Final Year Project Report

Full Unit Project

Prime Numbers and Cryptosystems

Abhishek Nand Kumar

A report submitted in part fulfillment of the degree of

BSc (Hons) in Computer Science

Supervisor: Zhiyuan Luo

Department of Computer Science
Royal Holloway, University of London

April 03, 2019

Declaration

This report has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 15000

Student Name: Abhishek Nand Kumar

Date of Submission: 02/04/19

Signature: Abhishek Nand Kumar

Abstract

The project aims to implement a library in Java to represent large numbers, perform mathematical operations on them and generate large primes, then using that library to build a cryptographic messaging application that can ensure confidentiality on an insecure network. For cryptography to be efficient, it must work at a reasonable speed, this can be difficult because some cryptographic protocols rely on the computation of large numbers, which can be time-consuming. The problem with most programming languages is that they cannot represent numbers that are large enough for cryptographic use. To solve this problem, the large numbers need to be represented as a data structure in Java and then must be easily manipulated when used with different cryptosystems. The library created for this project is called BigMath", it was used to implement RSA as a digital signature and Diffie-Hellman as a key exchange protocol to generate a DES key. To guarantee that the library is efficient, multiple algorithms were implemented and unit tested to guarantee they work. Speed tests were performed to ensure that the library is reliable and speedy. This is important because there cannot be any miscalculations, especially in a messaging environment, or else everything will fail.

Table of Contents

Chapter 1 Project Specification

1.1 Aims and Objectives

The project aims to create a cryptographic messaging application using RSA and a symmetric key system in order to ensure confidentiality and authentication online. To do so, algorithms will be implemented in Java that can compute large numbers efficiently and be able to generate large primes for RSA.

The project would need to represent large numbers as a data structure. The problem with Java is that it can only represent 32-bit numbers, this is a problem because RSA would need at least 398-bit numbers in order to be secure. In order to fix this problem, a system would need to be created that can handle large numbers and be able perform mathematical operations on them. A data structure would need to be created to represent large numbers. Research needs to be conducted to find the most efficient ways to represent data in order to efficiently manipulate numbers in a data structure. The best option would be to construct a system that has the smallest time complexity or the most efficient method to access the data. The data structure needs to be manipulated so that mathematical operations can be performed on them which is needed for the cryptographic protocols.

The project needs to generate large primes. In order to generate keys for RSA, prime numbers need to be generated first; it is simple to determine whether small numbers are prime, but it gets more complicated when the primality test is done on larger numbers. An algorithm within the large number library needs to be implemented in order to efficiently generates primes. After developing an algorithm, the primality test needs to be verified and measured to see how accurate it is. Since speed is important, the algorithm needs to generate a large prime number within a reasonable amount of time.

Large messages would need to be encrypted using RSA. Research will be conducted to see if splitting the message and then encrypting with RSA is more efficient than using a symmetric key encryption. However, RSA on its own can be inefficient, this is because it requires calculation of large numbers. In order to overcome this problem, a symmetric key system needs to be implemented to be used alongside RSA. Doing so should save a lot of time in the encryption process. Data Encryption Standard, or DES, might be considered as a symmetric key system.

Since a symmetric key system will be used, the project would need a key exchange protocol. The problem with using a symmetric key system is communicating the key over an insecure network. Diffie-Hellman will need to be implemented into the messaging service in order to establish shared keys. Diffie-Hellman is somewhat like RSA where it relies on large primes for secrecy. Pre-written algrotihms from RSA should work within the Diffie-Hellman process to generate a shared secret key.

The messaging service should be able to authenticate users. The problem with online communication is assuring that someone is who they say they are. Digital signatures are a way of authenticating users so that you can't pretend to be someone else. RSA can be used as an authentication tool when communicating online, RSA keys would need to be communicated during the handshake protocols so that users would be able to authenticate each other when a message is sent. The signature will have to be attached to the DES encrypted message and the authentication check would be done by the receiver.

1.2 Future Career

This project was chosen to help deepen understanding of cryptography and information security. The work in this project will allow for a hands-on approach to cryptography rather than just learning the theory of it or just using pre-existing libraries that does everything for you. I'm hoping to get a job in cyber security or information security, and I think that this project will help me learn and build my skills to be more effective in a work environment. I feel that this project would help me learn to work more efficiently in the future, how to figure out problems and solve them all within a given time frame. Most of all I think that this project will help me schedule and use my time efficiently in a professional environment. The project has also made me understand the ethical responsibilities of a programmer.

1.3 Summary of Completed Work

- A data structure was implemented called "BigMath", it deals with large numbers, it can do addition, subtraction, multiplication, division, modulo, shift left, shift right, comparing two big numbers, exponentiation by squaring and modular exponentiation.

- The program can generate large primes using BigMath.

- Speed tests were conducted on all algorithms.

- Implementing a system that generates RSA public keys using previously generated prime numbers.

- Using the modular exponentiation function, the program can encrypt and decrypt messages in RSA.

- The program can pad messages to add extra security to the RSA encryption.

- Implemented DES as a symmetric key system.

- Using BigMath, Diffie-Hellman was implemented as a key exchange protocol.

- Using BigMath, RSA was implemented as a digital signature.

- A server was constructed to relay data between two clients.

- A chat client was implemented to send signed-encrypted messages to another client through the server.

- The client can use three different versions of RSA, RSA-240, RSA-1024 or RSA-2048.

- The client allows gives the option to re-generate RSA keys.

- The client gives the option to perform Diffie-Hellman again to generate a new DES key.

1.4 What to Expect Ahead

In the next chapter, the theory of the project will be covered. The theory section is used to discuss the existing theory and research as context for the project, specifically on RSA, DES, Diffie-Hellman, digital signatures, and representing large numbers. Following the theory chapter is the methodology chapter, which will discuss the methods that were used in order to complete the project by applying knowledge from the theory section. The self-evaluation chapter is used to discuss the results of the project and how successful the project was. The professional issues chapter will cover ethical aspects of the project. The program manual, the bibliography and the appendix are shown at the end.

Chapter 2 Background Theory

This chapter is used to discuss the existing and established theory to provide context for the project. Asymmetric cryptography, symmetric cryptography, digital signatures, performing mathematical operations on large number data structures, key establishment protocols and Java sockets will be covered.

2.1 Public Key Cryptography: RSA

Rivest, Shamir, Adleman, or RSA, was created in 1978. Up until the 70s, cryptographic applications were primarily built around symmetric systems. Symmetric key systems can be insecure, mainly on how to communicate the shared secret key over an insecure network. Public key systems such as RSA fixes this problem.

RSA is a cryptosystem used in modern day computers; it is used to generate keys which are used to encrypt and decrypt data; it creates a sense of confidentiality in communication.

RSA is asymmetric, which means that there are two keys that every user receives, a public key, that everyone can see, to encrypt messages and a corresponding private key, that is hidden, to decrypt the encrypted message. A user would normally publicly publish their public key which anyone can use to encrypt messages, that can only be decrypted using the original private key. RSA is hard to break because it is tough to factorise, meaning that it is hard to find the prime number factors of a large composite number.

RSA is also used as a digital signature which will be discussed more in the digital signature section.

2.1.1 Generating large primes

Prime numbers are important in RSA; a number is prime if and only if it can be divided by itself and one, without having any remainders, for example, 13 can only be divided by itself and by one while 15 cannot be a prime because it can be divided by 5 and by 3.

The first part of generating the public key is picking two large primes. The problem with this is how to check if a number is prime without doing exhaustive calculations. The most common way to test this is by doing a probabilistic test for primality on a number. One of the most straightforward primality tests is Fermat's probabilistic primality test.

In a Fermat test, a number p is prime if $a^{(p-1)} \equiv 1 \pmod{p}$ where a is a random number between a and $(p-1)$. p would be tested against multiple a values using the equation. It works by applying multiple Fermat's tests on p with random values of a, however, a cannot be divisible by p. This is calculated using modular exponentiation. [3]

Modular exponentiation is exponentiation done over a modulus. It calculates the remainder when an integer a^e is divided by a positive integer m. $c=a^e \pmod{m}$. For example, $3^5 \pmod{7} = 5$. [4]

Using modular exponentiation, the project can test for primality using Fermat's test. For Fermat's test to be secure, at least 10 tests must be done with multiple random values of a to check if a number is a probable prime. Fermat's test was chosen because probabilistic tests are much faster than deterministic primality tests ($O(k \times \log 2n \times \log\log n \times \log\log n)$ compared to deterministic tests which operate in polynomial time.

2.1.2 Generating RSA Keys

RSA needs two keys, a public key, to decrypt the data and a private key, used for decrypting data. However, data encrypted with a public key can only be decrypted using its corresponding private key.

Public key generation

To generate the RSA key-pair, first, pick two random prime numbers (p, q).

Then multiply p and q to get the product N ($N = p*q$). N is crucial because it's used in the encryption lock and the decryption lock which is needed to encrypt and decrypt messages.

Then Euler's totient "phi" needs to be found, which is the number of positive integers up to a given number 'a' that are relatively prime, or coprime, with '*a*' and that are also smaller than *'a'*. Euler's totient is calculated by subtracting one from p and q and multiplying them together:

$$phi = (p - 1)(q - 1)$$

Now the public key '*e'* needs to be picked. *e* has two conditions; it must be between one and *phi*. Moreover, *e* must be coprime with *N*, which is the product of the two initial primes, and with *phi*, this means that *e* must share no factors with *N* and *phi* other than 1. This can be tested using Euclid's algorithm to determine if two numbers are coprime. It checks that the greatest common divisor of two positive integers is equal to 1. As mentioned earlier, a number is prime if it can only be divided by itself and 1. In this case, the greatest common divisor between *e* and *phi* and *N* can only be 1.

The greatest common divisor algorithm is implemented recursively:

gcd(a, b)

if a = 0: then the gcd is b

else: return gcd(b (mod a) , a)

For example, the greatest common divisor between 5 and 7:

gcd(5, 7)

return gcd(7 (mod 5), 5)

return gcd(5 (mod 2), 2)

return gcd(2 (mod 1), 1)

return 1

After running Euclid's test to find the numbers that are coprime with *phi* and *N* and that are smaller than *N*, any number is picked that satisfies the conditions. The encryption lock has been obtained and consists of *e* and *N*. It is represented as (*e*, *N*).

Private key generation

Modular multiplicative inverse:

Before finding the decryption key *d*, *d* must be computed from (*e*, *N*) using multiplicative Inverse or reciprocal. A multiplicative inverse for a number *x*, commonly represented as $\frac{1}{x}$ *or* x^{-1}, x is a number when multiplied by x is equal to 1. If two numbers have a greatest common divisor of 1, then the smaller integer has a multiplicative inverse in the modulo of the larger number. It can be expressed as x^{-1} element of Zp gcd(x, p) = 1. [1]

Using the encryption lock, the private key can be computed. *d* is calculated using the equation $d = \frac{1 + phi}{e}$. This is because $d * e = 1 \ (mod\ phi)$.

Since the encryption key has a gcd of 1 with φ(n), the modular multiplicative inverse of the public key can be determined using the Extended Euclidean Algorithm. When two numbers, *a* and *m* are coprime, it finds the gcd of a and b and finds x and y such that:

$ax + by = gcd(a, b)$

To find the multiplicative inverse of *a* under *m*, set $b = m$ (since *a* and *m* are relatively prime):

$ax + my = 1$

Add modulo *m* to both sides:

$ax \equiv 1 \ (mod\ m)$

'*x*' is the multiplicative inverse of '*a*'. [5]

Using the modular multiplicative inverse, *d* can be computed from the encryption key and *phi*. After calculating *d*, the decryption lock has been obtained and consists of *d* and *N*. It is normally represented as (*d*, *N*). The decryption lock must be kept a secret.

2.1.3 Encrypting Plaintext

Using the encryption lock that was generated, the messages can be encrypted, this is done by taking a plaintext message and converting it to a number. ASCII messages are converted to its equivalent as an integer, using ASCII character to decimal conversion. Once the plaintext is represented as a number, the ciphertext can be computed using the following formula:

$c = m^e \ (mod\ N)$

Where *m* is the message as a number, *e* is the encryption key, *N* is the product of the two initial prime numbers that were generated, and *c* is the ciphertext which is unreadable by a computer or human. *e* and *N* are found within the encryption lock. To calculate this, the modular exponential function is used.

In a normal scenario, with a sender and a receiver, the sender will take the message, and after encrypting the message with the receiver's public key, the sender will have computed the ciphertext. The sender then sends the ciphertext to the receiver, and only the receiver can decrypt the ciphertext with their private key.

2.1.4 Decrypting Ciphertext

Using the decryption lock that was generated from the encryption key, the ciphertext can be decrypted using the modular exponentiation function which was discussed in the "generating large primes" section. The plaintext can be computed from the ciphertext by using the following formula:

$$m = c^d (mod\ N)$$

Where *c* is the ciphertext, *d* is the decryption key, *N* is the product of the two initial prime numbers and *m* is the plaintext message represented as a number. The decryption key and *N* are kept within the decryption lock.

