**Implementation of RSA Algorithm**

**AP25122050012**

**Ramanadham Shiny**

**Rivest-Shamir-Adleman (RSA) Algorithm:**

This is asymmetric cryptographic algorithm to transfer data securely. This uses two keys – **Public Key** (Encryption), **Private Key** (Decryption).It relies on the difficulty of factoring large semiprime numbers (products of two large prime).This is a classic example of a trapdoor function—easy to compute one way, hard to reverse without a secret. One of the first practical public-key cryptosystems and still widely

**Limitations:**

* Typically used to encrypt small data.
* Slow performance.
* Take more processor time as it take large key size.

**Code:**

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.hazmat.backends import default\_backend

import hashlib

# 1. Generate RSA Key Pair

def generate\_keys():

private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048,

backend=default\_backend()

)

public\_key = private\_key.public\_key()

return private\_key, public\_key

# 2. Serialize Keys (PEM format)

def serialize\_keys(private\_key, public\_key):

pem\_private = private\_key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.TraditionalOpenSSL,

encryption\_algorithm=serialization.NoEncryption()

)

pem\_public = public\_key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

return pem\_private, pem\_public

# 3. Encrypt Message

def encrypt\_message(public\_key, message):

return public\_key.encrypt(

message.encode(),

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

)

# 4. Decrypt Message

def decrypt\_message(private\_key, ciphertext):

return private\_key.decrypt(

ciphertext,

padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)

).decode()

# 5. Sign Message

def sign\_message(private\_key, message):

return private\_key.sign(

message.encode(),

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

# 6. Verify Signature

def verify\_signature(public\_key, message, signature):

try:

public\_key.verify(

signature,

message.encode(),

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

return True

except Exception:

return False

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

pem\_private, pem\_public = serialize\_keys(private\_key, public\_key)

message = "Confidential RSA message"

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

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

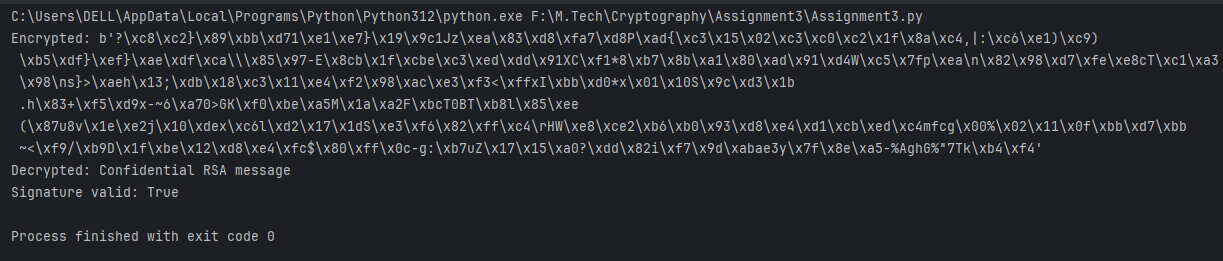
signature = sign\_message(private\_key, message)

is\_valid = verify\_signature(public\_key, message, signature)

print("Encrypted:", ciphertext)

print("Decrypted:", decrypted)

print("Signature valid:", is\_valid)

**Output:**

**GitHub Link -** [**https://github.com/shinyramanadham-AD/Applied-Cryptography**](https://github.com/shinyramanadham-AD/Applied-Cryptography)