Explanation & Instruction for the running code

Submitted by

Tasfia Tasneem Annesha

ID: 190041220

1. Explanation of Code

1.Install Required Libraries

```
[ ]  #Installing Required Libraries
     !pip install cryptography
```

```
Requirement already satisfied: cryptography in /usr/local/lib/python3.10/dist-packages (41.0.7)
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.10/dist-packages (from cryptography) (1.16.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.12->cryptography) (2.21)
```

The command `!pip install cryptography` is a shell command executed within a Jupyter notebook or a Colab environment. It is used to install the Python package named "cryptography" using the pip package manager. The "cryptography" package provides cryptographic recipes and primitives to Python developers.

2.Import necessary modules

```
[ ]  # Import necessary modules from the cryptography library
     from cryptography.hazmat.backends import default_backend #for providing cryptographic primitives and algorithms
     from cryptography.hazmat.primitives import hashes
     from cryptography.hazmat.primitives.asymmetric import rsa
     from cryptography.hazmat.primitives.asymmetric import padding
     from cryptography.hazmat.primitives import serialization
     from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

These import statements bring in the necessary components from the cryptography library to implement various cryptographic functionalities, covering symmetric and asymmetric encryption, hashing, serialization, and the necessary algorithms and modes.

3.Function to generate a key Pair for asymmetric encryption (RSA)

```
[ ]  # Function to generate a key pair for asymmetric encryption (RSA)
     def generate_key_pair():
         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 purpose of this function is to generate a key pair for asymmetric encryption using the RSA algorithm.This line initializes the generation of a private key using the `generate_private_key` method from the `rsa` module. The method takes parameters such as the public exponent and key size. `public_exponent=65537`, sets the public exponent for the RSA key. The value 65537 is commonly used as it is a prime number and provides certain mathematical properties that enhance security. `key_size=2048`, sets the size of the key in bits. In this case, the key size is set to 2048 bits. This is a standard key size for RSA, offering a good balance between security and computational efficiency. `backend=default_backend()` specifies the backend used for key generation. It is set to the default backend, which is chosen based on the available cryptographic libraries. The public key from the generated private key using the `public_key()` method. The public key is the part of the key pair that can be shared publicly.

So, the `generate_key_pair` function generates an RSA key pair consisting of a private key and its corresponding public key, and it returns these keys for use in cryptographic operations.

4. Function to encrypt a message using asymmetric encryption

```
# Function to encrypt a message using asymmetric encryption
def encrypt_asymmetric(message, public_key):
    cipher_text = public_key.encrypt(
        message.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return cipher_text
```

The `encrypt_asymmetric` function takes a message and a public key, encrypts the message using the specified asymmetric encryption algorithm (RSA in this case) with the provided public key, and returns the resulting ciphertext. The use of `OAEP` padding enhances the security of the encryption process.

5. Function to decrypt a message using asymmetric decryption

```
[ ]  # Function to decrypt a message using asymmetric decryption
     def decrypt_asymmetric(cipher_text, private_key):
         plain_text = private_key.decrypt(
             cipher_text,
             padding.OAEP(
                 mgf=padding.MGF1(algorithm=hashes.SHA256()),
                 algorithm=hashes.SHA256(),
                 label=None
             )
         )
         return plain_text.decode()
```

The `` `decrypt_asymmetric` `` function is designed for asymmetric decryption, specifically for RSA encryption. It takes the ciphertext and a private key as input parameters, and uses the `` `decrypt` `` method provided by the `` `private_key` `` object to decrypt the ciphertext. The decryption process employs the Optimal Asymmetric Encryption Padding (OAEP) scheme, specified by the padding parameters, including the Mask Generation Function (MGF1) using SHA256, the hash algorithm

SHA256, and no additional label. After decryption, the resulting plaintext is returned by decoding the byte sequence back to a string. This function completes the asymmetric encryption-decryption cycle, allowing the recipient with the corresponding private key to securely retrieve the original message from the received ciphertext.