2.1.5 Factorisation

The security of RSA heavily relies on the keys being hard to factorise, which means that it's hard to find the two initial prime numbers that were used in RSA to generate the key. Essentially the size of the prime number is important in determining how secure the RSA system will be, larger prime numbers will make RSA harder to break. The largest key-size that has been factorised so far is 768-bit RSA keys [7]. The most used key size is 1024-bit which has not been factorised yet. [7]. RSA-1024 is predicted to be factorised in the next five to ten years. [7] and RSA-2048, which uses 2048-bit keys, will not be broken in a very long time. The smallest version of RSA that has not been factorised yet is RSA-240, which uses 398-bit keys.

2.1.6 Real-World Example

Using the theory of RSA, a real-world scenario can be illustrated to help deepen understanding of RSA. Two users, Alice and Bob, want to send a secret message. First, they would both have to generate their key pair, which consists of a public key and a private key. Alice and Bob would then publish their public keys so that everyone can see them.

If Alice wanted to send a message to Bob, she would have to take Bob's public key and encrypt her message with Bob's public key; she then sends the ciphertext to Bob. This means that only Bob can decrypt the ciphertext because his private key is the only key that can decrypt the ciphertext. Once Bob receives the ciphertext, he can decrypt the message using his key.

This system shows that every user would have to generate a key pair and they would have to keep their decryption key a secret.

2.1.7 Padding RSA

Raw RSA can easily be broken; traffic analysis could reveal information about the ciphertext. Someone could tell when the message changes. For Example, if RSA was used to send secret information in a war zone, the message encrypt("stay put") will be sent continuously and it will always show the same ciphertext, but as soon as they send out encrypt("attack"), the adversary will see that the ciphertext has changed and they'll know something is going on. [6]

Padding can add another level of randomness or security to RSA. Padding is done by adding a bunch of random values to the beginning of the plaintext before encrypting it. This will guarantee that the ciphertext will always be different and it'll never reveal information. After the message is decrypted, there must be a system that removes the padding.

2.2 Digital Signatures

So far, this chapter has only dealt with the idea of private communication using RSA. However, the problem of authentication arises. A digital signature is a scheme to verify the authenticity of digital data. Digital signatures use an asymmetric cryptography system to authenticate users. Digital signature schemes allow a user to "sign" a message in a way that another user can verify that the message comes from the initial sender. This is because the second person would already possess the sender's public key and that it was established by the sender. This step is important because the receiver can guarantee that the message came from the sender and it wasn't modified in any way. [8].

"A signature scheme is a tuple of three probabilistic polynomial-time algorithms (Gen, Sign, Verify)" [8]. First, there's a key generation algorithm that generates a key pair, a public key and a private key. Then the signing algorithm starts by taking the private key and a challenge string and computing a signature where $signature \leftarrow Sign^{private\ key}(challenge)$. Finally, the deterministic verification takes the public key, the initially challenge and the signature generated in the previous step and verifies if the decryption of the signature, using the public key, is equal to the original challenge. If the Verify step doesn't equal the original challenge, then the signature is fraudulent [8].

For the signature to pass, it's required that:

$$Verify^{public\ key}\left(challenge,\quad Sign^{private\ key}(challenge)\right) = true$$

RSA can be implemented as a digital signature to confirm that a message originated from a sender. This is important because anyone could take someone else's public key and pretend to be that person. Signatures add authentication in real-world applications.

This works because, using RSA, the private key can be used to encrypt messages too, this is possible because of *e*d = 1 (mod phi)*. A user would send (*m, signature*) where *signature = m^d (mod n).* First RSA key pairs would need to be generated and then distributed, from there RSA can be used as a digital signature [8].

For example, If Bob wanted to authenticate himself with Alice, he would send Alice a challenge *m* and his signature encrypted using his private key using the formula: $signature = m^d \ (mod\ n)$, where *m* is the challenge, *d* is the private key and *n* is phi. Once Alice receives the signature, she can simply take Bob's public key and decrypt the signature using the formula:
$m = signature^e \ (mod\ n)$, where *m* is the challenge, *e* is the public key and *n* is phi. Alice simply must check that the challenge matches with the decrypted signature. If it is, then Bob is authenticated, if not then it is a malicious user that tried to forge a signature.

2.3 Representing large numbers

RSA is a cryptosystem that relies on prime numbers that are not easy to factorise, which means that the prime numbers need to be extremely large. The problem with Java is that it cannot represent numbers that are large enough for RSA to be viable. Specifically, Java can only represent x where $-2147483648 \leq x \leq 2147483647$. Numbers would need to be represented as a data structure.

There are multiple types of data structures that can be used to represent large numbers, but the main focus is how fast will it take to access data from the data structure, since sorting and searching isn't useful in a large number representation, I'll just be focusing on the most efficient ways to store and access data.

Thoughout this porject, Big-O notation will be used to measure the time complexities of different data structures. Big O notation is used to classify algorithms based on the running times or space requirements. Rob Bell [10] says that O(1) describes an algorithm that will always execute in the same time regardless of what it is computing. Rob Bell [10] also states that O(n) describes an algorithm that will grow linearly, for example, a for loop that iterates over an array has a running time of O(n) where n is the size of the array.

Potential data structures:

Arrays: The access time complexity is O(1) and the insertion time complexity is O(n). [9]

Stack: The time complexity to access data is O(n) and the time complexity to insert data is O(1). [9]

Queue: The time complexity to access data is O(n) and the time complexity to insert data is O(1). [9]

List: The time complexity to access data is O(n) and the time complexity to insert data is O(1). [9]

Hash Table: Since data cannot be accessed in a hash table, it is best not to use this data structure to represent large numbers. [9]

Trees: The time complexity to access data and to insert data is O(log(n)). [9]

A base and endianness will have to be chosen in order to represent numbers. The selection of a data structure is important because as the number gets larger, it will take more time to compute the answer of a mathematical operation on the data structure.

BigInteger is a pre-existing library for java that simulates large numbers by storing them into an array and is represented as two's complement notation. Not only can BigInteger represent large numbers, but it can also perform primitive integer operations on multiple large numbers and modular arithmetic. BigInteger also provides a lot of prime number computations like multiple primality testing and prime generation [11].

2.3.1 Knuth's Algorithm D

Knuth's algorithm D is a long-division algorithm used to efficiently divide two numbers in an array which was analysed and proven by Prof. Donald Knuth. in "The Art of Computer Programming" [12], it provides a long-division algorithm for non-negative numbers in an array. The algorithm first turns the numbers into an array, "123" would become the array [1, 2, 3]. Algorithm D is then split into eight steps:

D1:

$d \leftarrow \big(b\ (v1 + 1)\big)$

$u * d$

$v * d$

$if\ d == 1\ then\ u0 = 0$

Where b is the base, v is the divisor, and u is the dividend.

D2: $j = 0;\ loop\ through\ D2\ to\ D7$

D3:

$if\ uj$ == v1 then $q = b - 1$ where q is the quotient being calculated.

$else\ q = \dfrac{(uj * b + uj + 1)}{v1}$

D4: $uj = uj - (q * v)$

D5: $quotient_j = q$

If q is negative, then go to D6

D6: $quotient_j = quotient_j - 1$

$u * v$

D7: *increment j*

If j is smaller or equal to the length of u then return to D3

D8: the quotient array is now fully calculated which shows the quotient answer and to get the remainder: $u * d$.

2.4 Theory of Key Establishment

2.4.1 Diffie-Hellman

Diffie-Hellman is a key exchange protocol. It is the primary key exchange protocol in SSH, IPsec and TLS. Traditionally, symmetric key cryptography had a significant flaw where encrypted communications between two parties relied on insecure key exchange procedures such as transporting the key physically by courier. Diffie-Hellman allows two parties to agree on a shared symmetric key without having to send the shared key through an insecure channel or without having to meet up in person to establish a key physically. [13]

For Diffie-Hellman to work, firstly, both parties need to agree on a public modulus *p*, where *p* is prime, and a number *g*, where *g* is a primitive root modulo of *p*. In modular arithmetic, a number *g* is a primitive root modulo of *p* if there exists an integer *k* such that $g^k \equiv a(mod\ p)$, where *a* is coprime to *p* and is congruent to *k*. Congruency is a system where numbers wrap around after reaching a certain value, similar to a clock.

Example of primitive root:

$$g^k \equiv a(mod\ p)$$

$$3^1 \equiv 3(mod\ 7)$$

$$3^2 \equiv 2(mod\ 7)$$

$$3^3 \equiv 6(mod\ 7)$$

$$3^4 \equiv 4(mod\ 7)$$

$$3^5 \equiv 5(mod\ 7)$$

$$3^6 \equiv 1(mod\ 7)$$

$$3^7 \equiv 3(mod\ 7)$$

3 is a primitive root of modulo 7. *a* is congruent to *k* and *a* wraps around and starts repeating the numbers 3, 2, 6, 4, 5, 1, and 3.

There are complications with finding primitive roots since there is no simple algorithm or simple method to compute primitive roots. Since it is hugely computationally demanding to find the primitive root of a large number, it might be a viable option to use a pre-existing, precomputed primes and their primitive roots to use for Diffie-Hellman. Reusing primes will save a lot of time rather than computing primitive roots of large primes. There are precomputed primitive roots of large primes available online [14]. There is no loss in security when prime numbers are reused [15] but it's important that the reused primes are large numbers, specifically 1024bits or 2048bits, 1024bits recently has been discovered to fail, there are also rumours that the government relies on 1024-bit primes to break Diffie-Hellman, so it's probably much safer and more secure to use 2048bits [16]. In most cases, *g* is often a small number, most commonly 2, there is no risk of security when *g* is small.

Once the prime number *p* and primitive root *g* are chosen, both parties start the next step of the Diffie-Hellman key establishment protocol. Two parties, Alice and Bob, both generate a secret random number. Alice's random number will be *a*, and Bob's random number will be *b*. Then Alice will compute $A = g^a\ (mod\ p)$ and Bob will compute $B = g^b(mod\ p)$. Alice and Bob will share their computed values, so, Alice will send *A* to Bob and Bob will send *B* to Alice. Once Alice has *B*, she computes $key = B^a(mod\ p)$ and Bob will compute $key = A^b(mod\ p)$. Now both parties have computed the shared key. This works because $A^b(mod\ p) = g^{ab}(mod\ p) = g^{ba}(mod\ p) =$

$B^a (mod\ p)$. Only the values *a* and *b* are kept secret to produce the secret key, all other values are shared in the open. An eavesdropper would only be able to see *p*, *g A* and *B*.

2.4.2 Diffie-Hellman hand worked example:

1) Alice and Bob both agree on g = 5 and p = 23.
2) Alice chooses a = 4 as her secret number.
3) Bob chooses b = 3 as his secret number.
4) Alice computes $5^4 (mod\ 23) = 4$ and sends 4 to Bob.
5) Bob computes $5^3 (mod\ 23) = 10$ and sends 10 to Alice.
6) Alice then computes $10^4 (mod\ 23) = 18$.
7) Bob then computes $4^3 (mod\ 23) = 18$.
8) Both parties now have the number 18 as their shared secret key.

An eavesdropper would only be able to see the number 23, which is the initial prime, the number 5, which is the primitive root of the modulus 23, the number 4, which is the computed value by Alice and the number 10, which is the computed value by Bob. Throughout the whole procedure, the secret numbers 4 and 3 are kept secret.

2.5 Symmetric Key Cryptography

Symmetric key encryption relies on encryption and decryption with a shared secret key. Asymmetric systems, like RSA, are slower than symmetric key systems because RSA requires more CPU cycles to encrypt something. When it comes to speed, symmetric key systems are better [17].

2.5.1 Data Encryption Standard

Data encryption standard, or DES, is a symmetric key system that relies on a shared key to encrypt and decrypt messages. DES is a block cipher; a block cipher is an algorithm that encrypts a fixed number of bits in a group, called blocks. A block cipher will encrypt and decrypt a block into ciphertext or plaintext using a symmetric key. The block size of DES is 64 bits, and the key will be 64 bits long, but only 54 bits are used from the key, 8 bits are used for checking parity and then are not used again. [18]

DES Keys

First, DES will need a 64-bit key which can be generated using Diffie-Hellman, as shown in the previous section. Once a key is established, the first step of DES is to produce 16 subkeys which must be 48-bit long. This is done by permuting the 64-bit key using the permutation table PC-1, which is shown below: [18]

PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Using this table, the original 64-bit key is permuted to produce a 56-bit key called K+. The table PC-1 is essentially K+, but each value in PC-1 is a link to a bit in the original 64-bit key. For example, the first bit of K+ will be the 57^{th} bit from the original key, the second bit of K+ will be the 49^{th} bit from the original key, the third bit of K+ will be the 41^{st} bit from the original key and so on. The last bit of K+ will be the 4^{th} bit of the original key.

