**Practical No: 6**

**Practical Name: Study and implement a program for Hill Cipher.**

**Original Approach:**

**Introduction:** The **Hill cipher** is a polygraphic substitution cipher based on linear algebra, invented by the mathematician **Lester S. Hill** in 1929. It is an example of a **block cipher**, meaning it encrypts blocks of letters together rather than one letter at a time. The Hill cipher relies on matrix operations over a finite field (typically modulo 26, corresponding to the 26 letters of the alphabet).

**How the Hill Cipher Works:**

1. **Key**: A square matrix of size n×n is used as the encryption key. The matrix size determines the block size for encryption (e.g., 2x2 matrix means a 2-letter block, 3x3 means a 3-letter block).
2. **Plaintext**: The plaintext message is divided into blocks of n letters, and each letter is represented by a number from 0 to 25, where A=0, B=1, ..., Z=25.
3. **Encryption**: The key matrix is multiplied by each plaintext block (treated as a vector). The resulting vector is converted back into letters, modulo 26 (to stay within the alphabet).
4. **Decryption**: To decrypt, the inverse of the key matrix (mod 26) is computed. The ciphertext is multiplied by this inverse matrix, and the result is reduced modulo 26 to recover the original plaintext.

**Example of Hill Cipher:**

· **Plaintext**: "HI"

· **Key Matrix:** K =

**Step-by-Step Encryption:**

1. **Convert the plaintext "HI" to numbers**:

* H = 7, I = 8 (A = 0, B = 1, ..., Z = 25)
* Plaintext vector K =

1. **Multiply the key matrix by the plaintext vector**:

* C = K \* P = \* =

1. **Take modulo 26 of the result**:

* **C =**  =

1. **Return the index of the alphabet**

* Cipher text: **XB**

**Source Code:**

import numpy as np

def text\_to\_number(text):

return [ord(char) - ord('A') for char in text.upper()]

def number\_to\_text(number):

return ''.join(chr(num + ord('A')) for num in number)

def mod\_inverse(a, m):

a = a % m

for x in range(1, m):

if (a \* x) % m == 1:

return x

return 1

def matrix\_determinant(matrix):

if matrix.shape == (2, 2):

return matrix[0, 0] \* matrix[1, 1] - matrix[0, 1] \* matrix[1, 0]

elif matrix.shape == (3, 3):

return (matrix[0, 0] \* (matrix[1, 1] \* matrix[2, 2] - matrix[1, 2] \* matrix[2, 1]) -

matrix[0, 1] \* (matrix[1, 0] \* matrix[2, 2] - matrix[1, 2] \* matrix[2, 0]) +

matrix[0, 2] \* (matrix[1, 0] \* matrix[2, 1] - matrix[1, 1] \* matrix[2, 0]))

else:

raise ValueError("Matrix should be either 2x2 or 3x3.")

def matrix\_inverse(matrix, modulus):

det = matrix\_determinant(matrix) % modulus

if det == 0: # Check if determinant is zero

raise ValueError("Matrix is singular and cannot be inverted.")

det\_inv = mod\_inverse(det, modulus) if matrix.shape == (2, 2):

inverse\_matrix = np.array([[matrix[1, 1], -matrix[0, 1]], [-matrix[1, 0], matrix[0, 0]]])

elif matrix.shape == (3, 3):

cofactors = np.zeros(matrix.shape, dtype=int)

for i in range(3):

for j in range(3):

sub\_matrix = np.delete(np.delete(matrix, i, axis=0), j, axis=1)

sign = (-1) \*\* (i + j)

cofactors[i, j] = sign \* matrix\_determinant(sub\_matrix)

inverse\_matrix = np.transpose(cofactors)

return (det\_inv \* inverse\_matrix) % modulus

def encryption\_block(block\_number, matrix):

ciphernumber = np.dot(matrix, block\_number) % 26

return ciphernumber

def decryption\_block(block\_number, matrix):

deciphernumber = np.dot(matrix, block\_number) % 26

return deciphernumber

def encryption(plaintext, matrix):

n = matrix.shape[0]

plain\_number = text\_to\_number(plaintext)

ciphertext = ''

if len(plain\_number) % n != 0:

plain\_number.extend([0] \* (n - len(plain\_number) % n))

for i in range(0, len(plain\_number), n):

block\_number = plain\_number[i:i + n]

ciphernumber = encryption\_block(block\_number, matrix)

ciphertext += number\_to\_text(ciphernumber)

