Secure Image Encryption with AES and Quantum Key Distribution (BB84)

Prepared by: **SAPTADIP SAHA**

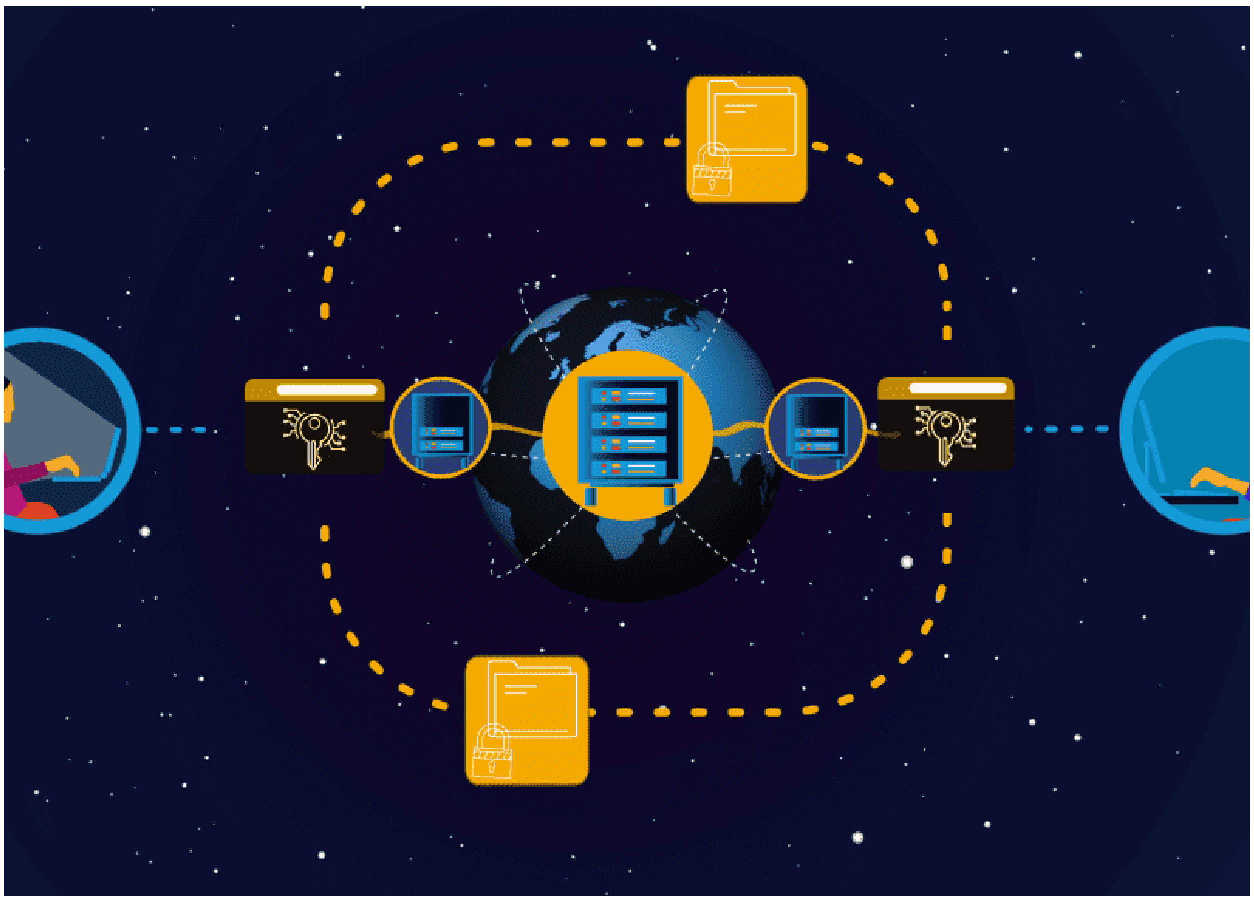


Image source: QuTech

Introduction

This project demonstrates how to securely encrypt images using the Advanced Encryption Standard (AES) algorithm, with encryption keys shared through the BB84 Quantum Key Distribution (QKD) protocol. AES relies on a single symmetric key for both encryption and decryption, making key distribution a critical aspect of security. BB84, a pioneering quantum key distribution method developed by Charles Bennett and Gilles Brassard in 1984, ensures secure key transmission using quantum mechanics. The BB84 protocol's unique use of quantum superposition and entanglement enables secure key sharing, where any attempt at eavesdropping is immediately detectable. By combining AES with BB84, this project ensures that sensitive images are encrypted and transmitted securely, with a high degree of confidence in the integrity and confidentiality of the encryption key, thus providing robust protection against unauthorized access or interception.

