IS53012C Security and Encryption Coursework: Assignment 1

Team Members:

1: Ailya Sayed

2: Freddy Rodriguez Figueroa

3: Joshua Ollivierre

4: Jerrod Sim

Introduction

The purpose of this group project is to demonstrate RSA encryption, decryption, and key generation in Java using simplified algorithms. The intended goals of the software are to:

- Demonstrate how two prime numbers p and q are generated.
- Illustrate the generation of a random number e, ensuring that:
  - $0 < e < r$, where $r = (p - 1)(q - 1)$.
  - e has no common factors with r (i.e., $\gcd(e, r) = 1$).
- Generate the private key d.
- Generate the public key (e, n), where $n = p \cdot q$.

Additionally, the program will showcase a scenario where RSA encryption would not be secure. This will be achieved by allowing users to input parameters that lead to such a problematic state.

Finally, the implementation includes a simple user interface to simulate a communication scenario. This scenario demonstrates how:

- Alice encrypts a message,
- Bob decrypts the ciphertext to recover the plaintext,
- Charlie may intercept the data flow and attempt unauthorised access.

By researching the RSA algorithm and aligning our implementation with the assigned task, we have developed a program that fulfils all the outlined objectives.

The RSA Algorithm

RSA is based on the concepts of modular arithmetic and number theory, particularly the unique properties of prime numbers, modular exponentiation, and Euler's totient function. Modular arithmetic is arithmetic with a focus on the remainder when a number is divided by another [1], and number theory is the branch of mathematics concerned with properties of the positive integers. Determining if a large number is prime or composite, and multiplying two numbers, are easy. Factorising a large number which is the product of two large primes, is very difficult. To find the factors of a composite number n which is the product of two large primes and has about 640 binary bits (approximately 200 decimal digits) is an impossible task using current computer power.

To generate the public and private keys, we must first choose two large primes, p and q, such that $n = p \cdot q$ forms a large composite number. Next, Euler's totient function ϕ, is calculated. This is calculated as $\phi(n) = (p - 1)(q - 1)$. This represents the number of integers less than n that are coprime with n. Then, a large random number e is chosen. e must be between 1 and ϕ, and has no common factor with ϕ. Subsequently, the private key d is calculated using the values previously established, using the formula $(e \cdot d) = 1 \pmod{\phi}$. Finally, we can dispose of the values p, q and r and publish the public key pair (e, n). d must be kept private.

To encrypt a message m, the sender uses the recipient's public key (e, n). The message m must be an integer such that $0 \leq m < n$. If m is larger than n, it is divided into blocks. The ciphertext c is computed as $c \equiv m^e \pmod{n}$. The recipient deciphers the ciphertext c using their private key d using the formula $m \equiv c^d \pmod{n}$. By the properties of modular arithmetic and the RSA setup, the operation correctly recovers the plaintext m.

There are several potential vulnerabilities in RSA. Firstly, if the public exponent e and message m are small, the encryption process may not apply the modulo n operation. This will make it easy to compute the original message m from the ciphertext c. Secondly, the security of RSA depends on the difficulty of factoring n into its prime components p and q. If an attacker finds a way to efficiently factor n, they could calculate the private key d. Smaller key sizes are more vulnerable to this attack. Additionally, if the prime numbers p and q used to generate the keys are too close to each other or not random enough, attackers may be able to guess or calculate them. Finally, side-channel attacks may be performed, where physical information from the device performing the RSA algorithm mat be exploited, such as how long it takes to do a calculation or how much power it uses.

Our Implementation

To create our project, we decided on minimum hardware and software requirements to ensure the code would run on a vast majority of devices. In terms of hardware, we ensured our simplification of the RSA algorithm would be compatible with systems that have at least 2GB of RAM and a basic processor. This is appropriate as our key size is limited to 512 bits, and there is a lack of advanced security features. In addition to this, we also decided on minimum software requirements. Our project is written in Java, so systems must have Java Development Kit 8 or higher, and a simple text editor or an Integrated Development Environment.

To divide the tasks equally amongst the group members, we wrote a brief outline of the tasks required to complete the implementation and divided these based on individual skills and availability. The first team member was responsible for initialising the Java project and implementing the RSAKeyGenerator class, including secure and insecure key generation. The second team member implemented the RSA class to handle the encryption and decryption of numerical messages. The third team member developed the main method to handle user inputs, simulate scenarios, and add checks. The final team member tested the code and documented said tests in the testing section.