return ciphertext

def decryption(ciphertext, matrix):

inverse\_matrix = matrix\_inverse(matrix, 26)

n = matrix.shape[0]

cipher\_number = text\_to\_number(ciphertext)

plaintext = ‘’

for i in range(0, len(cipher\_number), n):

block\_number = cipher\_number[i:i + n]

plainnumber = decryption\_block(block\_number, inverse\_matrix)

plaintext += number\_to\_text(plainnumber)

return plaintext

# Input plaintext and key

plaintext = input("Enter the plaintext: ")

print(f'Plaintext: {plaintext}')

key = input('Enter the key: ')

print(f'Key: {key}')

if len(key) <= 4:

matrix = np.array(text\_to\_number(key)).reshape(2, 2)

else:

matrix = np.array(text\_to\_number(key)).reshape(3, 3)

# Encryption process

ciphertext = encryption(plaintext, matrix)

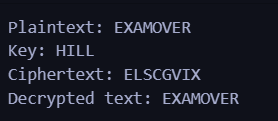
print(f'Ciphertext: {ciphertext}')

# Decryption process

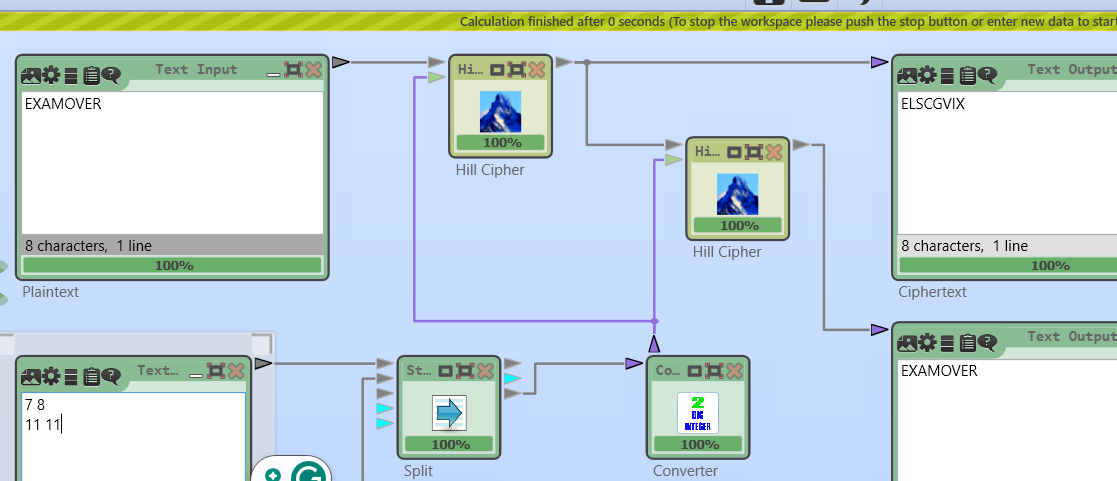
decrypted\_text = decryption(ciphertext, matrix)

print(f'Decrypted text: {decrypted\_text}')

**Output:**

****

**Cryptool:**

****

**Revised Approach:**

**Introduction**: The classic Hill cipher, while innovative, is vulnerable to certain cryptographic attacks due to its deterministic nature. To address these shortcomings, a **modified version** introduces **randomness** and additional security layers by incorporating a **random two-digit key** into the encryption and decryption process.

In this modified Hill cipher:

* The core structure remains the same, utilizing linear algebra where plaintext is represented as vectors and multiplied by a key matrix.
* **Enhancement**: A randomly generated two-digit key is introduced. This key's **unit digit** and **tens digit** are added to the result of the matrix multiplication during both encryption and decryption.
* This added randomness means that even if the same plaintext and key matrix are used multiple times, the ciphertext will differ each time, greatly improving the security against frequency analysis and known-plaintext attacks.

The modified Hill cipher, therefore, merges the efficiency of matrix-based encryption with an additional non-linear step, making it significantly harder for an attacker to reverse-engineer the plaintext without knowing both the key matrix and the random two-digit key. This ensures that the cipher is more resistant to standard cryptanalytic techniques while preserving the basic mechanics of the Hill cipher.

