

# Security & Encryption

## RSA Project

March 2022

David Cardoso, David Docherty, Yasmin Paksoy

[dcard001@gold.ac.uk](mailto:dcard001@gold.ac.uk), [ddoc001@gold.ac.uk](mailto:ddoc001@gold.ac.uk), [ypaks001@gold.ac.uk](mailto:ypaks001@gold.ac.uk)

# Table of Contents

Introduction .....	3
Algorithms.....	4
Design.....	6
Demonstration .....	7
Running the program .....	7
Method 1: Source Files .....	7
Method 2: JAR File .....	7
Creating a User.....	7
Encrypting a Message .....	8
Decrypting a Message.....	8
Error: User not found.....	9
Quit the program .....	9
Discussion .....	10
How it Works.....	10
KeySet Class .....	10
User Class .....	11
Encrypt Class .....	11
Decrypt Class.....	11
Main Class .....	11
Pros and Cons.....	12
Vulnerabilities: Charlie.....	12
Our experience.....	13
Bibliography .....	14
Appendix .....	16

## Table of Figures

Figure 1: Flowchart Key.....	4
Figure 2: Flowchart of User Interface .....	4
Figure 3: Flowchart showing find user sub-process.....	5
Figure 4: Flowchart showing encrypt sub-process .....	5
Figure 5: Flowchart showing decrypt sub-process .....	5
Figure 6: UML Diagram .....	6
Figure 7: Main menu.....	7
Figure 8: Creating a user demo .....	7
Figure 9: Encrypting a message successfully demo .....	8
Figure 10: Decrypting a message successfully demo.....	8
Figure 11: Decrypting a message demo - user not found error.....	9
Figure 12: Quitting the program demo.....	9

# Introduction

For this piece of coursework, we decided to attempt assignment 1. This assignment involves implementing the RSA algorithm. We worked in a group of three to complete this task.

RSA was introduced in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman at the Massachusetts Institute of Technology. The acronym 'RSA' is derived from the surnames of the three creators. RSA is a *public-key cryptosystem*, wherein the encryption key is made public, and the decryption key is kept secret. Messages can be encrypted by anyone using the public key but can only be decrypted and read by the holder of the private key. This can also be referred to as an *asymmetric cryptosystem*, as separate keys are used for encryption and decryption.

The security of RSA is based upon the difficulty of factoring the product of two large prime numbers. While it is easy to multiply two large primes together, it is very difficult to take the product and find the two prime factors that produce it. This is an example of a *trapdoor function*, a function which is easy to compute in one direction, but near impossible to compute in reverse.

The keysets are generated by randomly choosing two distinct prime numbers  $p$  and  $q$ . These numbers are multiplied together to obtain  $n$ , which will be used as the modulus for both the public and private keys. Modern RSA implementations will then compute  $\lambda(n)$  where  $\lambda$  is Carmichael's totient function. In the original RSA paper, the Euler totient function  $\phi(n) = (p-1)*(q-1)$  was used instead of  $\lambda(n)$ . In either case, we will refer to the resultant value as  $r$ .

Next, an integer  $e$  is chosen that satisfies these conditions:

- $1 < e < r$ .
- The greatest common divisor of  $e$  and  $r$  is 1 (in other words,  $e$  and  $r$  are coprime).

This integer  $e$  will serve as the encryption exponent. Finally, the decryption exponent  $d$  can be computed using the extended Euclidean algorithm. The correct value of  $d$  will satisfy the following formula:

$$e * d \bmod r = 1$$

The public key consists of the encryption exponent  $e$ , and the modulus  $n$ . The private key consists of the decryption exponent  $d$ . A message is encrypted by applying the public key to the message  $m$  with the following formula:

$$m^e \bmod n = c$$

The ciphertext  $c$  can then be decrypted and read by applying the private key with the formula:

$$c^d \bmod n = m$$

In this project we will create a basic implementation of the RSA algorithm, using the Euler Totient function to find  $r$  as described in the original RSA paper. We will be creating a command line program using Java where users can generate keys, encrypt, and decrypt messages. Our program will consist of a variety of classes and will save all generated keys, so users do not have to generate and share new keys when using the program.

# Algorithms

In order to show the flow of our program, we have developed multiple flowcharts. Flowcharts are visual representations of a series of processes and decisions needed to be made in order to perform a task/procedure. They use a range of shapes to represent different types of steps, which can be found on figure 1, the key, and shapes are linked together using directional arrows.

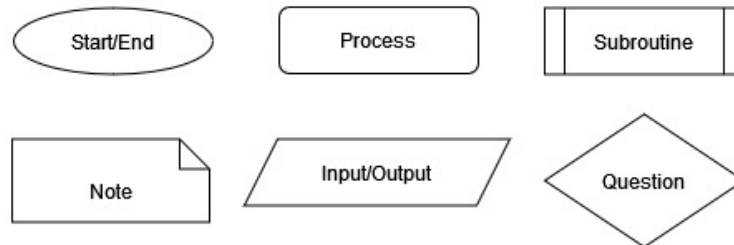


Figure 1: Flowchart Key

We split our program into 1 main and 3 sub-process flowcharts. The main depicts the user interface and calls the sub-processes as necessary. The sub-processes represent the finding a user, encryption, and decryption processes. The main flowchart, figure 2, can be seen below.

