**Quantum Encryption**

**Code\_2 :**

# Imports and Setup

from qiskit import QuantumCircuit, transpile, Aer

from qiskit\_ibm\_provider import IBMProvider

import numpy as np

provider = IBMProvider('bb07b6bb8a130a37da9847350827e0e7337dab2cd5b128cd65bdbcfc2731be61a8af827c8096262c023d7b63813f0ac1f6a6a4b571193aed7134bea113702e98')

# Encode

def encode\_message(bits, bases):

   message = []

   for i in range(len(bits)):

       qc = QuantumCircuit(1, 1)

       if bases[i] == 0:  # Prepare qubit in Z-basis

           if bits[i] == 1:

               qc.x(0)

       else:  # Prepare qubit in X-basis

           if bits[i] == 0:

               qc.h(0)

           else:

               qc.x(0)

               qc.h(0)

       qc.measure(0, 0)

       message.append(qc)

   return message

# Measure

def measure\_message(message, bases):

   measurements = []

   for q, basis in zip(message, bases):

       if basis == 1:  # measuring in X-basis

           q.h(0)

       measurements.append(q)

   return measurements

# Key Sifting

def remove\_garbage(a\_bases, b\_bases, bits):

   return [bit for i, bit in enumerate(bits) if a\_bases[i] == b\_bases[i]]

# Implement BB84 protocol

np.random.seed(seed=0)

   # Alice generates random bits and bases

alice\_bits = np.random.randint(2, size=4)

alice\_bases = np.random.randint(2, size=4)

message = encode\_message(alice\_bits, alice\_bases)

   # Bob generates random bases for measurement

bob\_bases = np.random.randint(2, size=4)

bob\_results = measure\_message(message, bob\_bases)

# Execute on backend

Backend='ibm\_kyiv'

backend = provider.get\_backend(Backend)

transpiled\_circuits = transpile(bob\_results, backend)

job = backend.run(transpiled\_circuits, shots=1)

result = job.result()

counts = result.get\_counts()

# Process results

bob\_measured\_bits = [int(list(count.keys())[0]) for count in counts]

# Step 3

alice\_key = remove\_garbage(alice\_bases, bob\_bases, alice\_bits)

bob\_key = remove\_garbage(alice\_bases, bob\_bases, bob\_measured\_bits)

print("Alice's bits:", alice\_bits)

print("Alice's bases:", alice\_bases)

print("Bob's bases:", bob\_bases)

print("Bob's results:", bob\_measured\_bits)

print("Alice's key:", alice\_key)

print("Bob's key:", bob\_key)

# Key Verification

if alice\_key == bob\_key:

   print("Success: Alice and Bob's keys match!")

else:

   print("Warning: Alice and Bob's keys do not match.")

   print("Mismatched positions:")

   for i, (a, b) in enumerate(zip(alice\_key, bob\_key)):

       if a != b:

           print(f"Position {i}: Alice has {a}, Bob has {b}")

   print("\nDetailed analysis:")

   print("Matching bases positions:", [i for i, (a, b) in enumerate(zip(alice\_bases, bob\_bases)) if a == b])

   print("Alice's bits at matching bases:", [alice\_bits[i] for i, (a, b) in enumerate(zip(alice\_bases, bob\_bases)) if a == b])

   print("Bob's measured bits at matching bases:", [bob\_measured\_bits[i] for i, (a, b) in enumerate(zip(alice\_bases, bob\_bases)) if a == b])

# Q Circuit

qc = QuantumCircuit(4, 4)

for i in range(4):

   if alice\_bases[i] == 0:

       if alice\_bits[i] == 1:

           qc.x(i)

   else:

       if alice\_bits[i] == 0:

           qc.h(i)

       else:

           qc.x(i)

           qc.h(i)