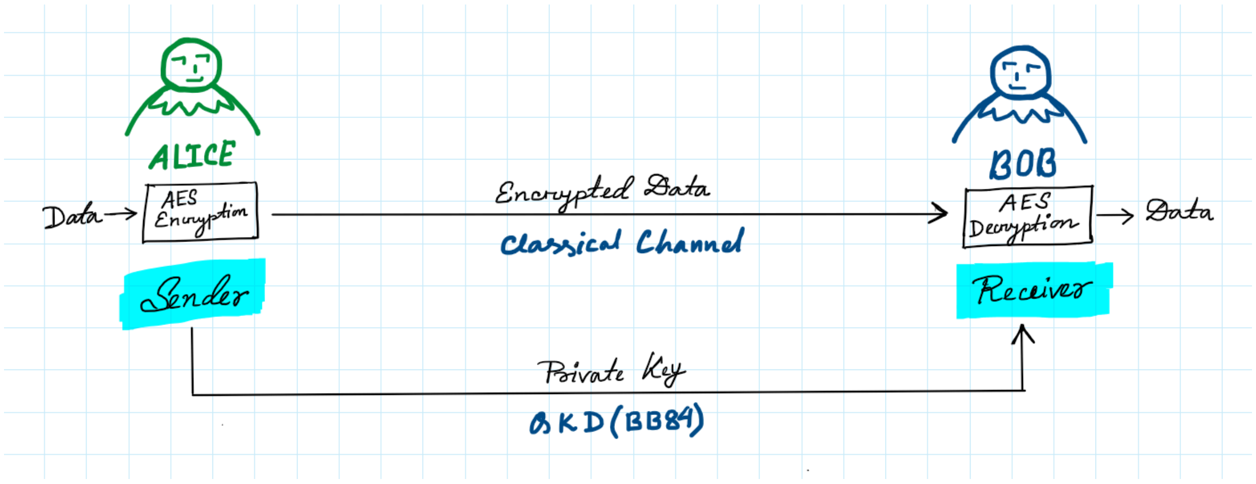


Image Encryption Using AES

The Advanced Encryption Standard (AES) is a symmetric block cipher algorithm adopted by the National Institute of Standards and Technology (NIST) in 2001 to replace the Data Encryption Standard (DES) as the new federal encryption standard. AES operates on fixed-size blocks of 128 bits, with key lengths of 128, 192, or 256 bits, allowing varying degrees of security.

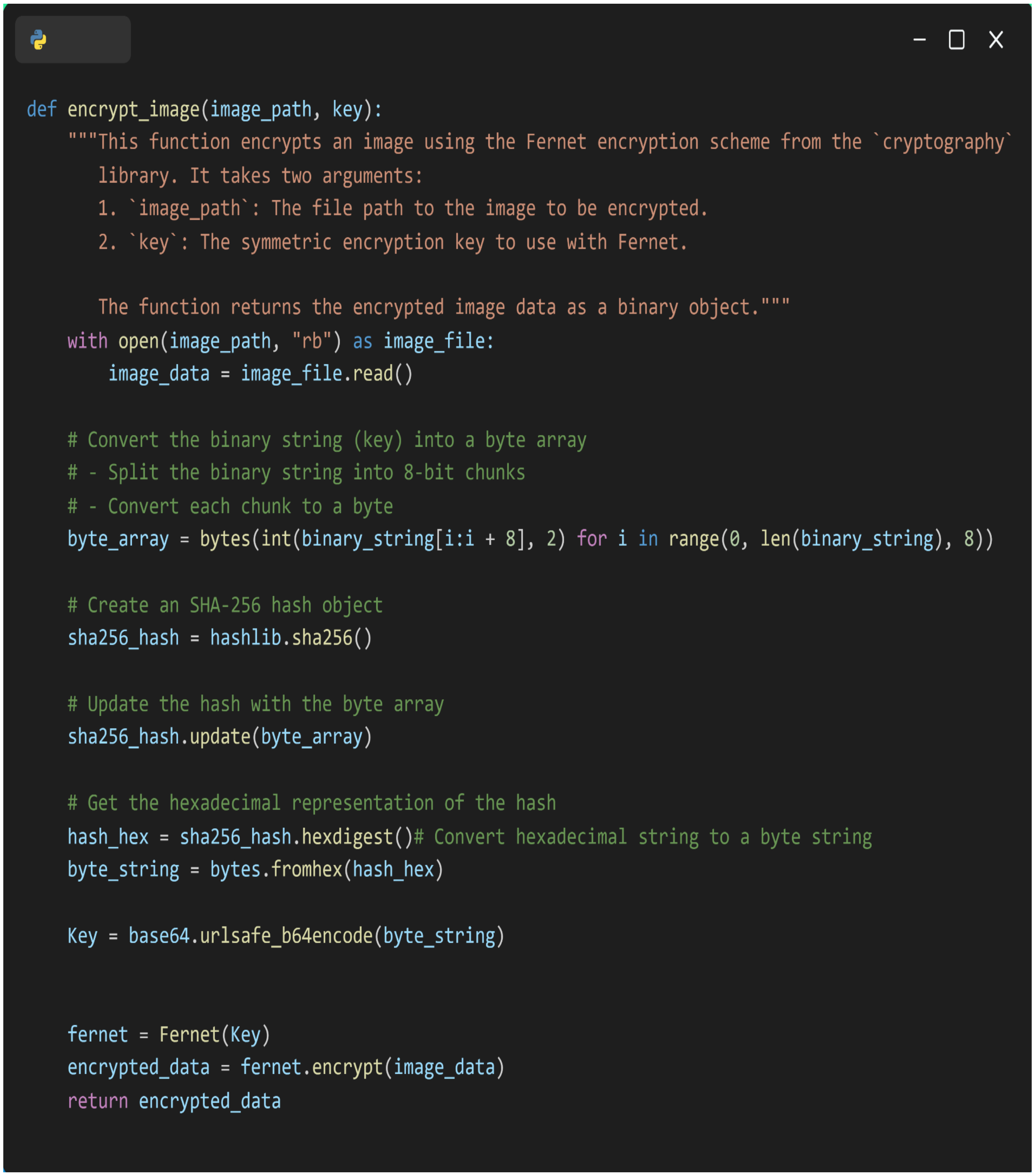
AES encryption consists of multiple transformation steps applied in a sequence, known as rounds. The number of rounds depends on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each round includes a series of operations: SubBytes, where bytes are substituted according to a predefined S-box (substitution box); ShiftRows, involving row-wise byte permutations; MixColumns, a column-wise transformation; and AddRoundKey, where a round-specific subkey derived from the main key is XORed with the block. The same series of transformations is used in reverse during decryption, with inverse operations for each step.

For this project, we will be using the `fernet` module from the `cryptography` library in Python to implement 128-bit Advanced Encryption Standard (AES) encryption. The `fernet` module provides a simple interface for symmetric encryption and decryption, automatically handling key generation, encryption, and decryption using AES with a 128-bit key length. It also includes integrity checks to ensure that the encrypted data has not been tampered with during transmission or storage. Therefore the key expected by the constructor consists of two 128-bit keys: one for signing, and the other for encryption, concatenated in that order. This makes it an ideal choice for securely encrypting and decrypting sensitive data, such as images while simplifying the implementation of AES encryption in Python projects. By using `fernet`, we can focus on securely handling and distributing the encryption key, which can be achieved through secure methods like the BB84 Quantum Key Distribution protocol.