Then K+ is split into two halves called C_0 and D_0. Using C_0 and D_0, 16 blocks of C_n and D_n are created, where $1 \leq n \leq 16$. To form C_n and D_n, C_{n-1} and D_{n-1} are permuted. To form these blocks, the following table is used: [18]

Iteration number	Number of left shifts
1	1
2	1

3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

The table shows the amount of left shift that are needed to get to the next block of C_n and D_n. For example, C_1 and D_1 are calculated by applying one left shift to D_0 and C_0. C_{13} and D_{13} are calculated by applying two left shifts on the previous blocks, C_{12} and D_{12}. Once all values of C_n and D_n are computed, the subkeys must be computed from them. This is done by following a permutation table called PC-2 which creates a 48-bit subkey called K_n, where $1 \leq n \leq 16$. First, C_n and D_n are combined to form C_nD_n which is 56-bits long. After applying PC-2 , a subkey is created that is 48-bit long. The following table shows PC-2: [18]

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

As similaraly demonstrated for PC-1, the permutation table is a representation of K_n where the numbers in the PC-2 table links to a bit in C_nD_n. For example, the first bit of K_1 will be the 14^{th} bit of C_1D_1, the second bit of K_1 will be the 17^{th} bit of C_1D_1 and the last bit of K_1 will be the 32^{nd} bit of C_1D_1. This step is repeated for every K_n and C_nD_n creating the sixteen 48-bit subkeys needed for DES.

DES Encryption

Using the 16 subkeys, 64-bit blocks of data can be encrypted. First, the 64-bit message must be permuted to be encrypted; this is done by using an initial permutation table. The IP table is shown below: [18]

IP Table

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Using the initial permutation table, IP can be computed. IP is essentially a representation of the IP table where the values inside the table link to a bit-position in the plaintext message. For example, the first bit of IP will be the 58^{th} bit of the plaintext message, the second bit of IP will be the 50^{th} bit of the plaintext message and the last bit of IP will be the 7^{th} bit of the plaintext message.

Then IP is split into two halves, L_0 and R_0. And the following step is repeated 16 times: [18]

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$

Using these equations, L_n and R_n can be generated, where $1 \leq n \leq 16$. The first equation just sets L_n to R_{n-1} and R_n is calculated by XORing L_{n-1} with the function f of R_{n-1} and the subkey K_n.

To calculate the function f, R_{n-1} needs to be expanded from a 32-bit block to a 48-bit block using an E bit-selection table shown below: [18]

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Let $E(R_{n-1})$ be the 48-bit expanded block. The table is a representation of $E(R_{n-1})$, and the values inside the selection table represent the position of a bit in R_{n-1}. For example, the 1^{st} bit of $E(R_{n-1})$ will be the 32^{nd} bit of R_{n-1}, the second bit of $E(R_{n-1})$ will be the 1^{st} bit of R_{n-1} and the last bit of $E(R_{n-1})$ will be the

1st bit of R_{n-1}. Essentially the table extends the 32-bit block into a 48-bit block by reusing certain bits in R_{n-1} to expand the block.

Then in the function f, $K_n \oplus E(R_{n-1})$ is calculated to form another 48-bit block. The next step is to use the previously calculated 48-bit as addresses in tables called S-boxes. This is done by splitting the 48-bit block, into eight boxes called B that are 6-bit each: [18]

$$K_n \oplus E(R_{n-1}) = B_1B_2B_3B_4B_5B_6B_7B_8$$

Then calculate:

$$S_1(B_1)\ S_2(B_2)\ S_3(B_3)\ S_4(B_4)\ S_5(B_5)\ S_6(B_6)\ S_7(B_7)\ S_8(B_8)$$

Where S is a function that refers to the output of a S_i block, where $1 \leq i \leq 8$. S_i tables are used to obtain the output of the function S, the first table is shown below but there are eight other tables for each value of i in S_i: [18]

S1

Column

Row:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

The table transforms the 6-bit box into a 4-bit block as an output, it does this by taking the first and last bits from the 6-bit box which represents a base 2 number, then this base 2 number is converted to decimal, or base 10, and it represents the row in the S1 table. The 4 bits in the middle of the 6-bit box is converted to base 10, and it represents the column in the S1 table. Using the table, a value is selected using the column and row from the box. For example, if the B_1 = 110011, then the row is 3 because the first and last bits of B_1 is 11, which is 3 in decimal, and the column is 9 since the middle bits of the box is 1001, which is 9 in decimal. The value is then searched for in row 3 and column 9, which is 11. The value 11 is then converted to base 2, which is equal to 1011.

This means that: $S_1(B_1) = 1011$

This step is repeated for every S_iB_i using the rest of the S tables. The rest of the tables are too large to show in this section so they are shown in appendix D.

$S_1(B_1)\ S_2(B_2)\ S_3(B_3)\ S_4(B_4)\ S_5(B_5)\ S_6(B_6)\ S_7(B_7)\ S_8(B_8)$ will create a 32-bit block. Then the block is permuted into P using the permutation table below:

P Table

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

The P table is a representation of P; it links each bit position of P to a bit position in $S_1(B_1)\ S_2(B_2)\ S_3(B_3)\ S_4(B_4)\ S_5(B_5)\ S_6(B_6)\ S_7(B_7)\ S_8(B_8)$. For example, the first bit of P will be the 16^{th} bit in the previously computed 32-bit block, the second bit of P will be the 7^{th} bit of the previously computed 32-bit block and the last bit of P will be the 25^{th} bit of the previously computed 32-bit block.

The output of the P table is the final output of the function f. Using the output of f, $R_n = L_{n-1} \oplus f$ is computed. Then the following is repeated:

$$L_n = R_{n-1}$$

and

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$

It is repeated until the final round of L_{16} and R_{16} is reached. At the final two blocks L_{16} and R_{16}, they are switched around and combined to form $R_{16}L_{16}$, and then the final permutation is applied using the FP table:

FP TABLE

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

The table is a representation of the final permutation FP where the table links to a bit position in $R_{16}L_{16}$. For example, the first bit of FP will be the 40^{th} bit of $R_{16}L_{16}$, the second bit of FP will be the 8^{th} bit of $R_{16}L_{16}$ and the last bit of FP will be the 25^{th} bit of $R_{16}L_{16}$. This final permutation will create a 64-bit block ciphertext.

DES Decryption

The decryption process for DES is done by repeating the encryption process and applying the subkeys backwards, as in, starting from K_{16} and applying the keys in reverse order every time the function f is computed, so the last computation of the function f applies the subkey K_1.

DES Modes of Operation

DES can only encrypt and decrypt 64-bit blocks of data; this means there needs to be some mode of operation to use when using DES for data that is larger than 64-bits. There are four main modes of operation for DES, electronic codebook, also known as ECB, Cipher Block Chaining, also known as CBC, Cipher Feedback, also known as CFB and Counter, also known as CTR.

ECB is the simplest mode since the data is just divided into 64-bit blocks of data and each block is encrypted separately. Once encrypted, all the ciphertext blocks are concatenated, once decryption is needed, the ciphertext is split into 64-bit blocks and is decrypted one by one.

CBC is different where the plaintext is split into 64-bit blocks, and then each plaintext block is XORed with the previous ciphertext block and then that XORed block is encrypted using the DES key. In the decryption phase, once the ciphertext block is decrypted using the DES key, the last ciphertext block is XORed with the decryption to get the plaintext.

Triple-DES

Triple DES is DES with three encryptions using two 64-bit keys. The plaintext is encrypted once with the first DES key; then the ciphertext result is encrypted again using the second key. And finally, the second ciphertext is encrypted once more using the first DES key to get the final ciphertext. Triple DES is said to be viable until 2030 [19].

2.6 Java Sockets

A socket is a point of communication in a communication network that links two programs together. Sockets are used to represent a server and a client. Java.net provides a socket library that implements client-side sockets and server-side sockets. The server runs on a computer and on a specific port, if a client wants to connect to a server, they will need to connect using the server's hostname and the server's port. The server and client both open an input stream to receive data and an output stream to send data. Both sides will read and write to the stream to transfer information through the sockets [20].

To summarise, RSA is an asymmetric cryptography system that relies on two keys and large primes to be secure. RSA can also be used as a digital signature system to authenticate users in a messaging service. However, for RSA to work, a data structure is needed to represent large numbers, arrays provide the best time complexity to represent a number as a data structure. Symmetric key systems are faster than RSA, and DES can be used to encrypt large amounts of data quickly. For DES to work, Diffie-Hellman will need to be implemented as a key establishment protocol, and everything will need to work over Java sockets.

Chapter 3 Methodology

This chapter is used to describe the methods that went into developing the project and how the theory was applied. The software engineering process, specific aspects of the code and project decisions will be covered.

3.1 Software Engineering Process

During this project, various software engineering principles were adopted in Java to create a fully functional product in a reasonable amount of time.

3.1.1 Test-Driven Development

The development process relied a lot on test-driven development or TDD. At the beginning of the year, all deliverables were planned ahead of time to get a general idea of all the things that needed to be done; all deliverables were given a general time frame to be completed. The whole point of TDD is that it depends on the repetition of short development lifecycles. The only way to move onto the next lifecycle is by ensuring that specific test cases pass. The development lifecycle started by taking a deliverable and writing a test for it; the test could be anything that results in a functionality implementation.

Junit was used to write the tests in Java. JUnit is a unit testing framework used as a tool in test-driven development. Unit testing was useful because it allowed testing of specific parts of code without having fully completed a functionality. It allowed testing as code was being written, ensuring that every segment of code works. Mainly, unit testing was used to test if a specific low level method works, for example, when a newly written method was implemented into the program, it was unit tested first before moving on to another function.

The next step of the TDD lifecycle is running the tests. Most of the time the tests would fail, when a test would fail, code was written in order to make the test pass. Then all the tests would be rerun to make sure that the newly written code had not broken another pre-existing functionality. If the tests did not pass, then the code would be refactored until all the tests passed. This whole process would be repeated until a functionality was fully implemented and all the tests pass.

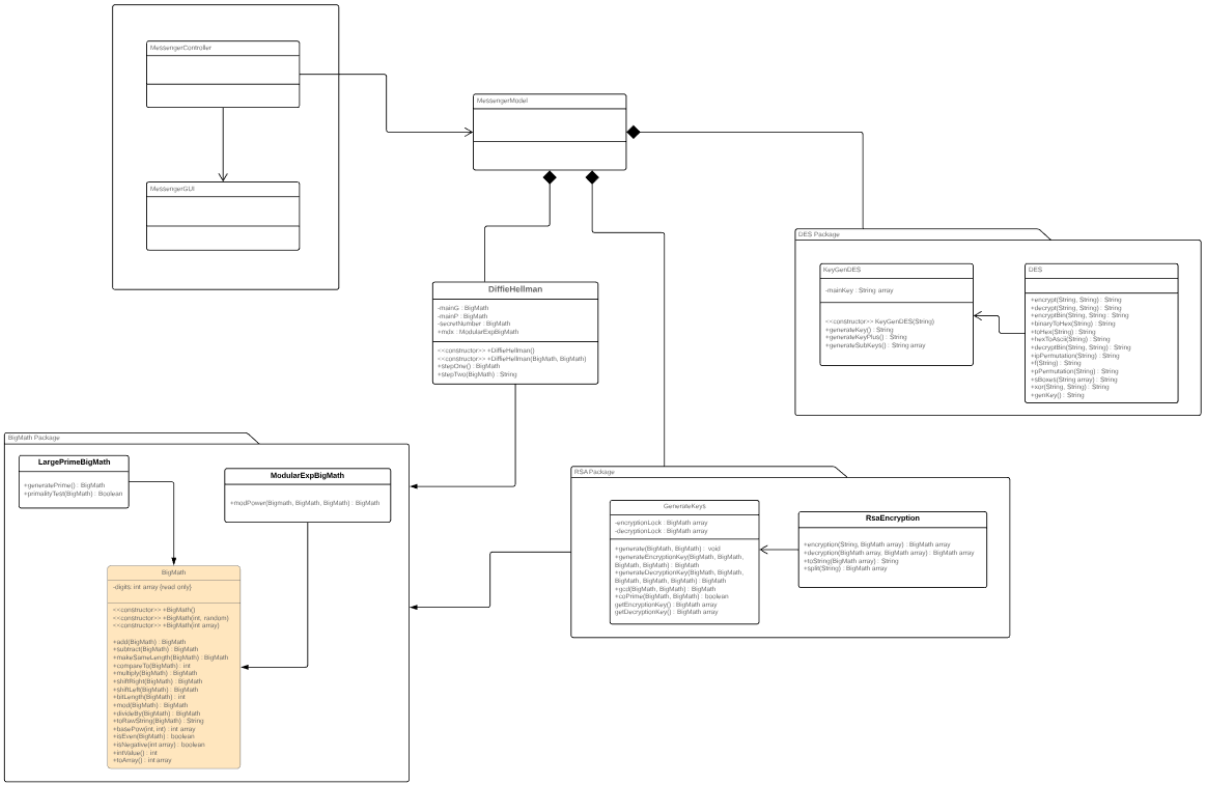
Unit tests were used to tests the low-level functionalities at the source of the application. Integration tests were used to check if many functionalities worked together, for example, integration tests were needed to see if BigMath can generate large primes for the key generation. Performance tests were done to check the speeds of mathematical operation algorithms. Acceptance tests were performed to verify if a system satifies a deliverable. For example, an acceptance test was done to check if the whole RSA process works.

3.1.2 Design Patterns

For this project, a class in Java was implemented in order to represent large numbers. It was decided to implement a factory method pattern to represent a large-number library, and the library was called “BigMath”. The idea is that when a number is constructed in the BigMath class, the class should be able to do calculations on that number, a factory method can help with this because it allows for calculations on a BigMath number and then returns a new BigMath class that represents the result of the calculation. The factory method is perfect because it efficiently represents large numbers in a way that makes it dynamic where it can make a BigMath class equal to another BigMath class, this is

important because it allows the program to do calculations with multiple large numbers. It also makes the library more flexible and more testable for unit testing.

