ECE 3790 Lab 4 Report – Winter 2019

Daniel Lovegrove (7763168)

1: INTRODUCTION

This lab serves as an example of how RSA encryption and Diffie-Hellman (DH) key exchange work. Both of these algorithms are used for public key encryption, which is required to communicate between parties securely over the internet or over some digital communication channel.

This first major section of this lab report dicusses the code written for the RSA encryption part of the lab, as well as answers the questions about RSA from the lab instruction. The second section discusses the code written for the DH key exchange part of the lab and answers any questions from the lab report as well.

All code is written in Java, and programmed with the Netbeans IDE (Version 8.1). As such, to run the code it is highly recommended to use Netbeans.

The documentation on how to run the code is in Appendix A. The code for RSA encryption is in Appendix B, and the code for the DH key exchange is in Appendix C.

2: RSA FILE ENCRYPTION

For this part of the lab, we were required to encrypt a small file and decrypt it using RSA.

2.1 Code Discussion

To accomplish file encryption and decryption, I split the implentation of the file encryption and the file decryption into separate Java classes. This is opposed to the code given to us for Lab 4 which encrypts a string and immediately decrypts it within the same class. With my implementation, one can run the file encryption to encrypt a file, and run the file decryption whenever they like afterwards to decrypt the file.

I limited the size of the file that can be encrypted to the number of bits in the modulus. This keeps the code simple. I did not attempt the part of the lab where a larger file could be encrypted block by block. In other words, the max file size for my implementation is 1 block. To calculate the block size, I take the key size in bits and divide it by 8 to get the max number of bytes allowed.

Before physically encrypting a file, the file encryptor first calculates *d*, *e*, and *n*, leveraging Java's `BigInteger` library. *e, d,* and *n* are all standard RSA encryption values. *e* and *n* combined are the public key, and *d* is the secret key.

To encrypt the file, the raw bytes from the file are retrieved and used to create a Java `BigInteger`, which is then taken to the e^{th} power and modulated by the modulus *n*. The result of this calculation is the encrypted block representing the file. As an implementation detail, the code I wrote physically overwrites the file that the user wants to encrypt with the encrypted representation of the file. This is more exciting than creating a separate file that stores the encrypted representation.

To further make things a little more interesting in my implementation, I encode the encrypted file in Base 64. This is a more realistic way of encoding the encrypted file than simply storing the integer representation of the encrypted file, since Base 64 is used heavily in digital communications. From the documentation for the PHP language, Base 64 is commonly used for transporting data over channels that are not suitable for transmitting pure binary[1]. This can include email protocols and the http protocol.

Since the file decryptor code I wrote needs to know the secret key *d*, I store the values of *e*, *d*, and *n* in a file after encrypting the file. This is absolutely not secure, but since this part of the lab is only meant as a proof of concept of RSA, I store the secret key in plain text to be used by the file decryption code.

To decrypt the file, the decryption code first gets the raw bytes of the file, decodes them from Base 64, converts the bytes into a `BigInteger`, and then takes that integer to the d^{th} power and modulates it by *n*. This produces the decrypted, original contents of the file. Note that the values of *d* and *n* come from the file written by the encryption code. After getting the original file contents, I store them back in the original file as plain text.

I ran my code for several key sizes to test how long encryption, decryption, and generation of *e*, *d*, and *n* took. The results are shown below in Table 1.

Table 1: Runtime for varying sizes of n

Key Size	Time to generate e, d, and n	Time to encrypt file	Time to decrypt file
64	14 ms	1 ms	1 ms
128	81 ms	1 ms	1 ms
256	95 ms	1 ms	1 ms
512	130 ms	1 ms	2 ms
1024	208 ms	1 ms	10 ms
2048	1511 ms	1 ms	44 ms

1 http://php.net/manual/en/function.base64-encode.php

2.2: Lab Questions

Q1: What is the distribution of primes among all numbers?

From the Wikipedia page for the prime number theorem, the distribution of primes up to a number *n* is approximated by π(n), where:

$$\pi(n) \simeq \frac{n}{\log(n)}$$ [2]

As *n* approaches infinity, so does the number of primes. This means that there are infinitely many primes.

Q2: How many numbers are on average selected by BigInteger before finding a likely prime?

I use a certainty value in the constructor of BigInteger of 20. From the documentation for BigInteger, the constructor used returns a number that is a prime with probability $1 - 1/2^{20}$. This means that the number is a prime with probability 0.9999. I'm not sure how this relates to the amount of numbers selected before finding a likely prime.

Q3: What are the basic number theory algorithms used in RSA?