6. Function to encrypt and decrypt a message using symmetric encryption (AES)

```
[ ]  # Function to encrypt a message using symmetric encryption (AES)
     def encrypt_symmetric(message, key):
         iv = b'\x00' * 16  # Initialization vector
         cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
         encryptor = cipher.encryptor()
         cipher_text = encryptor.update(message.encode()) + encryptor.finalize()
         return cipher_text

     # Function to decrypt a message using symmetric decryption (AES)
     def decrypt_symmetric(cipher_text, key):
         iv = b'\x00' * 16
         cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
         decryptor = cipher.decryptor()
         plain_text = decryptor.update(cipher_text) + decryptor.finalize()
         return plain_text.decode()
```

The `encrypt_symmetric` function is responsible for symmetric encryption using the Advanced Encryption Standard (AES) algorithm. It takes a plaintext message and a symmetric key as input parameters. The function initializes an initialization vector (`iv`) for the encryption process, creates an AES cipher object with the specified key and mode (Cipher Feedback mode in this case), and encrypts the message using the cipher object. The resulting ciphertext is obtained by updating the encryptor with the message's encoded bytes and finalizing the encryption process. The encrypted message is then returned.

On the other hand, the `decrypt_symmetric` function performs symmetric decryption using AES. It takes a ciphertext and the same symmetric key used for encryption as input parameters. Similar to the encryption process, it initializes an initialization vector (`iv`), creates an AES cipher object with the specified key and mode, and decrypts the ciphertext using the decryptor associated with the cipher object. The resulting plaintext is obtained by updating the decryptor with the ciphertext and finalizing the decryption process. Finally, the decrypted message is returned after decoding the byte sequence back to a string.

In summary, these functions illustrate a complete cycle of symmetric encryption and decryption using the AES algorithm, showcasing the fundamental concepts of symmetric key cryptography.

7. Key Pair generation

```
# Example : Generate key pair
private_key, public_key = generate_key_pair()
```

The provided example demonstrates the generation of an asymmetric key pair using the `generate_key_pair` function. This function, as previously explained, generates an RSA private key and its corresponding public key. In the example, the `generate_key_pair` function is called, and its return values, the private key and public key, are assigned to the variables `private_key` and `public_key`, respectively.

8. Encryption and decryption

```
# Encrypt with asymmetric public key
cipher_text_asymmetric = encrypt_asymmetric(message, public_key)
```

This encrypts the original message asymmetrically using the recipient's public key.

```
# Encrypt the symmetric key with asymmetric public key
symmetric_key = b'\x01' * 32  # Replace with a securely generated key
cipher_text_symmetric_key = encrypt_asymmetric(symmetric_key.decode(), public_key)
```

This generates a symmetric key for further encryption, to be securely replaced and then it encrypts the symmetric key using asymmetric encryption for secure key exchange.

```
# Decrypt the symmetric key with the asymmetric private key
decrypted_symmetric_key = decrypt_asymmetric(cipher_text_symmetric_key, private_key)
```

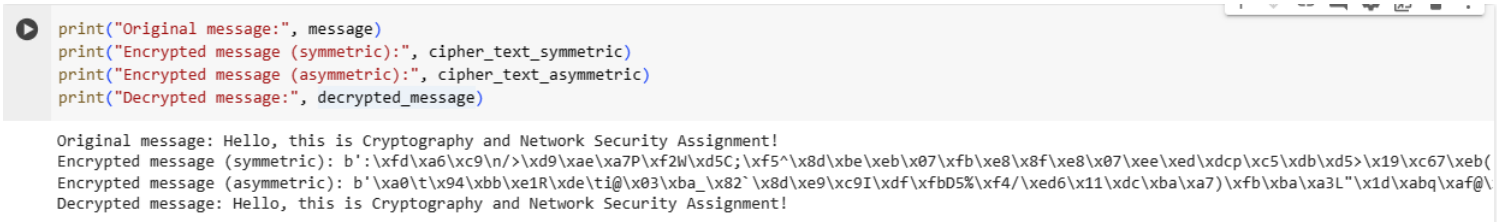
This decrypts the encrypted symmetric key using the recipient's private key for subsequent symmetric encryption and decryption.

```
# Encrypt the message with the symmetric key
cipher_text_symmetric = encrypt_symmetric(message, decrypted_symmetric_key.encode())

# Decrypt the message with the symmetric key
decrypted_message = decrypt_symmetric(cipher_text_symmetric, decrypted_symmetric_key.encode())
```

First it encrypts the original message symmetrically using the decrypted symmetric key. The resulting `cipher_text_symmetric` is the secure ciphertext ready for transmission. Then it decrypts the symmetrically encrypted message using the decrypted symmetric key. The `decrypted_message` holds the original plaintext message, demonstrating the successful symmetric encryption and decryption cycle.

9. Output



2. Instructions for Running the code

- Open Colab in a web browser,
- Upload the notebook,
- Install the required library with `!pip install cryptography`,
- Run each cell sequentially using the play button,
- Review the output for the original, encrypted, and decrypted messages,
- Ensure Python is installed if using a different environment and run the code in a compatible Python environment.