A Model View Controller pattern was also used to manage how data gets transmitted to and from the UI. In the project, the model class is used to retrieve data, and it can also have logic to update the controller when information is needed or requested from the controller. The view represents the UI or what the users see. The controller is the bridge between the view and the model; it can request data from the model and send that data to the view. It also takes requests or takes information from the view and sends it to the model to be processed.



3.1.3 Version Control System

Throughout this project, a version control system was used to track incremental versions of files that were created. SVN was used to track versions of this project. The SVN repository was structured so that there is a trunk called “RSAProject”, a branch folder called “branches” and a tag folder called “tags”. The trunk was used to store the main file of development; the branches folder is used to store a copy of the trunk that is used to implement new functionalities or deliverables, for example, a branch folder was created during this project to implement DES. Whenever a new functionality was implemented into the branch, the branch was merged back into the trunk. Once a certain number of deliverables or objectives were completed, a tag would be created as a snapshot of the project and saved in the “tags” folder. To save work to the SVN repository, the commit command was used from Eclipse’s Subversion addon. Work done on the project was committed as much as possible, but only when all the unit tests passed, this was done because it reduces any conflicts that might occur if the broken or untested code was committed to the repository.

To get a better flow, work was done on multiple branches at the same time; this was helpful because progress was being made on different deliverables simultaneously. The only problem with working on two things at the same time is that it led to code conflicts. To try and prevent this as much as possible, when work was done on those branches, the code was implemented in a way that would create

minimal conflicts with other branches. This was done by making sure that areas in the code are not tightly coupled to other functionalities or methods.

Sometimes changes need to be merged from one branch to another, for example, when working on the socket branch, bug fixes were merged from the Diffie-Hellman branch into the socket branch. However, the bug fixes caused conflicts within the code, so conflict resolution needed to be done on all the problematic classes using Eclipse's conflict resolution system.

3.1.4 Documentation

Javadoc is a documentation generator that was used to document the code throughout this project. Every method includes a Javadoc comment that provides a short description of what the method does, what the parameters are used for and what the method would return if there is any return statement.

Also, all the complicated segments of the program were commented using Java's comment system to explain what the segment does. For example, when writing an algorithm that would not make sense if it was reread, a comment would be added to help the reader understand how it worked or even for the creator if they ever forgot why they did something or how something works.

Documentation was used because, after months of doing this project, it is hard to remember how specific functionalities work. After going back to the old code that has been forgotten, having documentation of the code helps to understand what was done and why it was implemented.

3.1.5 Coding Standards

Checkstyle is a code analysis tool used to guarantee that the code follows coding standards. When the code for the project was written, Javadoc comments were written for all methods and classes so that everything is documented. Naming conventions stuck to Checkstyle's standards, the maximum line length of any part of the program is under 80 characters, there are not many function parameters for any method, and the spacing and indention are consistent throughout the whole project. Checkstyle has many other rules that were all followed, and it was used to ensure that the code is reliable, readable and re-useable.

3.2 How The Project Works

3.2.1 BigMath

To achieve the goals of this project, a data structure needed to be implemented so that it would be able to represent large numbers in Java. The chosen data structure should also be able to perform mathematical operations. This was achieved by representing numbers as an object in java which was named “BigMath”.

Creating a large number

To form a number in BigMath, the number must first be sent into the BigMath class as a string. Inside the BigMath class, the primary constructor will initialise the number as a BigMath object; it does this by going through each character inside the string and transforming it into an integer array. For example, the string “1234” is transformed into the integer array {1, 2, 3, 4}.

Data Structure

Integer arrays were chosen to represent large numbers because the time complexity to access data in an array is O(1), which is much better than all the other data structures. The time complexity to access data is important because mathematical operations require constant access to numbers within the data structure. Every other data structure’s time complexity to access data is either O(n) or O(log(n)), which is not as efficient as an array. The integer array was chosen to be represented as a “final” variable, which means that its value cannot be modified once it has been constructed. The array was made a final variable because once mathematical operations were performed on the array, it is vital that the first array is kept intact in an equation. Whenever mathematical operations were performed on the data structure, BigMath returns a new BigMath number without changing the original BigMath numbers that were used to get to that result.

Constructors

The running time for the BigMath constructor is O(n) where n is the length of the number; this is because the constructor will go through every number in the string and convert it into an integer in the array. There are also two other constructors in the BigMath class, one of them takes in an integer array as a parameter. The other constructor uses an integer and a Random class as parameters, this is used to create a random large number with a specified bit length.

Base

Since it was decided to represent numbers as an array of digits, a base needed to be picked for each element of the array. To keep things simple in the beginning, base 10 was chosen, and the base would be increased as more features were implemented. This means that for each element of the array, there is a number between 0 and 9. The maximum number that BigMath can represent with a Java array and by using base 10 is $10^{2147483647}$; this is because Java has a maximum array size of 2,147,483,647.

Endianness

Endianness is the order in which the digits are arranged in a number. The numbers in BigMath are stored as big-endian, which means that the most significant bit is shown first. BigMath numbers are stored in big-endian because it makes the numbers readable and straightforward; it is also how

numbers are typically read so keeping it big-endian causes less confusion when writing complicated algorithms.

Negative Numbers

RSA does not require calculation of negative numbers, but just in case it was needed in another algorithm, negative numbers were implemented to the BigMath library. To represent a negative number in BigMath, a negative symbol is added to the first element in the integer array; this means that if there is not a negative symbol in the first element of the array, then it is a positive number. In every mathematical operation method, the BigMath number was checked to see if it was a negative number before doing any calculations, if it is negative, the calculation is adjusted so that it would work with a negative number. Negative numbers are represented like this because accessing the first element of an array is O(1), which means that it only takes one computational operation to find out if a BigMath number is negative.

Comparisons

It is important that BigMath can compare two numbers. To compare numbers, a method was implemented so that BigMath can compare two BigMath numbers and return an integer value depending on the difference between the numbers. If the first number is larger than the second number, then the method would return "1". If the first number is smaller than the second, then the method would return "-1" and if the numbers are the same then the method returns "0". For example, when comparing BigMath("12") to BigMath("20"), the "compareTo" method would return -1 because 12 is smaller than 20. The compareTo method works by first checking if any of the numbers are negative and returning the integer value based on which number is negative. It then checks the length of both arrays; this is because a larger number would have a larger array. If the arrays of both objects are the same length, then the method goes through each number of both arrays until it finds a number that is bigger than the other object's array. If the numbers are the same after going through each number in both arrays, then the numbers are equal, and the method returns 0. The worst-case time complexity of the compareTo method is O(n), where n is the length of the array, this happens when both numbers are equal, and the method must go through each element of both arrays. The best-case time complexity is O(1), and this happens if one of the numbers is negative or if one of the arrays is bigger than the other. An example of CompareTo is shown below:

```
BigMath a = new BigMath("10");

BigMath b = new BigMath("9");

Print( a.compareTo(b));
```

The code above would print "1" in the console because 10 is bigger than 9.

Addition and Subtraction

It is essential that BigMath can add and subtract two BigMath objects. First, both BigMath object's arrays needed to be the same length. To do this, a method was created that takes an integer array and adds leading zero elements to the array until the array is the same length as another one. Making an array the same length as another takes O(n) because the algorithm must go through each element of the original array to create a new array with leading zeros. Once the arrays are the same length, the algorithm can calculate the result. The addition algorithm works by starting from the least significant number in the first array and adding it to the same position in the second array; if the result is higher than the base, then it will save the carry and add it to the next significant digit in the array. The

subtraction algorithm works the same way but using subtraction instead of addition; the only difference is when a number goes below zero after subtracting, it will borrow 10 from the next most significant bit in the array. Once the result is calculated, both algorithms return a new BigMath object that represents the result.

Multiplication

Multiplication algorithms are more complicated than addition and subtraction. The chosen methodologies for the multiplication algorithm were created to show the different speeds of different algorithms. Included in BigMath, there are three algorithms for multiplication, the first one is a slow implementation called mulitplySimple. The slow implementation works through addition; the algorithm will repeatedly add a number until it reaches the answer. For example, for $12 * 6$, the result starts from zero, 12 will be added to the result six times to get the answer of the multiplication. The second implementation is a much faster version called "multiplyNorm". Pseudocode for multiplyNorm:

Create result array by adding the length of both numbers together and using it as array length

Iterate through the second number in the equation

    Iterate through the first number

        Multiply digit of the second number with each digit of the first number

        Add up each result for every iteration

    If the carry is not zero

        Add the carry to the next element in the result array

Return the result array

As shown above, the algorithm essentially performs multiplications for each element in the second array with every element in the first array and then adds the values together to form the result. The algorithm above has a running time of $O(n^2)$, where n is the length of the longest array, this is because it iterates through both arrays to calculate the result.

BigMath also contains an attempt at the Karatsuba algorithm that has a running time of $O(n^{1.585})$; it is uncompleted and not fully functional yet. The Karatsuba algorithm relies on the divide and conquer approach to multiply two numbers.

Division and Modulo

Multiple modulo and division algorithms were created for this project, but only three were used due to speed issues. The slow modulo algorithm is called "modSlow", it works by subtracting a number by itself until it cannot subtract anymore without going past zero. Once it cannot subtract anymore, the algorithm takes the last number it was on which is the remainder and returns it as a new BigInteger object. The "modSlow" method is extremely slow because it has to use the subtraction algorithm which is already O(n) and use it n times, where n is the length of the divisor, which means that the algorithm has a running time of $O(n^n)$.

The other mod algorithm works by implementing long division to get the remainder of an operation. The long-division algorithm is called "mod" which works by breaking the dividend into smaller numbers and dividing them by the divisor. The pseudocode is shown below:

Split dividend to get the same length as the divisor

If the split dividend is bigger than the divisor

Subtract divisor from the split dividend to get the first remainder

Divide by the divisor to get the first digit of the quotient

Iterate through the rest of the numbers in the dividend

Bring down next number in the dividend to form the next remainder

Divide the newly formed remainder by the divisor to get the next quotient digit

Multiply the quotient digit with the divisor to get rem value

Subtract the remainder with the rem value to get the next remainder

Return the final remainder

The algorithm above has a running time of $O(n^2)$. This algorithm can also be used to find the quotient of the division which means that this algorithm can be used as a division algorithm or a modulo algorithm.

Knuth's Algorithm D

Knuth's algorithm provides slightly faster running times; the method that implements algorithm D is called "knuthMod". The pseudocode for "knuthMod" is shown below:

If d equals 1 then add a leading zero to the dividend

D2 and D7: iterate through the dividend

If u is equal to v1

Set quo to BASE minus one

Else

*Quo is equal to u*BASE + u_{+1} / v1*

*while v2*quo is greater than u*Base + u_{+1} – quo*v1*BASE + u_{+2}*

quo is equal to quotient minus 1

*D4: u = u – quo*v*

D5: first digit of quotient is equal to quo

If quo is negative

D6: Quo = quo – 1

v + u

D8: u is remainder and quotient is result of division

In the algorithm above, *u* represents the dividend, and the *v* represents the divisor, Knuth's algorithm D consists of guessing in step D3 to find the right quotient. The multiple steps in this algorithm are shown using D1, D2, D3, D4, D5, D6, D7 and D8.

Shift Left and Shift Right

The Karatsuba algorithm that is available in BigMath requires a logical left shift and a logical right shift in order to do calculations. The right shift operator works by dividing the BigMath object by two using the long-division algorithm discussed in the last section. The right shift method in the BigMath class allows multiple rights shifts on one BigMath number, to specify the number of right shifts, it takes in an integer parameter. The left shift operator is similar to the right shift algorithm, but instead of dividing by two, it multiplies by two. Both methods return a new instance of a BigMath object.

Modular exponentiation

The code below is the pseudocode for "ModularExpBigMath.java":

Initialise the result to 1

base = base mod modulus

while exponent is greater than zero:

    if exponent is an odd number:

        *result = (result * base) mod modulus*

    shiftRight exponent by one

    *base = (base * base) mod modulus*

return result

The pseudocode works by implementing Bruce Schneier's "Right-to-left binary method" algorithm from "Applied cryptography". It reduces the number of operations needed to calculate the answer while preserving a small memory footprint. The algorithm works through exponentiation by squaring. The algorithm above has a running time of O(log exponent) which is extremely efficient compared to computing $base^{exponent}$ directly, which would take O(exponent). For example, 2^{20} would take 20 steps to compute the answer using the "right-to-left" algorithm, but directly computing the exponent would take 1048576 steps, this is the main reason why the "right-to-left" algorithm was chosen.

Miscellaneous

There are other smaller functionalities provided in BigMath. Firstly, BigMath can calculate the bit length of any BigMath object; it does this by counting the number of left shifts is needed to get from one to the number being tested. Counting the left shifts works because it stops counting when 2^n is smaller than the number being counted, where n is the number of left shifts.