**Drawbacks of the Hill Cipher**

1. **Key Sensitivity**: The Hill cipher’s security relies heavily on the matrix used as the key. If the key matrix is poorly chosen (e.g., if it is not invertible or has common factors with 26), the decryption process becomes impossible or the cipher becomes easy to break.
2. **Lack of Randomness**: In the classic Hill cipher, the encryption process is deterministic for a given key and plaintext. This means that the same plaintext will always produce the same ciphertext if the key remains constant, making it vulnerable to known-plaintext attacks.
3. **No Diffusion of Digits**: The classic Hill cipher only performs matrix multiplication. If the ciphertext is intercepted, attackers can analyze relationships between plaintext blocks and corresponding ciphertext blocks, making frequency analysis feasible.

**How the Modified Code Improves Security**

Given modified version of the Hill cipher introduces **randomness** and an additional **nonlinear step**:

* By using **two-digit key** that influences the encryption by adding the unit and tens digits of this random key to the matrix multiplication results. This adds unpredictability to the encryption process, even if the same plaintext is encrypted multiple times with the same matrix key.
* This modification improves **resistance to frequency analysis** and **known-plaintext attacks** since the randomness introduces variation in the output ciphertext for identical plaintext inputs.

**Example:**

· **Plaintext**: "HI"

· **Key Matrix:** K =

**Step-by-Step Encryption:**

1. **Convert the plaintext "HI" to numbers**:

* H = 7, I = 8 (A = 0, B = 1, ..., Z = 25)
* Plaintext vector K =

1. **Multiply the key matrix by the plaintext vector**:

* C = K \* P = \* =

1. **Take modulo 26 of the result**:

* **C =**  =

1. **Add the double digit number to the output:**

* 23 + 1 = 24
* 1 + 5 = 6

1. **Convert the digit to the corresponding alphabet:**

* Cipher text: **XG**

**Source Code:**

import numpy as np

import random

def text\_to\_number(text):

return [ord(char) for char in text.upper()]

def number\_to\_text(number):

return ''.join(chr(num) for num in number)

def mod\_inverse(a, m):

a = a % m

for x in range(1, m):

if (a \* x) % m == 1:

return x

return 1

def matrix\_determinant(matrix):

if matrix.shape == (2, 2):

return matrix[0, 0] \* matrix[1, 1] - matrix[0, 1] \* matrix[1, 0]

elif matrix.shape == (3, 3):

return (matrix[0, 0] \* (matrix[1, 1] \* matrix[2, 2] - matrix[1, 2] \* matrix[2, 1]) -

matrix[0, 1] \* (matrix[1, 0] \* matrix[2, 2] - matrix[1, 2] \* matrix[2, 0]) +

matrix[0, 2] \* (matrix[1, 0] \* matrix[2, 1] - matrix[1, 1] \* matrix[2, 0]))

else:

raise ValueError("Matrix should be either 2x2 or 3x3.")

def matrix\_inverse(matrix, modulus):

det = matrix\_determinant(matrix) % modulus

if det == 0: # Check if determinant is zero

raise ValueError("Matrix is singular and cannot be inverted.")

det\_inv = mod\_inverse(det, modulus)

if matrix.shape == (2, 2):

inverse\_matrix = np.array([[matrix[1, 1], -matrix[0, 1]], [-matrix[1, 0], matrix[0, 0]]])

elif matrix.shape == (3, 3):

cofactors = np.zeros(matrix.shape, dtype=int)

for i in range(3):

for j in range(3):

sub\_matrix = np.delete(np.delete(matrix, i, axis=0), j, axis=1)

sign = (-1) \*\* (i + j)

cofactors[i, j] = sign \* matrix\_determinant(sub\_matrix)

inverse\_matrix = np.transpose(cofactors)

return (det\_inv \* inverse\_matrix) % modulus

def encryption\_block(block\_number, matrix, unit\_digit, tens\_digit):

ciphernumber = np.dot(matrix, block\_number) % 26

ciphernumber[0] = (ciphernumber[0] + unit\_digit) % 26

ciphernumber[1] = (ciphernumber[1] + tens\_digit) % 26

return ciphernumber

def decryption\_block(block\_number, matrix, unit\_digit, tens\_digit):

block\_number[0] = (block\_number[0] - unit\_digit) % 26

block\_number[1] = (block\_number[1] - tens\_digit) % 26

deciphernumber = np.dot(matrix, block\_number) % 26

return deciphernumber

def encryption(plaintext, matrix, unit\_digit, tens\_digit):

n = matrix.shape[0]

plain\_number = text\_to\_number(plaintext)

ciphertext = ''

if len(plain\_number) % n != 0:

plain\_number.extend([0] \* (n - len(plain\_number) % n))

for i in range(0, len(plain\_number), n):

block\_number = plain\_number[i:i + n]

ciphernumber = encryption\_block(block\_number, matrix, unit\_digit, tens\_digit)

ciphertext += number\_to\_text(ciphernumber)

return ciphertext

def decryption(ciphertext, matrix, unit\_digit, tens\_digit):

inverse\_matrix = matrix\_inverse(matrix, 26)

n = matrix.shape[0]

cipher\_number = text\_to\_number(ciphertext)

plaintext = ''

for i in range(0, len(cipher\_number), n):

block\_number = cipher\_number[i:i + n]

plainnumber = decryption\_block(block\_number, inverse\_matrix, unit\_digit, tens\_digit)

plaintext += number\_to\_text(plainnumber)

return plaintext

# Input plaintext and key

plaintext = input("Enter the plaintext: ")

print(f'Plaintext: {plaintext}')

key = input('Enter the key: ')

print(f'Key: {key}')

if len(key) <= 4:

matrix = np.array(text\_to\_number(key)).reshape(2, 2)

else:

matrix = np.array(text\_to\_number(key)).reshape(3, 3)

# Generate random two-digit key

random\_key = random.randint(10, 99)

unit\_digit = random\_key % 10

tens\_digit = random\_key // 10

print(f'Random two-digit key: {random\_key}')

print(f'Unit digit: {unit\_digit}, Tens digit: {tens\_digit}')

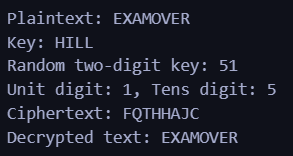
ciphertext = encryption(plaintext, matrix, unit\_digit, tens\_digit)

print(f'Ciphertext: {ciphertext}')

decrypted\_text = decryption(ciphertext, matrix, unit\_digit, tens\_digit)

print(f'Decrypted text: {decrypted\_text}')

**Output:**

****

**Cryptanalysis on Vigenère Cipher:**

I. **Known Plaintext Attack**:

* **Description**: In a known plaintext attack, the attacker has access to both the plaintext and its corresponding ciphertext. The Hill cipher encrypts blocks of plaintext using matrix multiplication. If enough plaintext-ciphertext pairs are known (at least as many as the size of the matrix), the attacker can set up a system of linear equations to solve for the elements of the encryption matrix.
* **Impact**: Once the encryption matrix is deduced, the attacker can decrypt any further ciphertexts encrypted with the same matrix. This renders the Hill cipher highly vulnerable when the same key is reused across multiple messages.

II. **Chosen Plaintext Attack**:

* **Description**: In a chosen plaintext attack, the attacker can select specific plaintexts and observe the resulting ciphertexts. By choosing particular blocks, such as identity matrices (e.g., [1,0,0][1, 0, 0][1,0,0], [0,1,0][0, 1, 0][0,1,0]), the attacker can easily recover the encryption matrix by solving the system of linear equations.
* **Impact**: This attack is highly effective because the attacker can control the plaintext input. Once the matrix is recovered, any ciphertext encrypted using the same key can be decrypted. This makes the Hill cipher vulnerable when an attacker can manipulate the plaintex

III. **Brute Force Attack**:

* **Description**: In a brute force attack, the attacker tries every possible key (matrix) to decrypt the ciphertext. For a 2x2 matrix, there are 26426^4264 possible combinations, and for a 3x3 matrix, there are 26926^9269 combinations. Although the number of possible keys grows quickly, brute force attacks are still possible for small matrices.
* **Impact**: While this attack is less practical for larger matrices, it becomes feasible for smaller key sizes (e.g., 2x2 matrices), reducing the cipher’s effectiveness. In scenarios where computational resources are abundant, brute force attacks remain a potential threat to the Hill cipher.

**Conclusion:**

The Hill cipher, though mathematically robust in theory, is vulnerable to several cryptanalysis techniques, especially when the encryption matrix is reused across multiple messages or when attackers have access to plaintext-ciphertext pairs. The cipher’s susceptibility to known plaintext and chosen plaintext attacks is its most significant drawback, limiting its security in modern cryptographic contexts.

