**RSA ALGORITHM**

**A PROJECT REPORT**

**NETWORK SECURITY (CS1702)**

Submitted By

***Kusuma Eswaravaka (CS22B1012)***

To

**Dr. Narendran Rajagopalan**

***Associate Professor***

***Department of Computer Science and Engineering***

***National Institute of Technology Puducherry***

***Karaikal – 609609***

****

**DEPARTMENT OF**

**COMPUTER SCIENCE AND ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY**

**KARAIKAL – 609 609**

**APRIL 2025**

**TABLE OF CONTENTS**

**ABSTRACT Page No**

1. **INTRODUCTION 3**
2. **IMPLEMENTATION 3**
3. **SAMPLE OUTPUT 5**
4. **EXPLANATION 5**
5. **CONCLUSION 6**

**INTRODUCTION:**

This report documents the implementation of an RSA (Rivest-Shamir-Adleman) encryption and decryption tool in Python. RSA is a widely used asymmetric cryptographic algorithm that ensures secure data transmission by using a pair of keys: a public key for encryption and a private key for decryption. The security of RSA relies on the mathematical difficulty of factoring large prime numbers. This report covers the key concepts, implementation details, and security considerations of RSA.

**IMPLEMENTATION:**

import random

import math

def is\_prime(n):

    """Check if a number is prime using optimized trial division."""

    if n <= 1:

        return False

    elif n <= 3:

        return True

    elif n % 2 == 0 or n % 3 == 0:

        return False

    i = 5

    while i \* i <= n:

        if n % i == 0 or n % (i + 2) == 0:

            return False

        i += 6

    return True

def gcd(a, b):

    """Compute greatest common divisor using Euclid's algorithm."""

    while b != 0:

        a, b = b, a % b

    return a

def modular\_inverse(e, phi):

    """Find modular inverse using extended Euclidean algorithm (incorporated version)."""

    original\_phi = phi

    x1, x2 = 1, 0

    y1, y2 = 0, 1

    while phi != 0:

        quotient = e // phi

        e, phi = phi, e % phi

        x1, x2 = x2, x1 - quotient \* x2

        y1, y2 = y2, y1 - quotient \* y2

    if e != 1:

        raise ValueError("Modular inverse doesn't exist")

    # Ensure the result is positive

    return x1 % original\_phi

def generate\_keypair(p, q):

    """Generate RSA public and private key pair."""

    if not (is\_prime(p) and is\_prime(q)):

        raise ValueError("Both numbers must be prime.")

    if p == q:

        raise ValueError("p and q cannot be equal")

    n = p \* q

    phi = (p - 1) \* (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1

    e = random.randrange(2, phi)

    while gcd(e, phi) != 1:

        e = random.randrange(2, phi)

    d = modular\_inverse(e, phi)

    return ((e, n), (d, n))

def rsa\_encrypt(pk, plaintext):

    """Encrypt plaintext using RSA public key."""

    key, n = pk

    cipher = [pow(ord(char), key, n) for char in plaintext]

    return cipher

def rsa\_decrypt(pk, ciphertext):

    """Decrypt ciphertext using RSA private key."""

    key, n = pk

    plain = [chr(pow(char, key, n)) for char in ciphertext]

    return ''.join(plain)

def get\_prime\_input(prompt):

    """Get and validate prime number input from user."""

    while True:

        try:

            num = int(input(prompt))

            if not is\_prime(num):

                print("Please enter a valid prime number.")

                continue

            return num

        except ValueError:

            print("Invalid input. Please enter an integer.")

def main():

    print("RSA Encryption/Decryption Tool")

    print("-----------------------------")

    # Get validated prime numbers

    p = get\_prime\_input("Enter first prime number: ")

    q = get\_prime\_input("Enter second different prime number: ")

    while p == q:

        print("Numbers must be different.")

        q = get\_prime\_input("Enter second different prime number: ")

    print("\nGenerating keys...")

    public\_key, private\_key = generate\_keypair(p, q)

    print(f"Public Key (e, n): {public\_key}")

    print(f"Private Key (d, n): {private\_key}")

    message = input("\nEnter a message to encrypt: ")

    # Encryption

    encrypted\_msg = rsa\_encrypt(public\_key, message)

    print("\nEncrypted Message:", ' '.join(map(str, encrypted\_msg)))