BigMath also provides a method that checks if a number is even, the method is called "isEven". There is also a method that returns the BigMath number as a string called "toString". BigMath also contains a method that checks if a BigMath number is negative and a method called "basePow" that returns the

base, which is 10, to the power of any number below the Java maximum integer range. “basePow” is needed to calculate step D3 in Knuth’s algorithm D for division.

3.2.2 Primality Test

Fermat’s Primality Test

To generate large prime numbers, Fermat’s primality test was implemented using BigMath. “LargePrimeBigMath.java” is a class that generates a random large number that is prime; it does this by picking a random specified bit-length odd number and performing Fermat’s test on it to check for primality, if the random number is not prime, it adds two to the number and does the primality test again. The algorithm goes through the numbers like this because every even number is guaranteed not to be prime, it also skips numbers that end with 5. Adding two to the odd number each time and skipping numbers that end in 5 results in fewer primality which saves computational time.

Using the modular exponentiation algorithm, LargePrime checks if a number is prime if $a^{p-1} = 1(mod\ p)$, where *p* is the number being checked for primality and *a* is a random number that is greater than one. The algorithm goes through ten values of *a* to test the prime *p*, this is because ten primality tests on a prime is enough to check if a prime is a probable prime. The pseudocode for LargePrime:

Calculate prime – 1 and call it p

Iterate ten times

Select random number a

Calculate modular exponentiation of a, p, and prime

If the answer is not equal to one, then signal that number being tested is not prime

Return the number if it passes all prime tests

In the pseudo code above, *p-1* is calculated before the iterative loop to save time rather than computing it every iteration. All the calculations are done using BigMath.

3.2.3 RSA

RSA Key Generation

“GenerateKeys” is a class in the project that computes a key pair for RSA. The keys are generated by generating two random prime numbers using the previously discussed LargePrime generator. It picks an encryption key *e* by picking a random number “minRange” and adding 100 to that and assigning that to ‘a’. MinRange creates a range to test for compatible encryption keys. The reason why a range is used is because an exhaustive search would take too long. In this range, it checks if the first number is coprime with the product of the two initial primes, then it checks if it is coprime with Euler’s totient ‘phi’ and checks to see if e is smaller than phi. If e passes all these tests, then it is added to the list of potential public keys. Once the range is finished, and the ArrayList is filled with potential values of e, a random encryption key is selected within the ArrayList to be the official encryption key.

The private key selection consists of computing the modular inverse of the encryption key with regards to “phi”. Once the keys are generated, the key generator returns two BigMath arrays, an encryption lock that contains the encryption key and the product of the primes. The other array contains the decryption lock that contains the decryption key and the product of the primes.

RSA encryption and decryption

The RSA encryption and decryption algorithm use the modular-exponentiation algorithm to calculate the ciphertext and plaintext. It does this by first splitting the plaintext string into smaller sections and converting the plaintext from ASCII to decimal. The modular exponentiation algorithm calculates the ciphertext by taking the plaintext to the power of the encryption pair, modulo the product of the primes. The decryption algorithm is essentially the same, but it uses the decryption lock and the ciphertext in the modular exponentiation algorithm. After decrypting the ciphertext, the BigMath number is converted back to ASCII.

Padding

Before the message is encrypted, the RSA algorithm uses "Padding.java" to pad the plaintext with null bytes, and after decrypting the ciphertext, the algorithm removes the padded null-bytes. Padding was implemented to add randomness to every ciphertext, by padding null bytes to the plaintext; the encryption algorithm will always compute a different ciphertext even if the same plaintext is encrypted continuously.

3.2.4 Diffie-Hellman

Diffie-Hellman is implemented as a singleton class in the project. The algorithm starts by generating a random 1024-bit number and then initialising the two public variables used in Diffie-Hellman, the main prime number and the primitive root. As discussed in the literature review, having a precomputed prime number and its primitive root does not reduce security. In the constructor, the Diffie-Hellman class sets the prime "p" to a precomputed prime number, and the primitive root of the prime "g" is set to 2. The project implements this methodology because calculating the primitive root of a prime would take too long; the security does not come from the pre-computed numbers but rather from the secret number that is generated at the beginning of the Diffie-Hellman class.

Using the modular exponentiation algorithm, the first step and second step of Diffie-Hellman class are calculated to generate the shared secret key. StepOne in Diffie-Hellman.java generates the number that is sent to the other party and stepTwo takes the number generated from stepOne to calculate the shared secret key in hexadecimal.

3.2.5 Digital Signature Scheme

Within the messaging application, RSA is used to sign messages as an authentication tool, as discussed in the theory section, authentication ensures that a user is who they say they are. The messaging application achieves one-way authentication by exchanging each user's public key during the handshake. Whenever a user sends a message, the application encrypts a challenge message using the user's private key to create a digital signature. The signature is then attached to the message and sent to the other user. The receiver then takes the signature and decrypts it using the sender's public key, if the message is equal to the original challenge message then the sender is authenticated.

3.2.6 DES

The DES procedure is split into two classes; the first class is used to generate the subkeys that are needed in DES. As previously discussed in the literature review, the subkeys are generated by permuting bits from the initial 64-bit key. In order to perform permutations, the algorithm represents

the permutation tables PC-1 and PC-2 as integer arrays, this is done because it is fast to construct the required subkeys because arrays have an access time complexity of O(1).

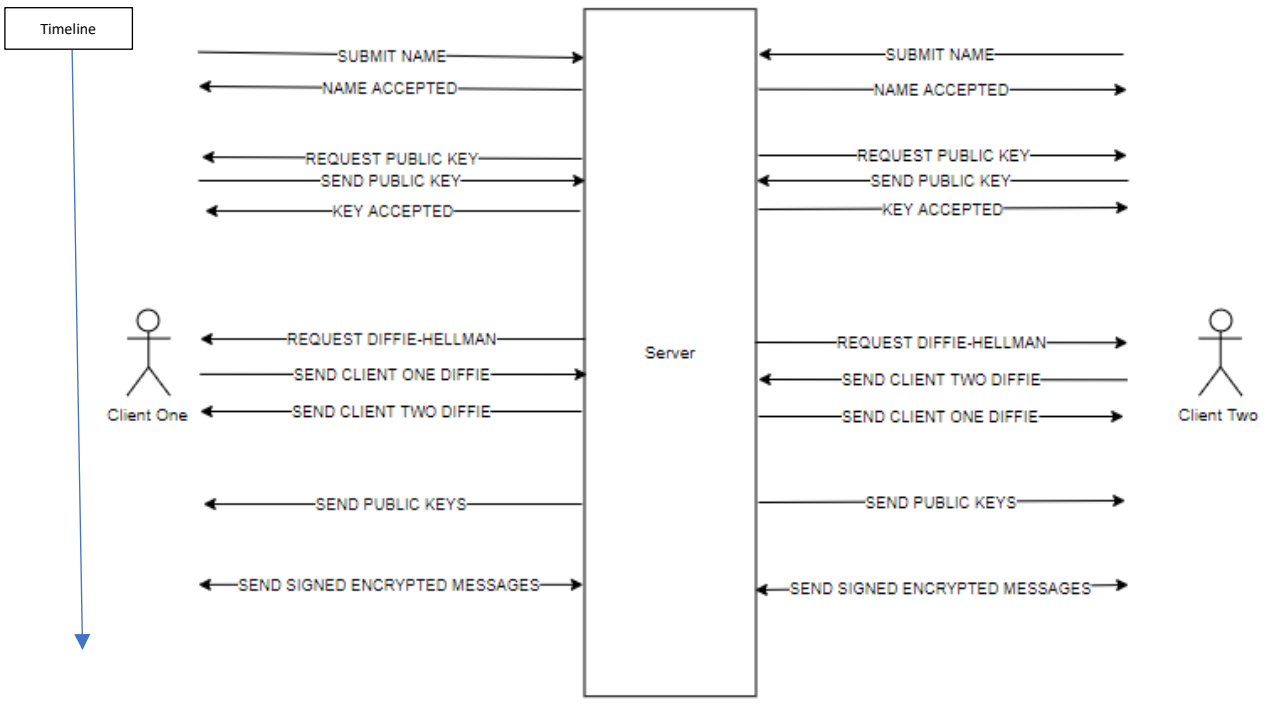
The DES encryption function consists of processing the plaintext then encrypting it using the DES procedure. As discussed in the literature review, DES needs a mode of operation to encrypt plaintexts that are larger than 64-bit; ECB, or electronic codebook, was chosen as the mode of operation. The class first starts by splitting the plaintext into 64-bit blocks; then it encrypts each block one by one with the subkeys. All permutation tables used in the encryption process are represented as integer arrays; the algorithm will use the arrays to permute data. However, the S-boxes are represented as 3D arrays. S boxes are used to permute 48-bit numbers to calculate the function f, in order to use the S-boxes in an efficient way they had to be in the same data structure, this is because the algorithm has to iterate through all the S-boxes and using multiple arrays or another data structure would not work.

The 3D array uses the first dimension of arrays to represent all the S-boxes, the second dimension represents the rows in each S-box, and the final dimension of arrays represents the columns in the S-boxes.

The DES algorithm computes the ciphertext and then converts it to hexadecimal. The DES algorithm decrypts a ciphertext by repeating the encryption process but applying the subkeys in reverse order. There is also a function that creates a DES key if one is not provided. However, this is not used in the messenger application because Diffie-Hellman generates the key for DES.

3.2.7 Sockets

Java Sockets was used as a communication layer to send encrypted messages to another client on an insecure channel. The project contains a server class and a client class. The client connects to the network server on a specific hostname and port.



The diagram above shows the communication between the server and the client. It starts by asking the client for its name and public key. Once the server receives the client name and server key, it sends an "accepted" message to verify that the data was taken in and stored successfully.

The server then performs Diffie-Hellman, which is a key establishment protocol. First, the server requests the client's result for the first step of Diffie-Hellman, then the server relays that result to the other client. Both clients can now compute the shared secret key using the final step of Diffie-Hellman. The client also allows the users to perform Diffie-Hellman again to generate another DES key, this can ensure perfect forward secrecy.

Then the server sends each client the other client's public key; the server does this because the clients will need to verify the signed messages using the other client's public key. The client also allows users to regenerate their RSA key-pair, once the RSA keys are generated, the public key is sent to the server and redistributed.

After this point, both clients can send DES encrypted messages with a signature to verify that it is not a malicious user sending a message. If a malicious user tries to join the connection or if the signature fails the authentication test, the client will inform the user that there is an imposter on the channel.

To summarise, this chapter covered multiple aspects of the project. It showed what data structure was used to represent large numbers. Using the large number data structure, the project was able to generate large prime numbers and perform mathematical operations for RSA and Diffie-Hellman. DES was implemented as a symmetric key system to encrypt large messages and using Java sockets, everything was implemented into a messaging application. This chapter also defends each decision by referring to discussions in the theory section. Importantly, this chapter didn't just explain what was done but also why certain decisions were made.

Chapter 4 Self-Evaluation

The role of this chapter is to discuss the findings and how it relates to the project aims and objectives by demonstrating my interpretations. This section is here to enforce what is already known about the area of study and to discuss the success of the project.

4.1 Results and Discussion

4.1.1 BigMath

The graphs in this section show the speed tests of the large-number library created for the project called BigMath. All graphs show the average speed of an algorithm, for each bit-length, a hundred tests were run.

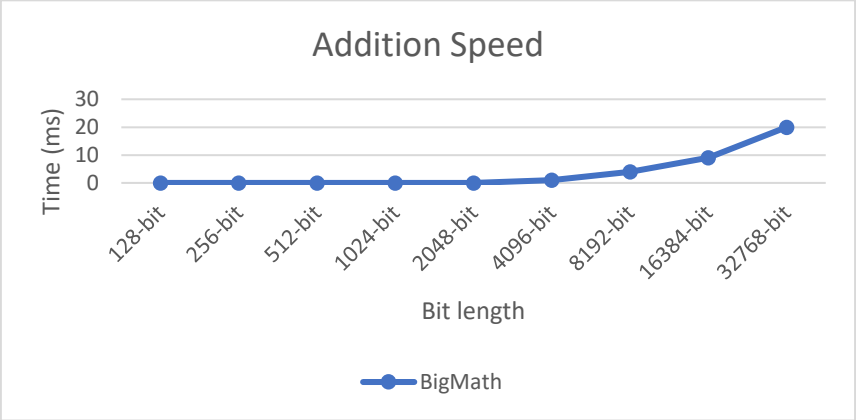


Figure 1: Addition

The results above were generated using AdditionSpeedTest.java in the "speed_stats" package within the project. It is important that the mathematical operations in the project are fast enough to work for RSA which is why they have all been tested to make sure they are efficient.