Our simplification of the RSA algorithm was due to system requirements, time scale for the project, and intention of developing code that represents the simplifications taught during the module. In a robust version of RSA, prime numbers p and q are typically over 1024 bits, assuming n is at least 2048 bits [3], and p and q are exactly half the size of the modulus n [4]. In our version, we use BigInteger.probablePrime, which limits the prime size to 512 bits for the secure scenario. Additionally, p, q and ϕ(n) are not disposed of for demonstration purposes. Secondly, in actual RSA, the public exponent e is commonly 65,537 [5], for efficiency and security. In our implementation, we start with 65,537 but fall back to smaller integers if the gcd condition fails. Thirdly, messages are padded in real-world implementations [6], whereas our messages are encrypted and decrypted directly with no padding. Finally, to demonstrate the RSA algorithm's functionality, we allow the code to demonstrate insecure scenarios, such as allowing small p and q values, and randomly choosing e within ϕ(n).

We encountered various challenges while trying to create a suitable implementation. One example of this is ensuring the coprimality of e and ϕ(n). To counter this, we added checks that use the gcd function and implemented fallback mechanisms to find a suitable e value. Another challenge we faced was the possibility of the user inputting a message larger than n. To counter this, we included a check to warn the user and demonstrate that the decryption process would fail. Finally, we struggled finding a way to simulate an insecure attack. Our solution was to implement a demonstration where an attacker, Charlie, calculates the private key d if p and q are known, showing the vulnerability of RSA when the primes are exposed.

Cryptorandom Key Generator

```
// Generate two distinct large primes
            p = BigInteger.probablePrime(bitLength, random);
            do {
                q = BigInteger.probablePrime(bitLength, random);
            } while (q.equals(p));

            // Compute d
            d = e.modInverse(phi);

            // Generate n
            n = p.multiply(q);
            phi =
p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
```

RSA Encryption and Decryption Algorithms

```
// Demonstrate encryption/decryption with large primes
        System.out.println("\n=== Secure Communication ===");
        System.out.print("Alice, enter a message (integer): ");
        String input = scanner.nextLine();
        BigInteger message;
        try {
            message = new BigInteger(input);
        } catch (NumberFormatException ex) {
            System.out.println("Invalid integer. Exiting.");
            scanner.close();
            return;
        }

        // Alice encrypts
        BigInteger secureCipher = secureRSA.encrypt(message);
        System.out.println("Alice sends ciphertext: " + secureCipher);
```

Communication Simulation

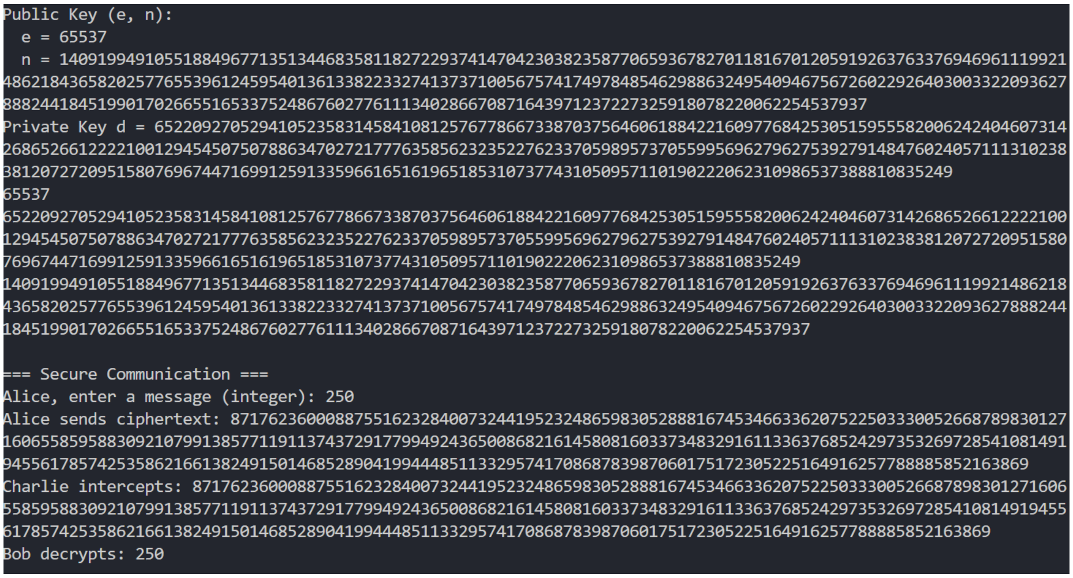
```
// Alice encrypts
        BigInteger secureCipher = secureRSA.encrypt(message);
        System.out.println("Alice sends ciphertext: " + secureCipher);

        // Charlie intercepts
        System.out.println("Charlie intercepts: " + secureCipher);

        // Bob decrypts
        BigInteger secureDecrypted = secureRSA.decrypt(secureCipher);
        System.out.println("Bob decrypts: " + secureDecrypted);
```

Testing

1. **Functional Testing**: Test cases for key generation, encryption, and decryption.

