Experiment No. 4

Title: RSA Cipher

Batch: B-2 **Roll No.: 16010422234** **Experiment No.: 4**

Results: (Program printout as per the format)

```
from math import gcd
import random
def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True
def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += m0
    return x1
def generate_keys(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)
    d = mod_inverse(e, phi)
    public_key = (e, n)
    private_key = (d, n)
    return public_key, private_key
def string_to_int_list(text):
    return [ord(char) for char in text]
def int_list_to_string(int_list):
    return ''.join(chr(num) for num in int_list)
def encrypt(int_list, public_key):
    e, n = public_key
    encrypted_message = [pow(char, e, n) for char in int_list]
    return encrypted_message
def decrypt(encrypted_message, private_key):
    d, n = private_key
```

```
    decrypted_message = [pow(char, d, n) for char in
encrypted_message]
    return decrypted_message
def print_menu():
    print("\nMenu:")
    print("1. Encrypt a message")
    print("2. Decrypt a message")
    print("3. Exit")
def main():
    public_key = None
    private_key = None
    while True:
        print_menu()
        choice = input("Enter your choice (1/2/3): ")
        if choice == '3':
            print("Exiting.")
            break
        if choice in ['1', '2']:
            if public_key is None or private_key is None:
                p = int(input("Enter prime number p: "))
                q = int(input("Enter prime number q: "))
                if not (is_prime(p) and is_prime(q)):
                    print("Both p and q must be prime numbers.")
                    continue
                public_key, private_key = generate_keys(p, q)
                print("Public Key:", public_key)
                print("Private Key:", private_key)
            if choice == '1':
                plaintext = input("Enter the plaintext to encrypt:
")
                int_list = string_to_int_list(plaintext)
                print("Integer Representation of Plaintext:",
int_list)
                encrypted_message = encrypt(int_list, public_key)
                print("Encrypted Message:", encrypted_message)
            elif choice == '2':
                encrypted_message = list(map(int, input("Enter the
encrypted message as space-separated integers: ").split()))
                decrypted_int_list = decrypt(encrypted_message,
private_key)
                decrypted_text =
int_list_to_string(decrypted_int_list)
                print("Decrypted Text:", decrypted_text)
        else:
            print("Invalid choice. Please enter 1, 2, or 3.")
if __name__ == "__main__":
    main()
```

```
PS C:\Users\chand\Downloads\V SEM\INS> & C:/Users/chand/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/chand/Downloads/V SEM/INS/EXP4/exp4.py"

Menu:
1. Encrypt a message
2. Decrypt a message
3. Exit
Enter your choice (1/2/3): 1
Enter prime number p: 313
Enter prime number q: 787
Public Key: (60851, 246331)
Private Key: (14891, 246331)
Enter the plaintext to encrypt: Hello World!
Integer Representation of Plaintext: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
Encrypted Message: [106658, 161135, 245314, 245314, 17712, 227709, 151837, 17712, 106785, 245314, 181131, 222808]

Menu:
1. Encrypt a message
2. Decrypt a message
3. Exit
Enter your choice (1/2/3): 2
Enter the encrypted message as space-separated integers: 106658 161135 245314 245314 17712 227709 151837 17712 106785 245314 181131 2
22808
Decrypted Text: Hello World!

Menu:
1. Encrypt a message
2. Decrypt a message
3. Exit
Enter your choice (1/2/3): 3
Exiting.
PS C:\Users\chand\Downloads\V SEM\INS>
```

```python
import random

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if 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

# Function to generate a large prime number
def generate_large_prime(key_size):
    while True:
        num = random.getrandbits(key_size)
        if is_prime(num):
            return num

# Function to calculate the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Function to find the modular inverse of e mod phi using the
Extended Euclidean Algorithm
def mod_inverse(e, phi):
    def egcd(a, b):
```

```
        if a == 0:
            return b, 0, 1
        g, x1, y1 = egcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return g, x, y

    g, x, y = egcd(e, phi)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        return x % phi

# Function to generate public and private keys
def generate_keypair(key_size):
    p = generate_large_prime(key_size // 2)  # Typically, use
key_size // 2 for p and q
    q = generate_large_prime(key_size // 2)
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and e is coprime to phi
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)

    d = mod_inverse(e, phi)

    return ((e, n), (d, n))

# Function to sign a message (Non-Repudiation)
def sign(private_key, message):
    d, n = private_key
    signature = [pow(ord(char), d, n) for char in message]
    return signature

# Function to verify a signature
def verify(public_key, message, signature):
    e, n = public_key
    decrypted_signature = ''.join([chr(pow(char, e, n)) for char in
signature])
    return decrypted_signature == message

def main():
    # Take key size as input from the user
    key_size = int(input("Enter the key size in bits: "))

    # Generate key pairs for Alice (sender) and Bob (receiver)
    print("Generating key pairs.")
    alice_public_key, alice_private_key = generate_keypair(key_size)
    bob_public_key, bob_private_key = generate_keypair(key_size)
```

```
    print(f"Alice's Public Key: {alice_public_key}")
    print(f"Alice's Private Key: {alice_private_key}")
    print(f"Bob's Public Key: {bob_public_key}")
    print(f"Bob's Private Key: {bob_private_key}")

    # Input the message
    message = input("Enter the message to send: ")

    # Alice signs the message using her private key
(non-repudiation)
    signature = sign(alice_private_key, message)
    print("Message signed successfully by Alice.")

    # Verify the signature using Alice's public key
    if verify(alice_public_key, message, signature):
        print("Signature verified successfully. Non-repudiation
ensured.")
    else:
        print("Signature verification failed. Non-repudiation could
not be ensured.")

if __name__ == "__main__":
    main()
```

```
PS C:\Users\chand\Downloads\V SEM\INS\EXP4> & C:/Users/chand/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/U
sers/chand/Downloads/V SEM/INS/EXP4/non-repudiation.py"
Enter the key size in bits: 64
Generating key pairs.
Alice's Public Key: (483118608779943429, 651495485589668917)
Alice's Private Key: (624640248072010029, 651495485589668917)
Bob's Public Key: (71524916522204741, 2394112959591041393)
Bob's Private Key: (2266606370580669581, 2394112959591041393)
Enter the message to send: Hi, Bob!
Message signed successfully by Alice.
Signature verified successfully. Non-repudiation ensured.
PS C:\Users\chand\Downloads\V SEM\INS\EXP4> 
```

```
import random

# Optimized function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Use 6k +/- 1 optimization for larger primes
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

