Don Bosco Institute of Technology, Mumbai 400070
Department of Information Technology

Name: Swasti Jain
Sem: 5, Branch: IT, Batch: 2

Experiment No.: 6

Date: 07/09/2022

Title:
Implement the RSA cryptosystem

Problem Definition:
Implement asymmetric algorithm RSA and encrypt the given plain-text and decrypt it back.

Prerequisite: Asymmetric cryptography

Theory:
RSA is one of the first practicable public-key cryptosystems and is widely used for secure data transmission.
In such a cryptosystem, the encryption key is public and differs from the decryption key which is kept secret.
In RSA, this asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the factoring problem.
RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described the algorithm in 1977.
Clifford Cocks, an English mathematician, had developed an equivalent system in 1973, but it was not declassified until 1997.f

A user of RSA creates and then publishes a public key based on the two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime numbers can feasibly decode the message.

Procedure/ Algorithm:

Key generation: RSA involves a public key and a private key. The public key can be known by everyone and is used for encrypting messages.
Messages encrypted with the public key can only be decrypted in a reasonable amount of time using the private key.
The keys for the RSA algorithm are generated the following way:

1. Choose two distinct prime numbers p and q. For security purposes, the integers p and q should be chosen at random, and should be of similar bit-length. Prime integers can be efficiently found using a primality test.

2. Compute n = p*q. n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.

3. Compute $\varphi(n) = \varphi(p)\varphi(q) = (p - 1) * (q - 1) = n - (p + q - 1)$, where φ is Euler's totient function.

4. Choose an integer e such that $1 < e < \varphi(n)$ and gcd $(e, \varphi(n)) = 1$;
i.e., e and $\varphi(n)$ are coprime. e is released as the public key exponent.
e having a short bit-length and small Hamming weight results in more efficient encryption – most commonly 216 + 1 = 65,537.
However, much smaller values of e (such as 3) have been shown to be less secure in some settings.

5. Determine d as $d \equiv e^{-1} \pmod{\varphi(n)}$;
i.e., d is the multiplicative inverse of e (modulo $\varphi(n)$). This is more clearly stated as: solve for d given $de \equiv 1 \pmod{\varphi(n)}$ This is often computed using the extended Euclidean algorithm.

Using the pseudo code in the Modular integers section, inputs a and n correspond to e and φ(n), respectively. d is kept as the private key exponent.
The public key consists of the modulus n and the public (or encryption) exponent e.
The private key consists of the modulus n and the private (or decryption) exponent d, which must be kept secret. p, q, and φ(n) must also be kept secret because they can be used to calculate d. An alternative, used by PKCS#1, is to choose d matching $de \equiv 1 \pmod{\lambda}$ with $\lambda = \text{lcm}(p-1, q-1)$, where lcm is the least common multiple.
Using λ instead of φ(n) allows more choices for d. λ can also be defined using the Carmichael function, λ(n).

Encryption:
Alice transmits her public key (n, e) to Bob and keeps the private key d secret.
Bob then wishes to send message M to Alice.
He first turns M into an integer m, such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme.
He then computes the ciphertext c corresponding to This can be done efficiently, even for 500-bit numbers, using Modular exponentiation. Bob then transmits c to Alice.

Decryption:
Alice can recover m from c by using her private key exponent d via computing Given m, she can recover the original message M by reversing the padding scheme.

Program code with Result/Output :

```
import math

def gcd(a, h):
    temp = 0
    while(1):
        temp = a % h
        if (temp == 0):
            return h
        a = h
        h = temp

p = 3
q = 7
n = p*q
e = 2
phi = (p-1)*(q-1)

while (e < phi):
    if(gcd(e, phi) == 1):
        break
    else:
        e = e+1

k = 2
d = (1 + (k*phi))/e
msg = 12.0

print("Message data = ", msg)

# Encryption c = (msg ^ e) % n
c = pow(msg, e)
c = math.fmod(c, n)
print("Encrypted data = ", c)

# Decryption m = (c ^ d) % n
m = pow(c, d)
m = math.fmod(m, n)
print("Decrypted data = ", m)
```

```
PS C:\Users\Swasti\OneDrive\Desktop\Security_Lab> & C:/PYTHON/Python36/python.exe c:/Users/Swas
ti/OneDrive/Desktop/Security_Lab/RSA.py
Message data =  12.0
Encrypted data =  3.0
Decrypted data =  12.0
PS C:\Users\Swasti\OneDrive\Desktop\Security_Lab>
```

References :

1. http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29
2. http://courses.cs.vt.edu/~cs5204/fall00/protection/rsa.html