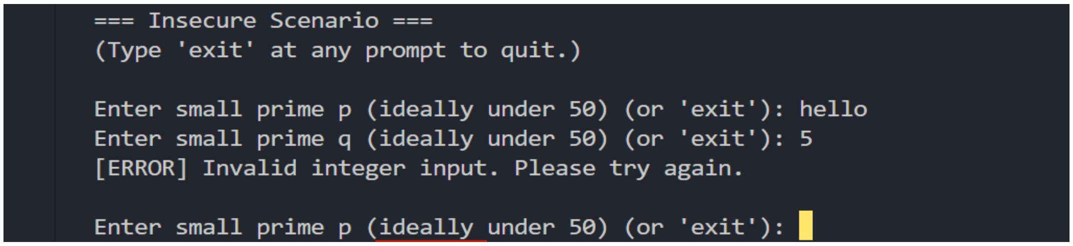


```
Public Key (e, n):
  e = 65537
  n = 1409199491055188496771351344683581182722937414704230382358770659367827011816701205919263763376946961119921
4862184365820257765539612459540136133822332741373710056757417497848546298863249540946756726022926403003322093627
8882441845199017026655165337524867602776111340286670871643971237227325918078220062254537937
Private Key d = 652209270529410523583145841081257677866733870375646061884221609776842530515955582006242404607314
2686526612222100129454507507886347027217776358562323522762337059895737055995696279627539279148476024057111310238
3812072720951580769674471699125913359661651619651853107377431050957110190222062310986537388810835249
65537
6522092705294105235831458410812576778667338703756460618842216097768425305159555820062424046073142686526612222100
1294545075078863470272177763585623235227623370598957370559956962796275392791484760240571113102383812072720951580
769674471699125913359661651619651853107377431050957110190222062310986537388810835249
1409199491055188496771351344683581182722937414704230382358770659367827011816701205919263763376946961119921486218
4365820257765539612459540136133822332741373710056757417497848546298863249540946756726022926403003322093627888244
1845199017026655165337524867602776111340286670871643971237227325918078220062254537937

=== Secure Communication ===
Alice, enter a message (integer): 250
Alice sends ciphertext: 8717623600088755162328400732441952324865983052888167453466336207522503330052668789830127
1606558595883092107991385771191137437291779949243650086821614580816033734832916113363768524297353269728541081491
945561785742535862166138249150146852890419944485113329574170868783987060175172305225164916257788885852163869
Charlie intercepts: 87176236000887551623284007324419523248659830528881674534663362075225033300526687898301271606
5585958830921079913857711911374372917799492436500868216145808160337348329161133637685242973532697285410814919455
61785742535862166138249150146852890419944485113329574170868783987060175172305225164916257788885852163869
Bob decrypts: 250
```

2. **User Input Validation**: Ensuring incorrect inputs are handled appropriately.

- **The input value must be numeric and prime**

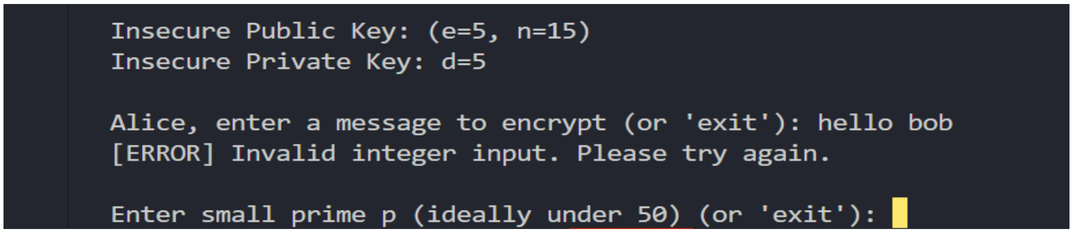


```
=== Insecure Scenario ===
(Type 'exit' at any prompt to quit.)

Enter small prime p (ideally under 50) (or 'exit'): hello
Enter small prime q (ideally under 50) (or 'exit'): 5
[ERROR] Invalid integer input. Please try again.

Enter small prime p (ideally under 50) (or 'exit'):
```

