JAVA PROTOTYPE FOR RSA ENCYPTION AND DECRYPTION

Brahath Shet, Umar Sultan, Aerhan Srirangan
33563315 33615754 33610411
bshet001@gold.ac.uk , usult001@gold.ac.uk ,
asrir001@gold.ac.uk

Contents

RSA is a popular public-key cryptosystem that facilitates safe communication between two parties. This is an outline of the development of a Java-based software prototype that uses streamlined examples and algorithms to show how the RSA encryption and decryption techniques operate. This prototype replicates a situation in which Alice sends an encrypted message to Bob, who decrypts it, with Charlie having the potential to intercept the message and steal the intended information. We will discuss the program's structure, the algorithms that were employed, and prospective upgrades that are all described in the publication. The fundamental features of the RSA algorithm, key generation, encryption, and decryption are shown via our java prototype presented here. These generated keys are what we use to encrypt and decrypt our messages when communicating.

To start, we need to generate two large primes. The reasons for this, is because RSA is a prime example of a one-way function. The basic gist of a one-way function is that they are easy to calculate one way, but extremely difficult to calculate the other. Think of it like stirring sugar into a cup of tea, stirring the sugar in is easy but it is next to impossible to recreate the sugar from the tea. This is what makes up the basis for the security of RSA, that it is one-way encryption. The encryption being made of up of a product of primes, means you need to factorise this product if you do not know the primes it is made up of, and this, even with modern computers can take months if not years to compute. You could still put in these computational resources and time to find those primes, but by the time a “Charlie” would finally crack this encryption, the keys will have changed and rendered this cracked information useless.

Now that we have a basic understanding of what RSA is and how it works, lets dive in deeper and take a closer look the keys we use are generated, used to encrypt our messages to form a cipher text, and then used to decrypt this cipher back into plain text.

Algorithms

- Random Key Generation:

All communication that uses RSA makes use of two pairs of keys, public and private keys. Each person has a private and public key. Both types of keys are made by performing various computation on prime numbers.

To start, our key generators create two random large primes p and q, based on the number of bits being used for encryption. These two primes are multiplied (p * q), and the resulting product is called is called ‘n’. Next, the resulting variable n is used to calculate ‘r,’ `which is basically φ (n). φ (n) basically, just means that both our primes p and q had one subtracted from them before they were multiplied, translating to r = (p – 1) * (q – 1). We now move onto calculating the two variables that are the most difficult to produce for our key generation, ‘e’ and ‘d’. We start by finding e since it is necessary for calculating d. e is basically a large number that falls in the range of 1 to r (non-inclusive). e does not need to be a prime number; it can be any composite number in the range given that it satisfies the condition of having no factors in common with r. This makes e a co-prime or r. The best way to determine if the number we’re choosing for e is a co-prime of r is find the GCD (greatest common denominator) of the two numbers. If the greatest common denominator between both numbers is one, that means they are co-prime as they have no factors in common. In addition to this, we will choose numbers from the end of the range i.e. From r backwards, checking each number until we find a co-prime. This way we can also satisfy the condition of e being a large random number. Now to calculate our last variable, d. We can calculate d using the equation $(e * d) \bmod r = 1$. Rearranging this, we end up with $d = (1 / e) \bmod \varphi$. To compute this, we will perform extended Euclidean division.

Having at last calculated all our variables we will carefully dispose of variables p, q and r, leaving us with d, e and n. d will be kept private to use as a private key while e and n will be published as the public key pair. It is in this fashion that keys are generated for each person to be used in communication.

- RSA Encryption Algorithm:

When Alice is going to send Bob a message, her message must first be encrypted so it can safely travel over unprotected channels. To do this, Alice first looks up the public key for Bob. She retrieves this from Bob through the unprotected channel. As previously discussed, we don't need to worry about Charlie intercepting Bob's public key. There is nothing Charlie can do with the public keys alone in an appropriate amount of time.

Once Alice has received Bob's public key, she encrypts the message she's trying to send Bob using his public key. The resulting cipher text after encryption is sent to Bob over the channel. The way this cipher text 'c' is hashed is using the formula $m^e \bmod n$, where e and n are the values contained in the Bob's public key. We discussed previously how these values were generated.