The source code reveals that Fernet requires a 32-byte key, comprising a 16-byte signing key and a 16-byte encryption key. Fernet generates a 32-byte byte string using the `os.urandom(32)` function, which is then encoded with `base64.urlsafe_b64encode()` to create a 64-byte key. This key is used to initialize a Fernet instance for encryption and decryption.

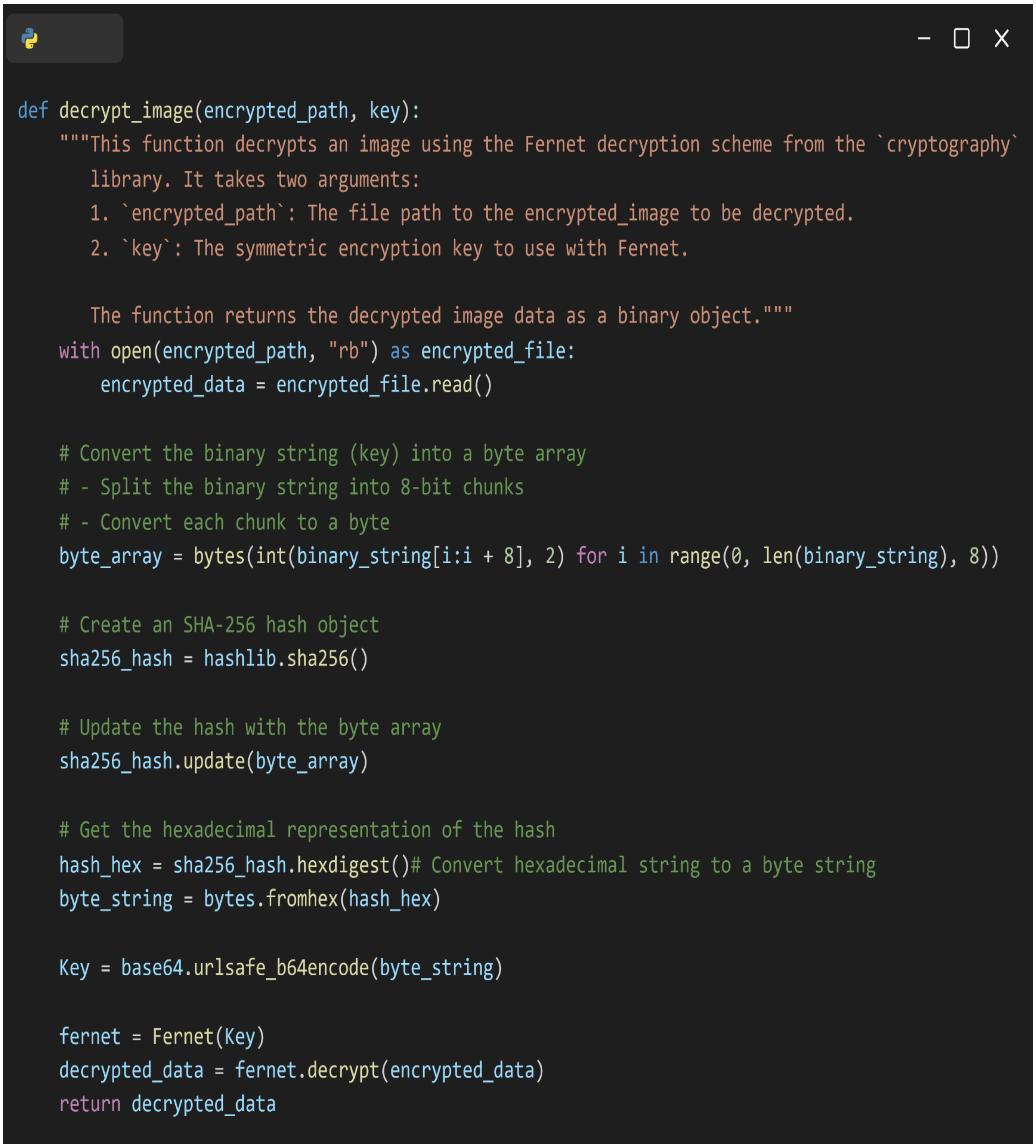
To integrate with Quantum Key Distribution (QKD), we start by preparing a 128-bit shared key using a QKD protocol like BB84. This shared key is then transformed into a 32-byte hexadecimal byte string using SHA-256. This 32-byte hexadecimal byte string is then used as the key to create a Fernet instance, enabling the encryption and decryption of data. Thus, the secure QKD-based shared key is transformed into the appropriate key format for Fernet's symmetric encryption process.

Encryption Function:



```python
def encrypt_image(image_path, key):
    """This function encrypts an image using the Fernet encryption scheme from the `cryptography`
       library. It takes two arguments:
       1. `image_path`: The file path to the image to be encrypted.
       2. `key`: The symmetric encryption key to use with Fernet.

       The function returns the encrypted image data as a binary object."""
    with open(image_path, "rb") as image_file:
         image_data = image_file.read()

    # Convert the binary string (key) into a byte array
    # - Split the binary string into 8-bit chunks
    # - Convert each chunk to a byte
    byte_array = bytes(int(binary_string[i:i + 8], 2) for i in range(0, len(binary_string), 8))

    # Create an SHA-256 hash object
    sha256_hash = hashlib.sha256()

    # Update the hash with the byte array
    sha256_hash.update(byte_array)

    # Get the hexadecimal representation of the hash
    hash_hex = sha256_hash.hexdigest()# Convert hexadecimal string to a byte string
    byte_string = bytes.fromhex(hash_hex)

    Key = base64.urlsafe_b64encode(byte_string)


    fernet = Fernet(Key)
    encrypted_data = fernet.encrypt(image_data)
    return encrypted_data
```