**References:**

1. Serdano, Akbar, Muhammad Zarlis, and Erna Budhiarti Nababan. "Performance of combining hill cipher algorithm and caesar cipher algorithm in text security." *2021 International conference on artificial intelligence and mechatronics systems (AIMS)*. IEEE, 2021.
2. Mujaddid, Azzam, and Sumarsono Sumarsono. "A Modifying of Hill Cipher Algorithm with 3 Substitution Caesar Cipher." *Proceeding International Conference on Science and Engineering*. Vol. 1. 2017.

**Practical No: 7**

**Practical Name: Study and implement a program for RSA Cipher.**

**Roll No: 22BCP250**

**Name: Anusi Patel**

**Division and batch: Div4 G7**

**Original Approach**

**Introduction:** The RSA algorithm is a public-key cryptography method for secure data transmission. It relies on the difficulty of factoring large prime numbers. RSA uses two keys: a public key for encryption and a private key for decryption. Only the intended recipient can decrypt the message, maintaining data security.

**Example:**

* **Key Generation**:
  + Choose two large prime numbers: p=3 and q=11
  + Compute n=p×q=33
  + Compute ϕ(n)=(p−1)×(q−1)=2×10=20
  + Choose e such that 1<e<ϕ(n) and e is co-prime to ϕ(n), let e=3
  + Compute d such that (e×d)mod  ϕ(n)=1, let d=7
* **Public Key**: (e=3, n=33)  
  **Private Key**: (d=7, n=33)
* **Encryption**:
  + Message m=7
  + Ciphertext c=m^e mod  n=7^3 mod  33=343 mod  33=13
* **Decryption**:
  + Ciphertext c=13
  + Decrypted message m=c^d mod  n=13^7 mod  33=62748517 mod  33=7

**Source Code:**

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

def find\_e(phi):

for e in range(2, phi):

if gcd(e, phi) == 1:

return e

return None

def mod\_inverse(e, phi):

t1, t2 = 0, 1

r1, r2 = phi, e

while r2 > 0:

quotient = r1 // r2

r1, r2 = r2, r1 - quotient \* r2

t1, t2 = t2, t1 - quotient \* t2

if r1 == 1:

return t1 % phi

return None

def mod\_exp(base, exp, mod):

result = 1

while exp > 0:

if exp % 2 == 1:

result = (result \* base) % mod

base = (base \* base) % mod

exp //= 2

return result

def message\_to\_numeric(message):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

numeric\_value = ""

for char in message:

numeric\_index = alphabet.index(char)

numeric\_value += f"{numeric\_index:02d}"

return int(numeric\_value)

def numeric\_to\_message(numeric\_value):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

chars = []

temp\_value = str(numeric\_value)

while len(temp\_value) >= 2:

pair = temp\_value[:2]

numeric\_index = int(pair)

chars.append(alphabet[numeric\_index])

temp\_value = temp\_value[2:]

if temp\_value:

numeric\_index = int(temp\_value)

chars.append(alphabet[numeric\_index])

return ''.join(chars)

def rsa\_encryption(p, q, message):

n = p \* q

phi = (p - 1) \* (q - 1)

e = find\_e(phi)

print(f"Automatically selected public exponent e: {e}")

numeric\_message = message\_to\_numeric(message)

print(f"Numeric equivalent of message: {numeric\_message}")

d = mod\_inverse(e, phi)

encrypted\_message = mod\_exp(numeric\_message, e, n)

print(f"Encrypted message (numeric): {encrypted\_message}")

decrypted\_message = mod\_exp(encrypted\_message, d, n)

print(f"Decrypted message (numeric): {decrypted\_message}")

original\_message = numeric\_to\_message(decrypted\_message)

print(f"Original message: {original\_message}")

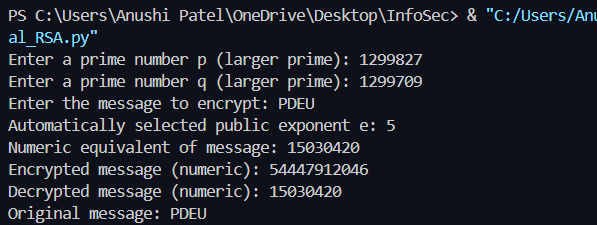
p = int(input("Enter a prime number p (larger prime): "))

q = int(input("Enter a prime number q (larger prime): "))

