ST. FRANCIS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY
SECURITY LAB

Experiment – 5 : Implementation of Digital Signature Scheme

Aim: To implement RSA Digital Signature Scheme.

Objective: After performing the experiment, the students will be able to –

- To understand the RSA Digital Signature Scheme

Lab objective mapped:
L502.3: Students should be able to Analyze and evaluate performance of digital signature scheme and demonstrate key management.

Prerequisite: Basic knowledge of Digital Signature.

Requirements: C/C++/JAVA/PYTHON

Pre-Experiment Theory:

Digital Signature Requirements:

- The signature must be a bit pattern that depends on the message being signed.
- The signature must use some information unique to the sender to prevent both forgery and denial.
- It must be relatively easy to produce the digital signature.
- It must be relatively easy to recognize and verify the digital signature.
- It must be computationally infeasible to forge a digital signature, either by constructing a new message for an existing digital signature or by constructing a fraudulent digital signature for a given message.

RSA can be used for signing and verifying a message. In this case it is called the RSA digital signature scheme.
The digital signature scheme changes the roles of the private and public keys.

1. First, the private and public key of the sender, not the receiver are used.
2. Second, the sender uses her own private key to sign the document; the receiver uses the sender's public key to verify it.

Generation of RSA Key Pair

- Key generation in the RSA digital signature scheme is exactly the same as key generation in the RSA cryptosystem.
- Alice chooses two primes p and q and calculates n= p x q.
- Alice calculates $\phi(n) = (p - 1)(q - 1)$.
- She then chooses e, the public exponent, and calculates d, the private exponent such that e x d = 1 mod $\phi(n)$.
- Alice keeps d, she publicly announces n and e.

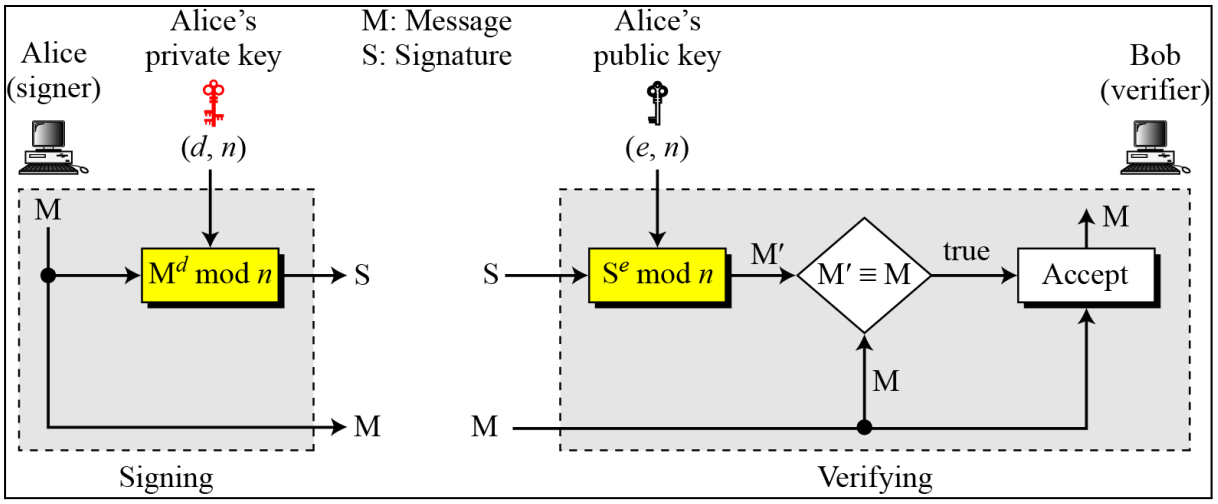
Signing

Alice creates a signature out of the message using her private exponent, $S = M^d \bmod n$ and sends the message and the signature to Bob.

Verifying

Bob receives M and S. Bob applies Alice's public exponent to the signature to create a copy of the message $M' = S^e \bmod n$.

Bob compares the value of M' with the value of M. If the two values are congruent, Bob accepts the message.



Procedure:

Write a program in C/C++/Java or Python for key generation, signing and verification using RSA algorithm.

a. For Key generation ask user to enter the value of prime numbers p & q and a public key 'e'. (Note that values of p, q & e cannot be random, they should satisfy criteria as per RSA algorithm)
b. Program should calculate private key element 'd' using Extended Euclidean Algorithm.
c. Provide a set of public (e, n) and private key (d, n) as the output to the user.
d. For digital sign generation, ask user to enter the Message. Using private key calculate the digital sign 'S' and output the same.
e. For sign verification, ask user to enter the sign 'S' and the Message 'M'. Using public key calculate the M' value. Compare this M' value with Message 'M'. If they are same, display that "The message is authenticate" else display "Message is altered. Discard."

2. Test the output of program for following exercise:

   For p=11, q=3 and e=3, find public and private key using RSA algorithm. For message, M=10, Find the digital signature value. Also show how any receiver of this signature will verify it.

Output:

1. Attach the complete code performing key generation, signature generation and verification.
2. Attach the program output for key generation (display public key & private key), digital sign 'S' and verification code M' for the inputs given in the exercise above.

Post Experimental Exercise:

1. Solve the exercise mentioned in procedure on journal sheets. [Theoretical result and attached code's output should match].
2. What are the different attacks possible on RSA digital signature scheme?

Conclusion:

In RSA Digital Signature Scheme the signing and verifying sites use the same function but with different parameters. The verifier compares the message and the output of the function for congruence. If the result is true, the message is accepted.

CODE :-

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

def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, y = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    return x % m

def generate_keys(p, q, e):
    n = p * q
    phi = (p - 1) * (q - 1)
    if gcd(e, phi) != 1:
        raise ValueError("e is not coprime to phi")

    d = mod_inverse(e, phi)
    return (e, n), (d, n)

def digital_signature(private_key, message):
    d, n = private_key
    signature = pow(message, d, n)
    return signature

def verify_signature(public_key, signature, message):
    e, n = public_key
    decrypted_signature = pow(signature, e, n)
    return decrypted_signature == message

# Input values
p = int(input("Enter prime number p: "))
q = int(input("Enter prime number q: "))
e = int(input("Enter public key 'e': "))
M = int(input("Enter the message M: "))

# Key generation
public_key, private_key = generate_keys(p, q, e)
print("Public Key (e, n):", public_key)
print("Private Key (d, n):", private_key)
```

```
# Digital signature generation
signature = digital_signature(private_key, M)
print("Digital Signature:", signature)

# Signature verification
received_signature = int(input("Enter the received digital signature: "))
is_authentic = verify_signature(public_key, received_signature, M)
if is_authentic:
    print("The message is authentic!")
else:
    print("Message is altered!")
```

OUTPUT:-