BB84_OTP-QSS-Generator-Transmission

February 2, 2020

[67]:
```
%matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
# Loading your IBM Q account(s)
provider = IBMQ.load_account()

import qiskit
qiskit.__qiskit_version__

# Import numpy for random number generation
import numpy as np

# importing Qiskit
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute,␣
↪BasicAer

# Import basic plotting tools
from qiskit.tools.visualization import plot_histogram
```

```
Credentials are already in use. The existing account in the session will be
replaced.
```

[68]:
```
get_ipython().run_line_magic('matplotlib', 'inline')

def generateBitString(length):

    qubit = QuantumRegister(1, 'qubit')
    classical = ClassicalRegister(1, 'classical')
    circuit = QuantumCircuit(qubit, classical, name = 'circuit')
    # Change the below line so that the bitString is generated using actual␣
↪quantum backend, not simulator
    simulator = Aer.get_backend('qasm_simulator')

    i = 0
```

```
    bitStringLength = length
    bitString = ""

    while i < bitStringLength:
        # Apply H gate to put the qubit in a state of superposition
        circuit.h(qubit[0])
        circuit.measure(qubit[0],classical)
        job = execute(circuit, backend=simulator, shots=1)
        result = job.result()
        counts = result.get_counts(circuit)
        #If the measured qubit evaluates to 1, add 1 to the string of bits,␣
 ↪else add a 0.
        if '1' in counts :
            bitString += "1"
        else :
            bitString += "0"
        i += 1
        circuit.reset(qubit[0])
    print("bitString generated: \n\"" + bitString + "\"")
    return bitString





n = 16

quantum = QuantumRegister(n, name = 'quantum')
classical = ClassicalRegister(n, name = 'classical')
```

[69]:
```
alice = QuantumCircuit(quantum, classical, name = 'Alice')

alice_key = generateBitString(n)
```

```
bitString generated:
"0100101011010010"
```

[70]:
```
for index, digit in enumerate(alice_key):
    if digit == '1':
        alice.x(quantum[index])

alice_table = []
for index in range(len(quantum)):
    if 0.5 < np.random.random():
        alice.h(quantum[index])
        alice_table.append('X')
    else:
```

```
        alice_table.append('Z')
```

[71]:
```
for index, digit in enumerate(alice_key):
    if digit == '1':
        alice.x(quantum[index])

alice_table = []
for index in range(len(quantum)):
    if 0.5 < np.random.random():
        alice.h(quantum[index])
        alice_table.append('X')
    else:
        alice_table.append('Z')
```