```
# Optimized function to generate a large prime number
def generate_large_prime(key_size):
    while True:
        # Generate a random odd number of the specified size
        num = random.getrandbits(key_size) | 1  # Ensures the number
is odd
        if is_prime(num):
            return num

# Function to calculate the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Optimized function to find the modular inverse of e mod phi using
the Extended Euclidean Algorithm
def mod_inverse(e, phi):
    def egcd(a, b):
        if a == 0:
            return b, 0, 1
        g, x1, y1 = egcd(b % a, a)
        return g, y1 - (b // a) * x1, x1

    g, x, _ = egcd(e, phi)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        return x % phi

# Function to generate public and private keys
def generate_keypair(key_size):
    p = generate_large_prime(key_size // 2)  # Typically, use
key_size // 2 for p and q
    q = generate_large_prime(key_size // 2)
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and e is coprime to phi
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)

    d = mod_inverse(e, phi)

    return ((e, n), (d, n))

# Function to encrypt the message
def encrypt(public_key, plaintext):
    e, n = public_key
    ciphertext = [pow(ord(char), e, n) for char in plaintext]
    return ciphertext
```

```
# Function to decrypt the message
def decrypt(private_key, ciphertext):
    d, n = private_key
    plaintext = ''.join([chr(pow(char, d, n)) for char in
ciphertext])
    return plaintext

# Function to sign a message (Non-Repudiation)
def sign(private_key, message):
    d, n = private_key
    signature = [pow(ord(char), d, n) for char in message]
    return signature

# Function to verify a signature
def verify(public_key, message, signature):
    e, n = public_key
    decrypted_signature = ''.join([chr(pow(char, e, n)) for char in
signature])
    return decrypted_signature == message

def main():
    # Take key size as input from the user
    key_size = int(input("Enter the key size in bits: "))

    # Generate key pairs for Alice (sender) and Bob (receiver)
    print("Generating key pairs.")
    alice_public_key, alice_private_key = generate_keypair(key_size)
    bob_public_key, bob_private_key = generate_keypair(key_size)

    print(f"Alice's Public Key: {alice_public_key}")
    print(f"Alice's Private Key: {alice_private_key}")
    print(f"Bob's Public Key: {bob_public_key}")
    print(f"Bob's Private Key: {bob_private_key}")

    # Input the message
    message = input("Enter the message to send: ")

    # Encrypt the message using Bob's public key
    encrypted_message = encrypt(bob_public_key, message)
    print("Encrypted message:", encrypted_message)

    # Convert the encrypted message integers to strings (to avoid
OverflowError)
    encrypted_message_str = ','.join(map(str, encrypted_message))

    # Sign the encrypted message using Alice's private key
(non-repudiation)
    signature = sign(alice_private_key, encrypted_message_str)
    print("Encrypted message signed successfully by Alice.")

    # Verify the signature using Alice's public key
```

```
    if verify(alice_public_key, encrypted_message_str, signature):
        print("Signature verified successfully. Non-repudiation 
ensured.")

