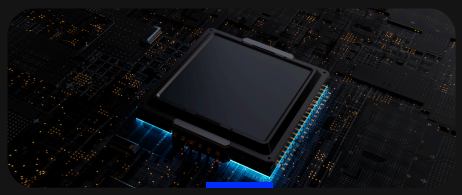
Quantum Cryptography with Qiskit

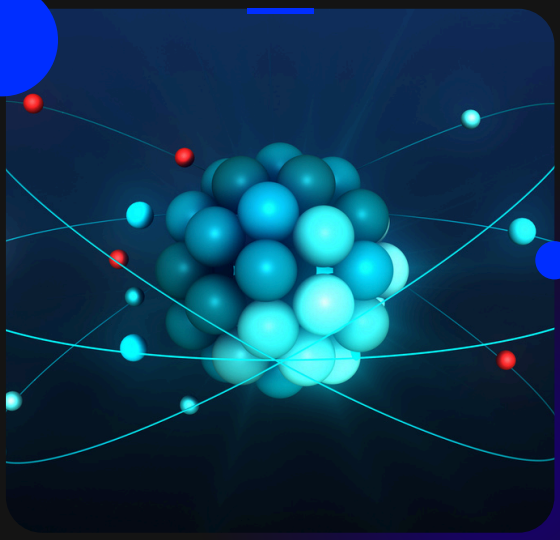
Name : Harshita Deewan
Presented Under The Guidance Of : Dr. Trivedi Harsh Chandrakant
College : LNMIIT, Jaipur



Welcome everyone to this talk on Quantum Cryptography with Qiskit!
My name is Harshita Deewan, and today we'll dive into the fascinating intersection of quantum physics and secure communication.
We'll see why traditional cryptography is at risk, how quantum cryptography steps in, and we'll even get hands-on by implementing the famous BB84 protocol using Qiskit, IBM's open-source quantum SDK.
The goal is to understand both the why and the how — so by the end, you'll know the theory and see real Python code that simulates quantum cryptography.

Why Quantum Cryptography?

- Threats from quantum computers
- Shor's algorithm breaks RSA
- Need for new paradigms



To understand quantum cryptography, let's start with the problem.
Classical cryptography relies on mathematical assumptions: factoring large numbers, computing discrete logarithms, etc.
In 1994, Peter Shor proposed an algorithm that can efficiently factor large numbers on a quantum computer.
This means that if large-scale quantum computers become practical, widely used schemes like RSA, DSA, and ECC could become insecure overnight.
Quantum cryptography addresses this by shifting from mathematical security to physical security — based on the laws of quantum mechanics.

Quantum cryptography uses core principles of quantum physics:

- No-cloning theorem: you cannot perfectly copy an unknown quantum state.

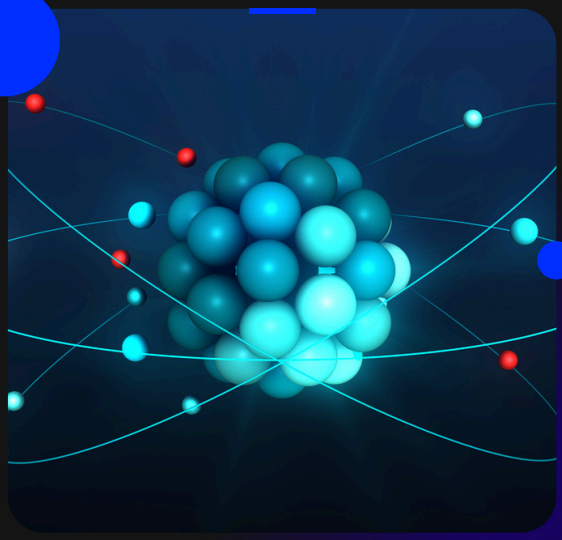
- Measurement disturbance: measuring a qubit collapses its state, often disturbing it.

These properties enable protocols like quantum key distribution (QKD), which allow two parties to establish a shared secret key securely — even in the presence of an eavesdropper.

Importantly, QKD doesn't protect data directly — it protects the key used to encrypt data. The encryption itself can still be classical (e.g., AES).

