CENG 471- TERM PROJECT

STAGE 1

APRIL 28, 2019

HASAN YENİADA
220201024

Table of Contents

1) **Introduction** **3**

2) **Summary of the Report** **3**

3) **First Step - Public / Private Key Generation** **4**

4) **Second Step - DH Key Exchange Scheme** **7**

5) **Third Step - Encryption / Decryption of Files With AES** **9**

6) **Performance Measurements** **12**

7) **Security Notes** **13**

8) **Conclusion** **13**

9) **References** **14**

1) INTRODUCTION

In that first stage of the term project, it is asked for us to code some primitive modules to implement a Symmetrical Encryption scenario. I have chosen AES for Symmetrical Encryption because it is more secure than DES. In the scenario, there are a sender and a receiver, and firstly they should generate their own public-private keys. These keys must be a large prime numbers to increase complexity of the encryption and decryption operations. (1)Thus, the first challenge is how to generate these large prime numbers in Java? (Because of that I selected AES for Symmetrical Encryption, the generated prime numbers are in length of 16 bytes(128 bits)).

Then, they should generate a common shared secret key using Diffie-Helman Key Exchange scheme. In that DH Key Exchange scheme, the sender generates a secret key using his/her private key and the public key of the receiver, and the receiver generates a secret key using his/her private key and the public key of the sender. After generations of these secret keys, these secret keys must be common for both sender and receiver, and this is the main property of the DH Key Exchange scheme. At that point, the function which is used to compute these shared secret keys has an important role in that scheme. (2)So, second challenge is how to generate these shared secret keys as eqaully for both sender and receiver.

After generating a common secret key by DH Key Exchange scheme, the next step is that, with any AES/DES usage, a data file is encrypted by sender, and decrypted by the receiver using that generated common secret key by DH Key Exchange scheme. (3)The important point in that step is performance of one of the selected AES/DES in different lengths of data files (1-10-100 and 1000 page lengths documents) in terms of memory, execution time and security.

2) SUMMARY OF THE REPORT

In that report, Firstly, I will explain how to generate public/private keys for sender and receiver. I will show how to generate these big prime numbers in Java and while generating these big prime numbers, how to use Fermat Little Theorem to check primality of generated big numbers.

Secondly, I will show how to calculate secret keys for both sender and receiver, and I will show how to use Fast Exponentiation in that calculations.

Thirdly, I will show how to use AES in Java to encrypt and decrypt different length of pdf files and to encrypt and decrypt a basic String.

Lastly, I will show the performance measurement results for the AES code for different lengths of pdf files in terms of memory, execution time and comment on security of AES on these en/decryption operations. I will use some screenshots from the codes that I implemented in Java to explain the steps better, and I will use some tables to show performance results better.

3) FIRST STEP – PUBLIC / PRIVATE KEY GENERATION

In the first step, the sender and receiver generates their own public and private keys. These keys are big prime numbers in 128 bit length because I have chosen AES for Symmetrical Encryption, and in AES, it is used 10, 12 or 14 rounds to encrypt and decrypt a data block of 128 bits, and depending on the number of rounds, it is used the key size 128, 192 or 256 bits, respectiveley. In Java, the default key size is 128 bits, so I generated public and private keys of the sender and receiver in 128 bits length. How to generate these 128 bit length big prime numbers in Java?

1) Generation of private keys for sender and receiver:

In Java, the BigInteger class has a method named probablePrime(), that method is used to generate a random BigInteger number. Let's look at the definition of the method from JavaDoc:

BigInteger java.math.BigInteger.probablePrime(int bitLength, Random rnd):
Returns a positive BigInteger that is probably prime, with the specified bitLength. The probability that a BigInteger returned by this method is composite does not exceed 2^{-100}.