- **The message sent by Alice must be integer**



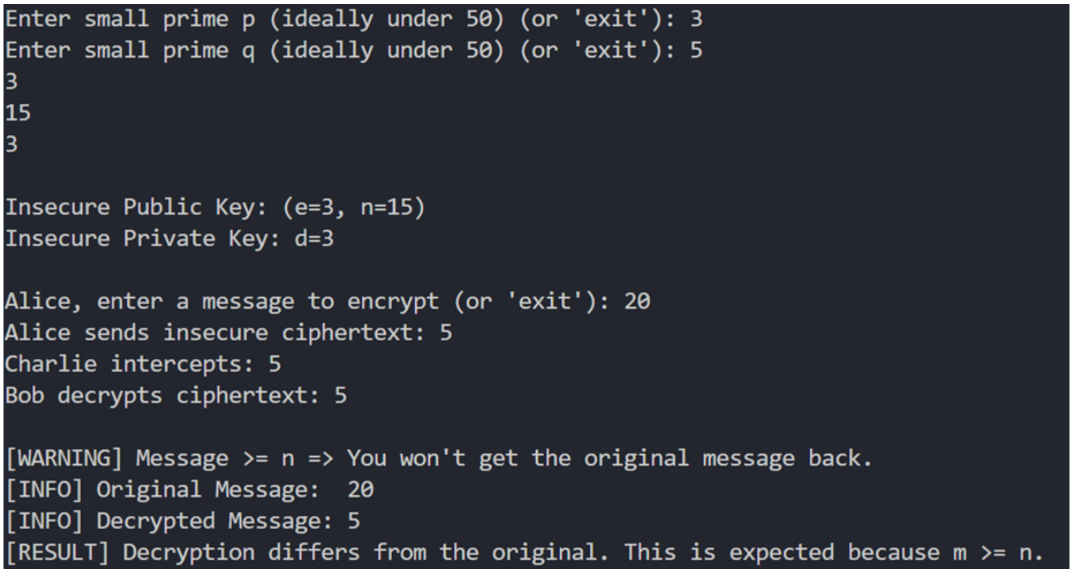
```
Insecure Public Key: (e=5, n=15)
Insecure Private Key: d=5

Alice, enter a message to encrypt (or 'exit'): hello bob
[ERROR] Invalid integer input. Please try again.

Enter small prime p (ideally under 50) (or 'exit'):
```

3. **Edge Cases**: Demonstration of a scenario where the RSA program doesn’t work securely

- **When the message is >= n**



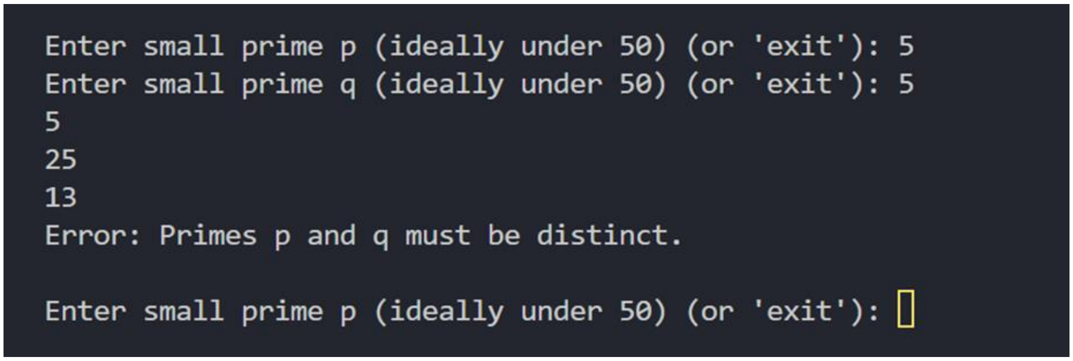
```
Enter small prime p (ideally under 50) (or 'exit'): 3
Enter small prime q (ideally under 50) (or 'exit'): 5
3
15
3

Insecure Public Key: (e=3, n=15)
Insecure Private Key: d=3

Alice, enter a message to encrypt (or 'exit'): 20
Alice sends insecure ciphertext: 5
Charlie intercepts: 5
Bob decrypts ciphertext: 5

[WARNING] Message >= n => You won't get the original message back.
[INFO] Original Message:  20
[INFO] Decrypted Message: 5
[RESULT] Decryption differs from the original. This is expected because m >= n.
```