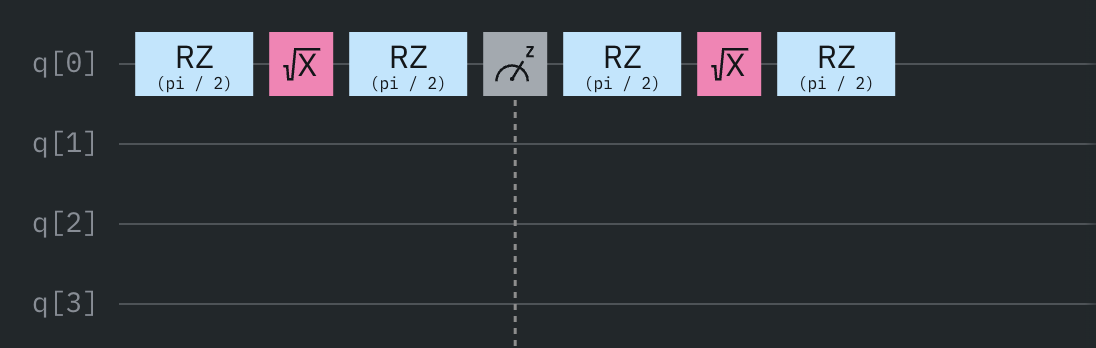
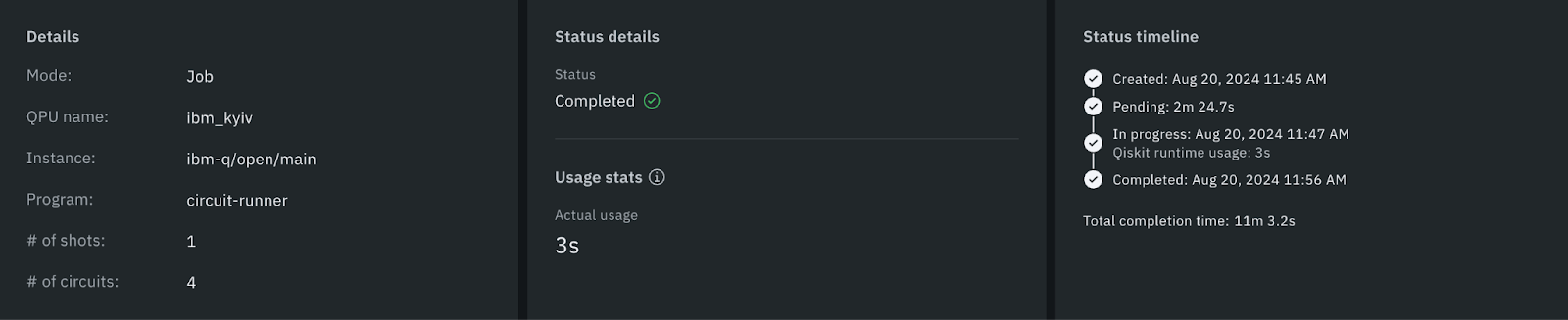
   if bob\_bases[i] == 1:

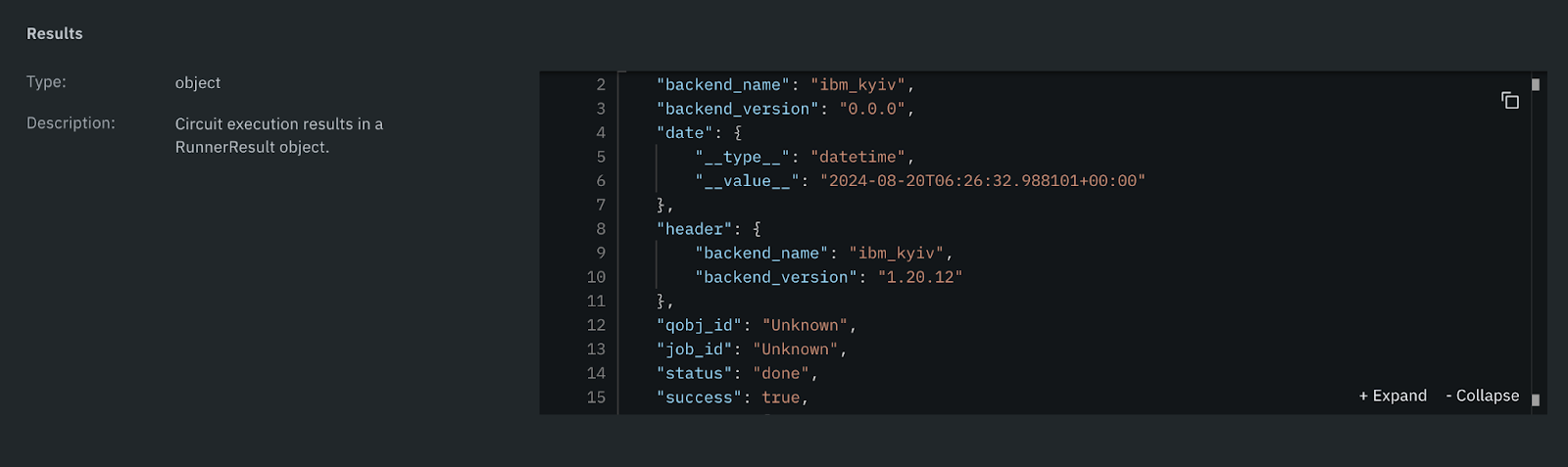
       qc.h(i)

   qc.measure(i, i)

print(qc)

**IBM Dashboard :**





**Ouput\_2 :**

Alice's bits: [0 1 1 0]

Alice's bases: [1 1 1 1]

Bob's bases: [1 1 1 0]

Bob's results: [0, 1, 1, 1]

Alice's key: [0, 1, 1]

Bob's key: [0, 1, 1]

**Success: Alice and Bob's keys match!**

**Circuit :**

     ┌───┐┌───┐┌─┐

q\_0: ┤ H ├┤ H ├┤M├───────────

     ├───┤├───┤└╥┘┌───┐┌─┐

q\_1: ┤ X ├┤ H ├─╫─┤ H ├┤M├───

     ├───┤├───┤ ║ ├───┤└╥┘┌─┐

q\_2: ┤ X ├┤ H ├─╫─┤ H ├─╫─┤M├

     ├───┤└┬─┬┘ ║ └───┘ ║ └╥┘

q\_3: ┤ H ├─┤M├──╫───────╫──╫─

     └───┘ └╥┘  ║       ║  ║

c: 4/═══════╩═══╩═══════╩══╩═

            3   0       1  2

**Conclusion\_2 :**

Success: Alice and Bob's keys match!