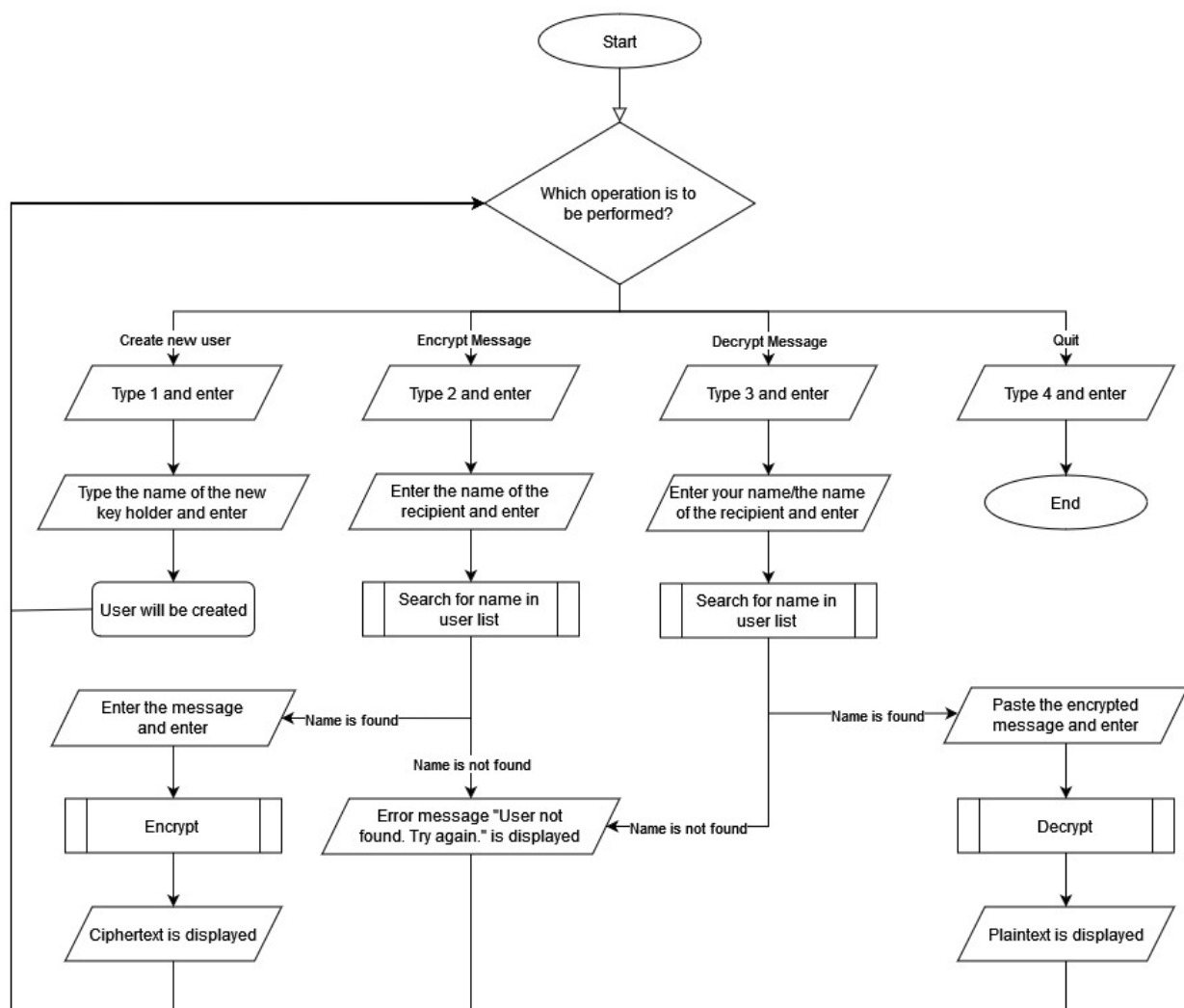


Figure 2: Flowchart of User Interface

The sub-process flowcharts can be seen below, figures 3, 4, and 5. Each sub-process is referred to from the main flowchart at least once.