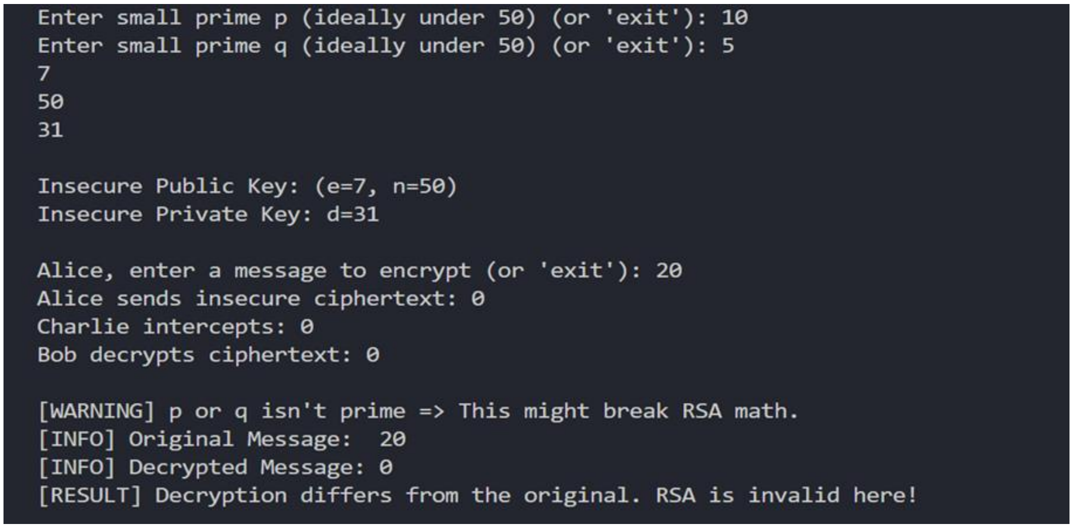
- **When the two prime numbers are equal**



```
Enter small prime p (ideally under 50) (or 'exit'): 5
Enter small prime q (ideally under 50) (or 'exit'): 5
5
25
13
Error: Primes p and q must be distinct.

Enter small prime p (ideally under 50) (or 'exit'):
```

- **If the numbers are not prime**



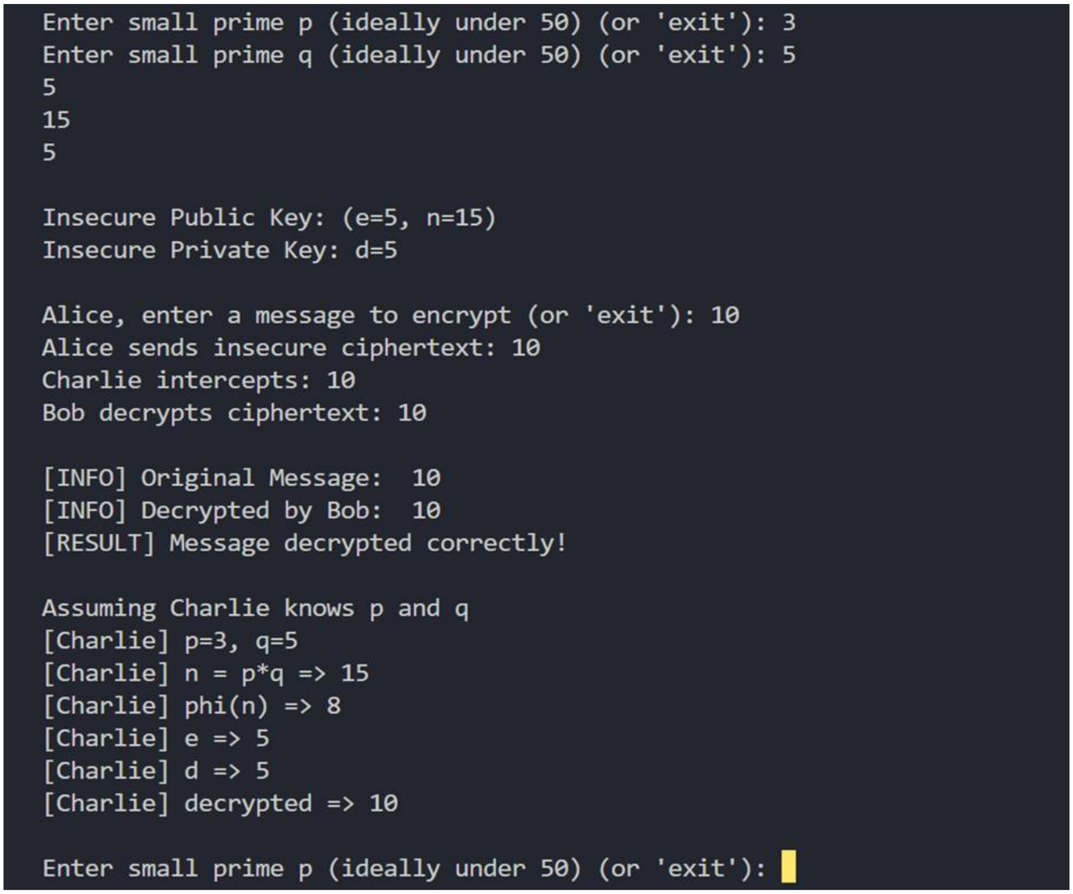
```
Enter small prime p (ideally under 50) (or 'exit'): 10
Enter small prime q (ideally under 50) (or 'exit'): 5
7
50
31

Insecure Public Key: (e=7, n=50)
Insecure Private Key: d=31

Alice, enter a message to encrypt (or 'exit'): 20
Alice sends insecure ciphertext: 0
Charlie intercepts: 0
Bob decrypts ciphertext: 0

[WARNING] p or q isn't prime => This might break RSA math.
[INFO] Original Message:  20
[INFO] Decrypted Message: 0
[RESULT] Decryption differs from the original. RSA is invalid here!
```

