Algorithms and Time Complexity Project 1

RSA Cryptosystem

Ronaldo Gjini, Indrit Caca

June 25, 2021

Contents

1 Introduction

This report paper represents the solution and the implementation of a simple and secure system which helps communicating parties exchange messages between themselves. One such system is the RSA cryptosystem which is the topic discussed in detail in this report. We consider the problem of encrypting messages between communicating parties so that a third party is not able to intercept and see the plaintext exchange between parties without solving difficult problems like the factorization problem. As for the requirement of this assignment, we have to implement a library of the respective methods

that perform different functions in the RSA cryptosystem like the modular multiplicative inverse, greatest common divisor, extended euclidean greatest common divisor, modular exponentiation and prime generation. All of these functions mentioned above as explained in the detail later on and make possible the solution of the problem that is the creation of a simple and secure RSA cryptosystem.

2 Solution

We have implemented the RSA cryptosystem in an efficient way, consisting of three classes: Main, User and RSALibrary.

2.1 Main Class

The Main class is the entry point of the program. Firstly, we generate the private and public keys for both Alice and Bob, and store them in their respective user objects. Then we read the input text from Alice, and then we call the encryptText method by passing the text and Bob's public keys (N,e). This method will return a list of BigIntegers which will contain the encrypted chunks of Alice's message. Then we call the decryptText method by passing the encrypted chunks and Bob's private keys (N,d). This method will return a list of decrypted chunks, which then we combine to retrieve the original message. We then do the same thing for Bob. We take Bob's message and encrypt it using Alice's public keys (N,e), and then we decrypt it using Alice's private keys (N,d).

2.2 RSALibrary Class

RSALibrary class is the most import class of the project, because it contains all the algorithms required to make RSA cryptosystem work. All the algorithms have been written by scratch, and none of the pre-built BigIntegers prohobited by the requirements have not been used. The methods in this class are responsible for generating the keys, and encrypting/decrypting messages using the algorithms provided. The algorithms and their time complexity are explained thoroughly in the section.

2.3 User Class

The user class is a simple class which contains the keys of each user. The class has three local variables represting the three RSA keys: n, e and d. Each time we generate new keys for a user, we create a new user object and store the keys in these variables. This approach makes it easier to use the keys when encrypting and decrypting messages.

3 Algorithms Implementation and Time Complexity Discussion

3.1 Gcd method

3.1.1 Gcd Java Implementation

```
public static BigInteger gcd(BigInteger a, BigInteger b) {
       if (b.compareTo(BigInteger.ZERO) >= 0 || a.compareTo(b) >=
           0) {
           if (b.compareTo(BigInteger.ZERO) == 0) {
               return a;
           }
           return gcd(b, a.mod(b));
       }
       return new BigInteger("-1");
   }
```

3.1.2 Gcd Time Complexity Analysis

Greatest Common Divisor algorithm implemented above in Java is an implementation of the Euclid's Greatest Common Divisor algorithm. In this algorithm user inputs 2 numbers, $\boldsymbol{a}$ and $\boldsymbol{b}$ such that $a \geq b \geq 0$ and it gets as an output **gcd(a,b)**. This implementation starts by performing two comparisons, one that checks if $b \geq 0$ and one that checks if $a \geq b$ which are the conditions that the Euclid's Greatest Common Divisor algorithm must fulfill. Since we are talking about number algorithms, for a given input size $\boldsymbol{n}$, the time complexity of performing comparison operations is $O(n)$. The time complexity of performing division operation needed for the modulation

is $O(n^2)$. Thus at most for a call of this method the time complexity involves 2 comparison operations (with time complexity $O(n)$) and 1 division operation (with time complexity $O(n^2)$) which ultimately has the time complexity $O(n^2)$ after lower-order terms are dropped. We also know that each time a modulo operation is performed on a ***n-bit*** number, its value halves afterwards (which means that the length in bit size is reduced by 1), so assuming we have 2 ***n-bit*** numbers it will take ***2n*** method calls to reach the base case. To conclude, we perform ***2n*** method calls to reach the base case, each having the time complexity of $O(n^2)$ so ultimately the time complexity of the Greatest Common Divisor Algorithm is $T(n) = O(n^3)$.

