BB84

March 8, 2024

1 BB84 Quantum Key Distribution Simulation Report

Amir Mohammad Moghaddam 9823144

Introduction

Quantum Key Distribution (QKD) is a revolutionary technique for secure communication by exploiting the principles of quantum mechanics. The BB84 protocol is a fundamental QKD method, ensuring secure key exchange between two parties. This report dissects the implementation of BB84 QKD in a simulation, detailing each block of the code along with its purpose and functionality.

[ ]:
```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit.visualization import *
from ibm_quantum_widgets import *
from numpy import pi
from qiskit_aer import Aer
from qiskit import transpile
import random
```

[ ]:
```
#defining all the ShorCodes function
#This function uses one Logical qubit to encode 8 other qubits so that we can␣
↪use it for bit and phase flip encoding
def Encoding(cir):
    # Phase- flip
    cir.cx(8,5)
    cir.cx(8,2)
    cir.h(2)
    cir.h(5)
    cir.h(8)
    #Bit-flip
    cir.cx(8,6)
    cir.cx(8,7)
    cir.cx(5,3)
    cir.cx(5,4)
    cir.cx(2,0)
    cir.cx(2,1)
    return cir
#This function Decodes the qubits so it can be used again
def Decoding(cir):
```

```
    #Bit-flip
    cir.cx(8,6)
    cir.cx(8,7)
    cir.cx(5,3)
    cir.cx(5,4)
    cir.cx(2,0)
    cir.cx(2,1)
    #Phase-flip
    cir.h(2)
    cir.h(5)
    cir.h(8)
    cir.cx(8,5)
    cir.cx(8,2)
    return cir
#Its time for bit flip for that we use a parity check function
def BitParityCheck(cir,SQubit,AuxQubit):
    cir.cx(SQubit,AuxQubit)
    cir.cx(SQubit+1,AuxQubit)
    cir.cx(SQubit+1,AuxQubit+1)
    cir.cx(SQubit+2,AuxQubit+1)
    return cir
#And We need a CorrectionBlock
def BitCorrectionBlock(cir,SQubit,AuxQubit):
    cir.x(AuxQubit)
    cir.ccx(AuxQubit+1,AuxQubit,SQubit+2)
    cir.x(AuxQubit)
    cir.ccx(AuxQubit+1,AuxQubit,SQubit+1)
    cir.x(AuxQubit+1)
    cir.ccx(AuxQubit+1,AuxQubit,SQubit)
    cir.x(AuxQubit+1)
    cir.measure(AuxQubit,AuxQubit)
    cir.measure(AuxQubit+1,AuxQubit+1)
    cir.reset(AuxQubit)
    cir.reset(AuxQubit+1)
    return cir
#Now bringing it all toghether we construct the Error Correction function
def BitErrorCorrection(cir,AuxQubit):
    for i in range(3):
        cir = BitParityCheck(cir,i*3,AuxQubit)
        cir.barrier()
        cir = BitCorrectionBlock(cir,i*3,AuxQubit)
    return cir
def PhaseParityCheck(cir,AuxQubit):
    #Storing the data in Q8,5,2
    cir.cx(8,6)
    cir.cx(8,7)
    cir.cx(5,3)
```

```
    cir.cx(5,4)
    cir.cx(2,0)
    cir.cx(2,1)
    cir.h(2)
    cir.h(5)
    cir.h(8)
    #Getting the parity stored in the Aux Qubits
    cir.cx(2,AuxQubit)
    cir.cx(5,AuxQubit)
    cir.cx(5,AuxQubit+1)
    cir.cx(8,AuxQubit+1)
    #restoring the Encoded Form
    cir.h(2)
    cir.h(5)
    cir.h(8)
    cir.cx(8,6)
    cir.cx(8,7)
    cir.cx(5,3)
    cir.cx(5,4)
    cir.cx(2,0)
    cir.cx(2,1)
    return cir
def PhaseCorrectionBlock(cir,AuxQubit):
    cir.x(AuxQubit)
    cir.ccz(AuxQubit+1,AuxQubit,8)
    cir.x(AuxQubit)
    cir.ccz(AuxQubit+1,AuxQubit,5)
    cir.x(AuxQubit+1)
    cir.ccz(AuxQubit+1,AuxQubit,2)
    cir.x(AuxQubit+1)
    cir.measure(AuxQubit,AuxQubit)
    cir.measure(AuxQubit+1,AuxQubit+1)
    cir.reset(AuxQubit)
    cir.reset(AuxQubit+1)
    return cir
def PhaseErrorCorrection(cir,AuxQubit):
    cir = PhaseParityCheck(cir,AuxQubit)
    cir.barrier()
    cir = PhaseCorrectionBlock(cir,AuxQubit)
    return cir
def BitNoise(cir,TarQubit,theta):
    cir.rx(theta,TarQubit)
    return cir
def PhaseNoise(cir,TarQubit,theta):
    cir.rz(theta,TarQubit)
    return cir
```