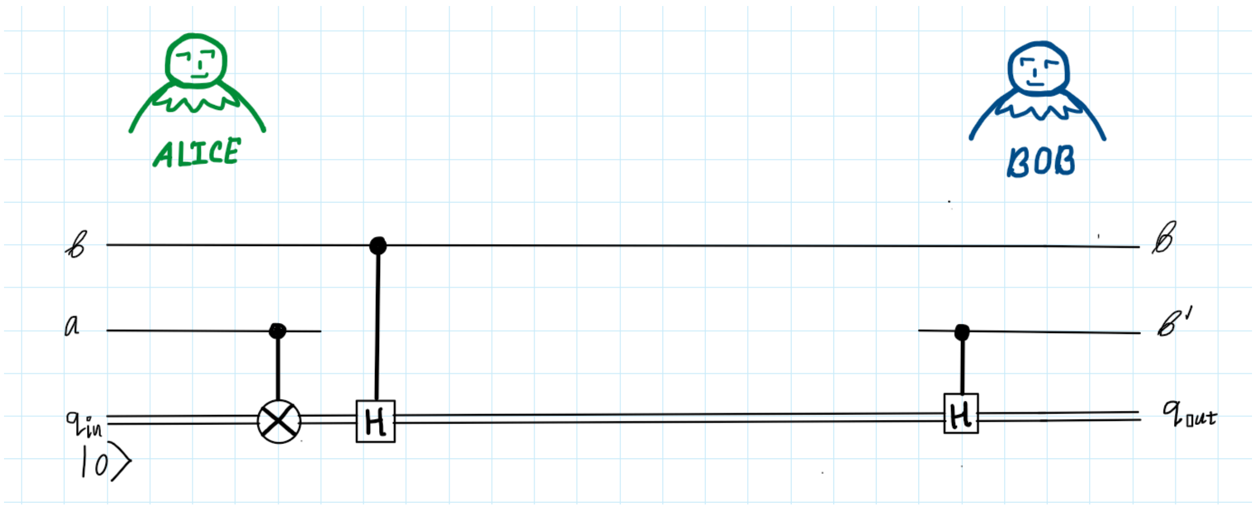
Decryption Function:



```python
def decrypt_image(encrypted_path, key):
    """This function decrypts an image using the Fernet decryption scheme from the `cryptography`
       library. It takes two arguments:
       1. `encrypted_path`: The file path to the encrypted_image to be decrypted.
       2. `key`: The symmetric encryption key to use with Fernet.

       The function returns the decrypted image data as a binary object."""
    with open(encrypted_path, "rb") as encrypted_file:
        encrypted_data = encrypted_file.read()

    # Convert the binary string (key) into a byte array
    # - Split the binary string into 8-bit chunks
    # - Convert each chunk to a byte
    byte_array = bytes(int(binary_string[i:i + 8], 2) for i in range(0, len(binary_string), 8))

    # Create an SHA-256 hash object
    sha256_hash = hashlib.sha256()

    # Update the hash with the byte array
    sha256_hash.update(byte_array)

    # Get the hexadecimal representation of the hash
    hash_hex = sha256_hash.hexdigest()# Convert hexadecimal string to a byte string
    byte_string = bytes.fromhex(hash_hex)

    Key = base64.urlsafe_b64encode(byte_string)

    fernet = Fernet(Key)
    decrypted_data = fernet.decrypt(encrypted_data)
    return decrypted_data
```

Key Sharing using BB84 Protocol

The BB84 quantum key distribution (QKD) protocol, developed by Charles Bennett and Gilles Brassard in 1984, utilizes principles of quantum mechanics to generate secure cryptographic keys. It was the first quantum cryptography method to exploit the no-cloning theorem, which states that an unknown quantum state cannot be perfectly copied, providing inherent security against eavesdropping.

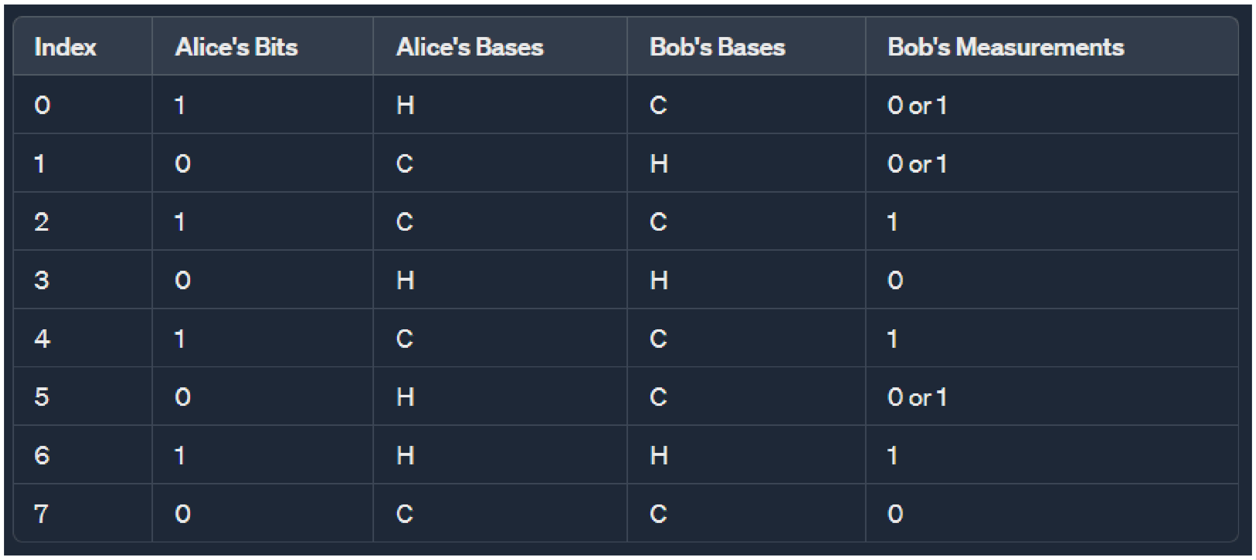


Note: a = Alice's bits (0 or 1), b = Alice's bases (C or H), b' = Bob's bases (C or H)