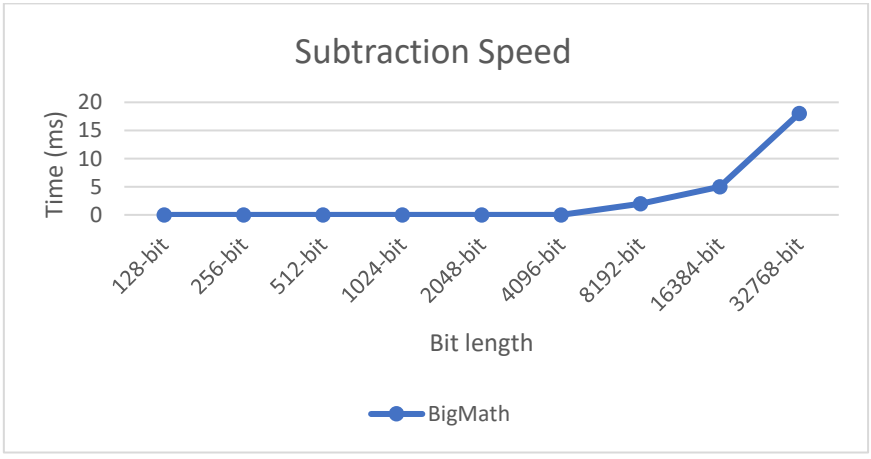


Figure 2: Subtraction

The figures above were generated using SubtractSpeedTest.java in the "speed_stats" package within the project.

BigMath was created for this project to do large-number calculations. Different bit-lengths were used to test the performance of the BigMath addition algorithm using Java's time recording package. Figure 1 shows that the speed of the BigMath stays consistently at zero milliseconds until the bit-length reaches 4096-bit, where the computational time increases. As already discussed in the "representing large numbers" section of the literature review, the slight increase in computational time is expected for larger numbers because as the bit-length goes up, there are more calculations needed to compute the answer.

In figure 2, the subtraction algorithm consistently stays at zero milliseconds even when calculating 2048-bit numbers. The addition and subtraction algorithm shows that representing numbers as an array is extremely efficient even when doing 4096-bit calculations. To compute the answer, the algorithms need to consistently access numbers within the data structure. As discussed in the literature review, arrays have an access time complexity of O(1).

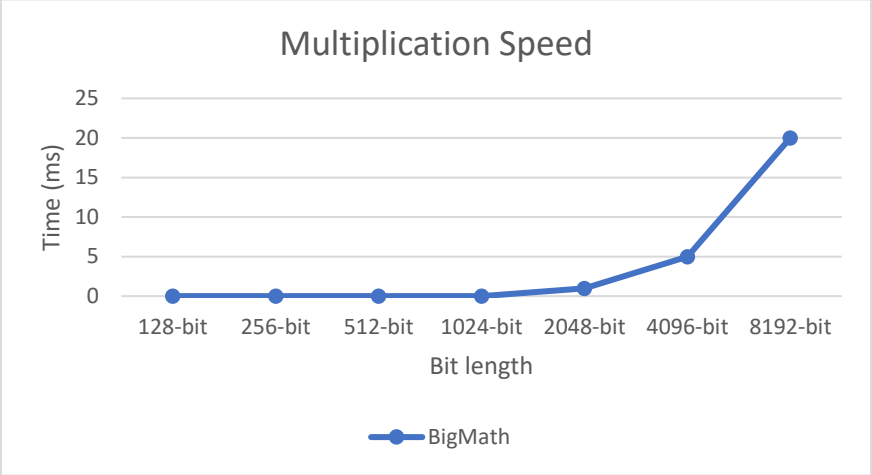


Figure 3: Multiplication Speed Test

The figures above were generated using MultiplySpeedTest.java in the "speed_stats" package within the project.

The multiplication algorithm is not as efficient as addition and subtraction. As shown in figure 3, the time needed to compute the result starts to increase when the bit-length is above 2048-bit. The results still hold up with what was discussed in the literature review; as the bit-length increases, the time to compute will increase too. The mod algorithm results, shown in figure 4, also demonstrates that as the bit-length increase the amount of time to compute increases too.

The performance results prove that the choice of data structure was the best choice for representing large numbers since every other data structure has a higher time complexity to access data and would have taken longer to compute the result.

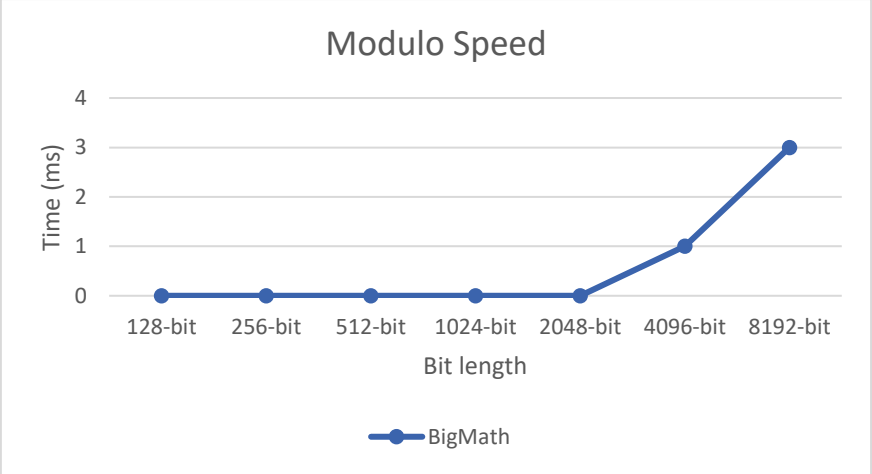


Figure 4: Division/Modulo

The figures above were generated using ModSpeedTest.java in the "speed_stats" package within the project.

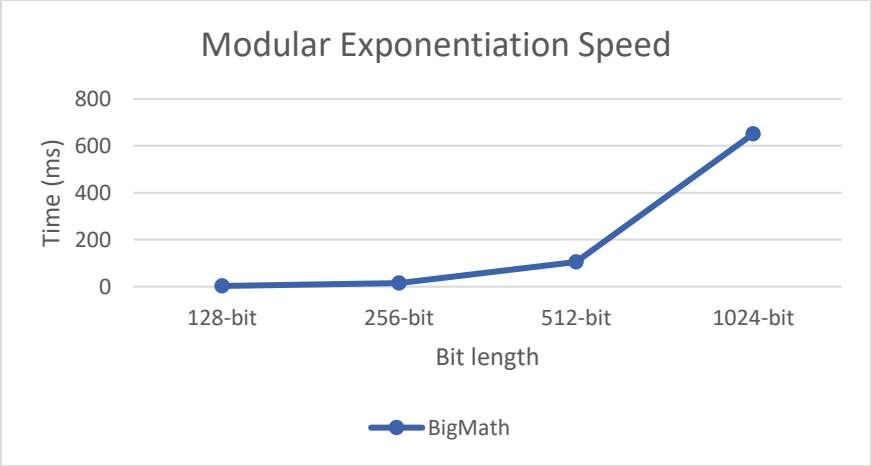


Figure 5: Modular Exponentiation

The figures above were generated using ModularExpSpeedTest.java in the "speed_stats" package within the project.

Modular Exponentiation was problematic throughout this project. The algorithm relies on multiple uses of the multiplication and modulo operation algorithms. The modular exponentiation algorithm showed accurate result but at slower speeds. As shown in figure 5, it took 651 milliseconds to compute 1024-bit modular exponentiation. After carefully investigating the algorithms, it was discovered that the mod algorithm was not efficient. Due to time restrictions, BigInteger mod was implemented into the modular exponentiation algorithm as a last minute solution.

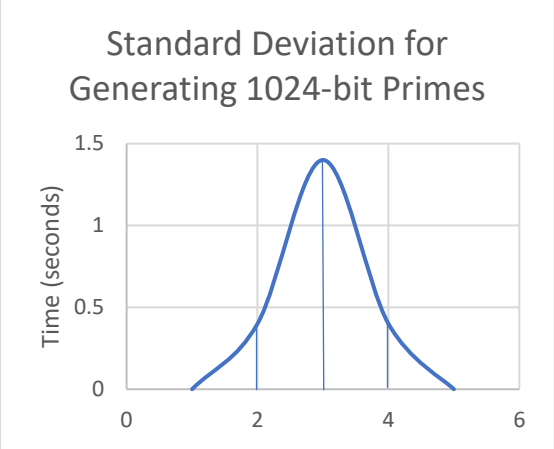


Figure 6

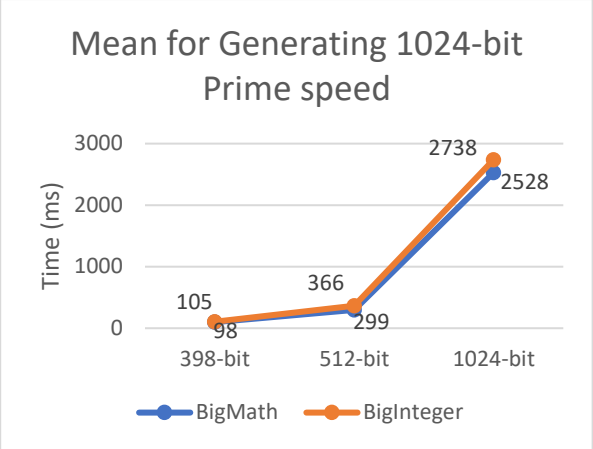


Figure 7

4.1.2 Large Primes

The figures above were generated using LargePrimeSpeed.java in the "speed_stats" package within the project, the results are shown in appendix C. The primes were generated using the same Fermat's test algorithm, but one uses BigMath, and the other uses BigInteger. 100 primes were generated for each test, and the average and variance were calculated. In the first test, 398-bit primes were generated, in the second test, 512-bit primes were generated and in the last test, 1024-bit primes were generated. The bit lengths were chosen because 398-bit primes are used for RSA-240, since RSA-240 uses a modulus that is 240 digits long, or 398-bit long. 512-bit primes were tested because RSA-1024 uses 512-bit primes. 1024-bit primes were tested because RSA-2048 uses 1024-bit primes.

Generating large primes is one of the main focuses of the project; it is essential that the primes are generated in a reasonable amount of time. The prime generator within the project can successfully generate a 1024-bit prime number in two and a half seconds, as shown in figure 7, and is 200 milliseconds faster than using BigInteger. BigMath is also faster when generating smaller primes. Fast generation of large numbers is essential because Diffie-Hellman and RSA rely on prime numbers to function, which is fundamental in completing this project. In the "factorisation" section in the literature review, it was discussed that RSA-240, RSA-1024 and RSA-2048 has not been factorised yet, so implmenting them into the messaging app will not cause security problems.

The BigMath library proved to be successful throughout this project. It was used to generate large prime numbers to be used in RSA key generation; the 2048-bit key pairs mean that this project is fully secure in modern-day cryptography and will be secure for a very long time. Using the modular exponentiation algorithm, the BigMath library was able to compute the encryption and decryption process of RSA efficiently and also used in Diffie-Hellman to produce a shared secret key. However, the only improvement that could have been implemented is increasing the base of the data structure; doing so would have increased the speed of all algorithms in BigMath. Otherwise, BigMath proved to be successful.

Diffie-Hellman was implemented in the project to establish a DES key between two users. The functions used in RSA were also used in Diffie-Hellman. "TestDiffie.java" is a performance test class that was used to show the speed of the Diffie-Hellman procedure. The test performs ten key establishments between two parties. The test results as shown in appendix A, shows that the program can successfully generate a shared key in an average time of 2.7 seconds. Diffie-Hellman is important because it generates a symmetric key without both parties leaking any compromising information, and the project shows that it can successfully generate a DES key in reassonable time.

4.1.3 Paper Examples checked using RSA prototype program

This section used to show that the theoretical side of this project also works in practicality. RSA key generation has been calculated on paper and then will be proofed checked using the prototype program. During these tests, the bit size of the two original prime numbers start small but gradually increases.

Test 1: In the first test, **8-bit prime numbers** prime numbers are used:

$$p = 5$$

$$q = 11$$

$$N = (p*q) = 55$$

$$phi = (p-1)*(q-1) = 40$$

The encryption key can be any number where:

$$1<e<phi$$

coprime with N and phi

Seven is picked since it applies to the rules above:

$$e=7$$

d is chosen to satisfy $ed \equiv 1 \pmod{phi}$ or $ed = 1 + k*phi$ where *k* is an integer:

$$d=23$$

$$23*7=1+(4*40)$$

Encryption lock = (7, 55)

Decryption lock = (23, 55)

Encrypt the number 12:

$$c = m^e \pmod{N}$$

$$12^7 \pmod{55}=23$$

Ciphertext = 23

Decrypt the ciphertext:

$$23^{23} \pmod{55} = 12$$

Decrypted plaintext = 12

Proof check using prototype program (PrototypeEncryption.java):

The program successfully replicated the example on paper.

The first test shows that the program can efficiently calculate the keys within two milliseconds.

Test 2: In the second test, **16-bit prime numbers** prime numbers are used;

$$p = 11$$

$$q = 13$$

$$N = (p*q) = 143$$

$$phi = (p-1)*(q-1) = 120$$

Encryption key can be any number where:

$$1<e<phi$$

$$coprime\ with\ N\ and\ phi$$

43 is picked since it applies to the rules above:

$$e = 43$$

d is chosen to satisfy $ed \equiv 1\ (mod\ phi)$ or $ed = 1 + k*phi$ where k is an integer that exists:

$$d = 67$$

$$43*67 = 1 + (24*120)$$

Encryption lock = (43, 143)

Decryption lock = (67, 143)

Encrypt the number 123:

$$c = m^e\ (mod\ N)$$

123^{43} (mod 143) = 85

Ciphertext = 85

Decrypt the ciphertext:

85^67 (mod 143)=123

Decrypted plaintext = 123

Proof check using prototype program:

This shows that the program worked slightly slower, by one millisecond compared to the last example (8-bit prime numbers). But it is still efficient for encrypting and decrypting a message.

