EXPERIMENT 02

Aim : Implementation and analysis of RSA cryptosystem and Digital signature scheme using RSA

Theory:

RSA (Rivest-Shamir-Adleman) is a widely-used public-key cryptosystem named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman. It is based on the difficulty of factoring large composite numbers into their prime factors, which forms the foundation of its security.

In RSA, each user has a pair of cryptographic keys: a public key, which can be freely distributed, and a private key, which must be kept secret. These keys are typically generated using large prime numbers and specific mathematical operations.

The main operations in RSA involve modular exponentiation. Encryption and decryption are performed using exponentiation modulo a product of two large prime numbers. The encryption operation uses the public key, while decryption uses the corresponding private key.

RSA is commonly used for secure communication and digital signatures. It provides confidentiality by allowing users to encrypt messages using the recipient's public key, ensuring that only the intended recipient can decrypt and read the message using their private key. Additionally, RSA supports digital signatures, enabling users to sign messages with their private key, allowing others to verify the authenticity of the message using the signer's public key.

Overall, RSA is a foundational cryptographic algorithm widely used in various security protocols and applications due to its security, versatility, and widespread support.

- **RSA Cryptosystem Implementation:**

Key Generation:

1. Generate two large random prime numbers, p and q.
2. Compute n = p * q.
3. Compute φ(n) = (p-1)(q-1), where φ is Euler's totient function.
4. Choose an integer e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1.
5. Compute d, the modular multiplicative inverse of e modulo φ(n), i.e., d ≡ e^(-1) (mod φ(n)).
6. Public key: (n, e)
7. Private key: (n, d)

Encryption:
1. Represent the plaintext message as an integer m, where $0 < m < n$.
2. Compute the ciphertext $c \equiv m^e \pmod{n}$.

Decryption:
1. Compute the plaintext message $m \equiv c^d \pmod{n}$.

- **Digital Signature Scheme using RSA:**

Key Generation (Same as RSA Cryptosystem):
1. Generate p, q, n, φ(n), e, and d as in the RSA key generation.

Signature Generation:
1. Compute the hash value of the message (e.g., using a secure hash function like SHA-256). 2. Represent the hash value as an integer h.
3. Compute the signature $s \equiv h^d \pmod{n}$.

Signature Verification:
1. Receive the message, signature pair (msg, s).
2. Compute the hash value of the message, obtaining h'.
3. Verify if $h' \equiv s^e \pmod{n}$.

Analysis:

1. Security: RSA's security relies on the difficulty of factoring large composite numbers. The security of RSA depends on the size of the keys used. Larger key sizes provide higher security, but also require more computational resources.

2. Efficiency: RSA encryption and decryption operations involve modular exponentiation, which can be computationally expensive, especially for large keys. Using more efficient algorithms for modular exponentiation, like square and multiply algorithm or Montgomery multiplication, can improve performance.
3. Signature Size: The size of RSA signatures depends on the size of the modulus n. Larger moduli result in longer signatures, which may impact transmission and storage requirements.

4. Padding Scheme: In practice, RSA encryption and digital signatures use padding schemes

like RSA-OAEP for encryption and RSA-PSS for signatures. These schemes provide security and prevent certain attacks like chosen ciphertext attacks and padding oracle attacks.

5. Randomness: Generating secure RSA keys requires high-quality randomness. Inadequate randomness can lead to weak keys susceptible to attacks.

6. Key Management: Secure RSA usage requires proper key management practices, including key generation, storage, and distribution, to ensure the confidentiality and integrity of encrypted data and signatures.

Implementing and analyzing RSA involves considerations of security, efficiency, and proper usage practices to ensure the cryptographic system's integrity and confidentiality. Additionally, understanding the strengths and weaknesses of RSA helps in its appropriate application in various security scenarios.

Program :