3.2 Modexp method

3.2.1 Modexp Java Implementation

```
public static BigInteger modExp(BigInteger x, BigInteger y,
    BigInteger N) {
        if (y.compareTo(BigInteger.ZERO) == 0) return
            BigInteger.ONE;
            BigInteger z = modExp(x, y.divide(BigInteger.TWO), N);
        if (y.mod(BigInteger.TWO).equals(BigInteger.ZERO)) {
            return z.multiply(z).mod(N);
        } else {
            return z.multiply(z).multiply(x).mod(N);
        }
    }
```

3.2.2 Modexp Time Complexity Analysis

Modular exponentiation implementation shown below in Java is an algorithm that takes two three numbers as the input, a number $\boldsymbol{x}$ which is the base, a number $\boldsymbol{y}$ which is the exponent and a number $\boldsymbol{N}$ which is the modulo. The result of this method is x^y mod N. This algorithm has the base case where the exponent is 0 which will return 1. In this case we only have to perform a single comparison operation which as we discussed on the previous algorithm has a time complexity of $O(n)$. However, for each recursive call we will perform at most a comparison operation if $y = 0$ (which has a time complexity of $O(n)$), a division operation where we halve the value of $\boldsymbol{y}$ (which has a

time complexity of $O(1)$), a comparison operation if $\boldsymbol{y}$ is even or odd and 2 modular multiplication operations (which have a time complexity of $O(n^2)$). So the time complexity of each recursive call after the lower-order terms are dropped is going to be $O(n^2)$. In order to reach the base case, the algorithm will keep on halving its value after each recursive call, which as we mentioned in the previous algorithm is equal to reducing the bit size length by 1 every recursive call. Based on this, it will take the algorithm $\boldsymbol{n}$ method calls to reach the base case. As such we perform $\boldsymbol{n}$ method calls, each having the time complexity of $O(n^2)$ for a combined time complexity of $T(n) = O(n^3)$ for the modular exponentiation.

3.3 ExtendedEuclidean method

3.3.1 ExtendedEuclidean Java Implementation

```
public static BigInteger[] extendedEuclidean(BigInteger a,
    BigInteger b) {
        if(a.compareTo(b) == 1) {
            if (b.compareTo(BigInteger.ZERO) == 0) {
                BigInteger x = BigInteger.ONE;
                BigInteger y = BigInteger.ZERO;
                BigInteger d = a;
                BigInteger[] result = new BigInteger[]{x, y, d};
                return result;
            }
            BigInteger[] tempResult = extendedEuclidean(b, a.mod(b));
            BigInteger x = tempResult[1];
            BigInteger y =
                tempResult[0].subtract(a.divide(b).multiply(tempResult[1]));
            BigInteger d = tempResult[2];
            BigInteger[] result = new BigInteger[]{x, y, d};
            return result;
        }
        else
            return new BigInteger[]{BigInteger.ONE, BigInteger.ZERO,
                a};
    }
```

3.3.2 ExtendedEuclidean Time Complexity Analysis

Extended Euclidean GCD implementation in Java is an extension of the Euclid's Greatest Common Divisor mentioned above. This algorithm takes as an input two numbers $\boldsymbol{a}$ and $\boldsymbol{b}$ and returns a triplet of numbers ($\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{d}$) such that $ax + by = d$. The base case for this recursive method is when b = 0 which will return the triplet (1, 0, a). In this base case there is are 2 comparison operators which have a time complexity of $O(n)$, there are 2 assignment operations for $\boldsymbol{x}$ and $\boldsymbol{y}$ with time complexity of $O(1)$, 1 assignment operation for $\boldsymbol{d}$ with time complexity of $O(n)$, and 1 array initialization operation with time complexity $O(n)$. So time complexity for the base case it is at most $O(n)$. However, the algorithm for each recursive call will perform at most 1 comparison operation with time complexity $O(n)$, 3 assignment operations with time complexity $O(n)$, 1 subtraction with time complexity $O(n)$, 1 division and 1 multiplication both with time complexity $O(n^2)$ and an array initialization with time complexity $O(n)$. After dropping lower-order terms, the time complexity for a recursive call of this algorithm is $O(n^2)$. The number of recursive calls in our algorithm it is determined by the modulo operation performed. Like we mentioned previously, each time a modulo operation is performed on a ***n-bit*** number, its value it is halved afterwards (so its bit size length is reduced by 1), so assuming we have 2 ***n-bit*** numbers it will take ***2n*** method calls to reach the base case. Since we perform ***2n*** method calls, each with time complexity of $O(n^2)$ to reach the base case, that means that the time complexity of the algorithm is $T(n) = O(n^3)$.

