

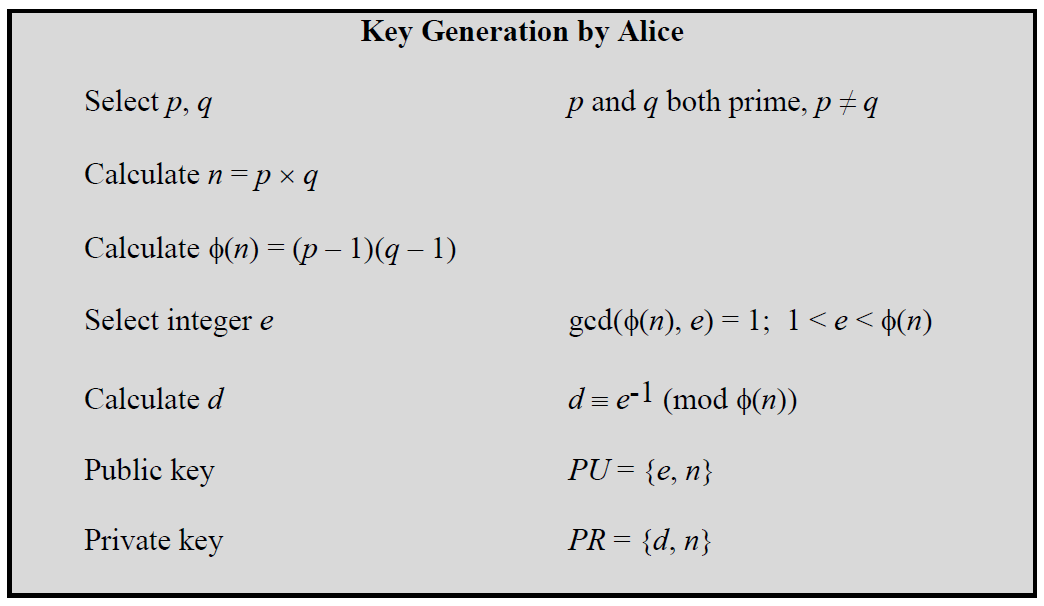
**SUMMER SEMESTER 2024**

**APT3090 CRYPTOGRAPHY AND NETWORK SECURITY**

Name: Hongki Kim

ID No.: 659070

Write a program using any Object oriented programming language to show implementation of RSA. The input p and q should be generated by randomly (15 Marks)



To encrypt a message, M, with the public key, create the ciphertext, C, using the equation: C = Me mod n

The receiver then decrypts the ciphertext with the private key using the equation: M = Cd mod n

**Evaluation Criteria**

1. Correctness of code
2. Random number Generation of p, q
3. Generation of multiple pairs of PU ( e,n) and PR Keys (d,n)
4. Encryption and Decryption of input message
5. A lab report showing your code , sample outputs and code explanation

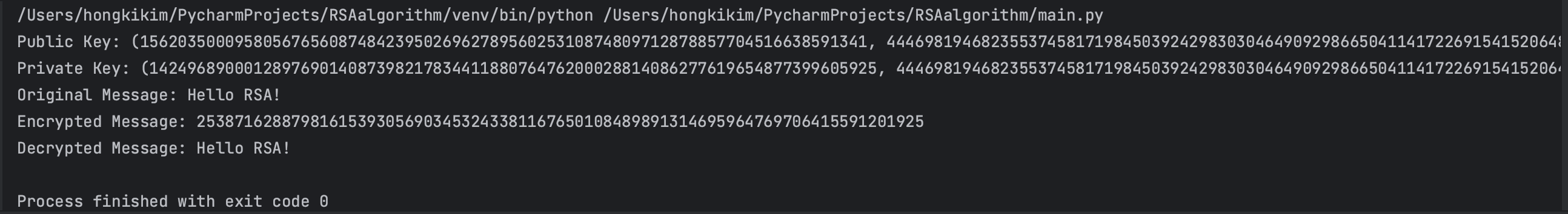
Answer:

What is RSA?  
RSA (Rivest–Shamir–Adleman) is a widely used public-key cryptosystem for secure data transmission. It relies on the mathematical properties of prime numbers and modular arithmetic. This lab report details the implementation of RSA in Python, demonstrating key generation, encryption, and decryption processes.