Quantum Key Distribution (QKD)

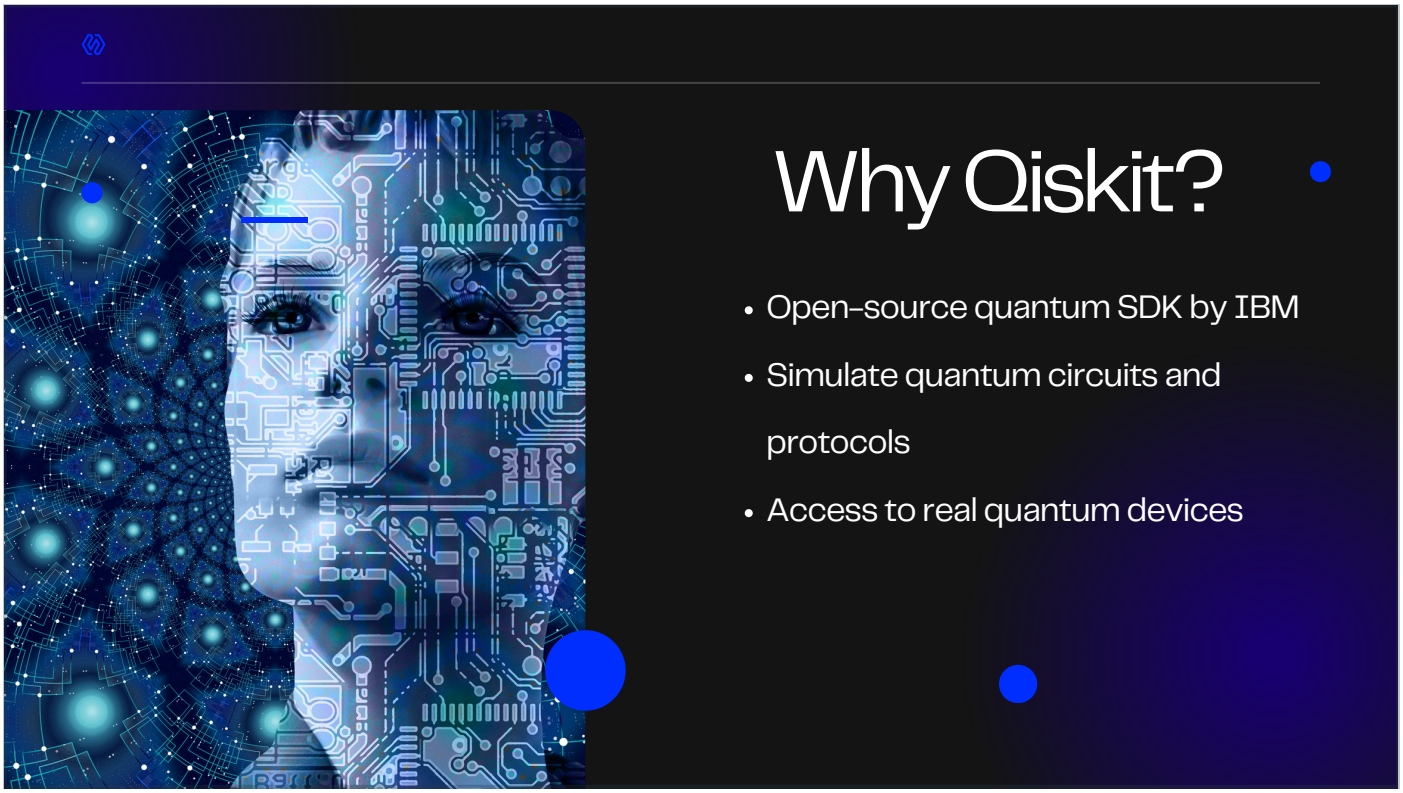
- Purpose: share a secret key securely
- Detects eavesdropping
- Most famous: BB84 protocol



QKD is the flagship application of quantum cryptography.
It solves a centuries-old problem: how can two distant parties establish a shared secret key over an insecure channel?
Classical solutions like Diffie-Hellman rely on hard math problems.
QKD protocols use quantum bits — qubits — to detect any eavesdropping.
If an attacker intercepts the qubits, measurement will disturb them, revealing their presence.
The BB84 protocol, invented in 1984 by Bennett and Brassard, is the first and still the most studied QKD protocol.

In BB84, Alice sends qubits encoded in random bases: either rectilinear (+) or diagonal (×). Bob, unaware of the bases, also randomly picks which basis to measure each qubit in. After transmission, Alice and Bob communicate over a classical channel and reveal which bases they used (but not the bits).
They discard bits where their bases didn't match.
The remaining bits, where bases matched, become the raw key.
They then check a subset to estimate error rate. High error rate suggests eavesdropping.

This process cleverly exploits quantum mechanics to guarantee detection of any interception.