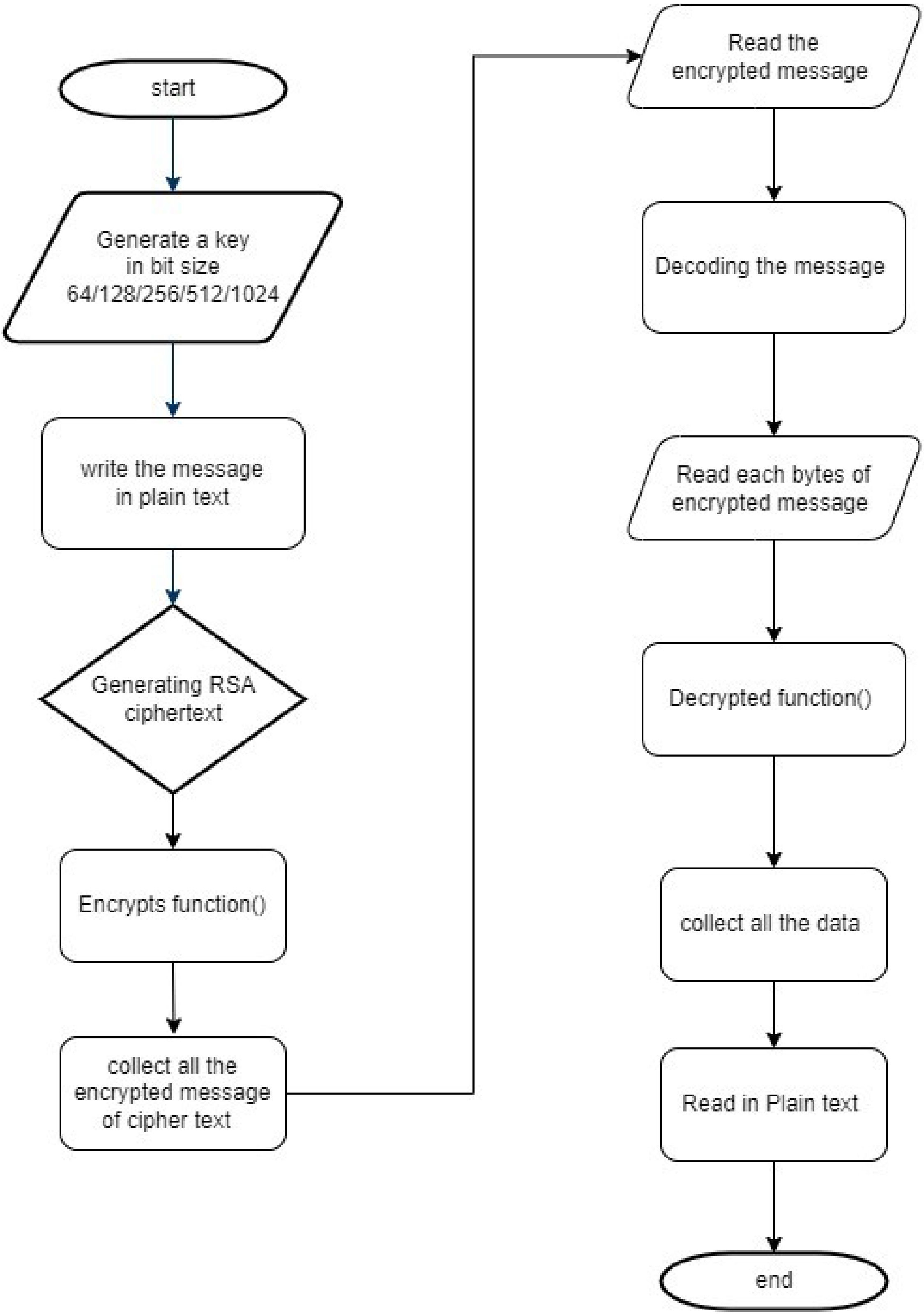
- RSA Decryption Algorithm:

When Bob receives a message from Alice over the unsecure channel, he receives it in the form of cipher text. In this form, he cannot read the message. To extract any meaning from it at all, or to simply convert it to plain text, he needs to decrypt the cipher. This decryption can only be performed by Bob as his private key is necessary for this process. It is for this very reason; we don't need to worry about Charlie intercepting the cipher text. The private key necessary for decryption was never made public (or published) and exists solely in Bob's computer system.

To decrypt the cipher text 'c' when received. He computes the plain text message 'm' using the formula $c^d \bmod n$, where d is Bob's private key and n is one part of his public. Applying this to the cipher will return the plain text.

