair.getPrivate();

```
// generate public and private keys
KeyPair keyPair = buildKeyPair();
PublicKey pubKey = keyPair.getPublic();
PrivateKey priKey = keyPair.getPrivate();

FileOutputStream keyfos = new FileOutputStream("publicKey.pub");
keyfos.write(pubKey.getEncoded());
keyfos.close();

FileOutputStream keyfos2 = new FileOutputStream("privateKey.key");
keyfos2.write(priKey.getEncoded());
keyfos2.close();
```

- **Load Private and Public Key from File**

```
/* Read all bytes from the private key file */
byte[] pri = Files.readAllBytes(Paths.get("privateKey.key"));

/* Generate private key */
PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(pri);
KeyFactory kf = KeyFactory.getInstance("RSA");
PrivateKey priKey = kf.generatePrivate(ks);

/* Read all bytes from the public key file */
byte[] pub = Files.readAllBytes(Paths.get("publicKey.pub"));

/* Generate public key */
X509EncodedKeySpec ks2 = new X509EncodedKeySpec(pub);
KeyFactory kf2 = KeyFactory.getInstance("RSA");
PublicKey pubKey = kf.generatePublic(ks2);
```

- **Encrypting Message with Pubic Key and Decrypting Message with Private Key**

```
public static byte[] encrypt(PublicKey pubKey, String message)
        throws Exception {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, (Key) pubKey);

    return cipher.doFinal(message.getBytes());
}

public static byte[] decrypt(PrivateKey priKey, byte [] encrypted)
        throws Exception {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE, priKey);

    return cipher.doFinal(encrypted);
}
```

- **Demonstration**
  - ✓ Compile ServerRSA.java code

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/RSA$ javac ServerRSA.java
```

  - ✓ Compile ClientRSA.java code

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/RSA$ javac ClientRSA.java
```

  - ✓ Create communication between server and client through port 8000

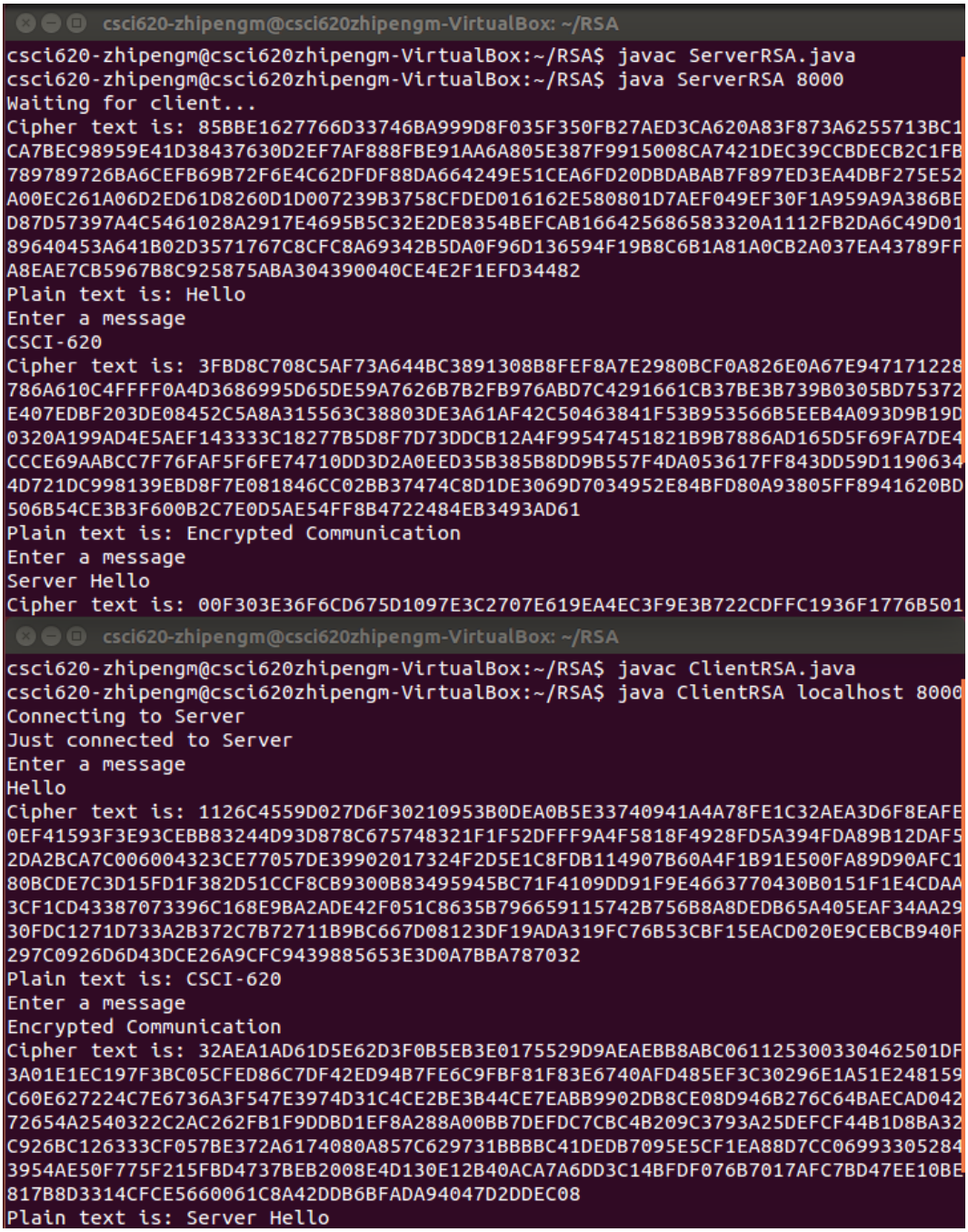
    $java Server 8000

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/RSA$ java ServerRSA 8000
Waiting for client...
```

    $java Client localhost 8000

```
csci620-zhipengm@csci620zhipengm-VirtualBox:~/Jar$ java Client localhost 8000
Connecting to Server
Just connected to Server
Enter a message
```

✓ Results



- **Reference**
  - ✓ https://www.novixys.com/blog/how-to-generate-rsa-keys-java/#2_Generating_a_Key_Pair