3.4 modInverse method

3.4.1 modInverse Java Implementation

```
public static BigInteger modInverse(BigInteger a, BigInteger b){
        BigInteger[] result;
        if(a.compareTo(b) < 0) {
            result = extendedEuclidean(b, a);
        }
        else {
            result = extendedEuclidean(a, b);
        }
        if(result[2].compareTo(BigInteger.ONE) == 0) {
```

```
            return (result[1].add(a)).mod(a);
        }
        else {
            return BigInteger.valueOf(-1);
        }
    }
```

3.4.2 modInverse Time Complexity Analysis

Modular inverse implementation in Java it is performed base on the Extended Euclidean GCD. This algorithm takes as an input two numbers $\boldsymbol{a}$ and $\boldsymbol{b}$ and returns the modular inverse of the smallest number modulo the largest number.At most we need to perform an array initialization with time complexity $O(n)$, 2 comparison operations each with time complexity $O(n)$, 1 Extended Euclidean GCD method call with time complexity of $O(n^3)$ and an addition and a division operation with time complexities $O(n)$ and $O(n^2)$. Dropping lower-order terms we find that the time complexity for the Modular inverse algorithm is $T(n) = O(n^3)$.

3.5 isRelativePrime method

3.5.1 isRelativePrime Java Implementation

```
public static boolean isRelativePrime(BigInteger num1, BigInteger
    num2){
        BigInteger GCD = RSALibrary.gcd(num1, num2);
        if(GCD.equals(BigInteger.ONE))
            return true;
        else
            return false;
    }
```

3.5.2 isRelativePrime Time Complexity Analysis

The algorithm mentioned above checks if 2 numbers $\boldsymbol{a}$ and $\boldsymbol{b}$ are relatively prime (In order for two numbers $\boldsymbol{a}$ and $\boldsymbol{b}$ to be relatively prime $gcd(a, b) = 1$). The algorithm takes as an input two numbers $\boldsymbol{a}$ and $\boldsymbol{b}$ and outputs

either true or false based on whether they are relatively prime or not. At most, this algorithm performs 1 method call to the Greatest Common Divisor algorithm explained above which has a time complexity of $O(n^3)$ and an comparison operation with time complexity of $O(n)$. After lower-order terms are dropped, the time complexity of this algorithm is ultimately $T(n) = O(n^3)$.

3.6 isProbablePrime method

3.6.1 isProbablePrime Java Implementation

```
public static boolean isProbablePrime(BigInteger
    probablePrimeNumber) {
        boolean isComposite = false;
        for (int i = 0; i < 100; i++) {
            Random random = new Random(System.currentTimeMillis());
            BigInteger randomNumber = new BigInteger(64, random);
            BigInteger a =
                probablePrimeNumber.subtract(randomNumber).abs();
            BigInteger aModCheck = modExp(a,
                probablePrimeNumber.subtract(BigInteger.ONE),
                probablePrimeNumber);
            if (!aModCheck.equals(BigInteger.ONE)) {
                isComposite = true;
                break;
            }
        }
```

3.6.2 isProbablePrime Time Complexity Analysis

This algorithm it is used for primality testing in our RSA cryptosystem implementation. For primality testing in our algorithm we have used Fermat's Test. This implementation takes a ***n-bit*** number (which in our cryptosystem is a 64-bit number), and then it performs Fermat's Test 100 times. This assures that the probability that the inputted number is not a prime around:

$$\frac{1}{2^k}$$

where $k = 100$, which is very insignificant. During each of these k-times iterations of the loop, we first generate a random 64 bit number using BigInteger's constructor, which takes the bitLength and random seed as parameters. Then we substract the randomly generated number from the number we are testing if it is prime, and get the absolute value. This ensures that this number ***n-bit*** is a random number smaller than the number we are testing for primality, thus concluding the first step of Fermat's Primality Testing, which states that $1 \leq a \leq p$, where $\boldsymbol{a}$ is a number from $\{1, 2, 3, \ldots, p-1\}$ and $\boldsymbol{p}$ is the number we are checking if it is prime. The we perform modular exponetiation to perform Fermat's Test (a^{p-1} mod p) We extract the result and check whether it is equal to 1 or not. If the result is not 1, it means the number is not prime, and the algorithm is finished. If the result is 1, it means it passed the test, and then we repeat the same procedure with another randomly generated ***n-bit*** value for a total of 100 times. Inside each loop we perform 1 generation of a 64-bit number and 1 subtraction have a time complexity of $O(n)$, 1 absolute value operation which has a time complexity of $O(1)$, 1 modular exponentiation call from the method implemented above with time complexity of $O(n^3)$, 1 subtraction with time complexity of $O(n)$, 1 comparison with time complexity of $O(n)$ and 1 assignment with time complexity of $O(n)$. After all the lower-order terms are dropped, the time complexity for a single loop iteration is $O(n^3)$, and since the number of iterations $k = 100$, $T(n) = (100 * n^3) = O(n^3)$.