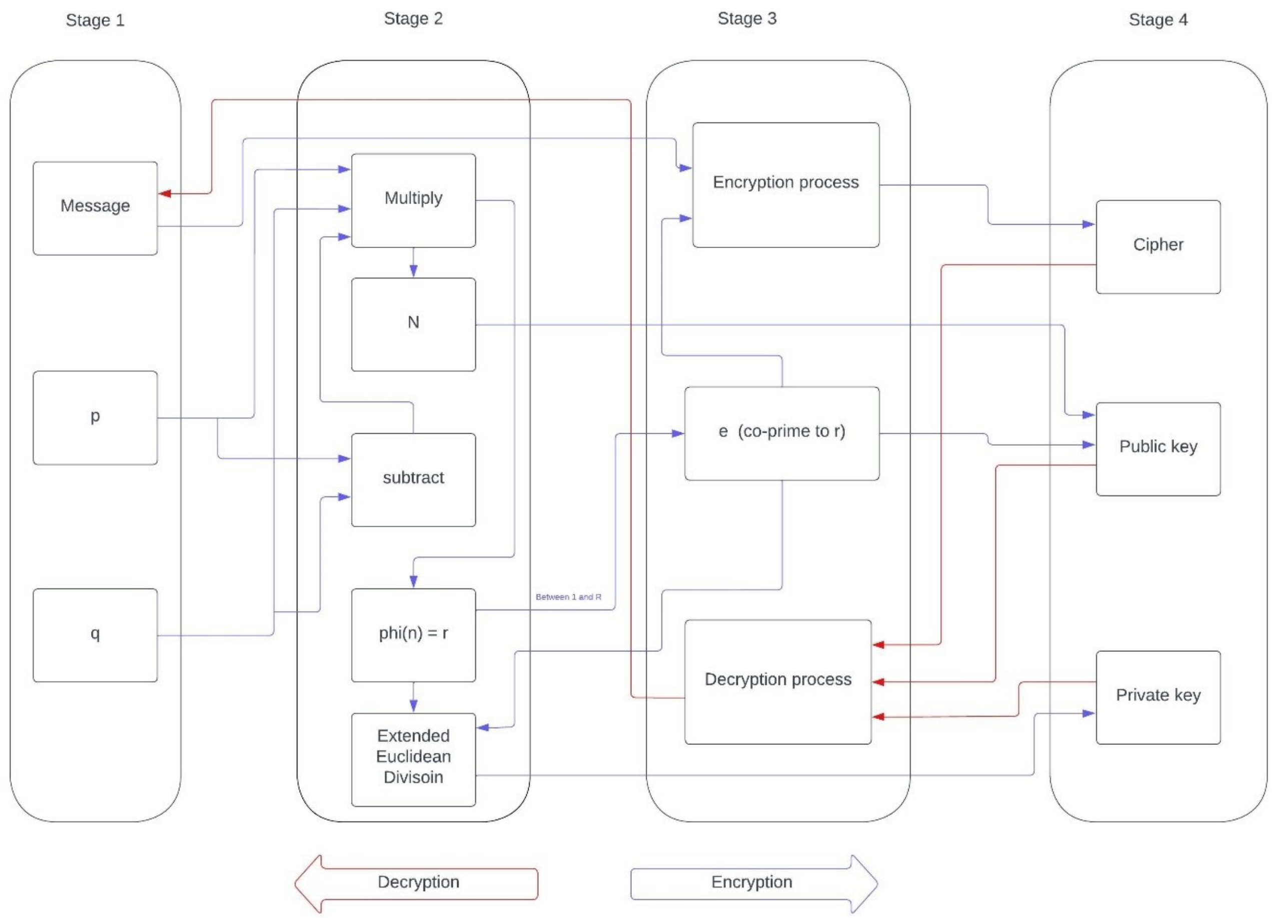
Note, mod stands for the modulus operator.

- Flowchart of our implementation of RSA


start
Generate a key in bit size 64/128/256/512/1024
write the message in plain text
Generating RSA ciphertext
Encrypts function()
collect all the encrypted message of cipher text
Read the encrypted message
Decoding the message
Read each bytes of encrypted message
Decrypted function()
collect all the data
Read in Plain text
end

Design

Block Diagram depicting RSA in various stages.