Test 3: In this test **1024-bit prime numbers**. Since the program works by generating large random primes, this test works by pre-generating the two primes using the program and then hand checking the math using online big-number calculators.

p:
467577987471171593857324984075551366213140048234264181018474086545581363812479475067595814723162096897393309752664279492615996808867140771895701820848859683487120283742211604173173009143515847482719405917580767078775858489792575440940586130859573679565654118090380341515079758691142787799916916400601654878 33

q:
651905599780222129699779533768466743878281340500112956373713206273667170904032224169584072907881422890133443118749002352554582607305567916298091910088044050400462581918812053375910308305654223846398329095898384626277910249434564955963392322303537269555241184888311184799828298433734368114832286647539168426 29

N:
304816708366423306310838497438097638420238817346526396903227830555545408896489929231884727221487204745623206881055524360366069053594326052804284665104111869856253356502061505215252947014994541907678869204605545316613303353363676853077292592386919826461915481263224386129679587792185347183385250963748698282936099744218491311590372840311183785122760396319849462030878392872964170585375226004298143051996880823218390342723216571881914855663000104480434191028403563770492092641897389041162983919596037609373798597572895480481524165915760835875811893669835095397476862116416684005964814704568776899418354129428887523295 7

phi:
304816708366423306310838497438097638420238817346526396903227830555545408896489929231884727221487204745623206881055524360366069053594326052804284665104111869856253356502061505215252947014994541907678869204605545316613303353363676853077292592386919826461915481263224386129679587792185347183385250963748698282936099744218491311590372840311183785122760396319849462030878392872964170585375226004298143051996880823218390342723216571881914855663000104480434191028403563770492092641897389041162983919596037609373798597572895480481524165915760835875811893669835095397476862116416684005964814704568776899418354129428887523295 7

920460554531661330335336367685307729259238691982646191548126322438612967958779218534718338525096374869828292490490834597737435480179513274360402184618243250569065695651994477168523821010901192634417568644562534312281400908375343020906150127301759849625371903452643161626398528715246567215074510433689608262124743810396343098647852348943190677210913820398590626790908662976874281573413332000534027086209894747929024 96

e:
360684041307801672218281224632381757862884576134117195642807202369322571217392687025307578303362142974582052724384234918129730943704766169868918653208393932134633298172903763200868107527419013576854899170114230956359023283385247173749550772604289440174784448950484844338309015243635144699076258728212040749

Calculate the mod inverse of e modulo N:

d:
179505657151538518157085573711464389905602066495824755533064609469999552014408287103913831309243131720412734657447592613524039849011105948802354900509656829739526644659966937703428448652834731604968990390737243617160935684709605175066591454762337202279233635884354879119363350988222938762197656617795196403215230473044847436649045639011236694254972278053406268686792425030692162476893221373652929087749543590104504818215958186986318380154899127608206276668887013528167448945783920041183804180030820632618679977871678345594904839423967762533440378820382659116434392473879481096032845475351986059250353554523923566499 7

ciphertext:
152140819633176047584903661394317180571644949057301596709358242825353447153427367454548356944261704730846786475105230379218140622890967728290948350093987840824647040946773347292760611747037132218247119218105867153952921694617743205876898331271984441045049438853937267641539681260815533787948752395865763801679396709067288461379709910659647724274627451683934945134945015679071037610800836927660712497369807067790195867088316436386555281619766012492441331138463383861968722982262330036110742733440491638514020078724825778301913515118241533458710314982227314549661166037039681223510502008554734119285037449017918214468 4

```
Encryption lock: (360684041307801672218281224632381757862884576134117195642807202
Decryption lock: (179505657151538518157085573711464389905602066495824755533064609

Key Generation Time: 3401 milliseconds


Enter a number that you would like to encrypt: 232134

Ciphertext: 152140819633176047584903661394317180571644949057301596709358242825353



Decrypting ciphertext using key...
Decryption: 232134
```

The program shows that it can successfully calculate the RSA key pairs using the 1024-bit primes; this means that the program can generate 2048-bit key pairs.

RSA's speed is essential in online cryptography. Since RSA heavily relies on large primes to be secure, it is important that the prime number generator works, multiple bit length primes were tested and proof checked by hand. The test results demonstrate that as the prime number's bit-length increase, the key generation time increased. It takes two milliseconds to generate RSA keys using 8-bit prime numbers. However, it took three and a half seconds to generate the keys using 1024-bit prime numbers. As discussed in the literature review, it is expected that as the bit-length rises, the time to do calculations will also rise.

4.1.4 DES

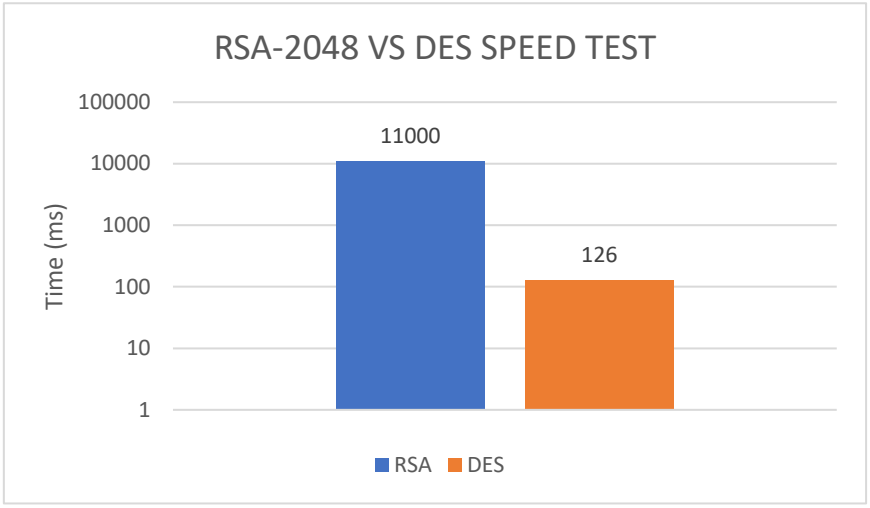


Figure 8: DES Encryption vs RSA Encryption

The figures above were generated using DESSpeed.java in the "speed_stats" package within the project. Both are encrypting a 1024-bit number.

The pattern throughout this section is that when dealing with a large bit-length number, the speed will decrease, this becomes a problem when dealing with large plaintext inputs. Figure 8 shows a speed comparison between RSA and DES. In the speed test, RSA and DES are used to encrypt a 1024-bit number; the results show that DES is much faster when it comes to encryption, as discussed in the literature review, RSA-2048 becomes inefficient when dealing with large amounts of data. The results demonstrated that DES should be used as an encryption and decryption algorithm since it is much more efficient. After conducting this performance test, RSA was only used as a digital signature tool, and DES was used as the primary encryption method in the official messaging application.

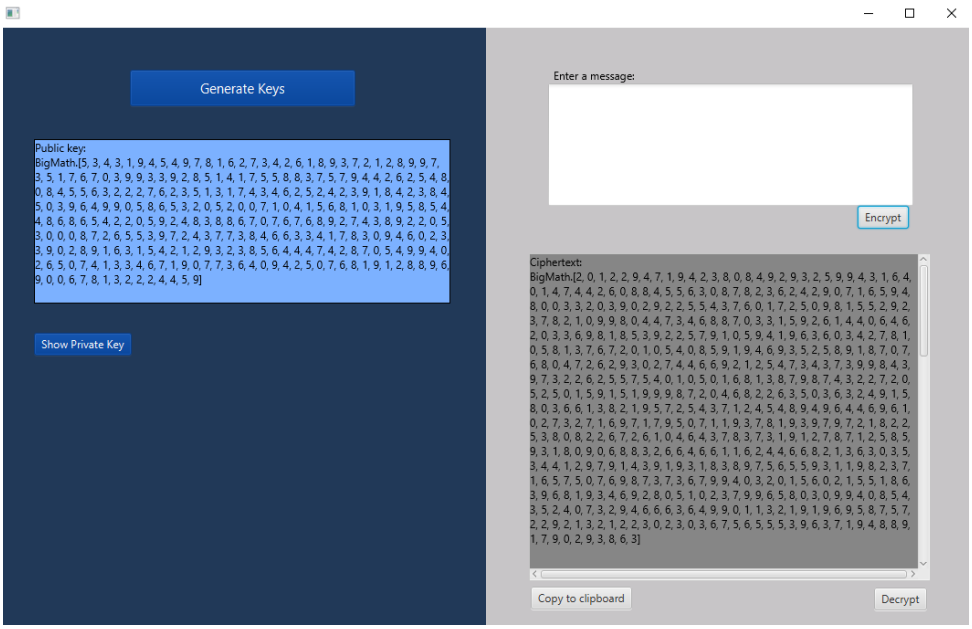


Figure 9: Encrypting "hello" with the RSA.

RSA padding was implemented as another layer of security. As discussed in the literature review, textbook RSA can leak information about the plaintext by computing the same ciphertext for a

specific plaintext. The encryption process in “MainGUI.java” shows that nothing is leaked when encrypting the same message repeatedly.

4.2 Conclusion

The main goal of the project was to create a library in Java that can represent large numbers and perform mathematical operations on them to create cryptographic protocols for a messaging application over an insecure network. Ensuring confidentiality is important because privacy is fundamental for communication online.

The performance tests and results show that BigMath’s algorithms can perform large-number calculations in a reasonable amount of time. The BigMath library was used to generate 1024-bit prime numbers for RSA-2048 in less than three seconds, this proves that the chosen data structure ended up being the most optimal solution.

Symmetric key encryption is extremely efficient, as discussed in the literature review. DES was implemented to ensure encryption and decryption of large amounts of data is done in a reasonable time. The performance tests show that RSA-2048 is inefficient when encrypting large amounts of data while a symmetric key system like DES outperforms RSA by a large margin.

As a result, RSA-2048 was implemented as a digital signature to ensure authentication between users in the messaging application. Since anyone can pretend to be anyone online, it is important that the application can authenticate users and guarantee that a user is not communicating with an imposter.

Diffie-Hellman was included as a key establishment protocol for DES. Moreover, the tests prove that the program can generate a DES key within two seconds. The messaging application does not have to rely on insecure key communication to establish a key. The implementation of Diffie-Hellman was a massive accomplishment since it helps guarantee a shared secret key and makes the messaging application much more secure.

I will be improving on the project in the future as there are more things that could be included. Some of the things that could have additionaly been implemented is a higher base for BigMath which would have made all the algorithms faster, also I could have implemented the Advanced Encryption Standard as an alternative to DES.

Thoroughout the project I had to create deliverables and ensure that they’re completed within a specific time frame. I have learnt that it is extremly important to manage my time based on what deliverables I need to accomplish, and background knowledge is extremely important when implementing the deliverables.

To summarise, this project was largely successful. BigMath can efficiently do large-number calculations in a reasonable amount of time. Using BigMath, multiple cryptographic protocols were implemented into the messaging application in order to guarantee confidentiality and authentication online.

4.3 Professional Issues

This section is used to discuss the professional issues encountered throughout the project. As a programmer, I have an ethical responsibly and moral obligation to ensure that my creations help ensure professionalism.

Throughout this project, multiple professional issues were raised as concerns. The primary professional issue encountered was the idea of guaranteeing privacy. In an online world where data is shared through insecure channels, it is important to ensure users privacy. I had a responsibility to make the messaging application robust and secure so that the application does not leak any messages. I had to develop the application in a way that protects a user's information and stops malicious users from using the application maliciously. A loss of information can lead to harm, so I had to do everything in my power to stop it.

Malicious users and attackers are not the only problems; I had to guarantee that I, as the creator, cannot see any of the private messages communicated between users. Since I had control of the server, I had to implement end to end encryption to guarantee that even I cannot read messages. Honouring confidentiality must be a goal of the project even for users who do not care about it.

Accepting full responsibility for my work is another important aspect. During the project, I conducted multiple tests to ensure that my algorithms work. In order to ensure privacy, I had to apply the theory accurately, however, it is impossible to predict everything, and if anything were to fail, the responsibility lies on me, and I should work to correct the errors if anything were to fail. The responsibility is on me to accurately describe what the application can and cannot do; this is important to shareholders and consumers.

Some of my code is obtained from online resources or based off of algorithms in computer theory books. It is essential that I have permission to use the code and that I give proper credit to the original creators by commenting the author's names or organisation above code that I have not written myself. Not only is it ethically wrong to use another person's code without crediting them but it can also raise legal issues.

Programmers can be exposed to illegal activities when developing software. I realised that if I were a bad programmer, I would be able to perform illegal activities with my software, such as creating a backdoor somewhere in the cryptographic protocols to save all the user's decryption keys in another location. It is ethically wrong to do these sorts of things because I would be invading a user's privacy. I have a moral obligation to guarantee a user's privacy no matter what. A backdoor is not just ethically wrong, but it could also potentially be exploited by malicious users.

In this project, I had to decide what data had to be recorded for the software to work. It should be clear what information is taken and how it is used. The problem with data collection is that I am responsible for the user's data so it must be protected at all cost. Data breaches are becoming more and more normal; if there were a data breach with my software, then I would be responsible.