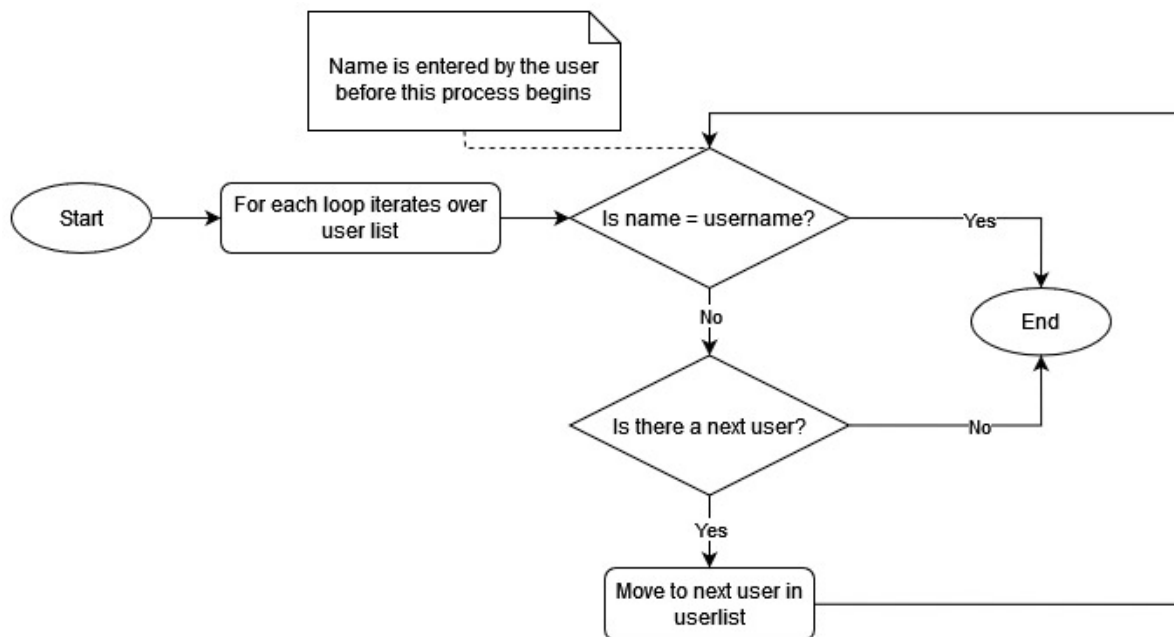


Figure 3: Flowchart showing find user sub-process

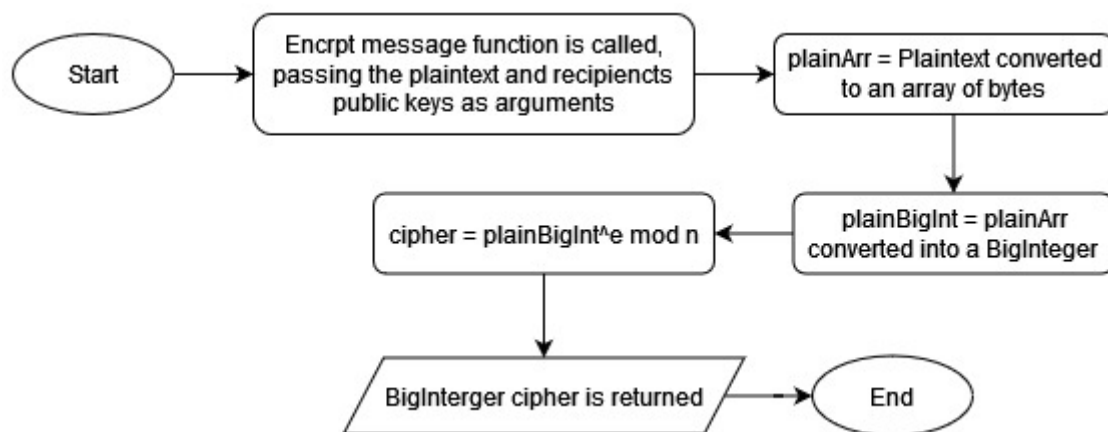


Figure 4: Flowchart showing encrypt sub-process

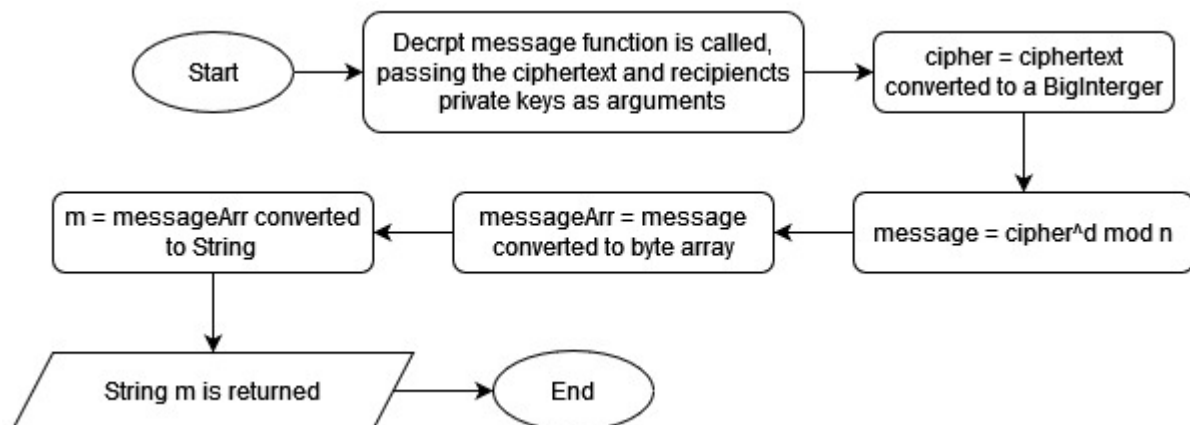


Figure 5: Flowchart showing decrypt sub-process

# Design

The figure below shows the UML diagram representing this program:

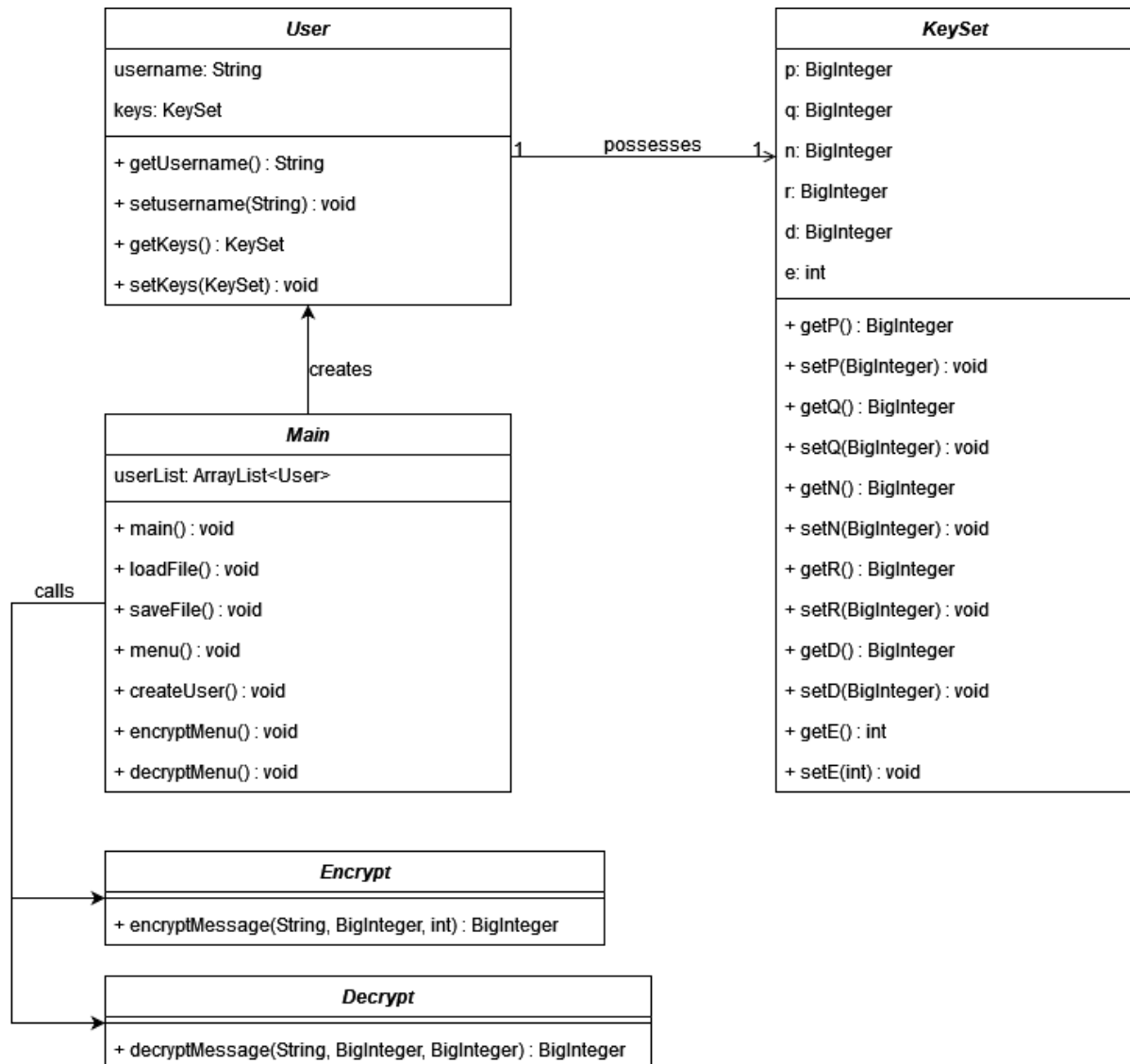
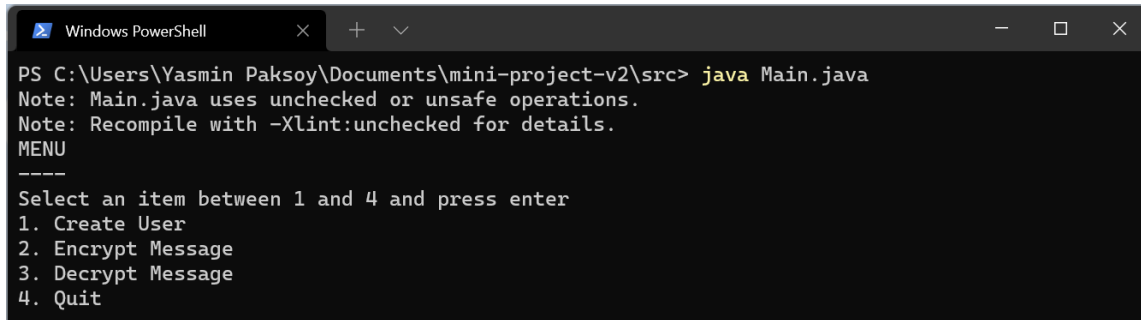


Figure 6: UML Diagram

# Demonstration

## Running the program

The program can be run in two ways; the first, using the source files and the second, using the JAR file. Note: for both of these methods to work, JDK version 17 needs to be installed on the computer. Both of these methods produce the following output:



```
PS C:\Users\Yasmin Paksoy\Documents\mini-project-v2\src> java Main.java
Note: Main.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
MENU
----
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
```

Figure 7: Main menu

## Method 1: Source Files

For the first method, we navigate to the location of the source files in the command line and enter "java Main.java".

## Method 2: JAR File

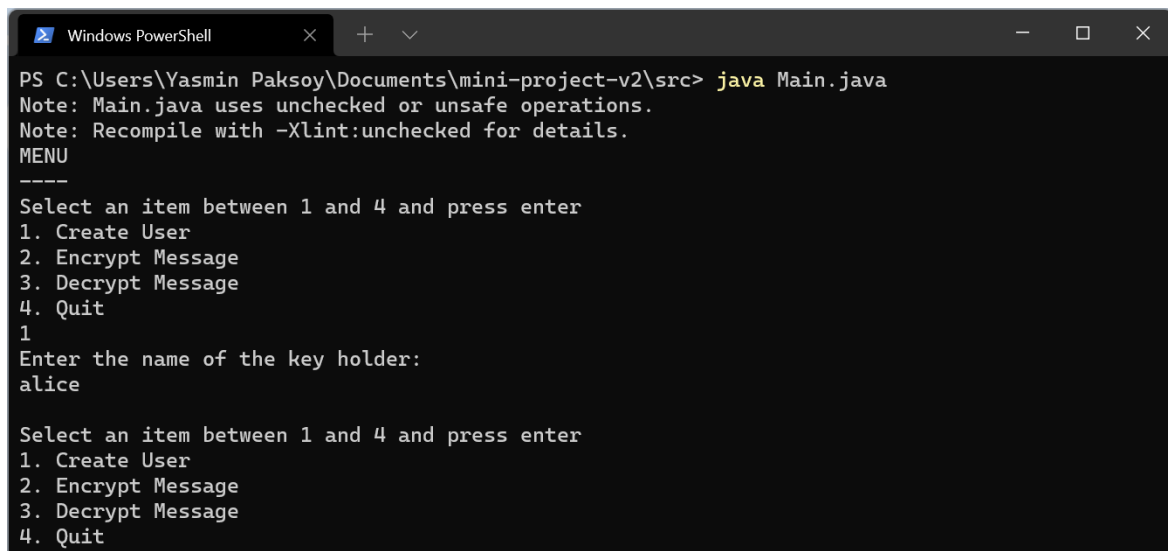
For the second method, we navigate to the location of the JAR file in the command line and enter "java -jar mini-project.jar".

