from cryptography.hazmat.primitives.asymmetric import rsa  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives import hashes  
from cryptography.hazmat.primitives.asymmetric import padding  
  
  
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  
  
  
def sign(message, private\_key):  
 message = bytes(str(message), 'utf-8')  
  
 signature = private\_key.sign(  
 message,  
 padding.PSS(  
 mgf=padding.MGF1(hashes.SHA256()),  
 salt\_length=padding.PSS.MAX\_LENGTH  
 ),  
 hashes.SHA256()  
 )  
  
 return signature  
  
  
def verify(message, sig, public\_key):  
 message = bytes(str(message), 'utf-8')  
 try:  
 public\_key.verify(  
 sig,  
 message,  
 padding.PSS(  
 mgf=padding.MGF1(hashes.SHA256()),  
 salt\_length=padding.PSS.MAX\_LENGTH  
 ),  
 hashes.SHA256()  
 )  
 return True  
 except Exception as e:  
 print("Error verifying signature:", str(e))  
 return False  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 private\_key, public\_key = generate\_keys()  
 print(private\_key)  
 print(public\_key)  
  
 message = "Hi, I'm a Blockchain developer"  
 sig = sign(message, private\_key)  
 print(sig)  
 correct = verify(message,sig,public\_key)  
 if correct:  
 print("Successful")  
 else:  
 print("Failed")

"C:\Users\hp\PycharmProjects\ digital siganture\venv\Scripts\python.exe" "C:\Users\hp\PycharmProjects\ digital siganture\main.py"

<cryptography.hazmat.backends.openssl.rsa.\_RSAPrivateKey object at 0x00000269AE0F3C40>

<cryptography.hazmat.backends.openssl.rsa.\_RSAPublicKey object at 0x00000269AE0F3EB0>

b'\xa9a\xfd\x10.\xe0o\xe6\xbc\xdf\xe7p\xf6g\xf0\x87\xc4m\x8e\xd9\xf7Pe0\xd5\x15\x97<A\t\x06\x06\x15\x83\x03qw\xfa\x0e\xa7\x8a\xfa\xa8\xe9}=\xbe\x11\x9b\xf6.\xe2w\xef\xfd\xed\x12\x8e\xfd\x9b\xf8\xc9\x96\x00\xf2\xfd7\x03\xbeZ\x04BxA\x96\xae\x94rd2\xed\'\xb3\xb9Ax0|a\x03\xa0d\x8f\x88\xc6\x8f?\x8d\x9c\x12\tV\x1b\x9a\'\xc6\xa3\x95\x13F\x8a~\x97`\xb5\t\x03\xe3\x12\xcd\x9a\x18f\xc8I\xfeL\xdftQ2\x1a3$\xc0\xe9\xff\xb7zT\xff\xc5f\xd8|\xe5\xdd\xc04\xba9\xe0\xec\xdb\x96+KM\xa3[O\x00\x0f\x90e\x8a\xb2v\xaa\xc2\xben\x8d\xc8}\xf4o]\xf1\x12\xf7?\xea\x85\xf6$\x86\xe0\xcb\xdf\xe0\xfc\xa8%CM\xf6\xa9{\xa2\xe2\xe0\xc6\x08\xefX\x07m\xe8\x00\xff2G\xef\x1b\x82\*\xe4\x97F\x967 \xc7S%\x1e\xa2xW"N\xf7(h\r\x8d\r\x95t{W\x95\x11\r\xb0V(\xf5\xb2\xce\xac\xe1\xa2\xb9\xca'

Successful

The above code is the ouput

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import padding

These lines import the necessary modules and classes from the **cryptography** library.

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

The **generate\_keys()** function generates a new RSA private key using the **rsa.generate\_private\_key()** method. It sets the public exponent and key size parameters, and the default backend is used. The corresponding public key is then obtained from the private key using the **public\_key()** method. Both the private key and public key are returned as a tuple.

def sign(message, private\_key):

message = bytes(str(message), 'utf-8')

signature = private\_key.sign(

message,

padding.PSS(

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

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

return signature

The **sign()** function takes a message and a private key as input. The message is converted to bytes using UTF-8 encoding. The private key's **sign()** method is then called to create a signature for the message. The **padding.PSS** scheme is used with the **mgf** parameter set to **padding.MGF1** using SHA256, and the salt length is set to the maximum length allowed. The hash algorithm used is SHA256. The resulting signature is returned.

def verify(message, sig, public\_key):

message = bytes(str(message), 'utf-8')

try:

public\_key.verify(

sig,

message,

padding.PSS(

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

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

return True

except Exception as e:

print("Error verifying signature:", str(e))

return False

The **verify()** function takes a message, a signature, and a public key as input. The message is converted to bytes using UTF-8 encoding. The public key's **verify()** method is called to verify the signature against the message. The same padding scheme and hash algorithm used during signing are provided. If the verification succeeds, **True** is returned. If an exception occurs during verification, an error message is printed, and **False** is returned.

if \_\_name\_\_ == '\_\_main\_\_':

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

print(private\_key)

print(public\_key)

message = "Hi, I'm a Blockchain developer"

sig = sign(message, private\_key)

print(sig)

correct = verify(message, sig, public\_key)

if correct:

print("Verification successful")

else:

print("Verification failed")

The code block under **if \_\_name\_\_ == '\_\_main\_\_':** is the main entry point of the script. It generates a new set of private key and public key using the **generate\_keys()** function. The private key and public key are then printed. A message is created, and its signature is generated using the **sign()** function. The signature is printed to the console. Finally, the **verify()** function is called to verify the signature against the message using the public key. The