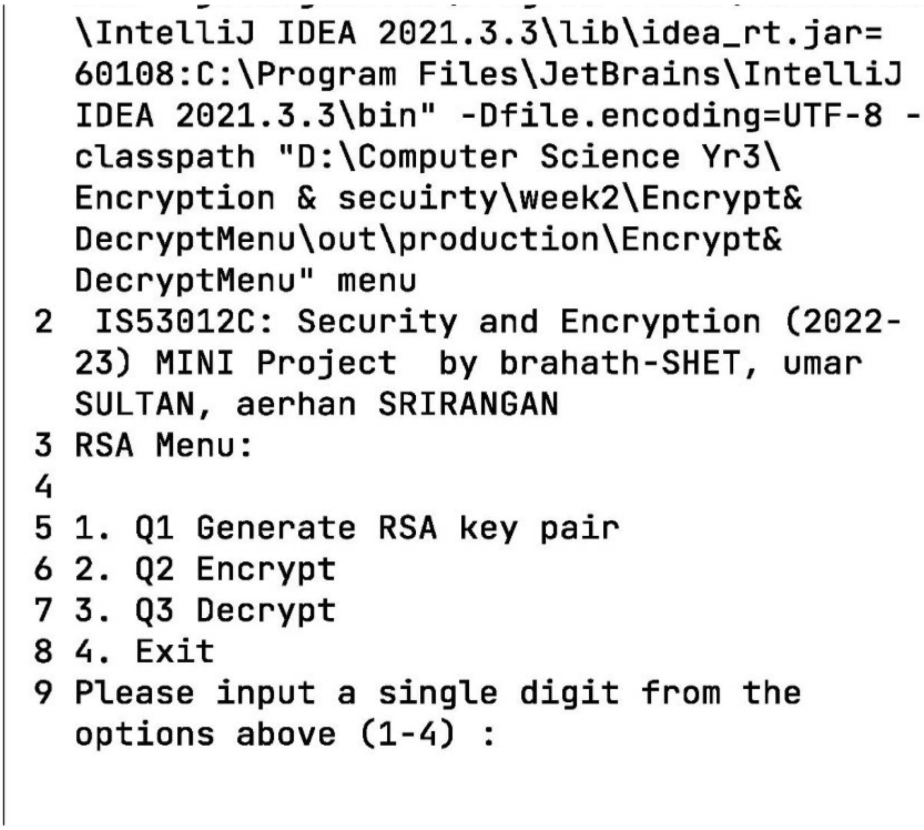


Demonstration

Program Structure

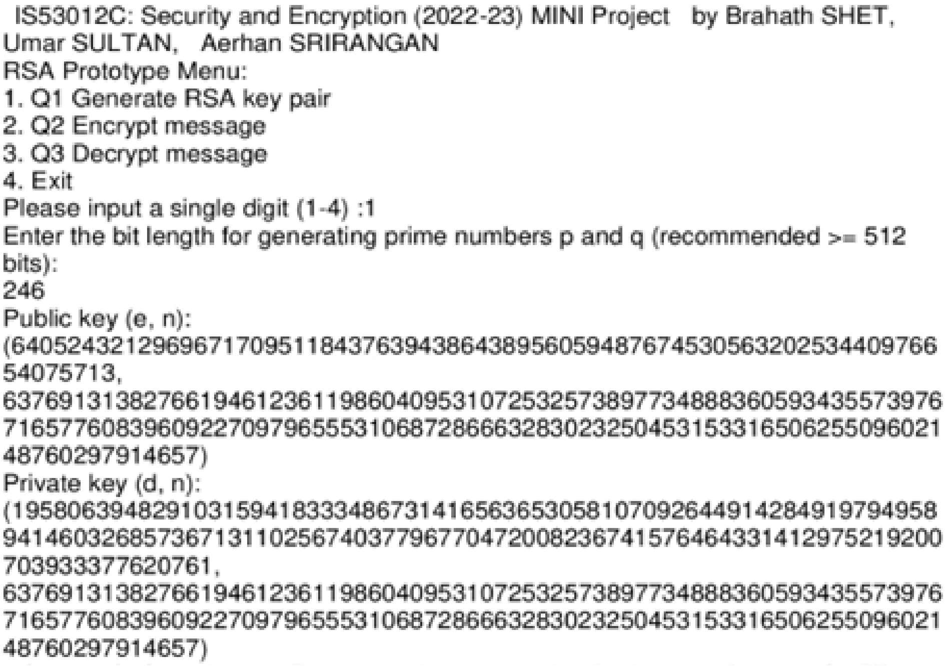
The prototype can be divided into the following:

- Demonstration of the RSA Menu.