## Creating a User

To create a new user, we follow the following steps:

1. Type 1 on the main menu and enter.
2. Enter the name to link the new set of keys to,

The program creates the keys and displays the menu again.



```
PS C:\Users\Yasmin Paksoy\Documents\mini-project-v2\src> java Main.java
Note: Main.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
MENU
----
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
1
Enter the name of the key holder:
alice

Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
```

Figure 8: Creating a user demo

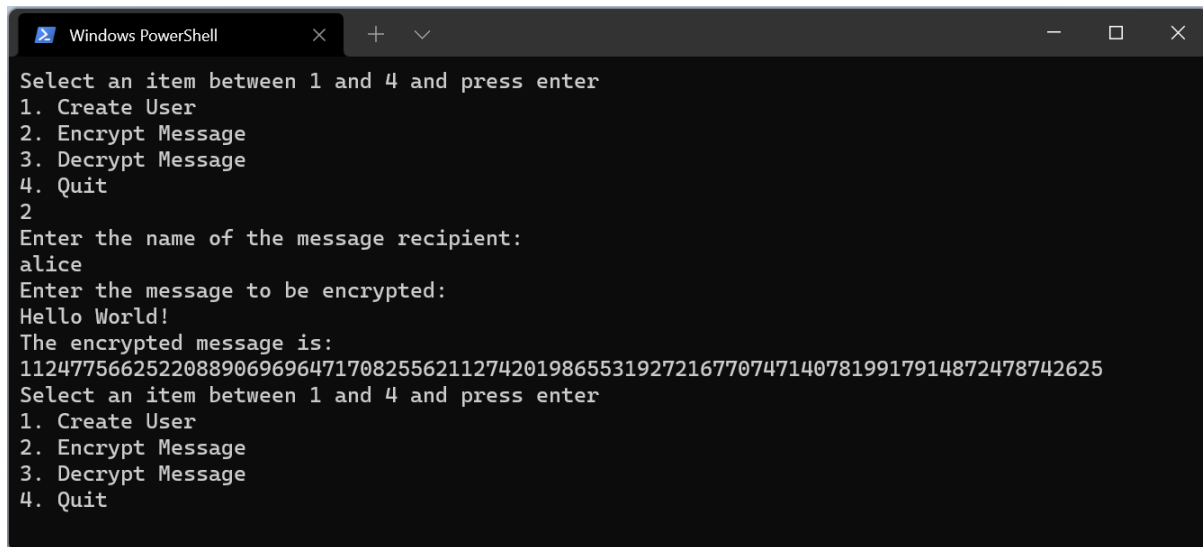


## Encrypting a Message

To encrypt a message, we follow the following steps:

1. Enter 2 at the main menu.
2. Enter the name of the user the message is for.
3. Enter the message.

The encrypted message will be displayed, and the program returns to the main menu.



```
Windows PowerShell
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
2
Enter the name of the message recipient:
alice
Enter the message to be encrypted:
Hello World!
The encrypted message is:
11247756625220889069696471708255621127420198655319272167707471407819917914872478742625
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
```

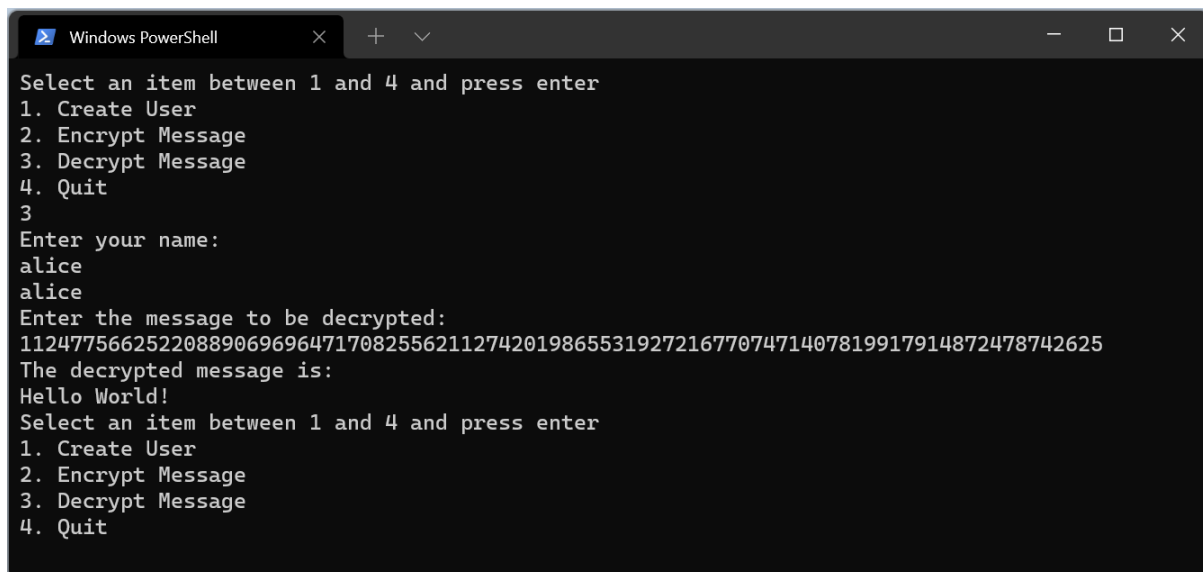
Figure 9: Encrypting a message successfully demo

## Decrypting a Message

To decrypt a message, we follow the following steps:

1. Enter 3 at the main menu.
2. Enter the name of the recipient of the ciphertext.
3. Enter the ciphertext.

The decrypted message will be displayed, and the program returns to the main menu.