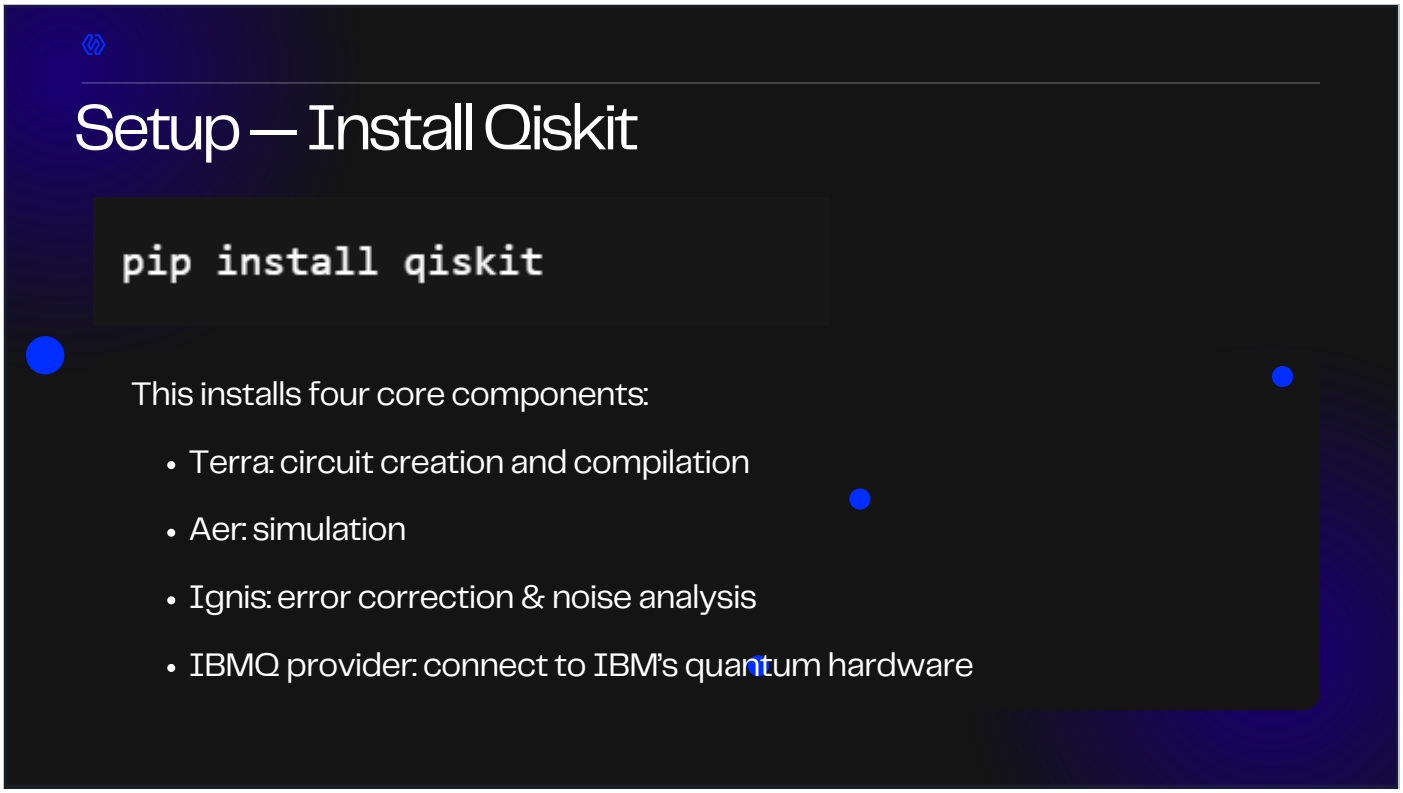
Qiskit is IBM's open-source Python framework for quantum computing.
It allows us to:

- Build and visualize quantum circuits

- Simulate quantum behavior on classical machines

- Connect to IBM Quantum's real devices in the cloud
Qiskit lowers the barrier to learning and experimenting with quantum algorithms and cryptographic protocols.

Today, we'll see BB84 in Qiskit step by step.



Let's get practical.
Installing Qiskit is as easy as running pip install qiskit.
Now, let's build BB84 together.

Step 1 – Generate Random Bits and Bases

- Alice's random bits & bases
- Bob's random bases

BB84 starts with randomness.
Alice chooses:

- A random bit string (0s and 1s)

- A random choice of basis (rectilinear or diagonal) for each bit
Bob independently picks random bases.
This randomness is essential: predictable choices would compromise security.

Let’s see this in code.

Python Code — Alice's Preparation

```python
import random
n = 10
alice_bits = [random.randint(0,1) for _ in range(n)]
alice_bases = [random.choice(['+', 'x']) for _ in range(n)]
```

Here, n=10 is the number of qubits we'll send.
alice_bits is a list of random bits, and alice_bases is a list of '+' or 'x'.
In practice, n can be thousands or millions for strong keys.
For demo purposes, we'll keep it small and readable.

