Walchand College of Engineering, Sangli (Government Aided Autonomous Institute)

AY 2025-26

Course Information Programme B.Tech. (Computer Science and Engineering)

Final Year B. Tech., Sem VII Class,

Semester Course Code 6CS451

Course Name Cryptography and Network Security Lab

22510064 PARSHWA HERWADE BATCH 1

EXPERIMENT NO. 7

Title - Implementation of RSA Algorithm

Objectives:

1. To understand the theoretical foundation of the RSA algorithm
2. • Explore the mathematical principles behind RSA, including prime numbers, Euler's theorem, and modular exponentiation.

2. To implement the RSA algorithm from scratch using a programming language (e.g., Python, Java, C++)

• Develop modules for key generation, encryption, and decryption.

3. To ensure secure key generation

• Implement a method to generate large, random prime numbers and compute public and private key pairs.

4. To demonstrate the encryption and decryption process

• Encrypt a plaintext message using the public key and decrypt it using the private key.

5. To validate the correctness and performance of the implementation

• Test the algorithm with different input sizes and ensure accurate decryption of encrypted messages.

6. To explore the limitations and possible optimizations of RSA

• Analyze time complexity, key size limitations, and potential vulnerabilities in naive implementations.

Problem Statement: In the current era of digital communication and data exchange, ensuring the confidentiality, integrity, and authenticity of information is critically important. Traditional symmetric encryption methods face challenges in secure key distribution and scalability. To address these issues, public key cryptography provides a robust solution where encryption and decryption are performed using separate keys. The RSA algorithm is one of the most widely used public key cryptosystems that enables secure data transmission over insecure networks. However, understanding and implementing RSA from scratch requires a deep understanding of number theory, modular arithmetic, and key generation processes. There is a need for a clear, educational, and functional implementation of the RSA algorithm that demonstrates its core principles and operations, including key generation, encryption, and decryption.

Theory

RSA (Rivest–Shamir–Adleman) is an asymmetric cryptosystem.

- **Key Generation**
  1. Choose two large primes p and q.
  2. Compute $n = p \times q$.
  3. Compute $\phi(n) = (p-1)(q-1)$.
  4. Choose public exponent e such that $1 < e < \phi(n)$ and gcd(e, φ(n)) = 1.
  5. Compute private key d as modular inverse of e modulo φ(n): $d \equiv e^{-1} \pmod{\phi(n)}$.
- **Encryption**
  Ciphertext $C = M^e \bmod n$.
- **Decryption**
  Plaintext $M = C^d \bmod n$.

Security rests on the hardness of factoring large n.

Equipment/Tools

- JDK 17+
- Java IDE or command-line javac & java

Procedure / Steps

1. Prompt the user for two large primes ppp and qqq.
2. Compute nnn, ϕ(n).
3. Prompt for a public exponent e (or choose one) that is coprime to ϕ(n).
4. Compute the modular inverse to find d.
5. Ask user for a plaintext integer message MMM where 0<M<n0 < M < n0<M<n.
6. Encrypt and display ciphertext.
7. Decrypt and verify the result.