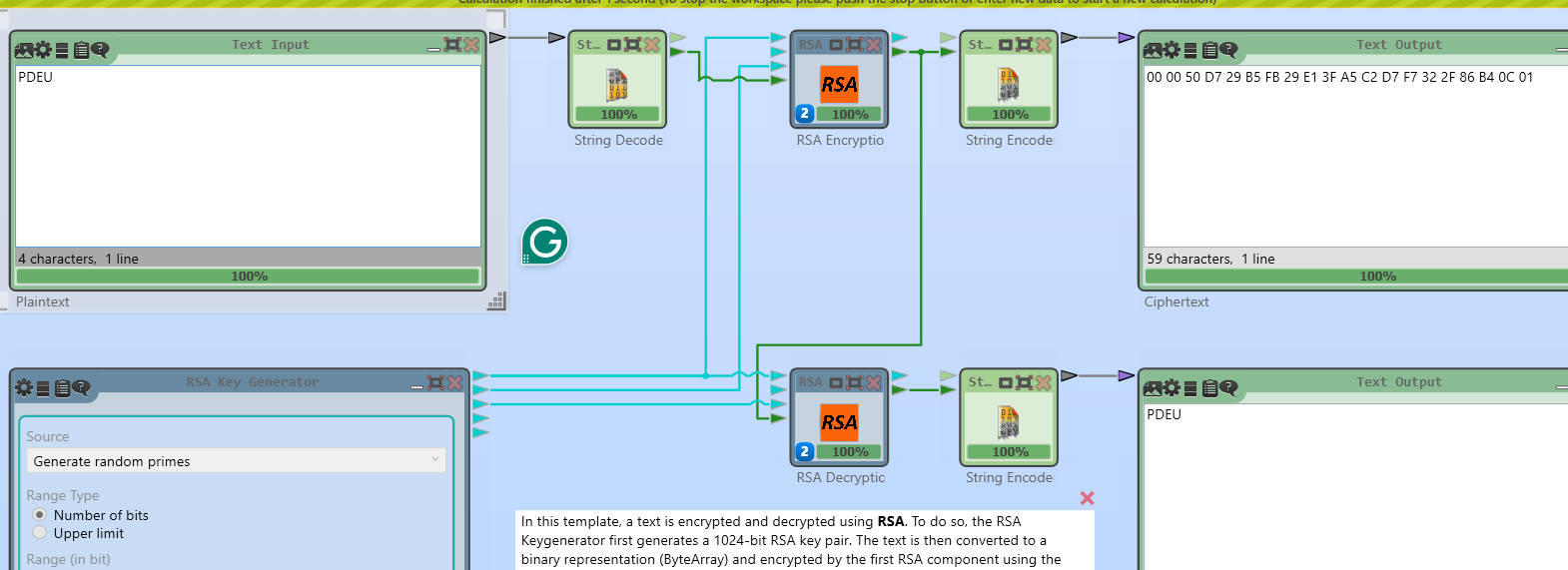
message = input("Enter the message to encrypt: ")

rsa\_encryption(p, q, message)

**Output:**

****

**Cryptool:**

****

**Cryptanalysis** **on RSA:**

1. **Brute-Force Attack:**

* **Description:** A brute-force attack on RSA involves trying every possible private key to decrypt a given ciphertext.
* **Impact:** While theoretically possible, brute-force attacks are impractical for RSA due to the large key size (e.g., 2048-bit keys). The immense number of combinations makes this method infeasible in practice with current computational power.

1. **II. Known-Plaintext Attack:**

* **Description:** Involves having a portion of the plaintext and its corresponding ciphertext to deduce the private key.
* **Impact:** If an attacker has access to a portion of plaintext and the corresponding ciphertext, they may attempt to reverse-engineer the encryption process. However, without access to the private key, this attack is challenging in RSA due to the one-way nature of the encryption function.

1. **III. Chosen-Plaintext Attack (CPA):**

* **Description:** In a CPA, the attacker can choose arbitrary plaintexts and obtain their corresponding ciphertexts to gain information about the private key.
* **Impact:** RSA is vulnerable to CPA if not implemented correctly. However, when proper padding schemes like OAEP (Optimal Asymmetric Encryption Padding) are used, the attack becomes impractical.

**Conclusion:**

The RSA algorithm is a cornerstone of public-key cryptography, enabling secure communication through a pair of keys: a public key for encryption and a private key for decryption. Its security relies on the difficulty of factoring large prime numbers, making it widely used in applications like SSL/TLS and digital signatures. While RSA is robust, it requires proper key sizes (recommended at least 2048 bits) and secure implementation to protect against potential vulnerabilities. As technology evolves, RSA remains vital for ensuring data confidentiality and integrity in the digital landscape.