In our case, bitLength is 128 bits, and I gave 128 and the rnd object to that method, and it returns a BigInteger that is probably prime. But that is not enough to use that nımber, because there is a small probability that the returned number can be composite, so I also should check the primality of the returned number with a Primality Test. In that first stage, it is asked for us to use Fermat Little Theorem to check the primality of any number, so I used FLT to check the primality of returned number from probablePrime(int bitLength, Random rnd). The resulting generateBigPrimeNumber method definition is at below:

```
//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 method loops until it finds a prime number, then returns it. Now, the missing point is how I implemented the Fermat Little Theorem in Java. Let's look at the FLT implementation detailly in the following page.

Fermat Little Theorem Implementation:

According to FLT, if p is a Prime number, and a is an Integer number, such that 1 <= a <= n-1, then:

1) a ^^ (p-1) = 1 (mod p), (Note: ^^ indicates thr power operation.)

Therefore, given an integer x whose primality is under question, finding any integer A in this interval such that this equation (1) is not true suffices to prove that x is composite. So, Let's look at the how I implemented FLT in Java:

```
public boolean checkPrimalityWithFermatLittleTheorem(BigInteger candidatePrimeNumber){

    boolean isPrimeFlag = true;

    BigInteger resultingRemainder;
    BigInteger k = candidatePrimeNumber.subtract(BigInteger.ONE);

    for(BigInteger i = BigInteger.ONE; i.compareTo(k) == -1; i = i.add(BigInteger.ONE)) {

        //Pick a random big integer between [2,p-1]
        BigInteger randomBigInteger = getRandomBigInteger(candidatePrimeNumber);

        resultingRemainder = fastExponentiation.getModulorExponentiation(randomBigInteger,
                                                                         candidatePrimeNumber.subtract(BigInteger.ONE),
                                                                         candidatePrimeNumber);

        if(!resultingRemainder.equals(BigInteger.ONE)) {

            isPrimeFlag = false;
            i = k; //To finish the loop.
        }

        k = k.divide(new BigInteger("2"));
    }

    return isPrimeFlag;
}
```

Here, every time I pick a random integer number between 1 and (p-1), and check the equation (1), and if equation does not hold, I mark given candidatePrimeNumber which is generated from probablePrime() method, as Not Prime, otherwise mark given prime number as Prime. Here the missing point is that I used Fast Exponentiation to evaluate: equation (1) = a ^^ (p-1) = 1 (mod p) .

What does it mean Fast Exponentiation?

Exponention means raising numbers to a power:

- For Example: 2 ^^ 3 means, raising 2 to power 3 => 2 ^^ 3 = 2x2x2

Modular Exponentiation means computing again 2^^3 but now in some modulo:

- For Example: 2^^3 (mod 6)

Fast Exponentiation is used to evaluate that exponentiation operation with big numbers quickly, so in that stage, I used Fast Exponentiation to evaluate the equation (1) because the numbers used in the exponentiation are big prime numbers, and I implemented it as a recursive method in Java such that:

```java
//a to the power b (mod q)
public BigInteger getModulorExponentiation(BigInteger a, BigInteger b, BigInteger q) {

    BigInteger zero = java.math.BigInteger.ZERO;
    BigInteger one = java.math.BigInteger.ONE;
    BigInteger two = new BigInteger("2");

    if(b.equals(zero)) {
        return one;
    }
    if(b.mod(two).equals(zero)){
        return (getModulorExponentiation(a, b.divide(two), q).pow(2))
                                        .mod(q);
    } else {
        return (a.multiply( getModulorExponentiation(a, b.subtract(one), q)))
                .mod(q);
    }
}
```

And I explained how to generate private keys for sender and receiver using the main asymmetrical primitives Fermat Little Theorem and Fast Exponentiation. Now let's look at how I implemented to generate public keys for sender and receiver:

2) Generation of public keys for sender and receiver:

To generate public keys for sender and receiver, we need two system parameters:

- A fixed large prime number p (128 bits in length).
- A fixed generator g (I have chosen 2 as generator)

I generated these two system parameters using the same method in generating private keys named: generateBigPrimeNumber(int bitLength). After generating these required system parameters:

The sender Alice computes her public key with his generated private key a, generator 2 and fixed large prime number p with the Fast Exponentiation:

- Public key of Alice => A = g ^^ a (mod p)

And the receiver Bob computes her public key with his generated private key b, generator 2 and fixed large prime number p with the Fast Exponentiation:

- Public key of Bob => B = g ^^ b (mod p)

The corresponding implementations of public key generation:

```java
public void generatePublicKey(FastExponentiation fastExponentiation) {
    publicKey = fastExponentiation.getModulorExponentiation(fixedGenerator, privateKey, fixedLargePrime);

}
```

fastExponentiation.getModularExponentiation() method calculates the equation:

- fixedGenerator ^^ privateKey (mod fixedLargePrimeNumber)

for Alice: fixedGenerator = 2, privateKey = a, fixedLargePrimeNumber = p, and result = A

for Bob: fixedGenerator = 2, privateKey = b, fixedLargePrimeNumber = p and result = B

4) SECOND STEP – DH KEY EXCHANGE SCHEME

We have generated the public and private keys for the sender Alice and the receiver Bob. The second step is to both Alice and Bob should generate a common secret key by DH Key Exchange scheme. In the first step, we calculated the following public and private keys for Alice and Bob:

- Private key of Alice = a (128 bit random prime number)
- Public key of Alice = A (2 ^^ a (mod p) , where p is 128 bits fixed random prime number
- Private key of Bob = b (128 bit random prime number)
- Public key of Bob = B (2 ^^ b (mod p) , where p is 128 bits fixed random prime number

And now, using DH key exchange scheme Alice sends A to Bob, and Bob sends B to Alice. After that, they should generate the same shared secret key. But how? The function in DH Key Exchange scheme should compute shared secret keys for both Alice and Bob equally.

Alice took public key of the Bob, and using her private key (a) she computes the shared secret key as:

(g ^^ b) ^^ a (mod p) = g ^^ (ba) (mod p) = computed shared secret key from Alice

Bob took public key of the Alice, and using her private key (b) he computes the shared secret key as:

(g ^^ a) ^^ b (mod p) = g ^^ (ab) (mod p) = computed shared secret key from Bob

Note that the resulting computed shared secret keys are equal from Alice and Bob:

- g ^^ (ba) (mod p) = g ^^ (ab) (mod p)

Now let's see how I implemented that computations in Java:

```
public void performDHKeyExchange(FastExponentiation fE, Person sender, Person receiver) {

    //Alice's shared secret key = Bob's Public Key ^^ Alice's private key (mod fixedLargePrime)
    BigInteger sharedSecretKeyOfSender = fE.getModularExponentiation(receiver.getPublicKey(),
                                                                     sender.getPrivateKey(),
                                                                     sender.getFixedLargePrime());
    sender.setSharedSecretKey(sharedSecretKeyOfSender);

    //Bob's shared secret key = Alice's Public Key ^^ Bob's private key (mod fixedLargePrime)
    BigInteger sharedSecretKeyOfReceiver = fE.getModularExponentiation(sender.getPublicKey(),
                                                                       receiver.getPrivateKey(),
                                                                       receiver.getFixedLargePrime());
    receiver.setSharedSecretKey(sharedSecretKeyOfReceiver);