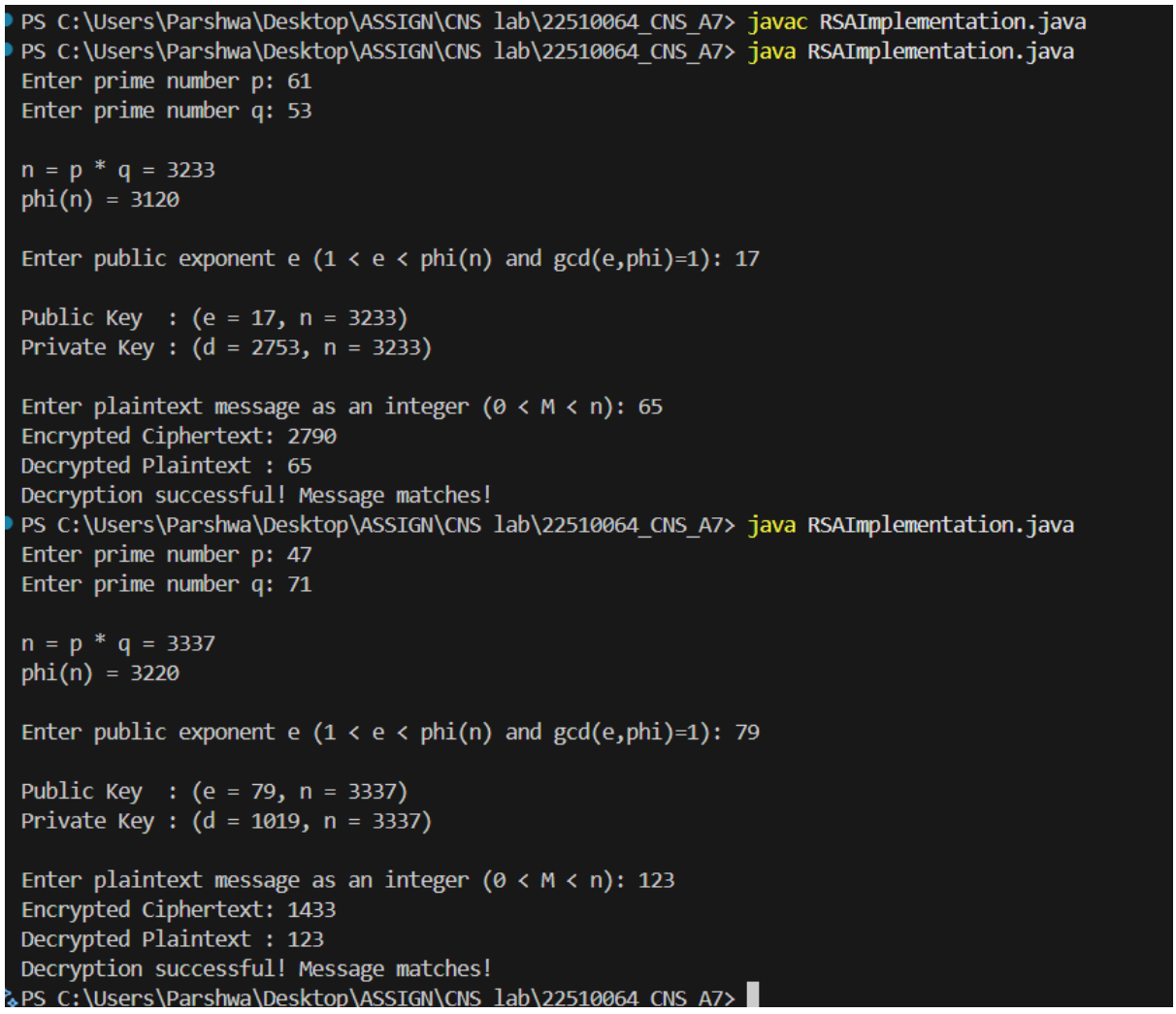
CODE:

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

public class RSAImplementation {
    private static BigInteger gcd(BigInteger a, BigInteger b) {
        return b.equals(BigInteger.ZERO) ? a : gcd(b, a.mod(b));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter prime number p: ");
        BigInteger p = sc.nextBigInteger();

        System.out.print("Enter prime number q: ");
        BigInteger q = sc.nextBigInteger();

        BigInteger n = p.multiply(q);
```

```
        BigInteger phi =
(p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
        System.out.println("\nn = p * q = " + n);
        System.out.println("phi(n) = " + phi);

        System.out.print("\nEnter public exponent e (1 < e < phi(n)
and gcd(e,phi)=1): ");
        BigInteger e = sc.nextBigInteger();
        while (e.compareTo(BigInteger.ONE) <= 0 ||
               e.compareTo(phi) >= 0 ||
               !gcd(e, phi).equals(BigInteger.ONE)) {
            System.out.print("Invalid e. Enter again: ");
            e = sc.nextBigInteger();
        }

        BigInteger d = e.modInverse(phi);
        System.out.println("\nPublic Key  : (e = " + e + ", n = " +
n + ")");
        System.out.println("Private Key : (d = " + d + ", n = " + n
+ ")");

        System.out.print("\nEnter plaintext message as an integer (0
< M < n): ");
        BigInteger M = sc.nextBigInteger();
        while (M.compareTo(BigInteger.ZERO) <= 0 || M.compareTo(n)
>= 0) {
            System.out.print("Message out of range. Enter again: ");
            M = sc.nextBigInteger();
        }

        BigInteger C = M.modPow(e, n);
        System.out.println("Encrypted Ciphertext: " + C);

        BigInteger decrypted = C.modPow(d, n);
        System.out.println("Decrypted Plaintext : " + decrypted);

        if (M.equals(decrypted)) {
            System.out.println("Decryption successful! Message
matches!");
        } else {
            System.out.println("Decryption failed.");
        }
```

```
        sc.close();
    }
}
```

OUTPUT:



Observations & Conclusion

- RSA correctly encrypts and decrypts messages when keys are generated properly.
- Larger primes increase security but also computation time.
- Key generation and modular exponentiation are the most time-intensive steps.

- Limitations:
    - Pure RSA is slow for large data—commonly used only to encrypt symmetric keys.
    - Security relies on difficulty of factoring large n.