Step 2 – Build Quantum Circuits

- Apply X gate for bit=1
- Apply H gate for diagonal basis

To encode bits as qubits:

- If Alice wants to send bit=1, she applies an X gate (which flips |0⟩ to |1⟩).

- If she uses diagonal basis, she applies Hadamard (H) to prepare |+⟩ or |−⟩.
This code builds a list of quantum circuits, one per qubit.

Python Code — Alice's Circuits

```
from qiskit import QuantumCircuit
circuits = []
for bit, base in zip(alice_bits, alice_bases):
    qc = QuantumCircuit(1,1)
    if bit == 1:
        qc.x(0)
    if base == 'x':
        qc.h(0)
    circuits.append(qc)
```

For each bit:

- Start with |0⟩.

- If bit=1: apply X.

- If basis is diagonal: apply H.
Now the qubit is ready for transmission to Bob.

Step 3 – Bob's Measurement

- Bob applies H if measuring in diagonal basis
- Measures qubit

Bob receives qubits without knowing Alice's bases.
To measure in diagonal basis, Bob applies Hadamard before measurement.
In rectilinear basis, he measures directly.

Python Code — Bob's Measurement

```
bob_bases = [random.choice(['+', 'x']) for _ in range(n)]
for qc, base in zip(circuits, bob_bases):
    if base == 'x':
        qc.h(0)
    qc.measure(0,0)
```

Here, Bob randomly picks bases.
For each qubit, he applies H if his basis is diagonal, then measures.
The measurement collapses the qubit into 0 or 1.

Step 4 – Simulate in Qiskit

- Use Aer simulator
- Get measurement results

We can’t directly see qubits; we observe them by measuring.
Qiskit’s Aer qasm_simulator simulates this process and gives classical outcomes.

Python Code — Run Simulation

```
from qiskit import Aer, execute
backend = Aer.get_backend('qasm_simulator')

results = []
for qc in circuits:
    job = execute(qc, backend=backend, shots=1)
    counts = job.result().get_counts()
    bit = int(list(counts.keys())[0])
    results.append(bit)
```

Each circuit is executed once (shots=1) since BB84 uses single qubits.
We collect Bob's measured bits into results.

Step 5 – Basis Reconciliation

- Compare bases
- Keep bits where bases matched

After transmission, Alice and Bob publicly reveal the bases they used.
They discard bits measured with mismatched bases.
This step keeps the truly correlated bits, forming the raw key.

Python Code — Extract Final Key

```
key = []
for a_base, b_base, a_bit, b_bit in zip(alice_bases, bob_bases, alice_bits, results):
    if a_base == b_base:
        key.append(a_bit)
```

Notice: we add a_bit to the key because Bob's bit should match it.
Any mismatch might indicate noise or eavesdropping.

Sample Output

Index	Alice Bit	Alice Basis	Bob Basis	Bob Measured Bit	Kept in Key?
0	1	+	+	1	✓
1	0	x	+	1	✗
2	1	+	+	1	✓
3	0	+	x	0	✗
4	1	x	x	1	✓
5	0	x	+	0	✗
6	1	+	+	1	✓
7	0	x	x	0	✓
8	1	x	x	1	✓
9	0	+	x	1	✗

Final Shared Key (Alice's bits where bases matched):

```
[1, 1, 1, 0, 1]
```

Here's a snapshot: we see Alice's random bits, Bob's bits, bases, and the final shared key. In a real system, we'd also do error correction and privacy amplification.