}
```

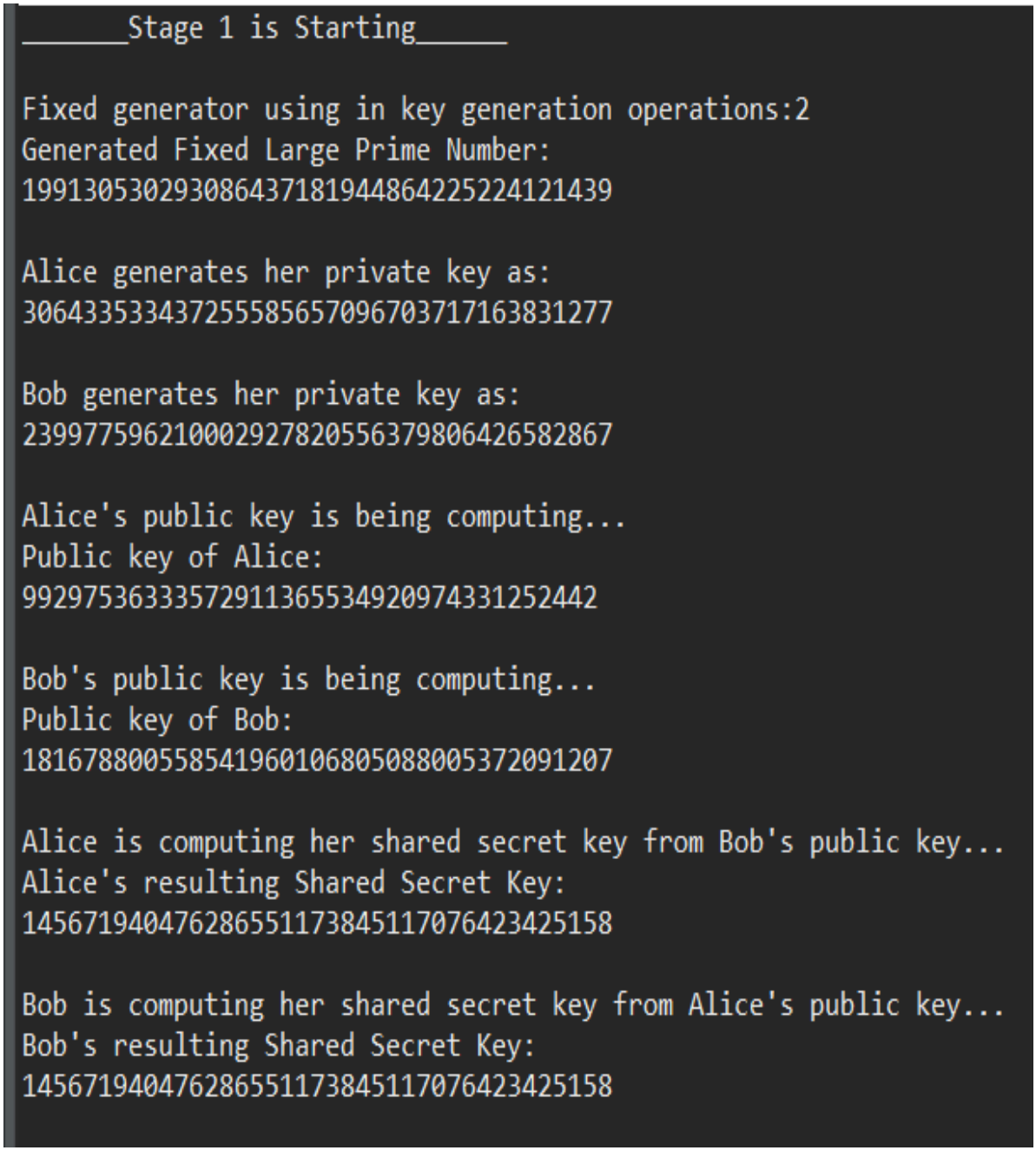
Here the comments and variable names explain how both Alice and Bob computes shared secret keys detailly. Again there are modular exponentiation evaluation here with big numbers, so I used FastExponentiation primitives for these calculations.

And now we generated a common secret key by DH Key Exchange scheme, and we can continue with AES and encrypt a file by sender and decrypt it by receiver, but before that let's see one execution result from these key generation operations with screenshots from Java Console in the following page.

In the first and second steps, we generated:

- Public key of Alice
- Private key Alice
- Public key of Bob
- Private key of Bob
- Shared secret key for Alice and Shared secret key for Bob and they should be identical.

Let’s see the one execution result of these operations in Java with a screenshot from Console.



Note that the resulting shared secret keys are identical for both Alice and Bob. Now we continue with the next step which is with AES usage, a data file is encrypted by Alice and decrypted by Bob.

5) THIRD STEP – ENCRYPTION / DECRYPTION OF FILES WITH AES

The third step is using the AES, encrypt a file by sender Alice, and decrypt it by receiver. Java supports AES with Java Cryptography Extension(JCE), and using the libraries of that extension, I implemented AES easily to encrypt and decrypt a file with different length of data. In AES, each cipher encrypts and decrypts data in blocks of 128 bits using cryptographic keys of 128, 192 and 256 bits, respectively. It uses the same key for encrypting and decrypting, so the sender and the receiver must both know and use the same secret key. The problem in AES is that how both sender and receiver know and use the same secret key. However, we solved that problem in the previous step by DH Key Exchange scheme, and both sender Alice and receiver Bob generated a common shared secret key already.

Because of that I have generated a common secret key by DH Key Exchange scheme, using that key as the same secret key in AES for both Alice and Bob, I easily implemented AES to encrypt and decrypt a file. Before that, to test the AES code is working expected, I tried the code with a simple string to encrypt it with sender Alice, and decrypt it by Bob to see the original message is the same for decrypted message from Bob. Let's see the implementation details and the resulting screenshots to see the results:

1) A basic String Encryption and Decryption:

```
System.out.println("\nAES En/Decryption with a String...");

//SharedSecretKey are the same for both Alice and Bob, I used Alice's SharedSecretKey.
AES aes = new AES(senderAlice.getSharedSecretKey());

String originalString = "CENG 471 Term Project Stage1...";
String encryptedString = ((Sender) senderAlice).encryptString(aes, originalString) ;
String decryptedString = ((Receiver) receiverBob).decryptString(aes, encryptedString) ;

System.out.println("Original Message= "+ originalString);
System.out.println("Encrypted Message= " + encryptedString);
System.out.println("Decrypted Message= " + decryptedString);
```

In the main function of the Application, Alice encrypts the original message. Bob takes that encrypted String and decryps it and save into decryptedString. Execution result of the code:

```
AES En/Decryption with a String...
Original Message= CENG 471 Term Project Stage1...
Encrypted Message= FOL/mF0dj0VESVzC6/dJS+tU+o3qt2mmQB7HS2wJ5zM=
Decrypted Message= CENG 471 Term Project Stage1...
```

encryptString() and decryptString() method implementations:

```
public String encryptString(String strToEncrypt)
{
    try
    {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKeyForAES);
        return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));

    } catch (Exception e) {
        System.out.println("Error while encrypting: " + e.toString());
    }

    return null;
}

public String decryptString(String strToDecrypt)
{
    try
    {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
        cipher.init(Cipher.DECRYPT_MODE, secretKeyForAES);

        return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));

    } catch (Exception e) {
        System.out.println("Error while decrypting: " + e.toString());

    }
    return null;
}
```

- encryptString() method takes the original message(plainText) as an argument and returns the encrypted original message(cipherText).
- decryptString() method takes the encrypted message(cipherText) as an argument and returns the decrypted message which is equal to original message(plainText)

2) Document Encryption and Decryption with AES:

I will add just the encryption and decryption details of the 10 Page length document, because other different size documents' en/decryption details are the same:

```
//Alice encrypts the originalFile1Page.pdf and save it as encryptedFile1Page(document1PageLengthEncrypted.pdf).
byte[] encryptedBytes1PageDocument = ((Sender) senderAlice).encryptDocument(aes, inputBytesOf1PageDocument);

//Then, Alice sends encrypted file to Bob, Bob decrypt it and save as document1PageLengthDecrypted.pdf.
byte[] decryptedBytes1PageDocument = ((Receiver) receiverBob).decryptDocument(aes, encryptedBytes1PageDocument);
```

In the main function of the Application,

Alice encrypts the original document(document10PageLength.pdf).

Bob takes that encrypted document and decryps it and at the end of these operations, decryptedBytes of the encryptedDocument is save into document10PageLengthDecrypted.pdf and the content of these 2 documents are the same. (You can check after execution of the code).

encryptDocument() and decryptDocument() method implementations:

```
public byte[] encryptDocument(byte[] plainBytes)
{

    byte[] encryptedBytes = null;

    try
    {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKeyForAES);

        encryptedBytes = cipher.doFinal(plainBytes);


    }   catch (NoSuchPaddingException | NoSuchAlgorithmException
            | InvalidKeyException | BadPaddingException
            | IllegalBlockSizeException ex) {

        System.out.println(ex.getMessage());
        ex.printStackTrace();
    }

    return encryptedBytes;
}
```

```
public byte[] decryptDocument(byte[] encryptedBytes)
{

    byte[] decryptedBytes = null;

    try
    {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, secretKeyForAES);

        decryptedBytes = cipher.doFinal(encryptedBytes);

    }   catch (NoSuchPaddingException | NoSuchAlgorithmException
            | InvalidKeyException | BadPaddingException
            | IllegalBlockSizeException ex) {

