# **Rivest–Shamir–Adleman (RSA) Algorithm**

1. **Introduction**

The Rivest–Shamir–Adleman (RSA) algorithm is an asymmetric-key cryptographic technique widely used for secure data transmission. Unlike symmetric encryption, RSA uses a pair of keys - a public key for encryption and a private key for decryption. Its security is based on the mathematical difficulty of factoring large prime numbers, making it a reliable choice for ensuring confidentiality and authenticity.

This report details the implementation of RSA in Python, explaining the processes of key generation, message encryption and decryption, and the underlying mathematical principles that ensure security.

1. **Code**

import random

from sympy import isprime

# Generates a large prime number of specified bit size

def generate\_prime(bitsize=2048):

while True:

number = random.getrandbits(bitsize)

if isprime(number):

return number

# Calculates greatest common divisor of two numbers

def gcd(a, b):

while b:

a, b = b, a % b

return a

# Finds modular inverse using Extended Euclidean Algorithm

def mod\_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

# Generates RSA public and private key pairs

def generate\_keys(bitsize=2048):

p = generate\_prime(bitsize)

q = generate\_prime(bitsize)

n = p \* q

phi = (p - 1) \* (q - 1)

e = random.randrange(2, phi)

while gcd(e, phi) != 1:

e = random.randrange(2, phi)

d = mod\_inverse(e, phi)

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

# Encrypts message using RSA public key

def encrypt(public\_key, message):

e, n = public\_key

message\_int = int.from\_bytes(message.encode(), 'big')

cipher = pow(message\_int, e, n)

return cipher

# Decrypts ciphertext using RSA private key

def decrypt(private\_key, cipher):

d, n = private\_key

decrypted\_int = pow(cipher, d, n)

decrypted\_message = decrypted\_int.to\_bytes((decrypted\_int.bit\_length() + 7) // 8, 'big').decode()

return decrypted\_message

if \_\_name\_\_ == "\_\_main\_\_":

# Generate key pairs

public\_key, private\_key = generate\_keys(bitsize=2048)

# Get user input message

message = input("Enter a message to encrypt: ")

print(f"Original Message: {message}")

# Encrypt and display result

encrypted\_message = encrypt(public\_key, message)

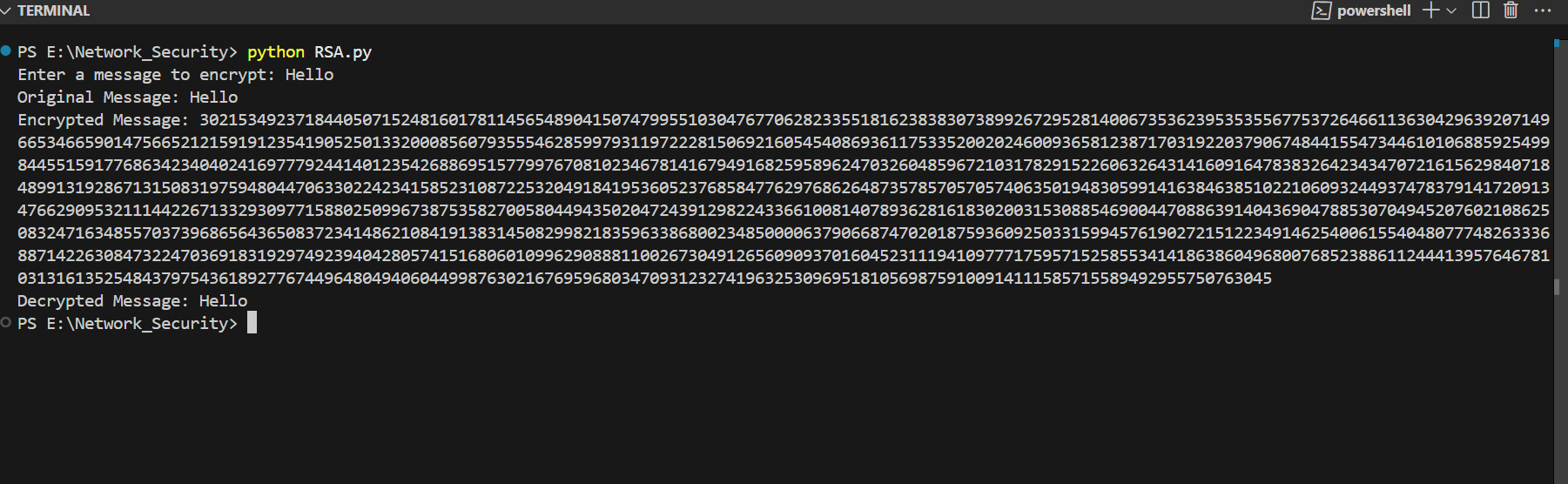
print(f"Encrypted Message: {encrypted\_message}")

# Decrypt and display result

decrypted\_message = decrypt(private\_key, encrypted\_message)

print(f"Decrypted Message: {decrypted\_message}")

1. **Output**

****

1. **Explanation of Implementation**

**Key Generation Process :**

The RSA key generation follows these steps:

1. **Prime Number Generation**
   * + Two large 2048-bit prime numbers (p and q) are generated using random.getrandbits() (cryptographically secure random number generator).
     + Primality is verified using the Miller-Rabin test (via sympy.isprime).
2. **Modulus Calculation**
   * + Compute the RSA modulus: n = p × q
     + Compute Euler's totient function: φ(n) = (p-1)(q-1)
3. **Public Key Selection**
   * + Choose public exponent *e* randomly from [2, φ(n)-1] such that:

gcd(e, φ(n)) = 1

* + - This ensures *e* is coprime with φ(n).

1. **Private Key Derivation**
   * + Compute private exponent d as the modular inverse of e:

d ≡ e⁻¹ mod φ(n)

* + - Implemented using the Extended Euclidean Algorithm.

Therefore, Public key and Private key will be

* + - Public Key: (e, n)
    - Private Key: (d, n)

**Encryption Process :**

To encrypt a plaintext message:

1. **Message Preparation**

Convert the plaintext string to a large integer using UTF-8 encoding.

1. **Encryption Operation**

Computes ciphertext using Python's built-in pow() for modular

exponentiation.

c = mᵉ mod n ,

where m is the plaintext integer and (e,n) is the public key.

**Decryption Process :**

To decrypt the ciphertext:

1. **Decryption Operation**

Recover the plaintext integer using modular exponentiation:

m = cᵈ mod n

Where (d, n) is the private key.

1. **Message Reconstruction**

Convert the decrypted integer back to the original string using UTF-8

decoding.

1. **Conclusion**

This implementation demonstrates RSA encryption and decryption in Python,

showcasing the fundamental principles of public-key cryptography. By generating

secure 2048-bit keys and using modular arithmetic, it successfully encrypts

plaintext into ciphertext using a public key and decrypts ciphertext back to

plaintext using a private key.