BB84 operates on the principle that it's impossible to gain information to distinguish between two non-orthogonal states without disrupting the signal. In the BB84 protocol, two parties, Alice and Bob, are connected by a classical communication channel, and Alice can send qubits (quantum bits) to Bob via a one-way quantum channel. The protocol proceeds as follows:

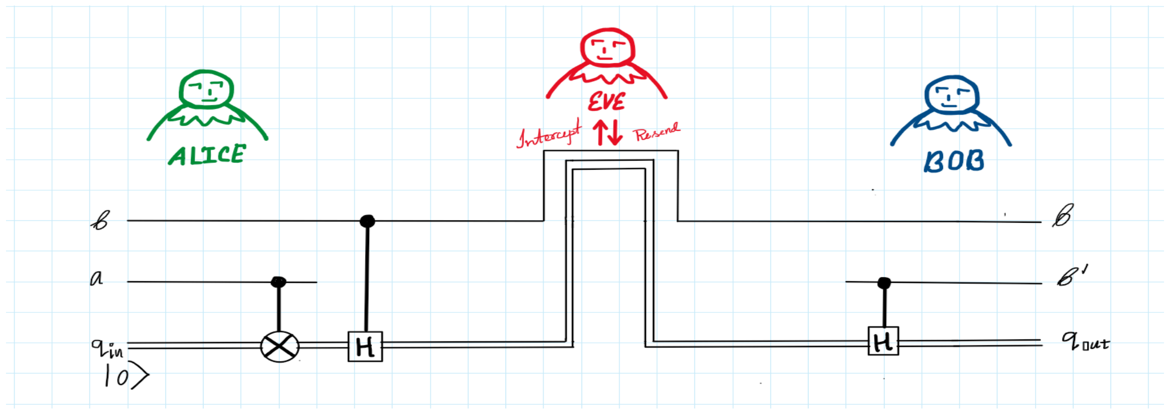
1. Alice generates two random binary strings, a and b, both of length n. String a represents the qubit states, while string b represents the basis used for encoding.
2. Alice prepares the qubits based on the following rules:
    - If a[i] = 0 and b[i] = 0, the qubit is in the $|0\rangle$ state.
    - If a[i] = 1 and b[i] = 0, the qubit is in the $|1\rangle$ state.

    - If a[i] = 0 and b[i] = 1, the qubit is in the |+⟩ state (superposition of |0⟩ and |1⟩).
    - If a[i] = 1 and b[i] = 1, the qubit is in the |-⟩ state (another superposition state).
3. Alice sends these qubits to Bob.
4. Bob generates a random binary string c, also of length n. He uses this string to decide how to measure the qubits: if c[i] = 0, he measures in the computational basis (|0⟩, |1⟩); if c[i] = 1, he measures in the Hadamard basis (|+⟩, |-⟩). He records his measurements in the string m.
5. After the measurements, Alice and Bob exchange their basis strings, b and c. They compare the positions where the bases match, and for these positions, the measurements should agree. These shared bits form a key that can be used for secure cryptography.



Index	Alice's Bits	Alice's Bases	Bob's Bases	Bob's Measurements
0	1	H	C	0 or 1
1	0	C	H	0 or 1
2	1	C	C	1
3	0	H	H	0
4	1	C	C	1
5	0	H	C	0 or 1
6	1	H	H	1
7	0	C	C	0

BB84 is protected against intercept-and-resend attacks. This security comes from the no-cloning theorem, which states that a qubit in an unknown state cannot be duplicated. This means that any attempt to measure a qubit will disrupt its original state. We can consider a scenario where an eavesdropper, Eve, intercepts all of Alice's qubits. If Eve measures these qubits in a randomly chosen basis, she won't necessarily get the same result as Alice's intended state. Even if Eve then sends new qubits to Bob based on her measurements (Intercept and Resend), there's a high chance that these qubits won't align with Alice's original states, leading to inconsistencies when Alice and Bob compare their measurements. This way, Alice and Bob can detect eavesdropping by checking a portion of their shared key for discrepancies. If they find mismatches, it's a sign that someone tampered with the qubits, alerting them to potential eavesdropping.



Simplified illustration of Eve executing intercept-resend attack.

Non-eavesdropped protocol:

Alice's basis:	CHCCCHCH
Bob's basis:	CHHHHHHH
Alice's bits:	10111100
Bases match::	XX___X_X
Expected key:	1010
Actual key:	1010

Eavesdropped protocol:

Alice's basis:	HCHCCHCH
Bob's basis:	HHHCCHCC
Alice's bits:	00100101
Bases match::	X_XXXXX_
Expected key:	010010
Actual key:	111011

Conclusion

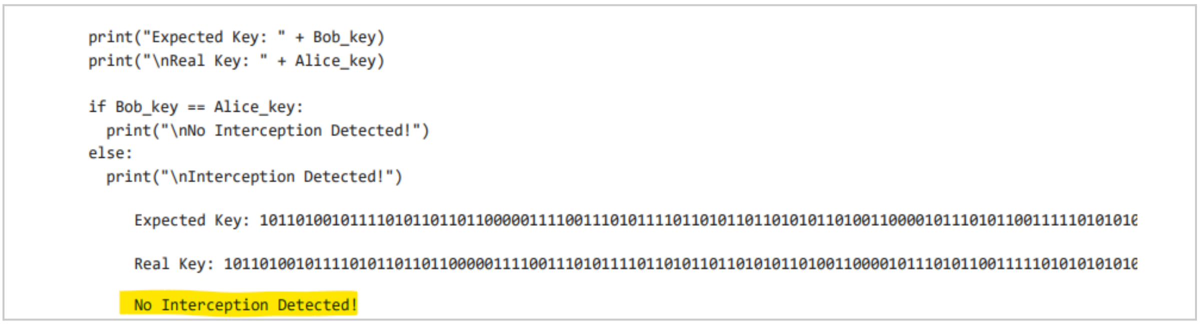
This project demonstrates image encryption using symmetric key encryption algorithms like AES and secure key distribution through the BB84 Quantum Key Distribution protocol. By combining these technologies, the project showcases a robust method for encrypting sensitive images and securely sharing encryption keys using quantum principles. This approach ensures the confidentiality and integrity of data, offering a secure framework for protecting sensitive information.