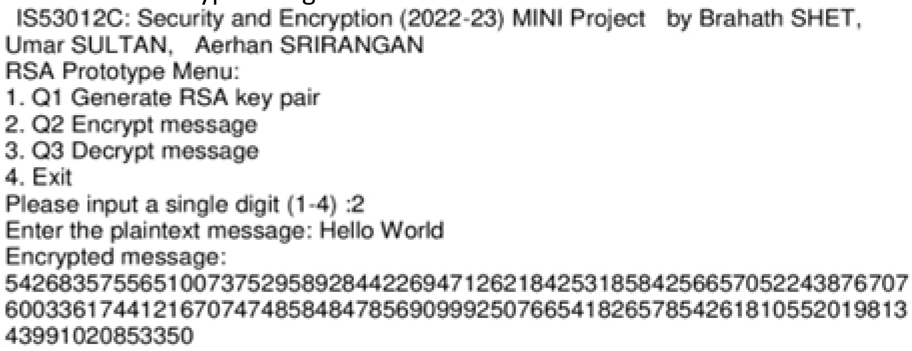
```
\IntelliJ IDEA 2021.3.3\lib\idea_rt.jar=
60108:C:\Program Files\JetBrains\IntelliJ
IDEA 2021.3.3\bin" -Dfile.encoding=UTF-8 -
classpath "D:\Computer Science Yr3\
Encryption & secuirty\week2\Encrypt&
DecryptMenu\out\production\Encrypt&
DecryptMenu" menu
 IS53012C: Security and Encryption (2022-
23) MINI Project  by brahath-SHET, umar
SULTAN, aerhan SRIRANGAN
RSA Menu:

1. Q1 Generate RSA key pair
2. Q2 Encrypt
3. Q3 Decrypt
4. Exit
Please input a single digit from the
options above (1-4) :
```

- Demonstration of the crypto random key generator



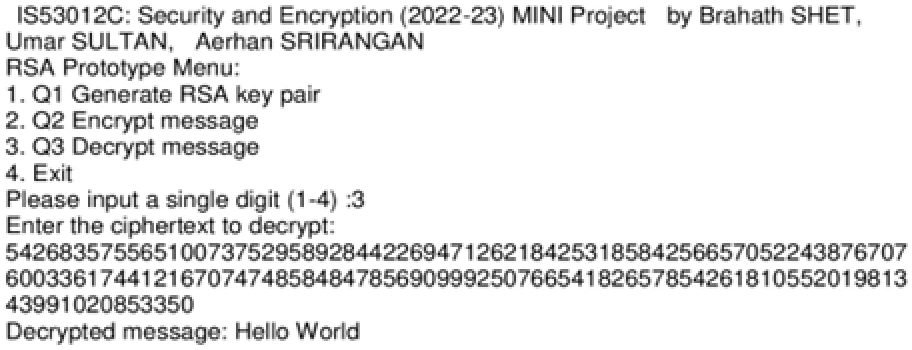
```
 IS53012C: Security and Encryption (2022-23) MINI Project   by Brahath SHET,
Umar SULTAN,   Aerhan SRIRANGAN
RSA Prototype Menu:
1. Q1 Generate RSA key pair
2. Q2 Encrypt message
3. Q3 Decrypt message
4. Exit
Please input a single digit (1-4) :1
Enter the bit length for generating prime numbers p and q (recommended >= 512
bits):
246
Public key (e, n):
(6405243212969671709511843763943864389560594876745305632025344097 66
54075713,
637691313827661946123611986040953107253257389773488836059343557 3976
716577608396092270979655531068728666328302325045315331650625509 6021
48760297914657)
Private key (d, n):
(195806394829103159418333486731416563653058107092644914284919794958
941460326857367131102567403779677047200823674157646433141297521920 0
703933377620761,
637691313827661946123611986040953107253257389773488836059343557 3976
716577608396092270979655531068728666328302325045315331650625509 6021
48760297914657)
```

- Demonstration of RSA encryption algorithm.