**References:**

* Zhou, X., & Tang, X. (2011, August). Research and implementation of RSA algorithm for encryption and decryption. In *Proceedings of 2011 6th international forum on strategic technology* (Vol. 2, pp. 1118-1121). IEEE.
* Meneses, Fausto, et al. "RSA encryption algorithm optimization to improve performance and security level of network messages." *IJCSNS* 16.8 (2016): 55.

**Practical No: 8**

**Practical Name: Study and implement a program of the Digital Signature with RSA algorithm (Reverse RSA).**

**Original Approach**

**Introduction:** RSA (Rivest–Shamir–Adleman) is a widely used public key cryptography algorithm that provides both encryption and digital signature functionalities. In the context of digital signatures, the process involves signing a message with a private key and verifying that signature with the corresponding public key. This ensures the integrity and authenticity of the message, confirming that it has not been altered and that it was indeed created by the claimed sender.

**Example:**

1. **Key Generation**

To generate a key pair for DSA, we need two keys: a private key (used for signing) and a public key (used for verifying signatures). The steps for key generation are as follows:

1. **Select a large prime number p:**  
   A prime number is chosen. For example, let p = 23.
2. **Select a prime divisor q:**  
   Choose a prime divisor of p - 1. For example, q = 11.
3. **Choose a generator g:**  
   Calculate g using the formula  
   g = h^(p−1)/q mod p  
   for some integer h (where h is between 1 and p - 1).  
   Let’s say g = 2.
4. **Select a private key x:**  
   Randomly choose an integer x such that 0 < x < q. For example, x = 7.
5. **Compute the public key y:**  
   The public key is calculated as  
   y = g^x mod p.  
   For our example:  
   y = 2^7 mod 23 = 18.
6. **Summary of Key Generation:**

p = 23

q = 11

g = 2

x = 7 (private key)

y = 18 (public key)

1. **Signing a Message**

To sign a message m, follow these steps:

1. **Hash the message:**  
   Use a hash function to create a hash of the message. For example, if the message is "Hello", assume the hash H(m) = 15.
2. **Generate a random integer k:**  
   Select a random integer k such that 0 < k < q. For example, let k = 3.
3. **Calculate r:**  
   Compute  
   r = (g^k mod p) mod q  
   r = (2^3 mod 23) mod 11 = 8 mod 11 = 8.
4. **Calculate s:**  
   Compute  
   s = (k^−1 (H(m) + x · r)) mod q.  
   First, compute k^−1 (the modular inverse of k modulo q):  
   k^−1 = 4 (since 3 · 4 mod 11 = 1).
5. Now calculate s:  
   s = (4 · (15 + 7 · 8)) mod 11  
   = (4 · (15 + 56)) mod 11  
   = (4 · 71) mod 11 = 9.
6. **Signature:**  
   The signature of the message m is the pair (r, s) = (8, 9).
7. **Verifying the Signature**

To verify the signature (r, s):

1. **Verify that r and s are valid:**  
   Check that 0 < r < q and 0 < s < q:  
   0 < 8 < 11 and 0 < 9 < 11 are both true.
2. **Compute the hash of the message:**  
   Assume the hash H(m) = 15 as before.
3. **Calculate w:**  
   Compute  
   w = s^−1 mod q.  
   s^−1 = 5 (since 9 · 5 mod 11 = 1).
4. **Calculate u1 and u2:**  
   Compute  
   u1 = (H(m) · w) mod q  
   u1 = (15 · 5) mod 11 = 9.
5. Compute  
   u2 = (r · w) mod q  
   u2 = (8 · 5) mod 11 = 7.
6. **Calculate v:**  
   Compute  
   v = ((g^u1 · y^u2) mod p) mod q.  
   v = ((2^9 · 18^7) mod 23) mod 11.
7. **Calculate each part:**  
   2^9 mod 23 = 18  
   18^7 mod 23 = 10.
8. Thus,  
   v = (18 · 10 mod 23) mod 11  
   = (180 mod 23) mod 11 = 8.
9. **Compare v and r:**  
   If v = r, the signature is valid. In our case, 8 = 8, so the signature is valid.