        System.out.println(ex.getMessage());
        ex.printStackTrace();
    }

    return decryptedBytes;
}
```

- encryptDocument() method takes the original document(document10PageLength.pdf) as an argument and returns the encrypted original document.
- decryptDocument() method takes the encrypted document as an argument and returns the decrypted document(document10PageLengthDecrypted.pdf) which is equal to original document(pdf)

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 AES Execution Time: 5ms.

1 Page Document Used memory is bytes: 6783136
1 Page Document Used memory is megabytes: 6
```

- 10 Page Documment Execution Time and Memory Result:

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

10 Page Document AES Execution Time: 6ms.

10 Page Document Used memory is bytes: 6000440
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 AES Execution Time: 8ms.

100 Page Document Used memory is bytes: 6776392
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 AES Execution Time: 23ms.

1000 Page Document Used memory is bytes: 12015728
1000 Page Document Used memory is megabytes: 11
```

Let's Summarize the Results in a table:

AES	Execution Time	Memory
1 Page Document	5 ms	6 mb
10 Page Document	6 ms	6 mb
100 Page Document	8 ms	6 mb
1000 Page Document	23 ms	11 mb

According to results, execution time and memory usage of the 1 page, 10 page and 100 page document encryption and decryption operations are very close to each other, and 1000 page document execution times takes nearly 3 or 4 times more than the others and the memory usage of it is nearly more than 2 times than others.

7) SECURITY NOTES

Let's comment on the security of the resulting Symmetrical Encryption. Initially, I have generated public and private keys as prime numbers in 128 bits length, and then I have used DH Key Exchange scheme to generate a common shared secret key for both Alice and Bob. An attacker can see the generated public key, but to evaluate private key from that public key is so much difficult, so I used big prime numbers in these operations to increase complexity. That property makes the key exchange operations secure, and after the sender and the receiver share a common secret key between them, using AES, the sender encrypts a document with AES and sends it to the receiver.

At that point, security of the AES is important for the security of our resulting code. Hopefully, AES has a very strong security operations while encripting a file. It uses 10, 12 or 14 rounds in an encryption operation and at each round, it uses 4 types of transformations to increase the complexity of the operation namely, substitution, permutation, mixing and key-adding. So, we can trust the AES at that point.

One security issue in our resulting Symmetrical Encryiption can be in the DH Key Exchange scheme. There can be a man in the middle attack when the Alice and Bob shares their public keys, and because of there is no identification operation in DH Key exchange scheme, that attack can be successful, and communicaiton between Alice and Bob can be controlled by an attacker and this makes the system unsecure.

8) CONCLUSION

As a result, in that stage I have implemented the required modules for a Symmetrical Encryption, and then using them, the sender Alice and the receiver Bob generated their own public and private keys, and using DH Key Exchange scheme they generated a common secret key, and using AES, different lengths of data files(String and pdfs) are encrypted by Alice and then send to the Bob, and Bob decrypted these encrypted data files and generated the decrypted data files which has the same content with the original data files

After that, I have shown the performance measurements of the resulting AES code with a screenshot and a table. And finally I commented on the security of the resulting system and finished the first stage of the Term Project.

(Note: you can also find the implementation of Extended Euclidean Algorithm in the uploaded project)

9) REFERENCES

- https://proandroiddev.com/security-best-practices-symmetric-encryption-with-aes-in-java-7616beaaade9
- https://www.oodlestechnologies.com/blogs/Encrypt-And-Decrypt-A-File-In-Java
- https://www.baeldung.com/java-cipher-input-output-stream
- https://howtodoinjava.com/security/aes-256-encryption-decryption/
- https://medium.com/@prudywsh/how-to-generate-big-prime-numbers-miller-rabin-49e6e6af32fb
- http://commons.apache.org/proper/commons-crypto/xref-test/org/apache/commons/crypto/examples/CipherByteArrayExample.html
- https://www.novixys.com/blog/java-aes-example/
- https://www.vogella.com/tutorials/JavaPerformance/article.html
- https://www.quickprogrammingtips.com/java/creating-a-random-biginteger-in-java.html