[ ]:
```python
#Send Single Photon and Recieve and Measure
def SendSinglePhoton(cir):
    cir = Encoding(cir)
    cir = BitNoise(cir,8,0.9273)
    cir = BitErrorCorrection(cir,9)
    return cir
def MeasureSinglePhoton(cir):
    # Use Aer's qasm_simulator
    simulator = Aer.get_backend('qasm_simulator')
    # Execute the circuit on the qasm simulator
    new_circuit = transpile(cir, simulator)
    job = simulator.run(new_circuit)
    # Grab results from the job
    result = job.result()
    # Return counts
    counts = result.get_counts(cir)
    return counts
#Encoding the Qubit with the given basis for it
def GettingReady(cir,bit,basis):
    if bit == 1:
        cir.x(8)
    if basis == 1:
        cir.h(8)
    return cir
def ReciviengSinglePhoton(cir,BobBasis):
    cir = Decoding(cir)
    if BobBasis == 1:
        cir.h(8)

    cir.barrier()
    cir.measure([0,1,2,3,4,5,6,7,8,9,10],[0,1,2,3,4,5,6,7,8,9,10])
    return cir
def BB84(sequence,AliceBases,BobBases,length):
    ListOfCounts = []
    for i in range(length):
        circuit = QuantumCircuit(11,11)
        circuit = GettingReady(circuit,sequence[i],AliceBases[i])
        circuit = SendSinglePhoton(circuit)
        circuit = ReciviengSinglePhoton(circuit,BobBases[i])
        counts = MeasureSinglePhoton(circuit)
        ListOfCounts.append(counts)
        del circuit
    #Now Alice and Bob share their Bases
    Check_list = []
    for i in range(len(ListOfCounts)):
        if AliceBases[i] == BobBases[i]:
            Check_list.append(ListOfCounts[i])
```

```
    result_list = []
    for item in Check_list:
        for key, value in item.items():
            third_bit = key[2]  # Extract the 3rd bit of the key
            result_list.append({third_bit: value})
    Key = ''
    for item in result_list:
        for key, value in item.items():
            Key = Key + key
    return Key
def BB84withEve(sequence,AliceBases,BobBases,length):
    ListOfCounts = []
    EveBases = generate_random_sequence(length)
    for i in range(length):
        circuit = QuantumCircuit(11,11)
        circuit = GettingReady(circuit,sequence[i],AliceBases[i])
        circuit = SendSinglePhoton(circuit)
        circuit = ReciviengSinglePhoton(circuit,EveBases)
        counts = MeasureSinglePhoton(circuit)
        ListOfCounts.append(counts)
        del circuit
    #Now Alice and Bob share their Bases
    result_list = []
    for item in ListOfCounts:
        for key, value in item.items():
            third_bit = key[2]  # Extract the 3rd bit of the key
            result_list.append({third_bit: value})
    EveKey = ''
    for item in result_list:
        for key, value in item.items():
            EveKey = EveKey + key
    list_from_Key = [int(char) for char in EveKey]
    Key = BB84(list_from_Key,AliceBases,BobBases,length)
    return Key
def AliceKey(Bit,AliceBases,BobBases):
    Check_list = []
    for i in range(len(Bit)):
        if AliceBases[i] == BobBases[i]:
            Check_list.append(Bit[i])
        Key = ''
    for item in Check_list:
        Key = Key + str(item)
    return Key
```

1. Encoding and Decoding Functions

The encoding and decoding functions are crucial in preparing qubits for transmission and processing received qubits accurately.

```
GettingReady Function:
    Purpose: Prepares a qubit for transmission based on the given bit and basis.
    Functionality: Applies necessary operations (X gate for bit flip, H gate for basis encoding)

SendSinglePhoton Function:
    Purpose: Simulates the transmission of a single photon.
    Functionality: Incorporates encoding and noise introduction.

ReciviengSinglePhoton Function:
    Purpose: Simulates the reception and measurement of a single photon by Bob.
    Functionality: Decodes received qubits and performs measurements based on Bob's bases.
```

