BB84 Protocol Using Python

September 27, 2022

0.1 BB84 Circuit in Python

0.1.1 Creating the Circuit

First the required libraries are imported. Numpy is used for randomization while qiskit is used for setting up the quantum circuits.

[2]:
```
import numpy as np

#standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ, execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator
```

The inital states and basis are chosen at random using the numpy random integer function. Then, a circuit is created which uses the number of specified quibits. For each of the qubits in the circuit, if the first parameter, the state, is 1 then a CNOT(X) gate is applied to invert the value. For each of the numbers of integers in the basis, if the value is 1 a hadamard gate is applied. Finally, Bob measures the circuit with his measurement basis.

[9]:
```
# creates states and bases
qubits = 256
alice_state = np.random.randint(2, size=qubits)
alice_basis = np.random.randint(2, size=qubits)
bob_basis = np.random.randint(2, size=qubits)

# creates bb84 function
def bb84(state, basis, measurement_basis):
    # Alice creates qubits
    quibits = len(state)
    circuit = QuantumCircuit(qubits)

    for n in range(len(basis)):
        if state[n] == 1:
            circuit.x(n)
        if basis[n] == 1:
            circuit.h(n)
```

```
    # Bob measures the qubits
    for n in range (len(measurement_basis)):
        if measurement_basis[n] == 1:
            circuit.h(n)

    circuit.measure_all()

    return circuit
```

0.1.2 Running the Circuit on a Quantum Computer

The circuit runs on the parameters previously defined, Alice's and Bob's states and bases. The backend, or computer, is taken from IBM quantum labs as simulator_stabilizer.

[10]:
```
# Bob compares bases with Alice keeping matching qubits
circuit = bb84(alice_state, alice_basis, bob_basis)
backend = provider.get_backend('simulator_stabilizer')
key = execute(circuit.reverse_bits(),backend=backend,shots=1).result().
↪get_counts().most_frequent()
```

0.1.3 Encryption Key Creation

Now, we can create the encryption key by measuring Alice's basis and Bob's basis and if they are the same, adding that bit of information to the encryption key.

[11]:
```
encryption_key = ''
for n in range(qubits):
    if alice_basis[n] == bob_basis[n]:
        encryption_key += str(key[n])
print("key:", encryption_key)
```

```
key: 111001000110100111111111101001011000000101111011010100111001010011111000101
1101010100101111011110101001011101101010110101 0101
```

0.1.4 Encoding and Decoding with Fernet

From the crytography library, Fernet, a simple symmetric encryptor and decoder, is imported. Furthermore, a base64 converter is also added. This allows the key to be used by Fernet, which only accepts 32 byte keys in base 64. The first 32 bytes of the encryption key is converted into base 64 and a simple message, "hello world.", is created. The message is also decoded for easier readability.

[12]:
```
from cryptography.fernet import Fernet
import base64
# keeps only first 32 bytes of key
encryption_key = encryption_key[:32]

# converts key into base64 for Fernet
```

```python
base64_key = base64.b64encode(bytes(encryption_key, 'utf-8'))
key = Fernet(base64_key)

# here's the encrypted message
bmessage = key.encrypt(b'hello world.')
message = bmessage.decode()

print('encrypted message:',message)


# here's the decrypted message
decrypted_message = key.decrypt(bmessage)
print('\ndecrypted message:',decrypted_message.decode())
```

```
encrypted message: gAAAAABjM5J4WdxCmNV_lKeMX9SwuXJSkOQKSD3WurT4MypBWGUr5gN61BdNi
7zF6B3_Pme417IrJYT9t3rUfg-V4C8eMM1IoA==

decrypted message: hello world.
```