Ideally, Alice and Bob's final keys should be identical. However, in practice, there are a few factors to consider:

1. **Theoretical expectation:**

In a perfect, noiseless quantum channel with no errors in preparation or measurement, Alice and Bob's keys should indeed be identical after the basis reconciliation step (where they keep only the bits measured in matching bases).

1. **Real-world considerations:**
   * Quantum noise: Real quantum systems are subject to various forms of noise and decoherence, which can introduce errors.
   * Measurement errors: There's always some probability of error in quantum measurements.
   * Simulator limitations: While quantum simulators aim to mimic real quantum systems, they might not perfectly capture all aspects of quantum noise and errors.

1. **Our implementation:**

In our code, we're using a quantum simulator with only one shot per circuit. This means we're only running each measurement once, which can lead to statistical fluctuations.

**Error Correction :**

Running this code multiple times, we observe:

1. Most of the time, Alice and Bob's keys should match.
2. Occasionally, due to the probabilistic nature of quantum measurements and the limited number of shots (we're using only 1 shot per circuit), you might see discrepancies.

If you consistently see mismatches, it could indicate:

1. An issue with the quantum simulator or its configuration.
2. A bug in the implementation of the protocol.
3. The need for error correction techniques, which are often employed in practical quantum key distribution systems.

To improve the reliability of the key generation, you could:

1. Increase the number of shots for each circuit measurement.
2. Implement error correction protocols.
3. Use a longer initial bit string, so you have more bits to work with after discarding the mismatched bases.

In a real-world quantum key distribution system, Alice and Bob would perform additional steps like error correction and privacy amplification to ensure their final keys are identical and secure.

**Explanation :**

1. Bits:

Bits are the fundamental units of information in classical and quantum computing. In the BB84 protocol:

- Alice generates a random string of classical bits (0s and 1s).

- These bits will eventually form the basis of the shared secret key.

- In our implementation, `alice\_bits` represents this random string of bits.

Example: alice\_bits = [1, 0, 1, 1]

2. Bases:

In BB84, bases refer to the different ways a qubit can be prepared or measured. The protocol uses two bases:

- The rectilinear or Z-basis (usually denoted as +): In this basis, 0 is represented by |0⟩ and 1 by |1⟩.

- The diagonal or X-basis (usually denoted as ×): In this basis, 0 is represented by |+⟩ and 1 by |-⟩.

In our code:

- We represent the Z-basis with 0 and the X-basis with 1.

- Both Alice and Bob randomly choose bases for each bit.

- `alice\_bases` and `bob\_bases` represent these random choices.

Example:

alice\_bases = [0, 1, 1, 0]  (Z, X, X, Z)

bob\_bases   = [1, 1, 0, 0]  (X, X, Z, Z)

3. Keys:

The key is the final secret string of bits that Alice and Bob share after the protocol is complete. The process to get the key involves:

a. Qubit preparation: Alice prepares qubits based on her bits and chosen bases.

b. Transmission: Alice sends these qubits to Bob.

c. Measurement: Bob measures each qubit in his randomly chosen basis.

d. Basis reconciliation: Alice and Bob publicly compare their bases (but not the bit values).

e. Key sifting: They keep only the bits where their bases matched and discard the rest.

The resulting string of bits is their shared key.

In our code:

- `alice\_key` and `bob\_key` represent the final keys.

- These are generated using the `remove\_garbage` function, which keeps only the bits where bases are matched.

Example:

If alice\_bits = [1, 0, 1, 1] and bob\_measured\_bits = [0, 0, 0, 1],

and their matching bases were at positions 1 and 3,

then alice\_key = bob\_key = [0, 1]

Here's how it all fits together in the BB84 protocol:

1. Alice generates random bits and bases.

2. She prepares qubits according to her bits and bases.

3. Bob generates random bases (independent of Alice's).

4. Bob measures the qubits he receives from Alice in his chosen bases.

5. Alice and Bob compare bases and keep only the bits where their bases matched.

6. These remaining bits form their shared secret key.

The security of BB84 comes from:

- The inability to measure a qubit in two different bases simultaneously.

- The fact that measuring in the wrong basis gives a random result.

- Any eavesdropping attempt would introduce detectable errors.

We simulate this entire process using quantum circuits, which is why we need to use real quantum hardware.