Euclid's greatest common divisor algorithm is used to find a suitable value of *e*. The extended Euclid algorithm is used to find a value for *d.*

Q4: How can RSA be used within private key crypto systems?

The value *d* is the private key part of RSA encryption, *d* can be used in a private key crypto system to decrypt data.

Q5: Does RSA require prime numbers or likely-prime numbers?

It requires numbers that are very likely prime, they need not be prime with 100% certainty because it is still very difficult to factor a number that is the composition of two likely-prime numbers.

2 https://en.wikipedia.org/wiki/Prime_number_theorem

Q6: When was prime generation found to be capable in polynomial time?

I found a paper titled “PRIMES is in P,” which posits that there is an algorithm that determines whether a number is prime in polynomial time[3]. This paper was published in **2004**.

3: DIFFIE-HELLMAN KEY EXCHANGE

The code I wrote for this part of the lab exactly follows the steps outlined in the lab instructions on page 4. For this reason, I do not think it is very pertinent to discuss the code.

The following is a sample output from running the code I wrote for this part of the lab. The value of q was generated with a size of 64 bits.

```
Value of q: 11812476823124262683
Value of p: 23624953646248525367
Value of g: 2

Bob's secret is b = 8230317014000892644
Bob calculated B = 21715997579812063126

Alice's secret is a = 28131081540434829450
Alice calculated A = 11279538051077613412

---------- Alice and Bob share A and B ----------

Bob calculated K = 1705536712930390710
Alice calculated K = 1705536712930390710

Alice and Bob calculated the same K.
```

The most important part of this output is the last three lines, where Alice and Bob both come up with the same value of K, even though they do not know the other’s secret. K is the shared secret between Alice and Bob, which they can use to encrypt messages and send between each other. K is never sent over the network.

3.2: Lab Questions

Q1: What are RFCs?

RFC stands for Request for Comments. They are technical documents that are peer-reviewed, dicussed, and verified before being added to the list of standards recognized by a standards body such as IETF (Internet Engineering Task Force).

3 http://annals.math.princeton.edu/2004/160-2/p12

Q2: Does the code work when p is not a safe prime?

Yes, it did work when I tried the code without checking if p=2q+1 is prime.

Q3: Does the code work if p is not a prime at all?

Yes, it did work when I generated p as a random number.

Q4: Why are safe primes selected?

Through my research on safe primes, I found that they are used because they have a large prime-order subgroup of size q, and no other small subgroups. You want to avoid having small subgroups because it makes it harder to compute discrete logarithms to perform an attack[4].

4 https://www.johndcook.com/blog/2017/01/12/safe-primes-sylow-theorems-and-cryptography/

APPENDIX A: CODE DOCUMENTATION

To run the code for this lab, the Netbeans IDE is recommended. There are two Netbeans projects included with this report, one is called FileEncryptionDecryption and the other is called DiffieHellman.

FileEncryptionDecryption

To test the code for encryption and decryption of a small file with RSA, first open the project FileEncryptionDecryption in Netbeans. To run the test, run the `RSAtest.java` class in the package `rsatest` within Netbeans by right clicking the file and clicking “Run File.” This encrypts and decrypts the text file `sample.txt` in the package `textfiles` and tests whether the file contents are the same before encryption and after decryption.

DiffieHellman

To test the code for the Diffie-Hellman key exchange, first open the project DiffieHellman in Netbeans. To run the example of DH key exchange, run the `DiffieHellman.java` class in the package `diffiehellman` within Netbeans by right clicking the file and clicking “Run File.” This prints out information relating to a DH key exchange between Alice and Bob.