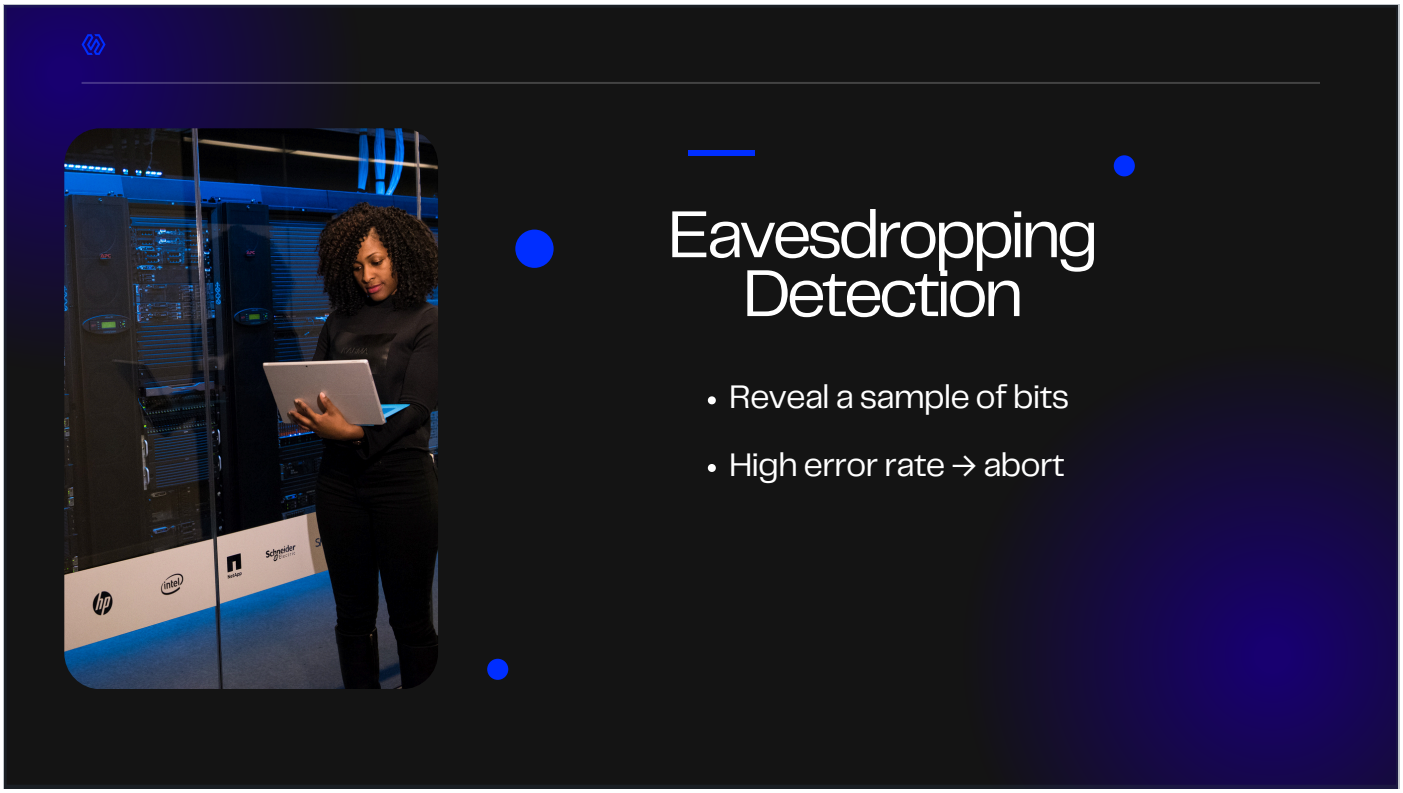
And here's a concrete example of how BB84 works after running the simulation:

- Alice and Bob each chose random bases.

- They reveal and compare the bases publicly.

- They discard the bits where bases didn't match (marked ✗).

- Only keep the bits where bases matched (✓).
The remaining bits become their shared secret key.
In real systems, this would be followed by error correction and privacy amplification to get the final secure key.

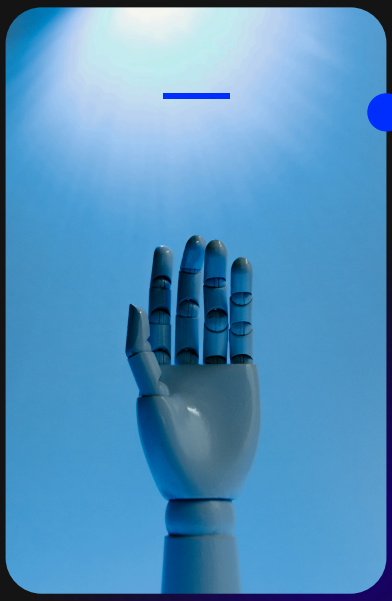


Alice and Bob publicly compare a random subset of shared bits.
If many mismatches appear, it’s evidence of an eavesdropper (Eve).
If error rate is acceptable, they keep the rest as secret key.

This test exploits the fundamental disturbance introduced by measurement.

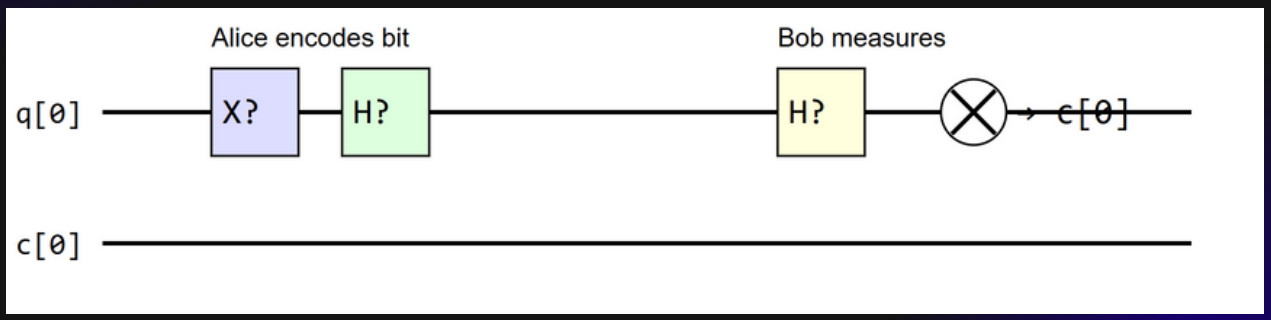
Visualization in Qiskit

```
circuits[0].draw('mpl')
```