**Source Code:**

import hashlib

# Function to calculate GCD

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

# Function to find a suitable public exponent e

def find\_e(phi):

for e in range(2, phi):

if gcd(e, phi) == 1:

return e

return None

# Function to calculate modular inverse

def mod\_inverse(e, phi):

t1, t2 = 0, 1

r1, r2 = phi, e

while r2 > 0:

quotient = r1 // r2

r1, r2 = r2, r1 - quotient \* r2

t1, t2 = t2, t1 - quotient \* t2

if r1 == 1:

return t1 % phi

return None

# Function for modular exponentiation

def mod\_exp(base, exp, mod):

result = 1

while exp > 0:

if exp % 2 == 1:

result = (result \* base) % mod

base = (base \* base) % mod

exp //= 2

return result

# Function to convert numerical message into string of digits

def message\_to\_numeric(message):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

numeric\_value = ""

for char in message:

numeric\_index = alphabet.index(char)

numeric\_value += f"{numeric\_index:02d}" # Format as two digits

return int(numeric\_value) # Convert the entire string to an integer

# Function to convert numeric value back to message

def numeric\_to\_message(numeric\_value):

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

chars = []

temp\_value = str(numeric\_value)

# Process the numeric value from the start

while len(temp\_value) >= 2:

pair = temp\_value[:2] # Take first two characters

numeric\_index = int(pair)

chars.append(alphabet[numeric\_index]) # Convert index back to character

temp\_value = temp\_value[2:] # Remove the first two characters

# Handle any remaining single digit if necessary

if temp\_value:

numeric\_index = int(temp\_value)

chars.append(alphabet[numeric\_index]) # Convert index back to character

return ''.join(chars)

# Function to hash the message using SHA-256 and reduce its size for RSA

def hash\_message(message, modulus):

# Create a SHA-256 hash of the message

sha256 = hashlib.sha256()

sha256.update(message.encode()) # Encode the message to bytes

hash\_value = int(sha256.hexdigest(), 16) # Hash as an integer

# Reduce the hash size for RSA

return hash\_value % modulus

# RSA encryption and decryption process

def rsa\_encryption(p, q, message):

n = p \* q

phi = (p - 1) \* (q - 1)

# Automatically find a valid e

e = find\_e(phi)

print(f"Automatically selected public exponent e: {e}")

# Calculate the private key d

d = mod\_inverse(e, phi)

# 1. Message Digest Creation

hashed\_message = hash\_message(message, n) # Use modulus to reduce hash size

print(f"Hashed message (numeric): {hashed\_message}")

signed\_message = mod\_exp(hashed\_message, d, n)

print(f"Signed message (numeric): {signed\_message}")

verified\_hash = mod\_exp(signed\_message, e, n)

print(f"Verified hash (numeric): {verified\_hash}")

if verified\_hash == hashed\_message:

print("Verification successful: The message is authentic and has not been altered.")

else:

print("Verification failed: The message may have been altered.")

p = int(input("Enter a prime number p (larger prime): "))

q = int(input("Enter a prime number q (larger prime): "))

message = input("Enter the message to encrypt: ")

rsa\_encryption(p, q, message)

**Output:**

****

**Cryptanalysis** **on DSA:**

1. **Prime Factorization**: RSA's security hinges on the difficulty of factoring large composites. The public key nnn (product of two large primes) must remain difficult to factor; larger primes increase security.
2. **Small Key Sizes**: Using a 16-bit key is insecure. Modern RSA requires at least 2048 bits to resist factoring attacks, as smaller keys can be easily compromised.
3. **Common Weaknesses**:

* **Low Exponent**: A small public exponent eee (e.g., 3) may create vulnerabilities, especially with repeated messages.
* **Timing Attacks**: These exploit variations in decryption times to uncover private keys.
* **Padding Schemes**: Without proper padding (like OAEP), RSA is susceptible to attacks such as plaintext guessing.

**Conclusion:**

The RSA algorithm is a cornerstone of public-key cryptography, enabling secure communication through a pair of keys: a public key for encryption and a private key for decryption. Its security relies on the difficulty of factoring large prime numbers, making it widely used in applications like SSL/TLS and digital signatures. While RSA is robust, it requires proper key sizes (recommended at least 2048 bits) and secure implementation to protect against potential vulnerabilities. As technology evolves, RSA remains vital for ensuring data confidentiality and integrity in the digital landscape.

**References:**

* Davies, D. W. (1983). Applying the RSA digital signature to electronic mail. *Computer*, *16*(02), 55-62.
* Singh, P., & Kumar, S. (2017). Study & analysis of cryptography algorithms: RSA, AES, DES, T-DES, blowfish. *Int. J. Eng. Technol*, *7*(1.5), 221.