**Practical No: 8**

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

**Roll No: 22BCP250**

**Name: Anusi Patel**

**Division and batch: Div4 G7**

**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.")

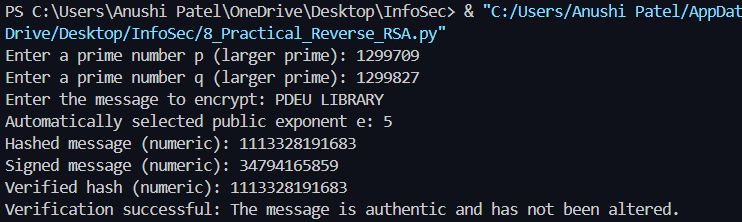
p = int(input("Enter a prime number p (larger prime): "))

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.