CENG 471- TERM PROJECT

STAGE 2

MAY 5, 2019

HASAN YENIADA
220201024

Table of Contents

1) INTRODUCTION

In that second stage of term project, it is asked for us to implement the same scenario with first stage, but in that time RSA encryption scheme should be used. The implemented modulos in the first stage can be used in that stage withoud any change, so I just added the RSA package to our project.

In that time, the sender and receiver generate their own RSA public-private key pairs and share their public keys. Therefore, in the first step of the report, I will show how to generage RSA public-private key pairs in Java, and how the sender and receiver share their public keys between each other. In RSA encryption scheme, the sender and receiver use **different** keys in encryption and decryption operations, however in the first stage, they have used the **same** key in encryption and decryption and that same key is shared using Diffie-Helman Key Exchange scheme.

After generation of the public-private key pairs, using RSA encryption scheme, different length of documents are encrypted by sender and decrypted by receiver. I used the files used in the first stage also in that stage, while measuring the performance of the resulting code. I will also add these performance measurement results using some screenshots and tables to compare the RSA and AES better.

In RSA, there are no key distribution problem, because both the sender and receiver use different keys in encryption-decryption operations, however, the main problem in RSA is that the message length is limited to encrypt, and I will explain that detailly in the third step of the report. Because of that limitation, I have used again AES in that stage to solve that message length problem to help RSA scheme.

2) SUMMARY OF THE REPORT

In the first stage, I will show how to generate public-private key pairs in Java. While generating these keys, we need to use GCD, multiplicative inverse and relatively prime checking operations of the modulo Euclid's Extended Algorithm, which I implemented in the first stage of the project. We also need Fermat Little Theorem while generating big prime numbers p and q.

In the second stage, I will show how to encrypt and decrypt a basic String and different length of documents using RSA encryption scheme, and in that part we also need Fast Exponentiation modulos which is implemented in the first stage. I have used the AES in RSA encryption scheme to solve the message length limitation problem.

At the end, I will show the performance measurements of the RSA and compare it with the AES using some tables and screenshots.

3) FIRST STEP – PUBLIC / PRIVATE KEY GENERATION

In the first step, the sender and receiver generates their own public and private keys. In RSA, the sender and receiver use different keys in encryption and decryption operations, and to generate these keys, the following steps are performed:

- The first step is to generate big prime numbers p and q as 1024 bits length. To generate these big prime numbers, I used the same method in the first stage, and I have explained that method detailly in the first stage.

```
//Generates a prime which has given bit length.
public BigInteger generateBigPrimeNumber(int bitLength) {

    BigInteger prime = new BigInteger("4"); //That is used to initialize the while loop.
    Random rnd = new Random();   //rnd object to use in probablePrime() method invocation.

    while (!tester.checkPrimalityWithFermatLittleTheorem(prime) ) {

        prime = BigInteger.probablePrime(bitLength, rnd);

    }

    //System.out.println("Integer Value of Generated Number:" + prime.intValue());

    return prime;
}
```

- The second step is to generate N using the p and q, and N = p * q, here is the implementation of the generation of p, q and N:

```
public BigInteger generateN(int bitLength){

    p = DH.generateBigPrimeNumber(bitLength);

    q = DH.generateBigPrimeNumber(bitLength);

    while(p.equals(q)) {

        q = DH.generateBigPrimeNumber(bitLength);

    }

    BigInteger N = p.multiply(q);

    return N;
}
```

- The third step is to generate T using again p and q, and here is the implementation of the generation of T:

```
public BigInteger generateT(){

    BigInteger T = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

    return T;
}
```

- The fourth step is to choose e, such as GCD(e, T) = 1, GCD(e, T) = 1 means that e and T are relatively prime, so in that part, I used the isRelativelyPrime method of ExtendedEuclidean, here is the implementation:

```
public BigInteger generateE(BigInteger T) {

    boolean flag = true;

    BigInteger e = getRandomBigInteger(T);

    while(flag) {

        if(euclidean.isRelativelyPrime(e, T)) {
            flag = false;

        } else {

            e = getRandomBigInteger(T);
        }
    }

    return e;

}
```

In that code, we choose a random BigInteger e, and until we find an e such as that e is relatively prime with the generated T. Then returns the chosen e.