- **Assuming Charlie knows the values of p and q**



```
Enter small prime p (ideally under 50) (or 'exit'): 3
Enter small prime q (ideally under 50) (or 'exit'): 5
5
15
5

Insecure Public Key: (e=5, n=15)
Insecure Private Key: d=5

Alice, enter a message to encrypt (or 'exit'): 10
Alice sends insecure ciphertext: 10
Charlie intercepts: 10
Bob decrypts ciphertext: 10

[INFO] Original Message:  10
[INFO] Decrypted by Bob:  10
[RESULT] Message decrypted correctly!

Assuming Charlie knows p and q
[Charlie] p=3, q=5
[Charlie] n = p*q => 15
[Charlie] phi(n) => 8
[Charlie] e => 5
[Charlie] d => 5
[Charlie] decrypted => 10

Enter small prime p (ideally under 50) (or 'exit'): 
```

Results and Discussion

The key success of this project lies in our implementation of the RSA key generation, encryption, and decryption. The prototype demonstrated a secure simulation by using large random prime numbers, and successfully encrypting and decrypting messages. This was an important goal, as without this, we would be unable to develop our project further. Another success was our implementation of the insecure scenario examples. This helped to demonstrate the potential vulnerabilities in the RSA algorithm, hence educating the user on the importance of choosing appropriate values and ensuring their communication channels are sufficiently protected against potential attackers. Additionally, we collaborated well as a team and were able to help each other if we encountered any problems. Our structured approach to task delegation and efficient workflow allowed us to contribute equally and produce a well-made implementation of RSA.

While the project achieved its objectives, there are areas for improvement. Given the time scale of the project, we were unable to add every feature we had aspired to include. One example of this is using a GUI to simulate a conversation. This could have made the demonstration more accessible and visually appealing, allowing our software to be used for teaching purposes. Another improvement would be using more complex and secure values for our RSA key generation. This would allow for more complex messages to be encrypted and decrypted, potentially allowing users to input alphanumeric characters to be used in the examples. Finally, we could have implemented a function that allows the user to “intercept” a ciphertext and allow them to try and decode it themselves, simulating the perspective of an attacker.

Conclusion

This project successfully demonstrated the implementation of RSA encryption, decryption, and key generation in Java, achieving the primary objectives outlined in the introduction. By simulating both secure and insecure scenarios, we highlighted the principles of the RSA algorithm and its potential vulnerabilities, thereby offering a practical understanding of how RSA operates. Despite its successes, there is room for improvement in areas such as enhancing the user interface, increasing the complexity and security of key generation, and adding interactive features to simulate an attacker’s perspective. These enhancements could broaden the software’s application as an educational tool and improve its overall functionality.

In conclusion, the project met its goals and provided valuable insights into RSA encryption. It also highlighted the importance of careful parameter selection and secure implementation practices to ensure robustness of cryptographic systems. Future iterations of the project could build on this foundation to explore more advanced features and use cases, further adding to its educational and practical value.

References

[1] ‘Number theory | Definition, Topics, & History | Britannica’. Accessed: Jan. 05, 2025. [Online]. Available: https://www.britannica.com/science/number-theory

[2] ‘Modular arithmetic | Number Theory, Congruence & Algorithms | Britannica’. Accessed: Jan. 05, 2025. [Online]. Available: https://www.britannica.com/science/modular-arithmetic

[3] ‘RSA (cryptosystem)’, Wikipedia. Jan. 07, 2025. Accessed: Jan. 09, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&oldid=1267990112

[4] T. Pornin, ‘Answer to “For RSA, is there a specified size range for p and q when calculating N?”’, Cryptography Stack Exchange. Accessed: Jan. 09, 2025. [Online]. Available: https://crypto.stackexchange.com/a/48293

[5] ‘65,537’, Wikipedia. Oct. 31, 2024. Accessed: Jan. 09, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=65,537&oldid=1254438835

[6] P. B. B. O. FRSE, ‘So How Does Padding Work in RSA?’, ASecuritySite: When Bob Met Alice. Accessed: Jan. 09, 2025. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/so-how-does-padding-work-in-rsa-6b34a123ca1f