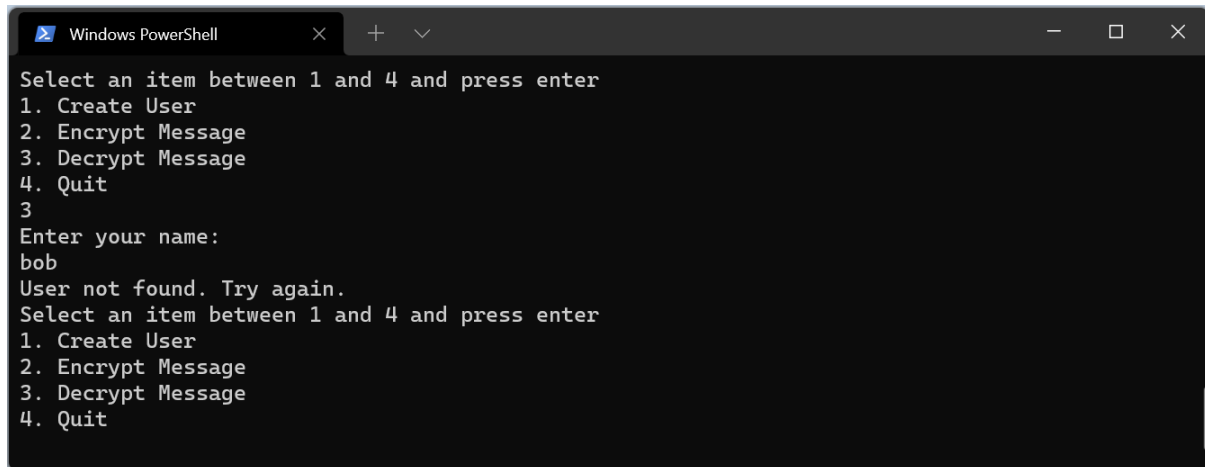


```
Windows PowerShell
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
3
Enter your name:
alice
alice
Enter the message to be decrypted:
11247756625220889069696471708255621127420198655319272167707471407819917914872478742625
The decrypted message is:
Hello World!
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
```

Figure 10: Decrypting a message successfully demo

## Error: User not found

If the name entered at step 2 of the encryption and decryption process is not found in the system the message "User not found. Try again." Will be displayed, as shown below. The program will then return to the main menu.

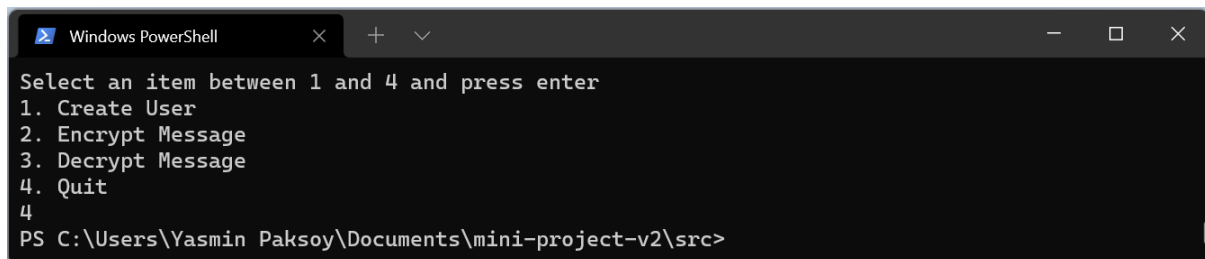


```
Windows PowerShell
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
3
Enter your name:
bob
User not found. Try again.
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
```

Figure 11: Decrypting a message demo - user not found error

## Quit the program

To exit the program, we type 4 and enter. This saves the keys made so far and closes the program.



```
Windows PowerShell
Select an item between 1 and 4 and press enter
1. Create User
2. Encrypt Message
3. Decrypt Message
4. Quit
4
PS C:\Users\Yasmin Paksoy\Documents\mini-project-v2\src>
```

Figure 12: Quitting the program demo

# Discussion

## How it Works

As mentioned previously, we created this program using Java, where we implemented multiple classes for different aspects. Our final program consists of 5 classes: key set, user, encrypt, decrypt, and main.

### KeySet Class

This class consists of 6 variables, all values used to generate public and private keys:  $p$ ,  $q$ ,  $n$ ,  $r$ ,  $d$ , and  $e$ . The variables  $e$  and  $n$  represent the public keys, and the variable  $d$  represents the private key. Size wise, we decided to use 4096 bits as the key length. For this, we generated the random primes,  $p$  and  $q$ , at 2048 bits. When they are multiplied together to get value  $n$ , this gives us the decided 4096-bit length. We decided to use a bit size so big as this is the minimum needed to ensure a relatively secure key. While 2048 bit keys are currently regarded as secure, with further developments in technology they will be able to be cracked soon using quantum computers. To get ahead of this issue, we implemented 4096 bit keys, consequently making it near impossible to find the factors of  $n$  and break encryption.

As we are using such big numbers, we could not use any of Javas primitive types; Java biggest numerical primitive type is a long, which is only 64 bits long, much smaller than what we needed. Due to this, we made use of the Java class BigInteger. This type supports such large numbers and has a range of methods that enables us to make the necessary calculations. All variables in this class, other than  $n$ , are of type BigInteger.

The constructor for this class does not take any arguments and generates all the variables itself. Here is a breakdown on how each variable is generated:

- Variables  $p$  and  $q$ , random primes, are generated using the BigInteger method `probablePrime()`.
- Variable  $r$  is calculated using the Euler totient function.
- Variable  $e$  is calculated by finding the greatest common divisor (GCD) between  $r$  and  $e$  using a while loop.
- Variable  $d$  is calculated using the extended Euclidean algorithm that uses a table composed of two times the variable  $r$ , with the variable  $e$  and the number 1. Following the ancient Euclidean algorithm, we coded a while loop that will keep looping until we find the variable 1 in the bottom left position of the table. At this point, the variable present in the bottom right position will be the variable  $d$ . Then, as confirmation that we have found a correct variable for  $d$ , we check if  $e*d \bmod r = 1$ .