- The fifth step is to find d such as d * e = 1 (mod T), means that d is the multiplicative inverse of e in mod T, so I used the findMultiplicativeInverse method of the of ExtendedEuclidean class, here is the implementation:

```
public BigInteger generateD(BigInteger e, BigInteger T) {

    BigInteger d = euclidean.findMultiplicativeInverse(e, T);

    return d;
}
```

- At the end, we have generated public and private keys which are:
  -Public key = (N, e)
  -Private key = (N, d)

Here is the complete key generation implementation:

```
//1) Sender and Receiver generates their own public/private key pairs.

System.out.println("\nThe sender Alice generates her own public/private key pair...");
senderAlice.generatePublicPrivateKeyPair(rsa, 1024);

System.out.println("\nThe receiver Bob generates her own public/private key pair...");
receiverBob.generatePublicPrivateKeyPair(rsa, 1024);
```

```
public void generatePublicPrivateKeyPair(RSA rsa, int keyBitLength) {

    BigInteger N = rsa.generateN(keyBitLength); //N = p * q
    BigInteger T = rsa.generateT();             //T = (p-1) * (q-1)
    BigInteger e = rsa.generateE(T);            //gcd(e, T) = 1

    publicKey.add(N);
    publicKey.add(e);  //public key is generated (N, e)

    BigInteger d = rsa.generateD(e, T);         //

    privateKey.add(N);
    privateKey.add(d); //private key is generated (N, d)
}
```

4) SECOND STEP – PUBLIC KEY SHARING

Both Alice and Bob generated their own public and private key pair as I shown, and next step is thay they share their public keys between each other:

```
//-------------------------------------------------------------------------
//2) Sender and Receiver share their public keys.

((Sender) senderAlice).sendPublicKeyToReceiver(receiverBob);
((Receiver) receiverBob).sendPublicKeyToSender(senderAlice);
```

```
public void sendPublicKeyToSender(Person senderAlice) {

    ((Sender) senderAlice).setPublicKeyOfReceiver(this.getPublicKey());
}
```

```
public void sendPublicKeyToReceiver(Person receiverBob) {

    ((Receiver) receiverBob).setPublicKeyOfSender(this.getPublicKey());
}
```

Now Alice has the public key of Bob, and Bob has the public key of Alice.

5) THIRD STEP – ENCRYPTION / DECRYPTION OPERATIONS

Now, the sender Alice has the public key of the Bob, and she can send a message to Bob encrypting the message with the public key of Bob, and only the Bob who has the private key can decrypt the message. Here is the Encryption and Decryption functions:

Encryption Function = Cipher Data = (Plain Data ^^ e) (mod N)

Decryption Function = Decrypted Data = (Cipher Data ^^ d) (mod N)

Note that, **(N, e)** is the **public key** of the **Bob**, and **Alice** encrypts **Plain Data** with that **public key** and sends that message to **Bob**, then **Bob** uses his **private key (N, d)** and decrypt **Cipher Data**.

Here, the main problem in RSA is that Alice can send a message with limited length, because is the message length exceed the generated number N, then the message Alice sent cannot be the same with the message Bob get, so this situation limits the message length which is sent by RSA encryption scheme. To solve that issue, I did some researches on the internet and found a method named Hybrid Encryption, and in that encyrption scheme, there is no restriction on the message length, and here are the steps of that approach:

- Generate a random symmetric key:

```
//A random Symmetric key is generated by sender and
//then encrypted using RSA encryption scheme with receiver's public key.
BigInteger randomSymmetricKey = ((Sender) senderAlice).generateRandomSymmetricKey(DH, 128);
BigInteger encryptedSymmetricKey = ((Sender) senderAlice).encrypSymmetricKey(rsa, randomSymmetricKey, receiverBob);
```

```
public BigInteger generateRandomSymmetricKey(DiffieHelmanKeyExchange DH, int bitLength) {

    BigInteger randomSymmetricKey = DH.generateBigPrimeNumber(bitLength);

    return randomSymmetricKey;
}
```

- Encrypt it using RSA encryption scheme:

```
public BigInteger generateRandomSymmetricKey(DiffieHelmanKeyExchange DH, int bitLength) {

    BigInteger randomSymmetricKey = DH.generateBigPrimeNumber(bitLength);

    return randomSymmetricKey;
}
```

- Encrypt the Plain Data with the AES encryption scheme:

```
public String encryptString(BigInteger randomSymmetricKey, String strToEncrypt) {

    AES aes = new AES(randomSymmetricKey);

    String encryptedString =  aes.encryptString(strToEncrypt);

    return encryptedString;
}

public byte[] encryptDocument(BigInteger randomSymmetricKey, byte[] plainBytes) {

    AES aes = new AES(randomSymmetricKey);

    byte[] encryptedBytes = aes.encryptDocument(plainBytes);

    return encryptedBytes;
}
```