Appendix

https://github.com/ailyaaaaaa/rsa-prototype/

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Scanner;

public class RSAPrototype {
    // RSA Key Generator
    static class RSAKeyGenerator {
        private BigInteger p;
        private BigInteger q;
        private BigInteger n;
        private BigInteger phi;
        private BigInteger e;
        private BigInteger d;
        private int bitLength = 512;

        // Generates a secure RSA key pair using large random primes.
        public void generateKeys() {
            SecureRandom random = new SecureRandom();

            // Generate two distinct large primes
            p = BigInteger.probablePrime(bitLength, random);
            do {
                q = BigInteger.probablePrime(bitLength, random);
            } while (q.equals(p));

            n = p.multiply(q);
            phi =
             p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

            // Common choice for e
            e = BigInteger.valueOf(65537);

            if (!phi.gcd(e).equals(BigInteger.ONE)) {
                // If not coprime, pick another small odd integer
                e = BigInteger.valueOf(3);
                while (phi.gcd(e).intValue() > 1) {
                    e = e.add(BigInteger.TWO);
                }
            }

            // Compute d
            d = e.modInverse(phi);
            System.out.println("\n=== Secure RSA (Large Primes) ===");
            System.out.println("Public Key (e, n):");
            System.out.println("  e = " + e);
            System.out.println("  n = " + n);
            System.out.println("Private Key d    = " + d);
        }

        // Demonstrates an "insecure" setup (small primes).
        public void setInsecureKeys(BigInteger p, BigInteger q) {
            this.p = p;
            this.q = q;
            this.n = p.multiply(q);
            this.phi =
             p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));

            // Pick a random e that is coprime with phi(n).
            SecureRandom random = new SecureRandom();
            do {
                // Generate a random e in the range (1, phi).
                // We ensure e > 1 and < phi, and gcd(e, phi) = 1
                e = new BigInteger(phi.bitLength(), random);
            } while ( e.compareTo(BigInteger.ONE) <= 0
                   || e.compareTo(phi) >= 0
                   || !phi.gcd(e).equals(BigInteger.ONE) );

            d = e.modInverse(phi);
        }

        public BigInteger getPublicKeyE() {
            return e;
        }

        public BigInteger getPublicKeyN() {
            return n;
        }

        public BigInteger getPrivateKeyD() {
            return d;
        }
    }

    // RSA (encrypt/decrypt)
    static class RSA {
        private final BigInteger e;
        private final BigInteger d;
```

```java
        private final BigInteger n;

        public RSA(BigInteger e, BigInteger d, BigInteger n) {
            this.e = e;
            this.d = d;
            this.n = n;
        }

        public BigInteger encrypt(BigInteger message) {
            return message.modPow(e, n);
        }

        public BigInteger decrypt(BigInteger ciphertext) {
            return ciphertext.modPow(d, n);
        }
    }

    // Utility: check primality
    private static boolean isPrime(BigInteger x) {
        // For demonstration, the built-in isProbablePrime() is typically enough
        return x.isProbablePrime(100);
    }

    // Main Simulation
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        RSAKeyGenerator keyGen = new RSAKeyGenerator();

        // Generate Secure RSA
        keyGen.generateKeys();
        RSA secureRSA = new RSA(keyGen.getPublicKeyE(),
                                keyGen.getPrivateKeyD(),
                                keyGen.getPublicKeyN());

        // Demonstrate encryption/decryption with large primes
        System.out.println("\n=== Secure Communication ===");
        System.out.print("Alice, enter a message (integer): ");
        String input = scanner.nextLine();
        BigInteger message;
        try {
            message = new BigInteger(input);
        } catch (NumberFormatException ex) {
            System.out.println("Invalid integer. Exiting.");
            scanner.close();
            return;
        }
```

```
// Alice encrypts
BigInteger secureCipher = secureRSA.encrypt(message);
System.out.println("Alice sends ciphertext: " + secureCipher);

// Charlie intercepts
System.out.println("Charlie intercepts: " + secureCipher);

// Bob decrypts
BigInteger secureDecrypted = secureRSA.decrypt(secureCipher);
System.out.println("Bob decrypts: " + secureDecrypted);

//Insecure Scenario in a loop
System.out.println("\n=== Insecure Scenario ===");
System.out.println("(Type 'exit' at any prompt to quit.)");

while (true) {
    // Prompt for p
    System.out.print("\nEnter small prime p (ideally under 50) (or
     'exit'): ");
    String pInput = scanner.nextLine();
    if ("exit".equalsIgnoreCase(pInput)) {
        System.out.println("Exiting insecure scenario...");
        break;
    }

    // Prompt for q
    System.out.print("Enter small prime q (ideally under 50) (or 'exit'): ");
    String qInput = scanner.nextLine();
    if ("exit".equalsIgnoreCase(qInput)) {
        System.out.println("Exiting insecure scenario...");
        break;
    }