3.7 ProbablePrime method

3.7.1 ProbablePrime Java Implementation

```
public static BigInteger probablePrime(int bitLength, Random
    random) {
        while (true) {
            BigInteger probablePrimeNumber = new
                BigInteger(bitLength, random);
            if (isProbablePrime(probablePrimeNumber)) {
                return probablePrimeNumber;
            }
        }
    }
```

3.7.2 ProbablePrime Time Complexity Analysis

The method implemented above in Java is the method used for generating ***n-bit*** probable primes (in our cryptosystem we use this method to create 32-bit and 64-bit probable primes). This algorithm takes as the input the specified bitLength and a random seed as parameters and returns a probable prime number with this specified bit length.This method works by generating a random ***n-bit*** number and then it calls the method isProbablePrime to check whether the generated number is prime or not. Now we start analysing the time complexity of this algorithm. Firstly, this algorithm is a probabilistic one, as it is based on the random number generated. From the Lagrange Prime number Theorem, given a ***n-bit*** number:

$$\pi(n) = \frac{2^n - 1}{\ln 2^n}$$

represents the number of primes less than the number specified. The biggest number that can be randomly generated using ***n-bits*** is $2^n - 1$. So the probability of selecting a random prime is:

$$\frac{2^n - 1}{\ln 2^n} * \frac{1}{2^n - 1} = \frac{1}{n \ln 2} \approx \frac{1}{n}$$

Given this calculation we can say that the program will approximately generate a probable prime with ***n-bits*** after ***n*** trials. Given this information, the algorithm will perform at most ***n*** random ***n-bit*** number generations so the time complexity for this operation is $O(n)$. After a suitable probable prime is generated, it will call the isProbablePrime method which has the time complexity of $O(n^3)$. Combining this two operations we deduce that the time complexity of this method is at most $T(n) = (n * n^3) = O(n^4)$.

3.8 GenerateKeys method

3.8.1 GenerateKeys Java Implementation

```
public static BigInteger[] generateKeys() {

        BigInteger[] keysArray = new BigInteger[3];

        Random randomOne = new Random(System.currentTimeMillis());
```

```
        Random randomTwo = new Random(System.currentTimeMillis() *
            10);
        Random randomThree = new Random(System.currentTimeMillis()
            * 100);

        BigInteger p = RSALibrary.probablePrime(32, randomOne);
        BigInteger q = RSALibrary.probablePrime(32, randomTwo);

        BigInteger n = p.multiply(q);

        BigInteger pMinusOne = p.subtract(new BigInteger("1"));
        BigInteger qMinusOne = q.subtract(new BigInteger("1"));

        BigInteger modPQ = pMinusOne.multiply(qMinusOne);
        BigInteger e = null;

        boolean isERelativelyPrime = false;

        while(!isERelativelyPrime){
            e = RSALibrary.probablePrime(64, randomThree);
            if(e.compareTo(n) < 0)
                isERelativelyPrime = isRelativePrime(e, modPQ);
            else
                isERelativelyPrime = false;
        }

        BigInteger d = RSALibrary.modInverse(modPQ, e);

        keysArray[0] = n;
        keysArray[1] = e;
        keysArray[2] = d;

        return keysArray;
    }
```