APPENDIX B: CODE FOR RSA FILE ENCRYPTION

FILE: FileEncryptor.java

Responsible for encrypting a file. Takes a filename from the args array and encrypts that file.

```
package rsa;

/*********************************************************************
*
* FileEncryptor.java
*
* This program implements the RSA algorithm to encrypt a small file.
*
* RSA is a public key encryption system which makes use of two keys, a public
* one and a private one. These become exponents for encryption and decryption.
*
* The modulus value is determined by two probabilistically prime numbers of a
* given (large) size. The public exponent may then be selected, and a private
* exponent calculated.
*
* The BigIntegers class provides native support for implementing encryption
* algorithms in Java.
*********************************************************************/

import java.io.*;
import java.util.Random;
import java.math.BigInteger;
import java.util.Base64;

public class FileEncryptor {

   public static void main(String[] args) {
       int KEY_SIZE = 128;
       int CERTAINTY = 20;
       File file;
       String fileContents;

       // Get a file from the command line arguments
       try {
           file = getFile(args, KEY_SIZE);
       } catch (RuntimeException e) {
           System.out.println(e.getMessage());
           return;
       }

       // Read the contents of the file
       try {
           fileContents = readFileContents(file);
       } catch (IOException e) {
           System.out.println(e.getMessage());
           return;
```

```
    }

    System.out.println(String.format("Encrypting file with a modulus of %d bits.\n", KEY_SIZE));

    // ------------------------------------------------------------------
    // Calculate keys and modulus
    // ------------------------------------------------------------------
    long start = System.currentTimeMillis();
    BigInteger one = new BigInteger("1");

    // Generate two likely primes of KEY_SIZE
    // The modulus for RSA is determined as the product of p and q.
    BigInteger p = new BigInteger(KEY_SIZE, CERTAINTY, new Random());
    BigInteger q = new BigInteger(KEY_SIZE, CERTAINTY, new Random());
    BigInteger n = p.multiply(q);

    // PHI(n) represents the Euler totient function, which is calculated as:
    // PHI(n) = (p-1)*(q-1)
    // It is used for determining public and private exponents
    BigInteger phi = p.subtract(one).multiply(q.subtract(one));

    // Select a public exponent (repeat until GCD condition is met).
    BigInteger e = new BigInteger(32, 4, new Random());
    BigInteger gcd = phi.gcd(e);
    while (!gcd.equals(one)) {
        e = new BigInteger(32, 4, new Random());
        gcd = phi.gcd(e);
    }

    // Calculate the private exponent.
    BigInteger d = e.modInverse(phi);
    System.out.println(String.format("Took %d milliseconds to calculate n, e, and d.\n",
        System.currentTimeMillis() - start));


    // ------------------------------------------------------------------
    // Encrypt the message
    // ------------------------------------------------------------------
    start = System.currentTimeMillis();
    Base64.Encoder encoder = Base64.getEncoder();
    byte[] fileContentsAsBytes = fileContents.getBytes();
    BigInteger fileContentsAsBigInt = new BigInteger(fileContentsAsBytes);
    BigInteger encrypted = fileContentsAsBigInt.modPow(e, n);
    String base64EncryptedFileContents = encoder.encodeToString(encrypted.toByteArray());

    System.out.println(String.format("Took %d milliseconds to encrypt the file",
        System.currentTimeMillis() - start));
    System.out.println("Original file contents: " + fileContents);
    System.out.println("Encrypted file contents (Base 64): " + base64EncryptedFileContents + "\n");


    // ------------------------------------------------------------------
    // Overwrite initial file and create new file to store keys and modulus
    // ------------------------------------------------------------------
    try {
```

```
            FileWriter encryptedFileWriter = new FileWriter(file, false);
            encryptedFileWriter.write(base64EncryptedFileContents);
            encryptedFileWriter.close();
        } catch (IOException exc) {
            System.out.println(exc.getMessage());
            return;
        }

        // Write RSA information to file encryptionInfo.txt
        try {
            String fileDirectory = file.getAbsoluteFile().getParent();
            File rsaInfoFile = new File(fileDirectory, "encryptionInfo.txt");
            FileWriter rsaInfoWriter = new FileWriter(rsaInfoFile, false);
            StringBuilder rsaInfoBuilder = new StringBuilder();
            rsaInfoBuilder.append(String.format("e=" + e + "\n"));
            rsaInfoBuilder.append(String.format("d=" + d + "\n"));
            rsaInfoBuilder.append(String.format("n=" + n));
            System.out.println("Encryption information:\n" + rsaInfoBuilder.toString());
            rsaInfoWriter.write(rsaInfoBuilder.toString());
            rsaInfoWriter.close();
        } catch (IOException exc) {
            System.out.println(exc.getMessage());
            return;
        }
    }

    /*
     * Get a valid file from the command line arguments.
     */
    public static File getFile(String[] args, int KEY_SIZE) throws RuntimeException {
        if (args.length != 1) {
            throw new RuntimeException("ERROR: Must supply an argument for a file name.");
        }

        String filename = args[0];
        File file = new File(filename);

        // If file does not exist or is not a file or is a directory, throw exception
        if (!file.exists() || !file.isFile() || file.isDirectory()) {
            throw new RuntimeException(String.format("ERROR: %s is not a valid file.", filename));
        }

        int maxLength = (KEY_SIZE / 8); // File length cannot be greater than modulus
        if (file.length() > maxLength) {
            throw new RuntimeException(String.format("ERROR: File is too big. Max # of bytes for
KEY_SIZE %d is %d", KEY_SIZE, maxLength));
        }

        return file;
    }

    /*
     * Return contents of file
     */
    public static String readFileContents(File file) throws IOException {
```

```
        byte[] data;
        try (FileInputStream inputStream = new FileInputStream(file)) {
            data = new byte[(int) file.length()];
            inputStream.read(data);
        }

        return new String(data, "UTF-8");
    }
}
```