    // Convert to BigInteger
    BigInteger p, q;
    try {
        p = new BigInteger(pInput);
        q = new BigInteger(qInput);
    } catch (NumberFormatException ex) {
        System.out.println("[ERROR] Invalid integer input. Please try
            again.");
        continue;  // go back to top of while-loop
    }

    // Build the insecure keys
```

```
     keyGen.setInsecureKeys(p, q);

     // Print out the public/private keys (even if p or q is not prime)
     BigInteger e = keyGen.getPublicKeyE();
     BigInteger n = keyGen.getPublicKeyN();
     BigInteger d = keyGen.getPrivateKeyD();

     //validation p and q are not distinct
     if (p.equals(q)) {
         System.out.println("Error: Primes p and q must be distinct.");
         continue;//
     }


     System.out.println("\nInsecure Public Key: (e=" + e + ", n=" + n +
")");
     System.out.println("Insecure Private Key: d=" + d);

     // Prompt for the message
     System.out.print("\nAlice, enter a message to encrypt (or 'exit'): ");
     String msgInput = scanner.nextLine();
     if ("exit".equalsIgnoreCase(msgInput)) {
         System.out.println("Exiting insecure scenario...");
         break;
     }

     // Convert to BigInteger
     BigInteger insecureMsg;
     try {
         insecureMsg = new BigInteger(msgInput);
     } catch (NumberFormatException ex) {
         System.out.println("[ERROR] Invalid integer input. Please try
again.");
         continue;  // loop again
     }

     // Encrypt the message
     RSA insecureRSA = new RSA(e, d, n);
     BigInteger insecureCipher = insecureRSA.encrypt(insecureMsg);

     // Print ciphertext
     System.out.println("Alice sends insecure ciphertext: " +
      insecureCipher);
     System.out.println("Charlie intercepts: " + insecureCipher);

     BigInteger bobDecrypted = insecureRSA.decrypt(insecureCipher);
```

```
System.out.println("Bob decrypts ciphertext: " + bobDecrypted);

// Compare bobDecrypted with insecureMsg to see if they're the same
boolean matchesOriginal = bobDecrypted.equals(insecureMsg);

// Check #1: Are p and q truly prime?
if (!isPrime(p) || !isPrime(q)) {
    // Show that it doesn't decrypt properly
    System.out.println("\n[WARNING] p or q isn't prime => This might
 break RSA math.");
    System.out.println("[INFO] Original Message:  " + insecureMsg);
    System.out.println("[INFO] Decrypted Message: " + bobDecrypted);
    if (!matchesOriginal) {
        System.out.println("[RESULT] Decryption differs from the
 original. RSA is invalid here!");
    }
    continue; // loop again
}

// Check #2: Is message >= n?
if (insecureMsg.compareTo(n) >= 0) {
    System.out.println("\n[WARNING] Message >= n => You won't get the
 original message back.");
    System.out.println("[INFO] Original Message:  " + insecureMsg);
    System.out.println("[INFO] Decrypted Message: " + bobDecrypted);
    if (!matchesOriginal) {
        System.out.println("[RESULT] Decryption differs from the
 original. This is expected because m >= n.");
    }
    continue; // loop again
}

// If we get here, p & q are prime and message < n
// => should decrypt correctly in normal RSA
System.out.println("\n[INFO] Original Message:  " + insecureMsg);
System.out.println("[INFO] Decrypted by Bob:  " + bobDecrypted);
if (matchesOriginal) {
    System.out.println("[RESULT] Message decrypted correctly!");
} else {
    System.out.println("[RESULT] Decryption differs from the
        original? Unexpected for valid RSA!");
}

// Demonstrate Charlie's perspective
System.out.println("\nAssuming Charlie knows p and q ");
// Recompute phi(n) from p and q
```

```
            BigInteger charliePhi =
             (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));

            BigInteger charlieE = e;
            BigInteger charlieD = charlieE.modInverse(charliePhi);

            // Now Charlie can decrypt the same ciphertext as Bob
            BigInteger charlieDecrypted = insecureCipher.modPow(charlieD, n);

            System.out.println("[Charlie] p=" + p + ", q=" + q);
            System.out.println("[Charlie] n = p*q => " + n);
            System.out.println("[Charlie] phi(n) => " + charliePhi);
            System.out.println("[Charlie] e => " + charlieE);
            System.out.println("[Charlie] d => " + charlieD);
            System.out.println("[Charlie] decrypted => " + charlieDecrypted);

            // Loop again to allow new p, q, or exit
        }
        scanner.close();
        System.out.println("Done.");
    }
}
```