Code for RSA (Python):

import random  
from sympy import isprime  
from math import gcd  
  
  
class RSA:  
 def \_\_init\_\_(self, bit\_length=128):  
 self.bit\_length = bit\_length  
 self.p = self.generate\_prime()  
 self.q = self.generate\_prime()  
 while self.q == self.p:  
 self.q = self.generate\_prime()  
  
 self.n = self.p \* self.q  
 self.phi\_n = (self.p - 1) \* (self.q - 1)  
 self.e = self.generate\_e(self.phi\_n)  
 self.d = self.modinv(self.e, self.phi\_n)  
  
 self.public\_key = (self.e, self.n)  
 self.private\_key = (self.d, self.n)  
  
 def generate\_prime(self):  
 while True:  
 num = random.getrandbits(self.bit\_length)  
 if isprime(num):  
 return num  
  
 def generate\_e(self, phi\_n):  
 while True:  
 e = random.randrange(2, phi\_n)  
 if gcd(e, phi\_n) == 1:  
 return e  
  
 def modinv(self, a, m):  
 m0, x0, x1 = m, 0, 1  
 if m == 1:  
 return 0  
 while a > 1:  
 q = a // m  
 m, a = a % m, m  
 x0, x1 = x1 - q \* x0, x0  
 if x1 < 0:  
 x1 += m0  
 return x1  
  
 def encrypt(self, message):  
 m = int.from\_bytes(message.encode('utf-8'), 'big')  
 c = pow(m, self.e, self.n)  
 return c  
  
 def decrypt(self, ciphertext):  
 m = pow(ciphertext, self.d, self.n)  
 message\_length = (m.bit\_length() + 7) // 8  
 message = m.to\_bytes(message\_length, 'big').decode('utf-8', errors='ignore')  
 return message  
  
  
# Example usage  
rsa = RSA(bit\_length=128) # 128-bit primes for more robust testing  
  
print(f"Public Key: {rsa.public\_key}")  
print(f"Private Key: {rsa.private\_key}")  
  
message = "Hello RSA!"  
print(f"Original Message: {message}")  
  
ciphertext = rsa.encrypt(message)  
print(f"Encrypted Message: {ciphertext}")  
  
decrypted\_message = rsa.decrypt(ciphertext)  
print(f"Decrypted Message: {decrypted\_message}")

Result:



Explanation for the Code:  
Libraries Used:

random: Generates random numbers.

sympy.isprime: Checks if a number is prime.

math.gcd: Computes the greatest common divisor of two numbers.

Class Initialization:

bit\_length: Specifies the bit length for prime generation (default is 128 bits).

Generates two distinct prime numbers p and q.

Computes n as the product of p and q.

Calculates Euler's totient function phi(n) = (p-1)(q-1).

Generates the public exponent e and computes the private exponent d.

Stores the public key (e, n) and private key (d, n).

Prime Generation:

Generates a random number of specified bit length.

Checks if the number is prime using sympy.isprime.

Returns the prime number once found.

Public Exponent Generation:

Generates a random number e such that 1 < e < phi(n).

Ensures that e is coprime with phi(n) using the greatest common divisor (gcd).

Modular Inverse Calculation:

Computes the modular inverse of a modulo m using the Extended Euclidean Algorithm.

This is used to compute the private exponent d.

Encryption:

Converts the plaintext message to an integer representation.

Encrypts the message using the formula c = m^e mod n.

Returns the ciphertext c.

Decryption:

Decrypts the ciphertext using the formula m = c^d mod n.

Converts the resulting integer back to a byte array and then to a UTF-8 string.

Ignores any decoding errors to handle potential padding issues.

Returns the decrypted message.