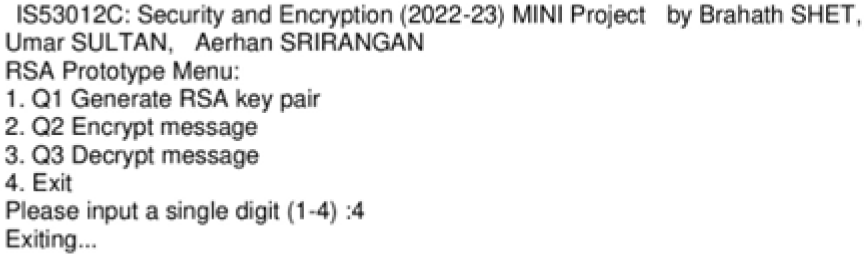
```
 IS53012C: Security and Encryption (2022-23) MINI Project   by Brahath SHET,
Umar SULTAN,   Aerhan SRIRANGAN
RSA Prototype Menu:
1. Q1 Generate RSA key pair
2. Q2 Encrypt message
3. Q3 Decrypt message
4. Exit
Please input a single digit (1-4) :2
Enter the plaintext message: Hello World
Encrypted message:
5426835755651007375295892844226947126218425318584256657052243876707
6003361744121670747485848478569099925076654182657854261810552019813
43991020853350
```

- Demonstration of the RSA decryption algorithm.



```
 IS53012C: Security and Encryption (2022-23) MINI Project   by Brahath SHET,
Umar SULTAN,   Aerhan SRIRANGAN
RSA Prototype Menu:
1. Q1 Generate RSA key pair
2. Q2 Encrypt message
3. Q3 Decrypt message
4. Exit
Please input a single digit (1-4) :3
Enter the ciphertext to decrypt:
5426835755651007375295892844226947126218425318584256657052243876707
6003361744121670747485848478569099925076654182657854261810552019813
43991020853350
Decrypted message: Hello World
```

- Showing a simple user interface to simulate the exit of the program.



```
 IS53012C: Security and Encryption (2022-23) MINI Project   by Brahath SHET,
Umar SULTAN,   Aerhan SRIRANGAN
RSA Prototype Menu:
1. Q1 Generate RSA key pair
2. Q2 Encrypt message
3. Q3 Decrypt message
4. Exit
Please input a single digit (1-4) :4
Exiting...
```

Discussion

RSA is certainly a versatile encryption protocol. It can be implemented relatively quickly, and it is relatively secure in terms of the encryption it provides for information. It also makes public key distribution for its consumers quite easy. For all its advantages, it still falls quite short of being a perfect encryption solution. Let's discuss some of these some of its pitfalls and why they exist.

Going in order, let's start by looking at public key distribution. While distribution is very easy, this brings into question the authenticity of these keys. Since public keys are published and, well, are Public. You request the public key from the person you wish to communicate with, since in RSA you must use the recipients public key to encrypt your message for it to be decrypted by their private key. This system is what ensures this communication can only be read by the two parties having it. But how do you ensure the authenticity of the recipient's public key? Since all communication occurs over an unsecure channel, who’s to say a ‘Charlie’ doesn’t intercept the requested public key first and switch it out for a fake public key. To ensure the authenticity of the keys we use for RSA, it makes sense to have a third party regulate these keys for us. It will be their job to ensure the authenticity of the requested keys, making sure the requested key does in fact belong to who it’s requested from.

For our second point, as quick and easy RSA is to implement, does not speak to how computationally intensive the process can be. RSA requires a fair amount of resources for encryption and decryption. We not only generate all the variables necessary for creating our keys, but we also expend resources using those keys on the senders end to compute the encryption and then again on the receiver's side to compute the decryption. This makes computation especially intense for the receiver as they are also the one to generate the keys used for the encryption and decryption process.

Lastly, RSA only makes sense to use for encrypting certain kinds of data. This attributed to the fact of how it was inherently built to serve as a basis of communication encryption online. Generating keys and swapping them between the communicating parties to encrypt and decrypt data. This model with make no sense on say, a voting machine, where the data you're encrypting is coming from multiple different sources and should only be able to be decrypted by one central authority. RSA would not be able to efficiently encrypt such data because it was built to cater to a fundamentally scenario.

Future Developments

- To increase the communication's overall security, use extra security mechanisms like digital signatures and secure key exchange methods.
- Enhancing the user interface for better user experience using GUIs.
- Able to lower bit size number that can send small text for encrypting the message. Since the decryption is not capable to give a correct output.

In summary, the Java RSA encryption and decryption offers a practical illustration of the RSA algorithm's simplification. Users may better grasp the encryption, decryption, and key generation procedures by modelling a communication situation between Alice, Bob, and Charlie. The program's usefulness and security may be improved in the future, making it a useful instructional tool for comprehending and studying RSA encryption.

Reference:

Pu I. (2023), Security and Encryption Module: Lecture Notes, London: Goldsmiths, University of London.