Jupyter Notebook

Notebook Link: [Image Encryption with AES and BB84]()

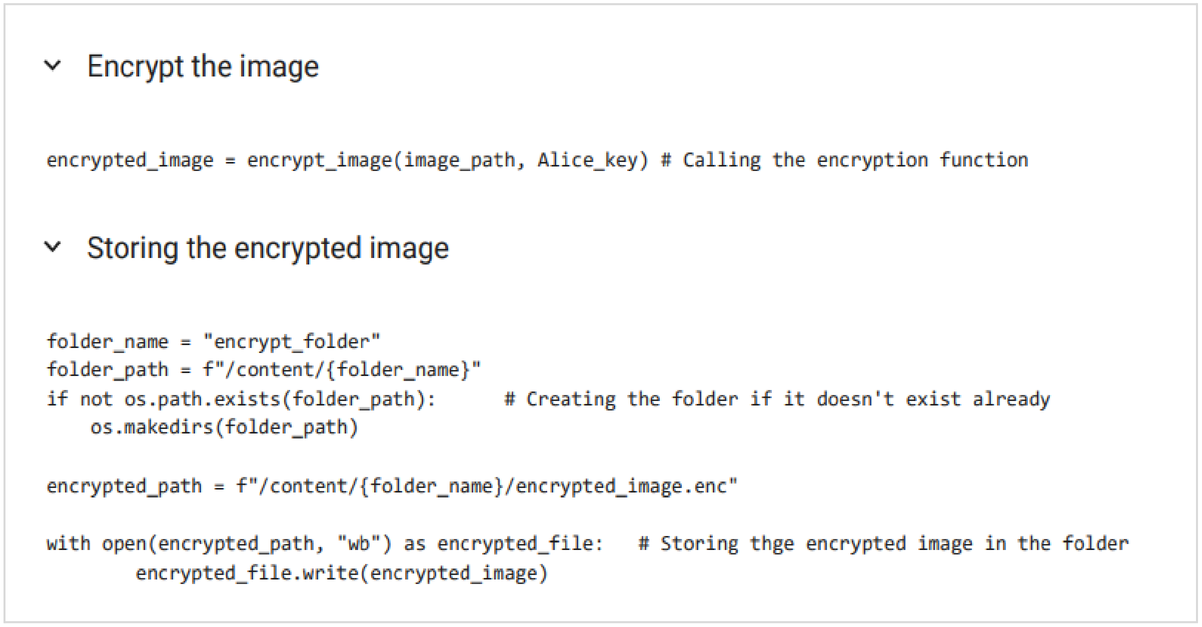
Simulation of Non-eavesdropped protocol:

```
# Experiment parameters
final_key_size = 128   # Size of the final Shared key after completing all iterations
circuit_size = 8  # No. of qubits
eavesdropping = 0   # 0: No Eve, 1: Eve is here
```



```
print("Expected Key: " + Bob_key)
print("\nReal Key: " + Alice_key)

if Bob_key == Alice_key:
  print("\nNo Interception Detected!")
else:
  print("\nInterception Detected!")

    Expected Key: 1011010010111101011011011000001111001110101111011010110110101011010011000010111010110011111010101

    Real Key: 1011010010111101011011011000001111001110101111011010110110101011010011000010111010110011111010101010

    No Interception Detected!
```