FILE: FileDecryptor.java

Responsible for decrypting a file. Takes a filename from the args array and decrypts that file.

```
package rsa;

/*******************************************************************************
*
* FileDecryptor.java
*
* This program implements the RSA algorithm to decrypt a small file.
*******************************************************************************/

import java.io.*;
import java.math.BigInteger;
import java.util.Base64;
import java.util.regex.*;

public class FileDecryptor {

    public static void main(String[] args) {
        Matcher matcher;
        File file;
        String fileContents;
        Pattern pattern_n = Pattern.compile("n=(\\d+)");
        Pattern pattern_d = Pattern.compile("d=(\\d+)");
        BigInteger d, n;

        // Get a file from the command line arguments
        try {
            file = getFile(args);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
            return;
        }

        // Read the contents of the file
        try {
            fileContents = readFileContents(file);
        } catch (IOException e) {
            System.out.println(e.getMessage());
            return;
```

```
    }

    System.out.println("Decrypting file.\n");

    // -----------------------------------------------------------------
    // Get encryption information from encryptionInfo.txt
    // -----------------------------------------------------------------
    try {
        String encryptionFileDirectory = file.getAbsoluteFile().getParent();
        File rsaInfoFile = new File(encryptionFileDirectory, "encryptionInfo.txt");
        String encryptionFileContents = readFileContents(rsaInfoFile);
        matcher = pattern_n.matcher(encryptionFileContents);
        matcher.find();
        n = new BigInteger(matcher.group(1));
        matcher = pattern_d.matcher(encryptionFileContents);
        matcher.find();
        d = new BigInteger(matcher.group(1));
    } catch (IOException exc) {
        System.out.println(exc.getMessage());
        return;
    }


    // -----------------------------------------------------------------
    // Decrypt the message
    // -----------------------------------------------------------------
    long start = System.currentTimeMillis();
    Base64.Decoder decoder = Base64.getDecoder();
    byte[] fileContentsAsBytes = fileContents.getBytes();
    byte[] encryptedBytesDecoded = decoder.decode(fileContentsAsBytes);
    BigInteger encryptedFileContentsAsBigInt = new BigInteger(encryptedBytesDecoded);
    BigInteger decryptedFileContentsAsBigInt = encryptedFileContentsAsBigInt.modPow(d, n);
    String decryptedFileContents = new String(decryptedFileContentsAsBigInt.toByteArray());

    System.out.println(String.format("Took %d milliseconds to decrypt the file",
        System.currentTimeMillis() - start));
    System.out.println("Encrypted file contents (Base 64): " + fileContents);
    System.out.println("Decrypted file contents: " + decryptedFileContents);


    // -----------------------------------------------------------------
    // Overwrite file with decrypted contents
    // -----------------------------------------------------------------
    try {
        FileWriter decryptedFileWriter = new FileWriter(file, false);
        decryptedFileWriter.write(decryptedFileContents);
        decryptedFileWriter.close();
    } catch (IOException exc) {
        System.out.println(exc.getMessage());
        return;
    }
}

/*
 * Get a valid file from the command line arguments.
```

```
     */
    public static File getFile(String[] args) throws RuntimeException {
        if (args.length != 1) {
            throw new RuntimeException("ERROR: Must supply an argument for a file name.");
        }

        String filename = args[0];
        File file = new File(filename);

        // If file does not exist or is not a file or is a directory, throw exception
        if (!file.exists() || !file.isFile() || file.isDirectory()) {
            throw new RuntimeException(String.format("ERROR: %s is not a valid file.", filename));
        }

        return file;
    }

    /*
     * Return contents of file
     */
    public static String readFileContents(File file) throws IOException {
        FileInputStream inputStream = new FileInputStream(file);
        byte[] data = new byte[(int) file.length()];
        inputStream.read(data);
        inputStream.close();

        return new String(data, "UTF-8");
    }
}
```