Finally, with both keys generated, we implement getter and setter methods that allow a better management on how vital variables are accessed and altered. This implementation also generates better protection over data and a more secure code.

## User Class

The user data structure consists of two variables: username and KeySet. The constructor for this function sets the username, using the data entered by the user and retrieved by a scanner, and generates the KeySet.

Both KeySet and User classes implement interface Serializable. This interface allows for an object to be represented “as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object” (tutorialspoint.com, 2019). This means that all the data in the local file, to save they keys generated, are stored in a non-human readable form. Object data is serialized and deserialized through high-level streams, ObjectOutputStream and ObjectInputStream respectively. Although a simple solution, is yet very effective protection on a file that can easily be accessed by Charlie. Additionally, upon the user creation, a new instance of KeySet class is created, a new and unique set of keys is created and appended to the local file.

## Encrypt Class

This class only consists of one function, the encryptMessage function. This function takes as argument the plaintext to be encrypted and the public key variables  $n$  and  $e$ . The exact process it undertakes can be seen in figure 4, the encrypt sub-process flowchart.

## Decrypt Class

This class also only consists of one function, the decryptMessage function. This function takes as argument the ciphertext to be decrypted and the private key variable  $d$  and  $n$ . The exact process it undertakes can be seen in figure 5, the decrypt sub-process flowchart.

## Main Class

The main class is where all other classes are brought together. It consists of one variable and 7 functions. The variable is an ArrayList of type User, called userList, which stores all of the generated users. Here is a breakdown of the functions and their roles:

- Main function, the function that is initially executed, calls the loadFile and menu functions.
- loadFile function uses the FileInputStream and ObjectInputStream methods mentioned previously to load all the previously generated keys into the userList variable.
- saveFile function also uses the same methods to save the userList variable into the Users.txt file when called.
- menu function consists of the menu that the users interact with. It uses the Scanner class to accept input and calls the relevant functions depend on which option the user has chosen, either createUser, encryptMenu or decryptMenu.
- createUser function, which accepts input from the user, again using the Scanner class, as the username that will be linked to the generated keys. It then creates the user object and adds it to the user list, calling the saveFile function afterwards to save the user.
- encryptMenu function takes the name of the recipient and the message. It then encrypts the message using their public keys and the encryptMessage function from the Encrypt class. If no such user is found, it returns the user to the main menu with an error message.
- decryptMenu function takes the name of the user and the ciphertext. It then decrypts the message using their private key and the decryptMessage function from the Decrypt class. If no such user is found, it also returns the user to the main menu with an error message.

## Pros and Cons

Our implementation of this algorithm has a few improvements when compared to the plain RSA. We have used a vast bit length for keys hence requiring an enormous computational power to try to find the values for variables  $p$  and  $n$ . Although not impossible to break, by having 4096 bits as we assured that any attempt will require more time and power. Additionally, we have implemented file persistence which allows users to keep the same public keys overtime, allowing past messages to still be read in the present and with maintained encryption. Another positive point of our implementation is the acceptance of special characters. By converting to byte arrays, we are using UTF-8, so each character of the typed message refers to a unique code point, also known as character encoding, translating into 1,112,064 usable valid character code points including special characters.

Despite our best efforts, there are less strong points. The way we implemented our code requires users to use the same computer. This means that, to use two or more computers, users would need to share the user file potentially over no secure methods, requiring a lot of effort to make this program usable across different locations. Additionally, since keys are stored locally an unfriendly party, e.g., Charlie, can have access to the stored data, potentially jeopardizing the cipher. Despite Users.txt being mostly not human-readable, some of it still is and since we used Java to serialize it, Charlie could simply use Java to read it and turn it back into an ArrayList. Overall, this is an implementation of “plain” RSA and thus is vulnerable to some attacks, which will be discussed further on.

## Vulnerabilities: Charlie

What can an attacker (Charlie) do when attempting to gain unauthorised access to information encrypted with RSA? Provided that the non-public variables and keys are kept secret, RSA is a highly secure algorithm. However, there are several attacks that may exploit vulnerabilities with “plain” RSA.

Many of these vulnerabilities come about as a result of non-ideal parameters used during key generation. The values  $p$  and  $q$ , and the resulting product  $n$ , must be of sufficient length to ensure the difficulty of factorisation. As more powerful computers have become widely available, the key length needed to ensure security has increased. If  $n$  is too small, Charlie would be able to factor it in a matter of hours, allowing him to access all keys and decrypt the message (Jiann Mok and Wen Chuah, 2019).

Similarly, the primes  $p$  and  $q$  should be similar in magnitude but differ in length by a few digits to make factoring harder. If they are too close in size, Charlie may be able to use *Fermat’s factorisation method* to reveal the value of  $n$ .

As mentioned earlier, our implementation uses  $p$  and  $q$  values that are 2048 bits in size. These multiply to produce an  $n$  value that is 4096 bits in size. This length is secure for current computational power, meaning Charlie would not be able to break our encryption using brute force. However, our implementation does not ensure that  $p$  and  $q$  differ in length. Therefore, Charlie may be able to use Fermat’s method to solve for  $n$ .

Other vulnerabilities come from the properties of the RSA algorithm itself. If the same plaintext message is encrypted and sent to multiple recipients, the original message may be decrypted using the *Chinese Remainder Theorem*. As RSA has no random elements, it is vulnerable to a *chosen plaintext attack*. In this attack, Charlie can encrypt likely plaintexts using the public key and check whether they are equal to the ciphertext. RSA also has the property that the product of two ciphertexts gives the same result as the encryption of the product of the respective plaintexts. This can render it vulnerable to a *chosen ciphertext attack*.

To combat these vulnerabilities, practical implementations of RSA use something called *padding*. This involves adding nonsense data to the message  $m$  before encryption. The padding ensures that the

plaintext is less insecure and that the encrypted message will have a larger range of possible ciphertexts. However, as the padding will take up some additional bits of the message, the un-padded message has to be somewhat smaller.