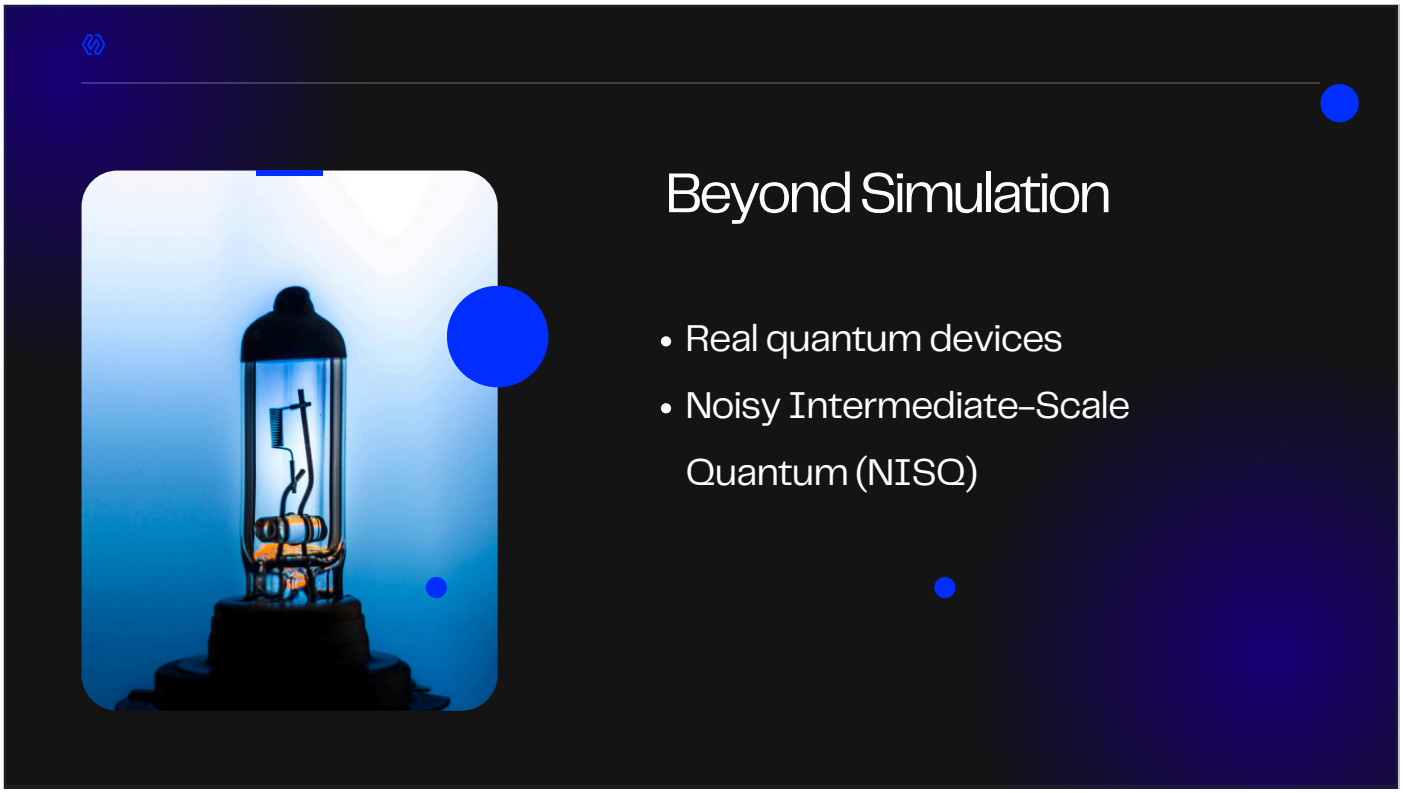


Qiskit can draw beautiful circuit diagrams.
They help visualize each step: X gates, Hadamards, measurements.
Great for debugging and teaching.