3.8.2 GenerateKeys Time Complexity Analysis

This method is the implementation of the RSA cryptosystem for private and public key generation for each individual that wants to exchange mes-

sages. This method starts by creating two probable primes $\boldsymbol{p}$ and $\boldsymbol{q}$. From these primes we generate a number $\boldsymbol{N}$ which is the product of $\boldsymbol{p}$ and $\boldsymbol{q}$: $N = pq$. Next we subtract 1 from both $\boldsymbol{p}$ and $\boldsymbol{q}$ and multiply them together: $(p-1)(q-1)$. Next will generate a probable prime $\boldsymbol{e}$ which is relatively prime to the ***(p-1)(q-1)***. After we have $\boldsymbol{e}$, we can find its modular inverse $\boldsymbol{d}$ using ***(p-1)(q-1)*** as its modulo. After these numbers are generated user makes public its public key (N, e) and keeps secret its private key d.Now we start analysing the time complexity of the algorithm. Firstly we initialize the array which has a time complexity of $O(n)$, then we create 3 random instances which have time complexity of $O(1)$, then we randomly generate 2 probable prime numbers each with a time complexity of $O(n^4)$, then we perform a multiplication which has a time complexity of $O(n^2)$, followed by 2 subtractions which have a time complexity of $O(n)$ followed by a multiplication with time complexity of $O(n^2)$, then we have 2 assignments with time complexity $O(1)$, then we have the operation of checking until a randomly generated prime number e is relatively prime to $(p-1)(q-1)$ which has a time complexity of $O(n^4)$, followed by a modular inverse method call which has a time complexity of $O(n^3)$, and 3 assignment operations which have a time complexity of $O(n)$. After dropping the lower-order terms, we can deduce that the time complexity of this algorithm is $T(n) = O(n^4)$.

3.9 EncryptText method

3.9.1 EncryptText Java Implementation

```
public static List<BigInteger> encryptText(String text, BigInteger
    e, BigInteger n) {

        List<String> chunks = RSALibrary.chunkMessage(text, 7);
        List<BigInteger> plainAsciiChunks = new ArrayList<>();
        List<BigInteger> cypherAsciiChunks = new ArrayList<>();

        for (String chunk: chunks) {
            BigInteger chunkPlainAscii = new
                BigInteger(chunk.getBytes(StandardCharsets.US_ASCII));
            plainAsciiChunks.add(chunkPlainAscii);
        }
```

```
        for(BigInteger plainAsciiChunk: plainAsciiChunks){
            cypherAsciiChunks.add(RSALibrary.modExp(plainAsciiChunk,
                e, n));
        }
        return cypherAsciiChunks;
    }
```

3.9.2 EncryptText Time Complexity Analysis

The method implemented above shows the encryption of the user messages using RSA cryptosystem. This method gets as input the text message and numbers e and N (both which make the receiver's public key) and returns an arraylist with the Ascii representation of the message chunked in smaller messages. This method starts by chunking the message in chunks of strings with 7 characters and adding them to an arraylist. As the message gets longer the number of chunks in the arraylist increases linearly so the time complexity for this operation is $O(k)$ where k is the number of message chunks generated. Next we have the arraylist initialization which has a time complexity of $O(1)$. Next we have a for-loop which will iterate $k - times$ (as we said k is the number of chunks of message). Inside this for loop, the operations of creating a biginteger from the string chunk's Ascii representation and the process of adding this number into arraylist are constant operations so their time complexity is $O(1)$, so the entire loop time complexity is $O(k)$. Finally we have another loop encrypts the Ascii representations of the message chunks. Just like the previous loop, this for loop will iterate $k - times$ (k is the number of chunks of messages will are part of the arraylist). Inside this for loop we have a addition operation which has a time complexity of $O(n)$ and a modular exponentiation which has a time complexity of $O(n^3)$. So this for loop has a total time complexity of $O(kn^3)$. After dropping the lower-order terms, the time complexity for this algorithm is $T(n) = O(kn^3)$, which shows that the number of modular exponentiations performed to encrypt the message will be multiplied by the number of message chunks k.

3.10 DecryptText method

3.10.1 DecryptText Java Implementation

```
public static List<String> decryptText(List<BigInteger>
    cypherText, BigInteger d, BigInteger n) {

        List<String> plainTextChunks = new ArrayList<>();

        for (BigInteger cypherAsciiChunk: cypherText) {
            BigInteger plainAsciiChunk =
                RSALibrary.modExp(cypherAsciiChunk, d, n);
            plainTextChunks.add(new
                String(plainAsciiChunk.toByteArray()));
        }

        return plainTextChunks;
    }
```

3.10.2 DecryptText Time Complexity Analysis

The method implemented above shows the decryption of the user messages using RSA cryptosystem. This method takes the arraylist of the encrypted numbers and the numbers d and N (d is the private key and the N is used alongside to decrypt the message). Firstly, we initialize an arraylist which has the time complexity of $O(1)$. Then we denote with k the number of elements of the arraylist. This means that the loop will iterate $k - times$. Inside every single for loop iteration, a modular exponentiation operation is performed which has the time complexity of $O(n^3)$, and the conversion of the numeric representation and the insertion of this string into the arraylist which both have time complexity of $O(1)$. After dropping the lower-order terms, the time complexity of this decryption algorithm implementation is $T(n) = O(kn^3)$ which shows that the number of modular exponentiations performed to decrypt the message will be multiplied by the number of message chunks k.