Throughout this project, I encountered conflicting professional issues. It is vital that the project I am working on does not break any laws or rules in the country that I created it in. However, a weird scenario occurred throughout this project; in December 2018, Australia became the first western country to ban secure encryption [21]. This meant that my project is inherently illegal for consumer use in Australia. I can not comment on the politics that led to this decision, but if I were in Australia, the Australian government would require me to create a backdoor in any encryption procedure which completely breaks user confidentiality. The Association for Computing Machinery's Code of Ethics in section 2.3 [22] states that computing professionals must abide by rules unless there is a compelling ethical justification.

Moreover, [22] things that have been judged unethical, like the Australian law, should be challenged. The Australian law is immoral because it breaks user privacy. The most ethical solution to this scenario is challenging the law through existing channels before violating the rule, as stated in the ACM Code of Ethics. However, the computer professional must consider the consequences and accept responsibility if they were to break the rules.

Since technology advances faster year by year, it is important that my program will be safe and secure in the years to come. Software systems get old and limited and can lead to breaches of security, which is why I implemented RSA-2048 as well as RSA-1024 and RSA-240. In the theory section, it was discussed that RSA-1024 would only be viable for the next five years. RSA-2048 is nowhere close to being broken which is why the messaging application will stay relevant and secure for years to come.

Throughout this project, I strived to create a high-quality application. In a work environment, my colleagues would rely on me to release the highest standard that I can achieve. I would not only have a responsibility to my coworkers but also the shareholders and the consumers. In order to achieve the highest levels of code, I have a responsibility to test every aspect of my code and also to perform work only in the areas that I am competent in. This requires me to do thorough literature reviews before attempting anything.

Professionalism was important in this project to ensure that my program can achieve a better societal impact. Communication online should be done in complete privacy. Without privacy, anyone would be able to read anything and criminals could potentially hurt consumers with their stolen information. As a computing professional, I must ensure that I act responsibly by consistently choosing to do the right thing. Ethical decision making must be enforced when programming and I have to be willing to place the interest of others ahead of my own.

Chapter 5 Manual

5.1 Requirements

Java 1.8 was used to develop this project, and it is needed to run the project.

The prototype classes are in the “prototype” package under the main folder. To run the prototype classes, no extra libraries are needed, need to run the prototype class in eclipse.

To run the GUI files, you will need to install the following software within Eclipse’s addon marketplace:

- JavaFX

- e(fx)

- xtext

- scene-builder

A video-tutorial on how to install this software: https://www.youtube.com/watch?v=2PxU7q9xl38

Alternatively, if you do not want to install these packages and just want to run the program, I included a runnable java file called SecureChat.java that runs the messenger without installation. It is found within the main RSAProject folder.

The Source code is located within the src folder.

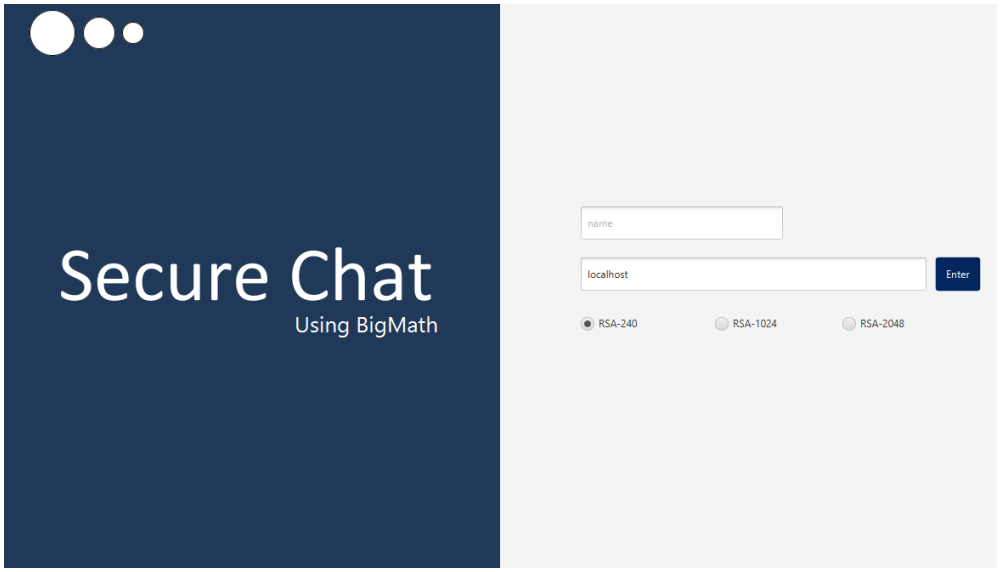
The Initial reports are included in the reports folder.

The J-Unit tests are located in the tests folder.

5.2 Running the Server

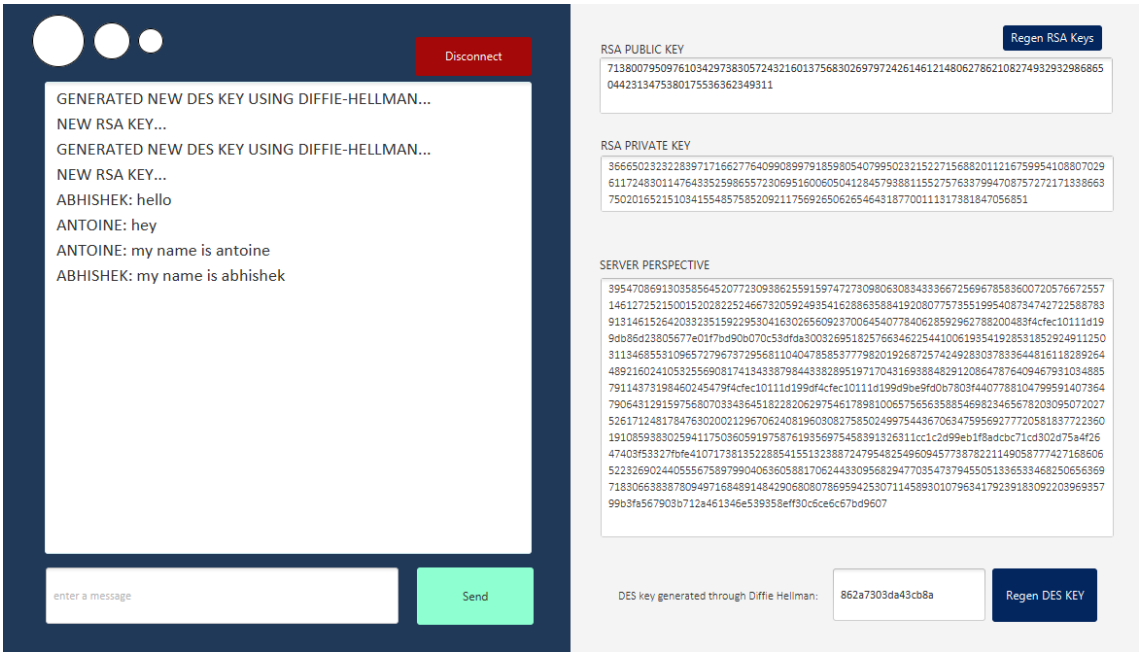
The server is already hosted online so it doesn’t need to be run, but if the user wanted to test the server, the java file to run the server is in the “Server” package in the “src” folder. Right click and select run in eclipse to launch the server. If the server is hosted locally, the user would have to change the address in the messenger to “localhost”.

5.3 Using the Messenger



Run the program by double clicking on the JAR file. The program will ask for a hostname, which is 138.68.168.181, the user's preferred username and the version of RSA that the user would like to use. Once entered, the program will connect to the other user on the server. The program will wait until the other client connects before joining the chat. The window can be moved anywhere by holding the right mouse button on any area of the program and dragging it.

Youtube link of the program running: https://youtu.be/AGiyuLT9lF0



Simply enter a message and click the green "send" button to send an encrypted message to the other user.

The panel on the right of the UI shows the user's RSA key pair, what the server sees, and the DES key generated through Diffie-Hellman. The program also alows the user to change their RSA key pair or perform Diffie-Hellman again with the other user to generate another DES key.

The red "disconnect" button will stop the connection and close the program.

Bibliography

[1] Barry Steyn. How RSA Works. http://doctrina.org/How-RSA-Works-With-Examples.html

[2] Ben Lynn. Modular Exponentiation. https://crypto.stanford.edu/pbc/notes/numbertheory/exp.html

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. Section 31.8. 2001

[4] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C Section 11.6 1996

[5] Wade Trappe. Lawrence C Washington. Introduction to Cryptography with Coding Theory. 2006

[6] David Ireland. Weaknesses in RSA. https://www.di-mgt.com.au/rsa_alg.html#weaknesses

[7] Wouter Castryck. Another RSA Challenge Broken. https://www.esat.kuleuven.be/cosic/another-rsa-challenge-broken/

[8] Jonathon Katz, Yehuda Lindell. Introduction to Modern Cryptography: Principles and Protocols. 2007. Section 12.3.

[9] Varun N R. Big O Cheatsheet – Data Structures and Algorithms with Their Complexities. https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/

[10] Rob Bell. A Beginner's Guide to Big O Notation. https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

[11] Josh Bloch, Michael McCloskey, Alan Eliasen, Timothy Buktu. BigInteger Library Documentation. https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

[12] Donald E. Knuth. The Art of Computer Programming Volume 2. 1981. Section 4.3.1, page 257.

[13] Menezes A.J., Oorschot P.C., Vanstone S.A. Handbook of Applied Cryptograph. 1996. Section 12.6 p.515.

[14] D. Harkins, D. Carrel. The Internet Key Exchange. 1998. https://tools.ietf.org/html/rfc2409 Section 6.2.

[15] T. Kivinen, M. Kojo. More Modular Exponential Diffie-Hellman Groups For Internet Key Exchange. 2003. https://tools.ietf.org/html/rfc3526.

[16] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall,Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelink Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails In Practice. https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf

[17] John Carl Villanueva. Symmetric Vs Asymmetric Encryption. 2015. https://www.jscape.com/blog/bid/84422/Symmetric-vs-Asymmetric-Encryption

[18] J. Orlin Grabbe. The DES Algorithm Illustrated. http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm

[19] Elaine Barker, Nicky Mouha. Recommendation For The Triple Data Encryption Algorithm Block Cipher. 2012. https://csrc.nist.gov/publications/detail/sp/800-67/rev-2/final.

[20] Java Sockets Documentation. What Is A Socket? https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[21] Joel Hruska. Australia Becomes First Western Nation to Ban Secure Encryption. 2018. https://www.extremetech.com/internet/281991-australia-becomes-first-western-nation-to-ban-secure-encryption.

[22] Don Gotternbarn. ACM Code of Ethics and Professional Conduct. Section 2.3. https://www.extremetech.com/internet/281991-australia-becomes-first-western-nation-to-ban-secure-encryption.

Appendix

Appendix A. ModExp test output:

128-BIT NUMBER SUBTRACT SPEED TEST

BigMath: 3

BigInteger: 0

256-BIT NUMBER SUBTRACT SPEED TEST

BigMath: 15

BigInteger: 0

512-BIT NUMBER SUBTRACT SPEED TEST

BigMath: 105

BigInteger: 0

1024-BIT NUMBER SUBTRACT SPEED TEST

BigMath: 766

BigInteger: 1

Appendix B. DiffieTest.java output:

SPEED: 2718 ms

Appendix C: LargePrime speed:

1024-BIT PRIME GENERATION SPEED TEST

BigMath mean: 2.3045303800000005

BigMath variance: 1.847904782724279

BigInteger mean: 1.5016426600000001

BigInteger variance: 1.1993500907648824

BigMath mean: 98.32304530379999

BigInteger mean: 105.30501642659999

BigMath mean: 299.33304530379996

BigInteger mean: 366.22501642659995

BigMath mean: 2528.5730453038

BigInteger mean: 2737.7250164266

Appendix D: DES S-Boxes

S1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9

14 11 2 12 4 7 13 1 5 0 15 10 3 9 8 6
4 2 1 11 10 13 7 8 15 9 12 5 6 3 0 14
11 8 12 7 1 14 2 13 6 15 0 9 10 4 5 3

S6

12 1 10 15 9 2 6 8 0 13 3 4 14 7 5 11
10 15 4 2 7 12 9 5 6 1 13 14 0 11 3 8
9 14 15 5 2 8 12 3 7 0 4 10 1 13 11 6
4 3 2 12 9 5 15 10 11 14 1 7 6 0 8 13

S7

4 11 2 14 15 0 8 13 3 12 9 7 5 10 6 1
13 0 11 7 4 9 1 10 14 3 5 12 2 15 8 6
1 4 11 13 12 3 7 14 10 15 6 8 0 5 9 2
6 11 13 8 1 4 10 7 9 5 0 15 14 2 3 12

S8

13 2 8 4 6 15 11 1 10 9 3 14 5 0 12 7
1 15 13 8 10 3 7 4 12 5 6 11 0 14 9 2
7 11 4 1 9 12 14 2 0 6 10 13 15 3 5 8
2 1 14 7 4 10 8 13 15 12 9 0 3 5 6 11