⌄ Alice storing the shared secret key

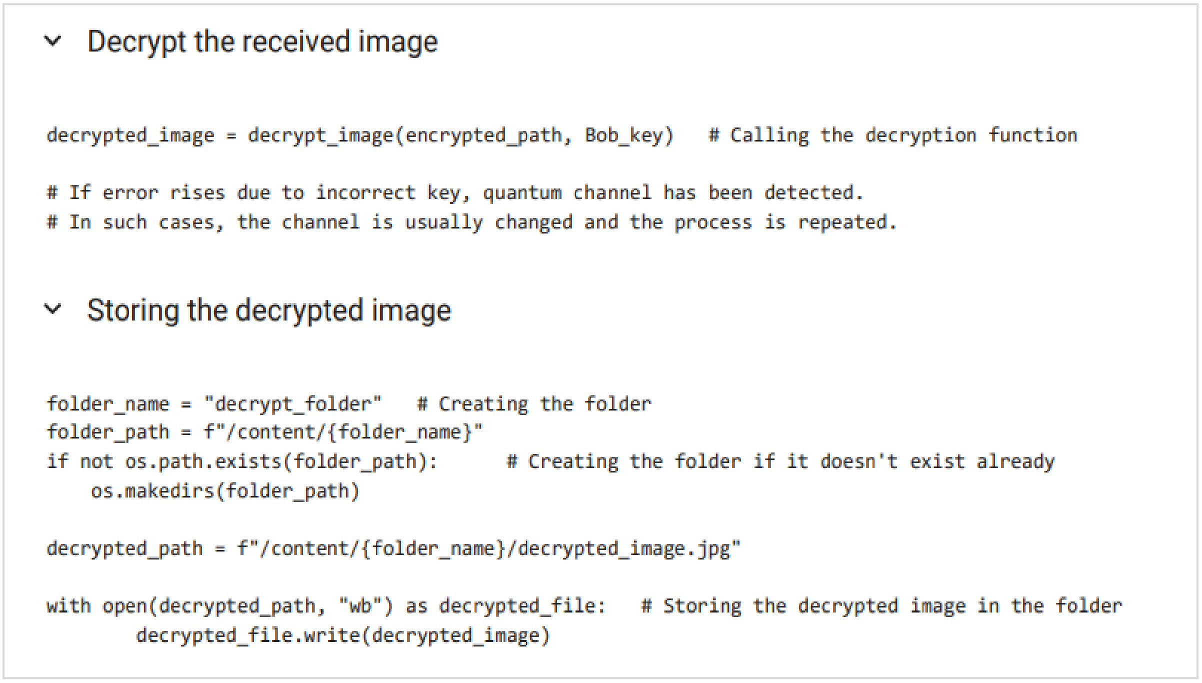


Encrypt the image

```python
encrypted_image = encrypt_image(image_path, Alice_key) # Calling the encryption function
```

Storing the encrypted image

```python
folder_name = "encrypt_folder"
folder_path = f"/content/{folder_name}"
if not os.path.exists(folder_path):      # Creating the folder if it doesn't exist already
    os.makedirs(folder_path)

encrypted_path = f"/content/{folder_name}/encrypted_image.enc"

with open(encrypted_path, "wb") as encrypted_file:   # Storing thge encrypted image in the folder
        encrypted_file.write(encrypted_image)
```

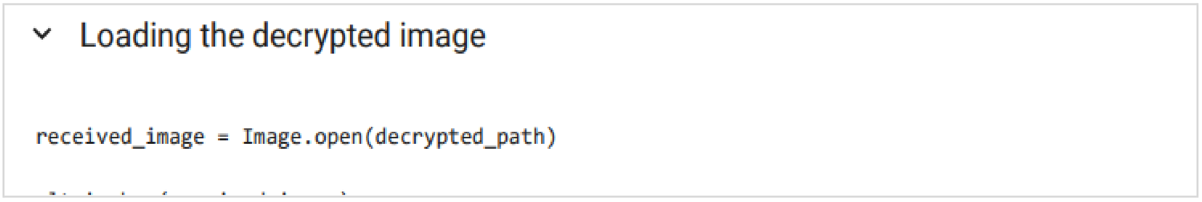


Decrypt the received image

```python
decrypted_image = decrypt_image(encrypted_path, Bob_key)   # Calling the decryption function

# If error rises due to incorrect key, quantum channel has been detected.
# In such cases, the channel is usually changed and the process is repeated.
```

Storing the decrypted image

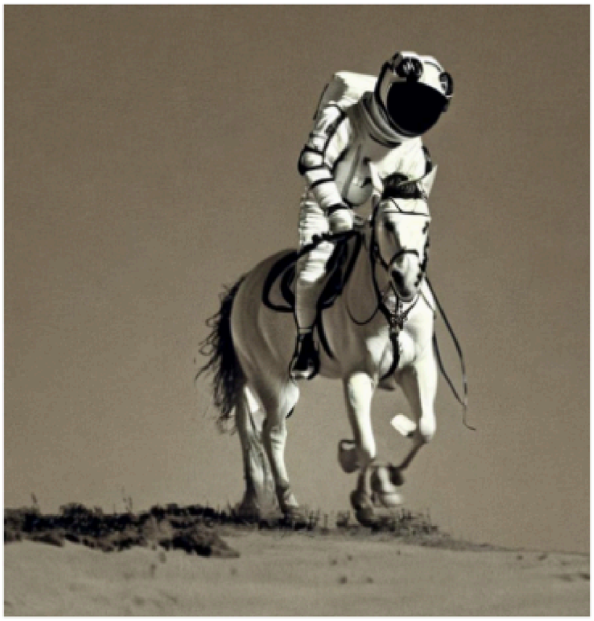
```python
folder_name = "decrypt_folder"   # Creating the folder
folder_path = f"/content/{folder_name}"
if not os.path.exists(folder_path):      # Creating the folder if it doesn't exist already
    os.makedirs(folder_path)

decrypted_path = f"/content/{folder_name}/decrypted_image.jpg"

with open(decrypted_path, "wb") as decrypted_file:   # Storing the decrypted image in the folder
        decrypted_file.write(decrypted_image)
```



Loading the decrypted image

```python
received_image = Image.open(decrypted_path)
```

```
plt.imshow(received_image)
plt.axis("off")
plt.show()
```

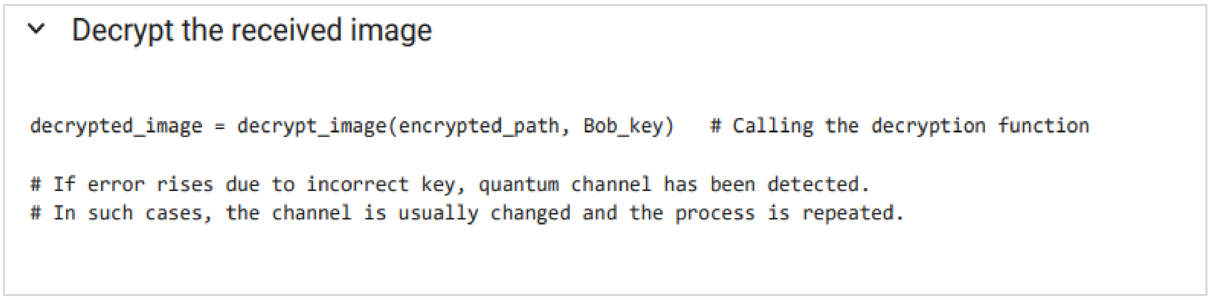


Encryption-Decryption Successful!