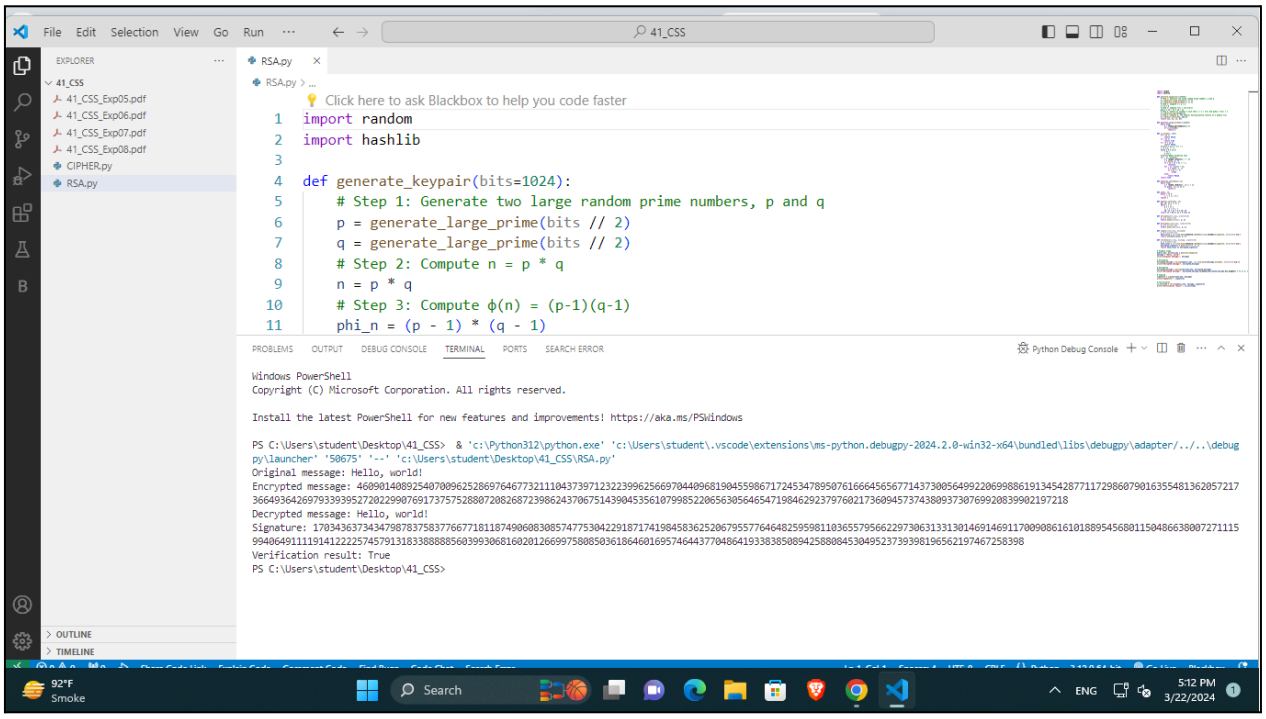
```
import random
import hashlib

def generate_keypair(bits=1024):
    # Step 1: Generate two large random prime numbers, p and q
    p = generate_large_prime(bits // 2)
    q = generate_large_prime(bits // 2)

    # Step 2: Compute n = p * q
    n = p * q

    # Step 3: Compute φ(n) = (p-1)(q-1)
    phi_n = (p - 1) * (q - 1)
# Step 4: Choose an integer e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1
    e = generate_coprime(phi_n)

    # Step 5: Compute d, the modular multiplicative inverse of e modulo φ(n)
    d = modular_inverse(e, phi_n)
```

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

def generate_large_prime(bits=1024):
    while True:
        p = random.getrandbits(bits)
        if is_prime(p):
            return p

def is_prime(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Write n as d * 2^r + 1
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Miller-Rabin primality test
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
else:
            return False
    return True

def generate_coprime(phi_n):
```

```
    while True:
        e = random.randint(2, phi_n - 1)
        if gcd(e, phi_n) == 1:
            return e

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def modular_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def encrypt(public_key, plaintext):
    n, e = public_key
    return pow(plaintext, e, n)

def decrypt(private_key, ciphertext):
    n, d = private_key
    return pow(ciphertext, d, n)

def sign(private_key, message):
    n, d = private_key
    hash_value = int.from_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')
    return pow(hash_value, d, n)

def verify(public_key, message, signature):
    n, e = public_key
hash_value = int.from_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')
    decrypted_signature = pow(signature, e, n)
    return hash_value == decrypted_signature

# Example usage
```

```
public_key, private_key = generate_keypair()
message = "Hello, world!"
print("Original message:", message)

# Encryption
encrypted_message = encrypt(public_key, int.from_bytes(message.encode(), byteorder='big'))
print("Encrypted message:", encrypted_message)

# Decryption
decrypted_message = decrypt(private_key, encrypted_message)
print("Decrypted message:", decrypted_message.to_bytes((decrypted_message.bit_length() + 7)
// 8, byteorder='big').decode())

# Signing
signature = sign(private_key, message)
print("Signature:", signature)

# Verification
is_verified = verify(public_key, message, signature)
print("Verification result:", is_verified)
```

Output:



Conclusion:

Q. What specific steps could be taken to further enhance the security and efficiency of the RSA cryptosystem and digital signature scheme implemented in the provided Python code?

Ans:
To further enhance the security and efficiency of the RSA cryptosystem and digital signature scheme implemented in the provided Python code, several steps can be considered:

1. Optimized Algorithm Implementation: Utilize more efficient algorithms and libraries for prime number generation, primality testing, and modular exponentiation to improve computational efficiency.

2. Enhanced Key Management: Implement stringent key management practices to ensure the generation, storage, and distribution of keys are securely handled, safeguarding against unauthorized access and key compromise.

3. Increased Key Size: Consider increasing the key size beyond the default of 1024 bits to adapt to advancements in computing power and maintain cryptographic strength against emerging threats.

4. Secure Padding Schemes: Incorporate secure padding schemes such as RSA-OAEP for encryption and RSA-PSS for signatures to mitigate potential vulnerabilities such as chosen ciphertext attacks.

5. Thorough Error Handling: Implement robust error handling mechanisms to gracefully handle exceptions and edge cases, ensuring the reliability and stability of cryptographic operations.

By implementing these measures, the security and efficiency of the RSA cryptosystem and digital signature scheme can be further strengthened, ensuring their suitability for secure cryptographic applications.