FILE: RSAtest.java

Tests the file encryption and decryption classes. Checks the file contents before encryption and after decryption, and compares the two.

```
package rsatest;

/*****************************************************************************
*
* RSAtest.java
*
* This program uses the FileEncryptor and FileDecryptor to first encrypt a real
* text file, and decrypt that file.
*****************************************************************************/

import rsa.FileEncryptor;
import rsa.FileDecryptor;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class RSAtest {
```

```
    public static void main(String[] args) {
        String filename = "src/textfiles/sample.txt";
        String[] sampleArgs = {filename};

        File file = new File(filename);
        String initialContents = readFileContents(file);

        // Encrypt file with encryptor class
        FileEncryptor.main(sampleArgs);
        System.out.println("\n-------------------------------------------------\n");

        // Decrypt the file with decryptor class
        FileDecryptor.main(sampleArgs);

        String finalContents = readFileContents(file);

        if (initialContents.equals(finalContents)) {
            System.out.println("\n> PASS: File was encrypted and decrypted correctly.");
        } else {
            System.out.println("\n> FAIL: File was not encrypted or decrypted correctly.");
        }
    }

    public static String readFileContents(File file) {
        try (FileInputStream inputStream = new FileInputStream(file)) {
            byte[] data = new byte[(int) file.length()];
            inputStream.read(data);
            inputStream.close();

            return new String(data, "UTF-8");
        } catch (IOException e) {
            System.out.println(e.getMessage());
            return "";
        }
    }
}
```

APPENDIX C: CODE FOR DIFFIE HELLMAN KEY EXCHANGE

FILE: DiffieHellman.java

Prints out the numbers and information associated with a DH key exchange. Does not encrypt anything, but does generate the shared secret between the two parties (Alice and Bob).

```
package diffiehellman;

import java.util.Random;
import java.math.BigInteger;

public class DiffieHellman {
   public static void main(String[] args) {
       int KEY_SIZE = 64;
       BigInteger one = new BigInteger("1");
       BigInteger two = new BigInteger("2");

       BigInteger[] aValues = {
           new BigInteger("2"),
           new BigInteger("3"),
           new BigInteger("5"),
           new BigInteger("7"),
           new BigInteger("9"),
           new BigInteger("11"),
           new BigInteger("15"),
           new BigInteger("19")
       };

       // ------------------------------------------------------------------
       // Generate a prime q and safe prime p
       // ------------------------------------------------------------------
       boolean pIsPrime = false;
       BigInteger q = null;
       BigInteger p = null;

       while (!pIsPrime) {
           q = new BigInteger(KEY_SIZE, 20, new Random());

           p = q.multiply(two).add(one);
           BigInteger pMinusOne = p.subtract(one);
           pIsPrime = true; // Assume prime until proven otherwise

           // Test if p is prime with the test a values
           for (BigInteger a : aValues) {
               // If a^(p-1)mod(p) != 1, p is not a prime number
               if (a.modPow(pMinusOne, p).compareTo(one) != 0) {
                   pIsPrime = false;
                   break;
               }
           }
       }
```

```
System.out.println("Value of q: " + q);
System.out.println("Value of p: " + p);

// ------------------------------------------------------------------
// Find a generator g
// ------------------------------------------------------------------
BigInteger g = new BigInteger("2");
boolean gIsGenerator = false;

while (!gIsGenerator) {
    if (g.modPow(q, p).compareTo(one) != 0) {
        g = g.add(one);
    } else {
        gIsGenerator = true;
    }
}

System.out.println("Value of g: " + g);

// ------------------------------------------------------------------
// Bob generates a secret b, and finds B=g^b mod(p)
// ------------------------------------------------------------------
BigInteger b = new BigInteger(p.bitLength(), new Random());
BigInteger B = g.modPow(b, p);
System.out.println("\nBob's secret is b = " + b);
System.out.println("Bob calculated B = " + B);

// ------------------------------------------------------------------
// Alice generates a secret a, and finds A=g^a mod(p)
// ------------------------------------------------------------------
BigInteger a = new BigInteger(p.bitLength(), new Random());
BigInteger A = g.modPow(a, p);
System.out.println("\nAlice's secret is a = " + a);
System.out.println("Alice calculated A = " + A);

System.out.println("\n---------- Alice and Bob share A and B ----------\n");

// ------------------------------------------------------------------
// Bob receives A from Alice and finds K = A^b mod(p)
// ------------------------------------------------------------------
BigInteger bob_K = A.modPow(b, p);
System.out.println("Bob calculated K = " + bob_K);

// ------------------------------------------------------------------
// Alice receives B from Bob and finds K = B^a mod(p)
// ------------------------------------------------------------------
BigInteger alice_K = B.modPow(a, p);
System.out.println("Alice calculated K = " + alice_K);


// Check if K's match.
if (bob_K.compareTo(alice_K) == 0) {
    System.out.println("\nAlice and Bob calculated the same K.");
} else {
```

```
            System.out.println("The values of K did not match. Something went wrong.");
        }
    }
}
```