        # Decrypt the message using Bob's private key
        decrypted_message = decrypt(bob_private_key, 
encrypted_message)
        print("Decrypted message by Bob:", decrypted_message)
    else:
        print("Signature verification failed. Non-repudiation could 
not be ensured.")

if __name__ == "__main__":
    main()
```

```
PS C:\Users\chand\Downloads\V SEM\INS\EXP4> & C:/Users/chand/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/chand/Downloads/V SEM/INS/EXP4/both.py"
Enter the key size in bits: 64
Generating key pairs.
Alice's Public Key: (7154270148034616123, 13579992167460350759)
Alice's Private Key: (5683806308901304187, 13579992167460350759)
Bob's Public Key: (5122650765427065991, 5853676662059533297)
Bob's Private Key: (5180779958353675151, 5853676662059533297)
Enter the message to send: Hi, Bob!
Encrypted message: [4409486193856752877, 5544817890664678112, 4014237248415405542, 5355690551118180015, 713881183240575860, 1467419842751294506, 3810987649526639054, 3579960693231228216]
Encrypted message signed successfully by Alice.
Signature verified successfully. Non-repudiation ensured.
Decrypted message by Bob: Hi, Bob!
PS C:\Users\chand\Downloads\V SEM\INS\EXP4>
```

Questions:

1. In RSA cryptosystem each plaintext character is presented by the number between 00(A) and 25(Z). The number 26 represents the blank character. Bob wants to send Alice the message "Hello World". So the plaintext is as below, 07 04 11 11 14 26 22 14 17 11 03 . Suppose p=11, q=3. Generate receiver's key pair and show encryption and decryption of the message using RSA cipher.

Given:

$p = 11$

$q = 3$

Step 1: Calculate n

$n = p \times q = 11 \times 3 = 33$

Step 2: Calculate $\phi\ (n)$

$\phi\ (n) = (p - 1) \times (q - 1) = (11 - 1) \times (3 - 1) = 10 \times 2 = 20$

Step 3: Choose e such that $1 < e < \phi\ (n)$ and gcd $(e, \phi\ (n)) = 1$

Let's choose $e = 3$ (commonly used small prime)

Step 4: Calculate d such that $e \times d \equiv 1 \pmod{\phi(n)}$

$3 \times d \equiv 1 \pmod{20}$

$d = 7$(since $3 \times 7 = 21 \equiv 1 \pmod{20}$)

So, the public key is $(e, n) = (3, 33)$ and the private key is $(d, n) = (7, 33)$

Encryption:

Plaintext: "Hello World" => 07 04 11 11 14 26 22 14 17 11 03

Using the public key (3, 33), each number M is encrypted to $C = M^e \pmod{n}$.

$C_1 = 07^3 \pmod{33} = 343 \pmod{33} = 13$

$C_2 = 04^3 \pmod{33} = 64 \pmod{33} = 31$

$C_3 = 11^3 \pmod{33} = 1331 \pmod{33} = 7$

$C_4 = 11^3 \pmod{33} = 1331 \pmod{33} = 7$

$C_5 = 14^3 \pmod{33} = 2744 \pmod{33} = 2$

$C_6 = 26^3 \pmod{33} = 17576 \pmod{33} = 10$

$C_7 = 22^3 \pmod{33} = 10648 \pmod{33} = 14$

$C_8 = 14^3 \pmod{33} = 2744 \pmod{33} = 2$

$C_9 = 17^3 \pmod{33} = 4913 \pmod{33} = 23$

$C_{10} = 11^3 \pmod{33} = 1331 \pmod{33} = 7$

$C_{11} = 03^3 \pmod{33} = 27 \pmod{33} = 27$

So, the ciphertext is: 13 31 7 7 2 10 14 2 23 7 27

Decryption:

Using the private key (7, 33), each number C is decrypted to $M = C^d \pmod{n}$.

$M_1 = 13^7 \pmod{33} = 7$

$M_2 = 31^7 \pmod{33} = 4$

$M_3 = 7^7 \pmod{33} = 11$

$M_4 = 7^7 \pmod{33} = 11$

$M_5 = 2^7 \pmod{33} = 14$

$M_6 = 10^7 \pmod{33} = 26$

$M_7 = 14^7 \pmod{33} = 22$

$M_8 = 2^7 \pmod{33} = 14$

$M_9 = 23^7 \pmod{33} = 17$

$M_{10} = 7^7 \pmod{33} = 11$

$M_{11} = 27^7 \pmod{33} = 3$

So, the decrypted plaintext is: 07 04 11 11 14 26 22 14 17 11 03 ("Hello World")

2. List the attacks on RSA.

- Brute Force Attack: Trying all possible private keys.
- Mathematical Attacks:
  - Factorization Attack: Breaking the RSA by factorizing n into p and q.
  - Timing Attack: Measuring the time taken for decryption to infer the private key.
  - Chosen Ciphertext Attack: The attacker chooses a ciphertext and gets it decrypted to obtain plaintext.
  - Key Size Attack: Using keys that are too small, making them vulnerable to factorization.

Outcomes: Illustrate different cryptographic algorithms for security.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

The RSA cipher is a foundational public key algorithm used extensively in secure communications. This experiment demonstrated the process of key generation, encryption, and decryption in the RSA algorithm. Through practical implementation, the strengths and computational requirements of RSA were observed, illustrating its suitability for encrypting small data blocks such as keys and passwords. Understanding RSA also highlights the importance of key size in maintaining security and the potential vulnerabilities to various attacks. This experiment underlines the critical role of cryptographic algorithms in ensuring data security in digital communications.