4 Hacker Simulation

4.1 NextProbablePrime method

4.1.1 NextProbablePrime Java Implementation

```
public static BigInteger nextProbablePrime(BigInteger num){
       BigInteger temp = num.add(BigInteger.ONE);
       if(!isProbablePrime(num))
           num = nextProbablePrime(temp.add(BigInteger.ONE));
       return num;
   }
```

4.1.2 NextProbablePrime Time Complexity Analysis

The method implemented above shows the algorithm for finding the next probable prime. This method takes as an input a number. Then it performs an addition with 1 which has a time complexity of $O(n)$. Then it performs a check if the number is a prime. This has a time complexity of $O(n^3)$ as discussed earlier. Inside this check we have a recursive call to the method itself. This can be repeated at most $n - times$. As a result the time complexity for generating the next probable prime is $T(n) = O(n^4)$

4.2 FactorizeN method

4.2.1 FactorizeN Java Implementation

```
public static List<BigInteger> factorizeN(BigInteger N){
       BigInteger startPrime = BigInteger.TWO;
       List<BigInteger> factors = new ArrayList<>();
       while(startPrime.compareTo(N.sqrt()) < 0){
           if(N.mod(startPrime).equals(BigInteger.ZERO)) {
               BigInteger p = startPrime;
               BigInteger q = N.divide(p);
               factors.add(p);
               factors.add(q);
           }
           startPrime =
               nextProbablePrime(startPrime.add(BigInteger.ONE));
       }
       return factors;
   }
```

4.2.2 FactorizeN Time Complexity Analysis

The method shown above it is implemented to simualate a hacker attack. This method takes the number N that we want to find the factors that it contains and it starts checking. Firstly, the algorithm performs an initialization operation with time complexity of $O(1)$, then it initializes an empty array which also has the time complexity of $O(1)$. Then it starts comparing numbers up until the $\sqrt{N}$, because that is the biggest factor that the number can contain. Inside the while loop we check if the prime generated is a factor of N (starting from 2 then we find the next probable prime). If the number is divisible we perform a division which has the time complexity of $O(n^2)$ and two insertions into array with time complexity of $O(1)$. At the end of the loop we also generate the next probable prime which has a time complexity of $O(n^2)$. From this information we can deduce that the time complexity of this algorithm is:

$$T(n) = \frac{2^{n/2}}{\ln 2^{n/2}} * O(n^4) \approx 2^{n/2} * O(n^3)$$

4.2.3 Attack Simulation Analysis

From the calculation shown above we saw that the time complexity to perform a factorization is

$$T(n) = 2^{n/2} * O(n^3)$$

. Performing the calculations, to break a 64-bit N number we simply use $n = 64$. So to break the N we need to perform

$$2^{64/2} * O(64^3) = 2^{32} * 2^{18} = 2^{50}$$

operations. Assuming we have a computer which can perfrom 2^{32} calculations per second (which are the norm today), then the time needed to break a 64-bit N number using factorization implemented above is

$$\frac{2^{50}}{2^{32}} = 2^{18} \approx 3days$$

5 Summary

To conclude, RSA is a reliable system to send messages between two communicating parties in a secure way, thanks to the factorization problem which is

almost impossible to break. To implement our own version of RSA, we have created algorithms from scratch such as gcd, modexp, extendedEuclidean, modInverse, isRelativePrime and more. These algorithms are used to generate secure private and public keys in a timely manner. These keys are used to encrypt and decrypt the messages between the communicating parties. The most important aspect of these keys is that they are very secure, and the larger they are in size, the harder it will be for hackers to crack them. This was shown by the hacking simulation above, that even if a hacker has access to the ciphertext and knows the encryption scheme, it will take a long time to break down a 64 bit-key RSA protocol, let alone key sizes that are typically used, which range between 2,048 to 4,096 bits.