As we implemented “plain” RSA in this assignment, no padding is added to the message before encryption. This means our encryption is vulnerable to the above attacks that take advantages of the properties of the basic RSA algorithm, and Charlie may be able to exploit these weaknesses.

Our implementation also relies on the program loading the users and their keys from a .txt file. This requires the file to be somehow passed between the parties if they are using different computers. If Charlie was able to intercept this file, he would have access to the private keys contained within and could use this to break the encryption.

## Our experience

In this project we have followed the plain RSA algorithm. Despite being relatively easy to implement the code, it proved hard to produce novel improvements. As RSA encryption strength relies on key size, and with minimum recommendations being moved to at least 2048 bits (Barker and Dang, 2015), we aimed for keys with 4096 bits length. To accomplish such dimension, we used data types unknown to us until then, such as BigInteger. BigInteger’s data structure does not allow normal arithmetic operations; instead require the use of functions provided in its library to perform mathematic operations, which we have learned for this implementation.

The development of this project was organic, with a constant search for enhancements. Our first version was based only on functions, which proved to be hard to develop on and due to an inflexible code. With this in mind, we produced a second version of our implementation, this time using classes per task. Due to the use of modularized code, this version showed to be flexible and more scalable than our first version.

Our experience producing this code was challenging and enhanced us as developers. This project revealed that a constant look for enrichment allowed us to constructively reassess our product and improve it. By doing so, our upgraded version is flexible enough to be developed into other projects and needs, resulting in better code and development practices than the first originally done for submission.

# Bibliography

- Andreas Baumhof (2019). Breaking RSA Encryption – an Update on the State-of-the-Art. [online] QuintessenceLabs. Available at: <https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art/> [Accessed 21 Mar. 2022].
- Barker, E. and Dang, Q. (2015). Recommendation for Key Management. National Institute of Standards and Technology. [online] Available at: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf> [Accessed 15 Mar. 2022].
- DanielPocock.com. (2013). RSA Key Sizes: 2048 or 4096 bits? [online] Available at: <https://danielpocock.com/rsa-key-sizes-2048-or-4096-bits/> [Accessed 25 Mar. 2022].
- design215.com. (n.d.). ASCII and UTF-8 2-byte Characters. [online] Available at: <https://design215.com/toolbox/ascii-utf8.php> [Accessed 25 Mar. 2022].
- docs.oracle.com. (n.d.). BigInteger (Java Platform SE 7 ). [online] Available at: <https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html> [Accessed 7 Mar. 2022].
- Emerging Technology from the arXiv (2019). How a Quantum Computer Could Break 2048-bit RSA Encryption in 8 Hours. [online] MIT Technology Review. Available at: <https://www.technologyreview.com/2019/05/30/65724/how-a-quantum-computer-could-break-2048-bit-rsa-encryption-in-8-hours/> [Accessed 25 Mar. 2022].
- GeeksforGeeks. (2021). Java Program to Implement the RSA Algorithm. [online] Available at: <https://www.geeksforgeeks.org/java-program-to-implement-the-rsa-algorithm/> [Accessed 11 Mar. 2022].
- Giann Mok, C. and Wen Chuah, C. (2019). An Intelligence Brute Force Attack on RSA Cryptosystem. Communications in Computational and Applied Mathematics, 1(1).
- Lake, J. (2018). What is RSA encryption and how does it work? | Comparitech. [online] Comparitech. Available at: <https://www.comparitech.com/blog/information-security/rsa-encryption/> [Accessed 16 Mar. 2022].
- Marchenkova, A. (2015). Break RSA encryption with this one weird trick. [online] Quantum Bits. Available at: <https://medium.com/quantum-bits/break-rsa-encryption-with-this-one-weird-trick-d955e3394870> [Accessed 25 Mar. 2022].
- Mezher, A.E. (2018). Enhanced RSA Cryptosystem based on Multiplicity of Public and Private Keys. International Journal of Electrical and Computer Engineering (IJECE), 8(5), p.3949.
- Oracle.com. (2018). Serializable (Java Platform SE 7 ). [online] Available at: <https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html> [Accessed 8 Mar. 2022].
- Raghupathi, K. (2016). Creating an Executable JAR in IntelliJ IDEA. [online] blog.karthicr.com. Available at: <https://blog.karthicr.com/posts/2016/07/10/creating-an-executable-jar-in-intellij-idea/> [Accessed 22 Mar. 2022].
- tutorialspoint.com (2019). Java Serialization. [online] www.tutorialspoint.com. Available at: [https://www.tutorialspoint.com/java/java\\_serialization.htm](https://www.tutorialspoint.com/java/java_serialization.htm) [Accessed 17 Mar. 2022].
- Wikipedia Contributors (2019). RSA (cryptosystem). [online] Wikipedia. Available at: [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) [Accessed 10 Feb. 2022].

www.atthemminute.com. (2020). Unicode vs ASCII | at the minute. [online] Available at: <https://www.atthemminute.com/article/character-encodings-what-is-ascii-unicode-and-utf-8> [Accessed 25 Mar. 2022].

www.ibm.com. (2018). UCS-2 and its relationship to Unicode (UTF-16). [online] Available at: <https://www.ibm.com/docs/en/i/7.1?topic=unicode-ucs-2-its-relationship-utf-16> [Accessed 25 Mar. 2022].



## Appendix

### Time tracker:

<b>Total number of hours spent</b>	<b>38</b>
Hours spent for algorithm design	5
Hours spent for programming	20
Hours spent for writing report	10
Hours spent for testing	3

### Code:

The final source files can be found in the submission folder, under FinalVersion, src.

The final JAR file can be found in the submission folder, under FinalVersion.

The initial version of the program can be found in the submission folder, under version1.