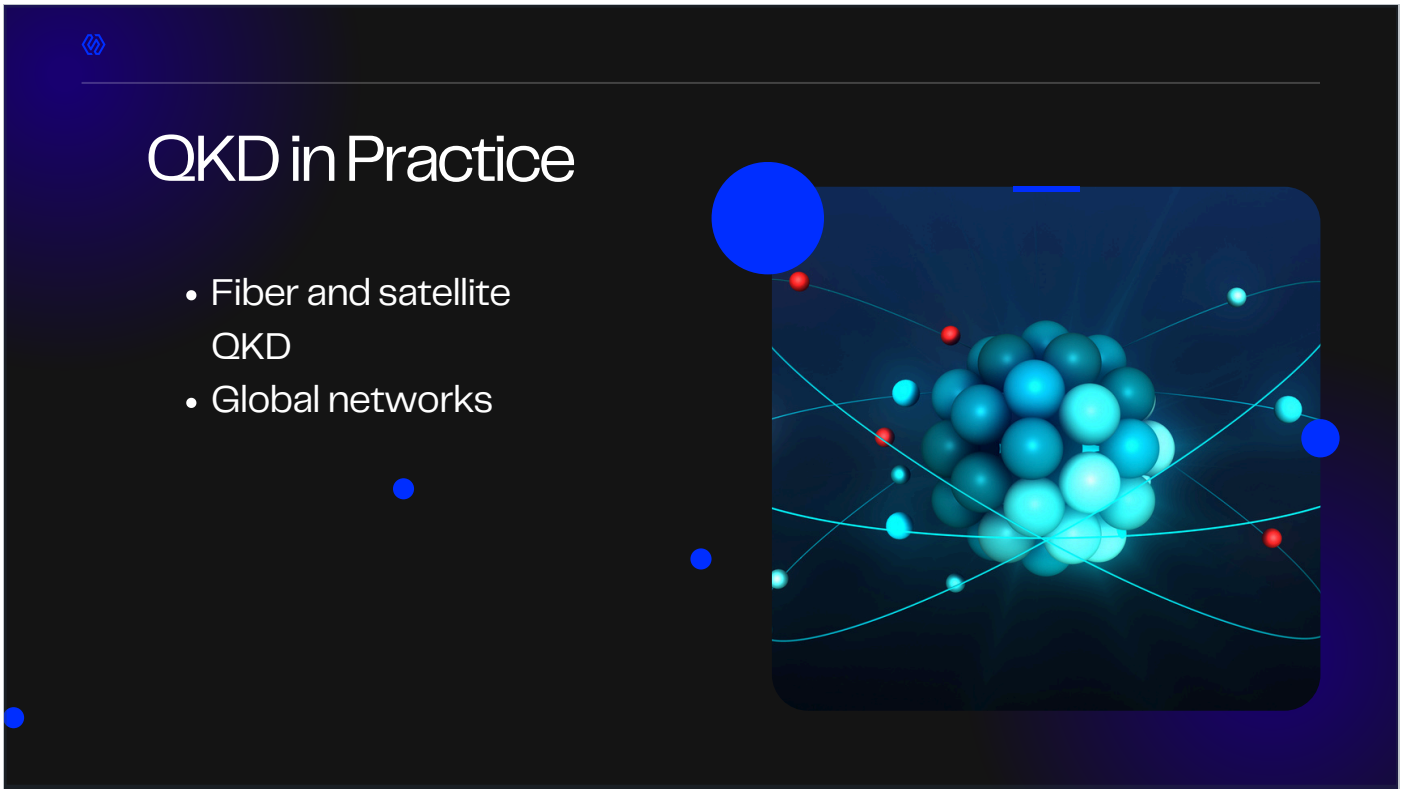
Circuit Image



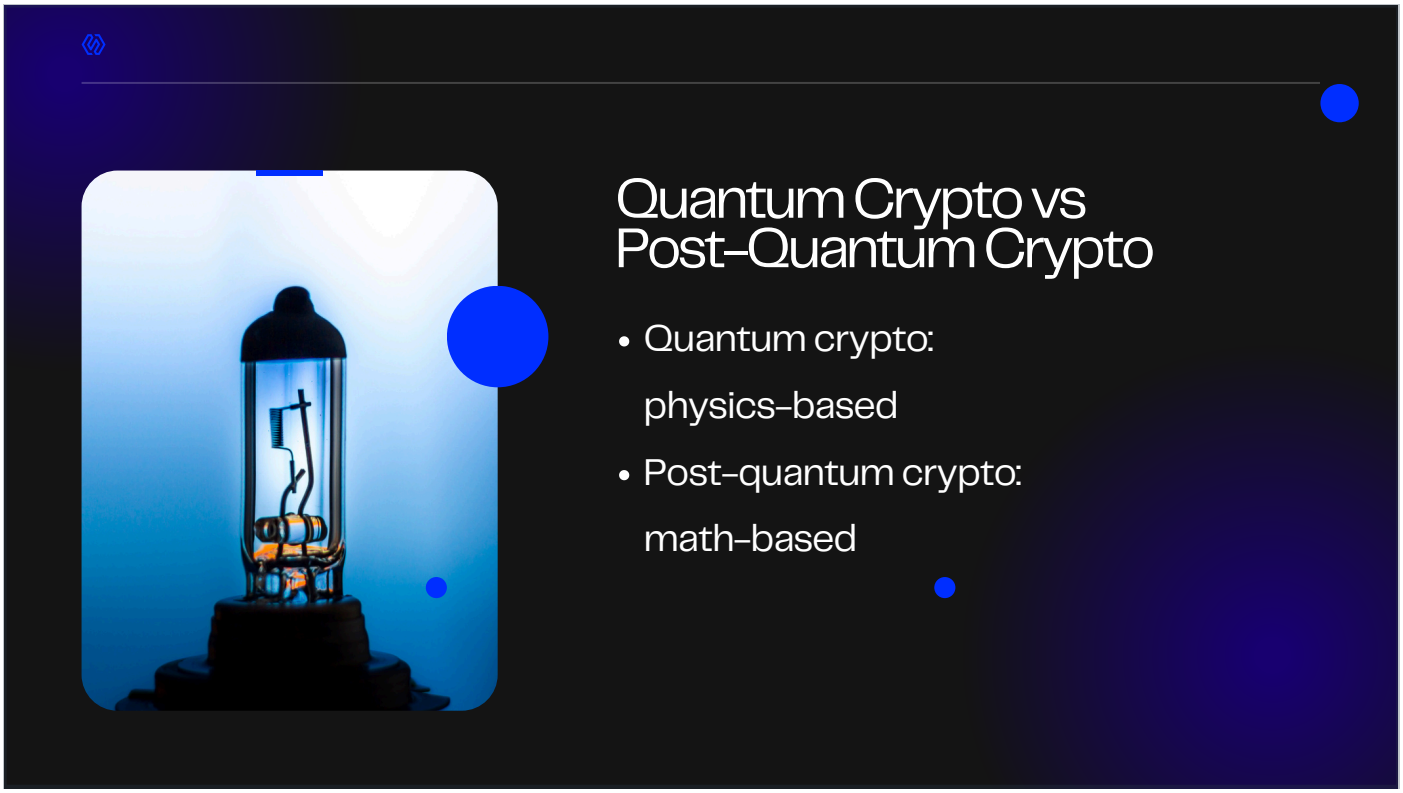
Here's Alice's first qubit preparation as a circuit diagram.
Visualization helps us "see" the invisible quantum process.



Qiskit isn’t limited to simulation.
You can send circuits to real IBMQ devices.
Real devices introduce noise and imperfections, making BB84 even more interesting to test.



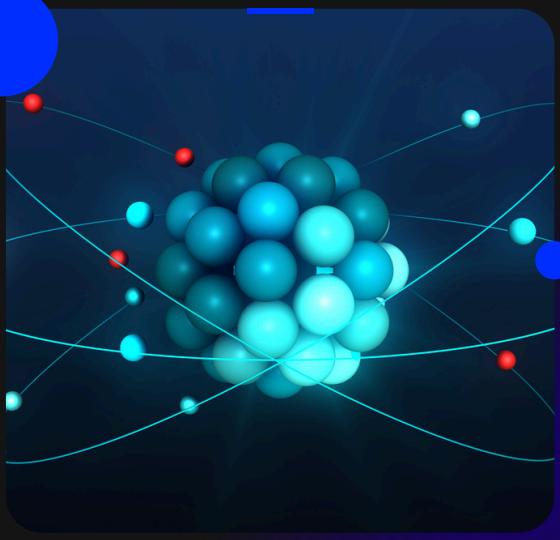
Quantum cryptography isn’t theoretical anymore.
China has a QKD backbone connecting cities.
The Micius satellite performed space-ground QKD over thousands of kilometers.
Europe and others have pilot networks too.



Quantum cryptography secures keys using quantum physics.
Post-quantum cryptography uses new algorithms resistant to quantum computers (e.g., lattice-based).
Both are complementary; each has strengths and trade-offs.

Challenges and Ethics

- Cost, scalability
- Who controls infrastructure?



Quantum crypto is expensive: requires specialized hardware.
Raises ethical questions: global access, digital sovereignty.
We must balance innovation with inclusivity and fairness.

To sum up: BB84 is simple but powerful, showing how physics can protect secrets.
Qiskit brings it to life, letting anyone simulate and learn.
Quantum cryptography is active research and has real deployments.

Thank You

The only limit is the imagination.