- Send both (EncryptedSymmetricKey by RSA) and (EncryptedPlainData with AES) to Bob:

```
String plainText = "CENG 471 Term Project Stage1...";
String encryptedString = ((Sender) senderAlice).encryptString(randomSymmetricKey, plainText);
String decryptedString = ((Receiver) receiverBob).decryptString(rsa, encryptedSymmetricKey, encryptedString);
```

```
//Alice encrypts original document with generated randomSymmetricKey using AES.
byte[] encryptedBytes1PageDocument = ((Sender) senderAlice).encryptDocument(randomSymmetricKey1, inputBytesOf1PageDocument);

/*
Then, Alice sends encryptedSymmetricKey and encryptedDocument to Bob,
Bob decrypts encryptedSymmetricKey using RSA and decrypts encryptedDocument
with that decryptedSymmetricKey using AES. */
byte[] decryptedBytes1PageDocument = ((Receiver) receiverBob).decryptDocument(rsa, encryptedSymmetricKey1, encryptedBytes1PageDocument);
```

- Bob decrypts the EncryptedSymmetricKey by RSA with his private key and decrypts EncryptedPlainData by AES with that DecryptedSymmetricKey and got the original message.

This is the Hybrid Encryption and I used the AES code of the first stage to implement that scheme.

```
public String decryptString(RSA rsa, BigInteger encryptedSymmetricKey, String encryptedString) {

    BigInteger decryptedSymmetricKey = rsa.decryptSymmetricKey(encryptedSymmetricKey, getPrivateKey());

    AES aes = new AES(decryptedSymmetricKey);

    String decryptedString =  aes.decryptString(encryptedString);

    return decryptedString;
}

public byte[] decryptDocument(RSA rsa, BigInteger encryptedSymmetricKey, byte[] encryptedBytes) {

    BigInteger decryptedSymmetricKey = rsa.decryptSymmetricKey(encryptedSymmetricKey, getPrivateKey());

    AES aes = new AES(decryptedSymmetricKey);

    byte[] decryptedBytes = aes.decryptDocument(encryptedBytes);

    return decryptedBytes;
}
```

6) PERFORMANCE MEASUREMENTS

Until that, I have explained how to implement each step of stage1 one by one, and now it is time to discuss the performance of the resulting code. I will add the execution time and memory usage results of each different size documents.

- **1 Page Documment Execution Time and Memory Result:**

```
1 Page Document Encryption/Decryption is starting...

1 Page Document RSA Execution Time: 80ms.

1 Page Document Used memory is bytes: 6003472
1 Page Document Used memory is megabytes: 5
```

- **10 Page Documment Execution Time and Memory Result:**

```
10 Page Document Encryption/Decryption is starting...

10 Page Document RSA Execution Time: 89ms.

10 Page Document Used memory is bytes: 6005072
10 Page Document Used memory is megabytes: 5
```

- **100 Page Documment Execution Time and Memory Result:**

```
100 Page Document Encryption/Decryption is starting...

100 Page Document RSA Execution Time: 93ms.

100 Page Document Used memory is bytes: 6781968
100 Page Document Used memory is megabytes: 6
```

- **1000 Page Documment Execution Time and Memory Result:**

```
1000 Page Document Encryption/Decryption is starting...

1000 Page Document RSA Execution Time: 109ms.

1000 Page Document Used memory is bytes: 12020952
1000 Page Document Used memory is megabytes: 11
```

Let's Summarize the Results in a table:

RSA	Execution Time	Memory
1 Page Document	80 ms	5 mb
10 Page Document	89 ms	5 mb
100 Page Document	93 ms	6 mb
1000 Page Document	109 ms	11 mb

7) CONCLUSION

When we compare the performance results of the AES in the first stage, and the performance results of the RSA with Hybrid approach, performance of the AES is better than the RSA, however security of the RSA is better than the AES because of there is no need to any key exchange scheme, because the sender and receiver use different keys in encryption and decryption. The execution times of the 1 page, 10 page, 100 page and 1000 page documents are very close to each other, because the most important factor that affects the execution time is public-private key generation part, and that is common for all of them, and the contents of the documents are encrypted and decrypted with AES and so that operation takes very less time than key generation part. As a result, RSA can be a secure and slow encryption scheme choice, with a message length limitation, however symmetric encryption schemes AES and DES are less secure and good performance encryption scheme choices.

8) REFERENCES

- https://www.quaxio.com/exploring_three_weaknesses_in_rsa/
- https://crypto.stackexchange.com/questions/11904/why-does-plain-rsa-not-work-with-big-messages-mn
- https://hackernoon.com/how-does-rsa-work-f44918df914b
- https://stackoverflow.com/questions/51761721/large-data-not-encrypted-with-rsa-encryption
- https://security.stackexchange.com/questions/33434/rsa-maximum-bytes-to-encrypt-comparison-to-aes-in-terms-of-security
- https://stackoverflow.com/questions/10007147/getting-a-illegalblocksizeexception-data-must-not-be-longer-than-256-bytes-when/46828430#46828430
- https://www.sitepoint.com/encrypt-large-messages-asymmetric-keys-phpseclib/