    # Decryption

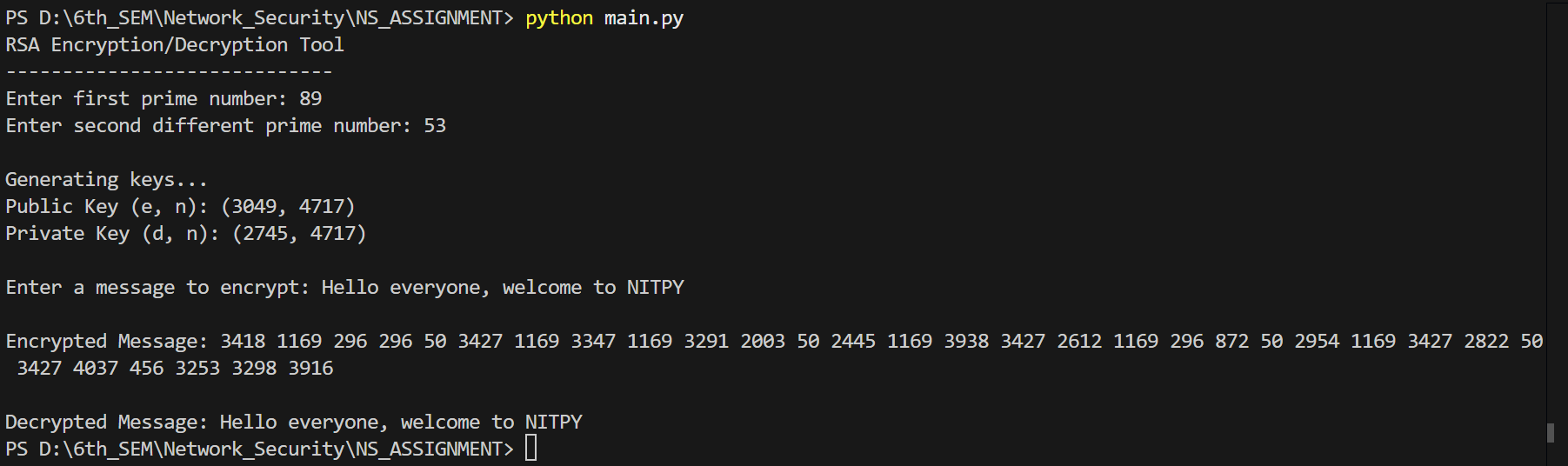
    decrypted\_msg = rsa\_decrypt(private\_key, encrypted\_msg)

    print("Decrypted Message:", decrypted\_msg)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**SAMPLE OUTPUT:**



**EXPLANATION:**

1. ***Prime Number Check*** (*is\_prime function*)

This function checks if a number is prime using an optimized trial division method:

* Handles simple cases first (numbers ≤ 3)
* Checks divisibility by 2 and 3
* Checks divisibility for numbers of form 6k ± 1 up to √n

2. ***Greatest Common Diviso***r (*gcd* function)

Implements Euclid's algorithm to find the GCD of two numbers, which is essential for finding a suitable public exponent.

### 3. *Modular Inverse* (*modular\_inverse*function)

Uses the extended Euclidean algorithm to find the modular inverse, which is needed to compute the private key. Ensures the result is positive.

### 4. *Key Generation* (*generate\_keypair* function)

1. Validates that inputs p and q are distinct primes
2. Computes n = p × q and φ(n) = (p-1)(q-1)
3. Randomly selects public exponent e that is coprime with φ(n)
4. Computes private exponent d as the modular inverse of e mod φ(n)

### 5. *Encryption* (*rsa\_encrypt* function)

Encrypts each character in the plaintext by:

1. Converting character to ASCII value
2. Computing ciphertext = (plaintexte) mod n for each character

### 6. *Decryption* (*rsa\_decrypt* function)

Decrypts each number in ciphertext by:

1. Computing plaintext = (ciphertextd) mod n for each number
2. Converting result back to character

### 7. User Interface (main function)

1. Gets validated prime numbers from user
2. Generates and displays key pairs
3. Encrypts user message and displays ciphertext
4. Decrypts ciphertext and displays original message

## Security Considerations

1. **Prime Selection**: The implementation requires manual input of primes, which in a real application would be automatically generated large primes.
2. **Message Handling**: Encrypts each character individually (not secure for production; should use proper padding schemes like OAEP).
3. **Key Size**: The security depends on the size of the primes used (small primes like in the example are for demonstration only).
4. **Randomness**: Uses Python's random module which is not cryptographically secure (should use secrets module for production).

**CONCLUSION:**

This implementation demonstrates the core mathematical principles of RSA while highlighting areas that would need enhancement for production-grade security.