2. Error Correction

Error correction mechanisms are vital to ensure the reliability of the transmitted information.

```
BitNoise Function:
    Purpose: Introduces partial bit flip noise to the qubits.
    Functionality: Randomly applies X gates to introduce bit flip errors.

BitErrorCorrection Function:
    Purpose: Implements error correction using the Shor code.
    Functionality: Applies error correction operations based on the syndrome of the error.
```

Certainly! Below is a breakdown of the code into individual block reports, along with an overall introduction and conclusion for your presentation. Introduction

Quantum Key Distribution (QKD) is a revolutionary technique for secure communication by exploiting the principles of quantum mechanics. The BB84 protocol is a fundamental QKD method, ensuring secure key exchange between two parties. This report dissects the implementation of BB84 QKD in a simulation, detailing each block of the code along with its purpose and functionality. 1. Encoding and Decoding Functions

The encoding and decoding functions are crucial in preparing qubits for transmission and processing received qubits accurately.


```
GettingReady Function:
    Purpose: Prepares a qubit for transmission based on the given bit and basis.
    Functionality: Applies necessary operations (X gate for bit flip, H gate for basis encoding)

SendSinglePhoton Function:
    Purpose: Simulates the transmission of a single photon.
    Functionality: Incorporates encoding and noise introduction.

ReciviengSinglePhoton Function:
    Purpose: Simulates the reception and measurement of a single photon by Bob.
    Functionality: Decodes received qubits and performs measurements based on Bob's bases.
```

2. Error Correction

Error correction mechanisms are vital to ensure the reliability of the transmitted information.

```
BitNoise Function:
```

```
    Purpose: Introduces partial bit flip noise to the qubits.
    Functionality: Randomly applies X gates to introduce bit flip errors.

BitErrorCorrection and PhaseErrorCorrection Function:
    Purpose: Implements error correction using the Shor code.
    Functionality: Applies error correction operations based on the syndrome of the error.
```

3. BB84 Protocol

The BB84 protocol forms the core of the QKD simulation, enabling secure key generation between Alice and Bob.

```
BB84 Function:
    Purpose: Executes the BB84 protocol between Alice and Bob.
    Functionality: Utilizes encoding, transmission, reception, and error correction to generate

BB84withEve Function:
    Purpose: Extends BB84 to include an eavesdropper (Eve).
    Functionality: Simulates Eve's interception and introduces potential security vulnerabilitie
```

[ ]:
```
#A function for generating a random sequense
def generate_random_sequence(length):
    return [random.randint(0, 1) for _ in range(length)]
#The length of the sequence
length=64
BitSequence= generate_random_sequence(length)
AliceBases = generate_random_sequence(length)
BobBases = generate_random_sequence(length)
```

[ ]:
```
Key = BB84(BitSequence,AliceBases,BobBases,length)
```

[ ]:
```
Alicekey = AliceKey(BitSequence,AliceBases,BobBases)
```

[ ]:
```
print("your Key is :")
print(Key)
print("with The length of :" + str(len(Key)))
```

```
your Key is :
10101010001011110011110000110101
with The length of :32
```

[ ]:
```
#Alice And Bob Share Some of their Key to check the Seciruty
def CheckSecurity(Key,Akey,Checklength):
    if Akey[-Checklength:] == Key[-Checklength:]:
        print("The Connection is Secure with the probability of:")
        print((1-((3/4)**Checklength))*100)
    else:
        print("The Connection is not Secure!")
```

[ ]:
```
CheckSecurity(Key,Alicekey,15)
```

```
The Connection is Secure with the probability of:
98.6636538989842
```

4. Key Generation and Security Check

Key generation and security assessment mechanisms ensure the integrity of the generated key.

```
AliceKey Function:
    Purpose: Generates Alice's key based on matching bases with Bob.
    Functionality: Extracts key bits based on agreed bases between Alice and Bob.

CheckSecurity Function:
    Purpose: Checks the security of the generated key against Alice's key.
    Functionality: Compares a portion of Alice's and Bob's keys to assess the probability of a s
```

[ ]:
```
Key2 = BB84withEve(BitSequence,AliceBases,BobBases,length)
```

[ ]:
```
print("your Key is :")
print(Key2)
print("with The length of :" + str(len(Key2)))
```

```
your Key is :
01110011001101001010101010110011
with The length of :32
```

[ ]:
```
CheckSecurity(Key2,Alicekey,15)
```

```
The Connection is not Secure!
```