Simulation of eavesdropped protocol:

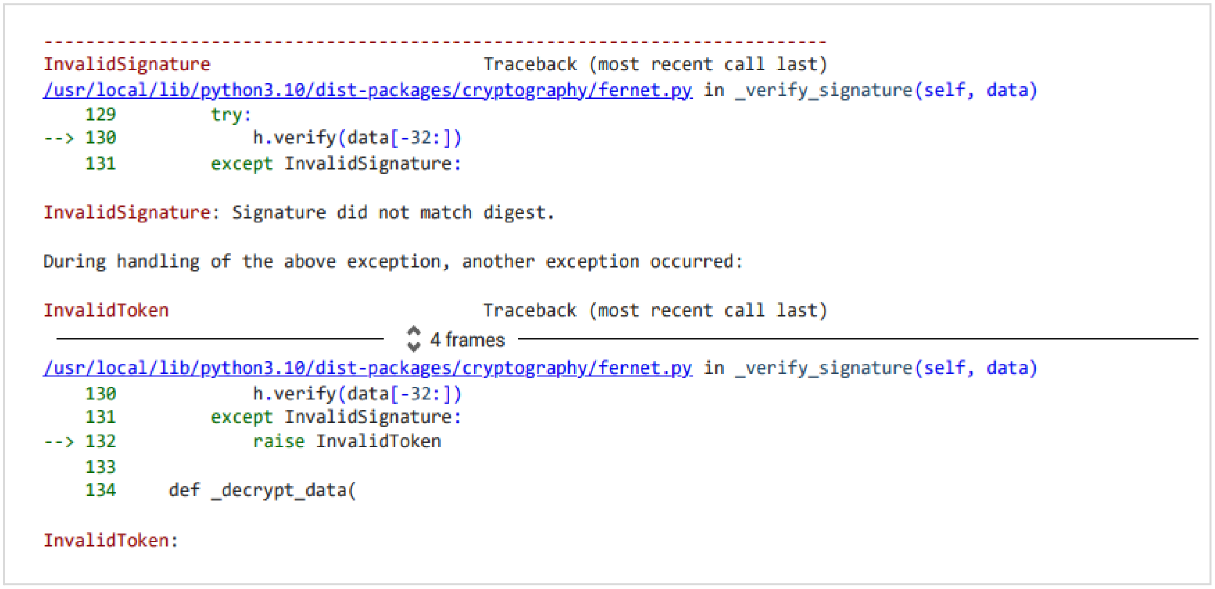
```
# Experiment parameters
final_key_size = 128   # Size of the final Shared key after completing all iterations
circuit_size = 8  # No. of qubits
eavesdropping = 1   # 0: No Eve, 1: Eve is here
```

```
print("Expected Key: " + Bob_key)
print("\nReal Key: " + Alice_key)

if Bob_key == Alice_key:
  print("\nNo Interception Detected!")
else:
  print("\nInterception Detected!")

    Expected Key: 0010110011111000000100010011111110101000101000100100110110100101000101110

    Real Key: 11111110011001000011100110100001101010100001011111011010010010110010111101111

    Interception Detected!
```



Decrypt the received image

```
decrypted_image = decrypt_image(encrypted_path, Bob_key)   # Calling the decryption function

# If error rises due to incorrect key, quantum channel has been detected.
# In such cases, the channel is usually changed and the process is repeated.
```



```
---------------------------------------------------------------------------
InvalidSignature                          Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/cryptography/fernet.py in _verify_signature(self, data)
    129         try:
--> 130             h.verify(data[-32:])
    131         except InvalidSignature:

InvalidSignature: Signature did not match digest.

During handling of the above exception, another exception occurred:

InvalidToken                              Traceback (most recent call last)
                                 4 frames
/usr/local/lib/python3.10/dist-packages/cryptography/fernet.py in _verify_signature(self, data)
    130             h.verify(data[-32:])
    131         except InvalidSignature:
--> 132             raise InvalidToken
    133 
    134     def _decrypt_data(

InvalidToken:
```

Encryption-Decryption Unsuccessful!

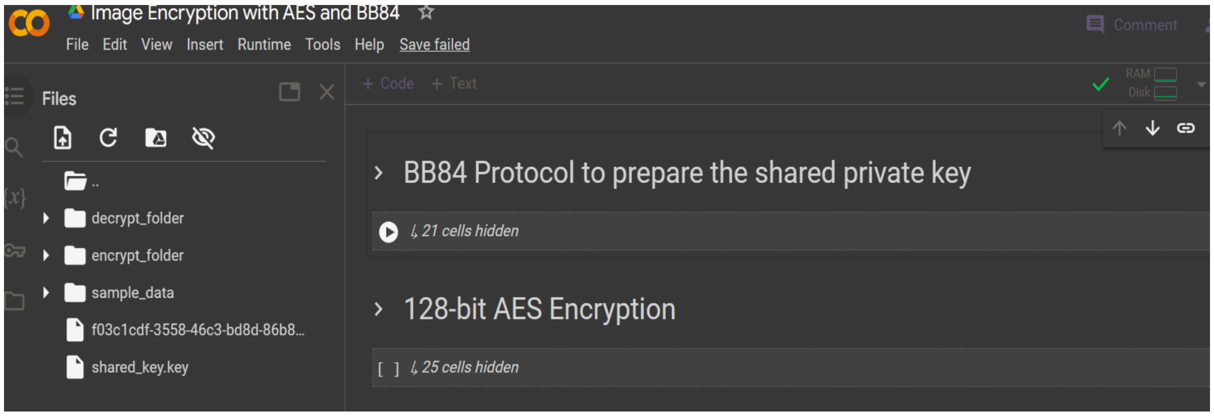


Image Encryption with AES and BB84
File Edit View Insert Runtime Tools Help Save failed
Comment
Files
+ Code + Text
RAM
Disk
..
decrypt_folder
encrypt_folder
sample_data
f03c1cdf-3558-46c3-bd8d-86b8...
shared_key.key

BB84 Protocol to prepare the shared private key

↳ 21 cells hidden

128-bit AES Encryption

[ ] ↳ 25 cells hidden

Screenshot of jupyter notebook: Image encryption with AES and BB84