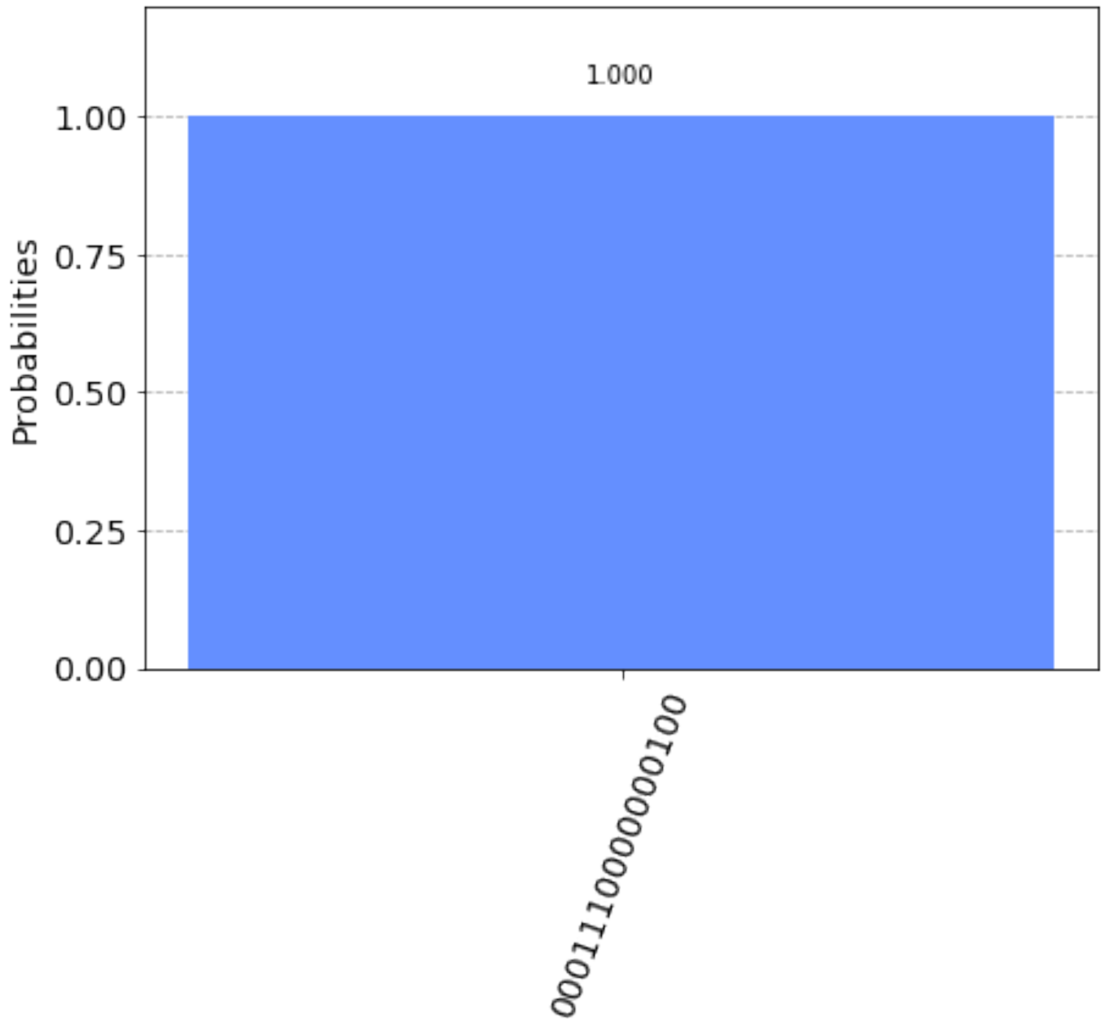
[72]:
```
def SendState(quantumCircuit1, quantumCircuit2, initalQuantumName):
    qs = quantumCircuit1.qasm().split(sep=';')[4:-1]
    for index, instruction in enumerate(qs):
        qs[index] = instruction.lstrip()

    for instruction in qs:
        if instruction[0] == 'X':
            old_qr = int(instruction[5:-1])
            quantumCircuit2.x(qr[old_qr])
        elif instruction[0] == 'H':
            old_qr = int(instruction[5:-1])
            quantumCircuit2.h(qr[old_qr])
        elif instruction[0] == 'M':
            pass
```

[73]:
```
bob = QuantumCircuit(quantum, classical, name='Bob')

SendState(alice, bob, 'Alice')

# Bob doesn't know which basis to use
bob_table = []
for index in range(len(quantum)):
    if 0.5 < np.random.random():  # With 50% chance...
        bob.h(quantum[index])        # ...change to diagonal basis
        bob_table.append('X')
    else:
        bob_table.append('Z')
```

[74]:
```
# Measure all qubits
for index in range(len(quantum)):
    bob.measure(quantum[index], classical[index])

# Execute the quantum circuit
```

```
backend = BasicAer.get_backend('qasm_simulator')
result = execute(bob, backend=backend, shots=1).result()
plot_histogram(result.get_counts(bob))
```

[74]:



[75]:

```
bob_key = list(result.get_counts(bob))[0]
bob_key = bob_key[::-1]
```

[76]:

```
keep = []
discard = []

for qubit, basis in enumerate(zip(alice_table, bob_table)):
    if(basis[0] == basis[1]):
        print("Same choice for qubit: {}, basis: {}" .format(qubit, basis[0]))
        keep.append(qubit)
```

```
    else:
        print("Different choice for qubit: {}, Alice has {}, Bob has {}" .
↪format(qubit, basis[0], basis[1]))
        discard.append(qubit)
```

```
Same choice for qubit: 0, basis: X
Same choice for qubit: 1, basis: X
Different choice for qubit: 2, Alice has Z, Bob has X
Different choice for qubit: 3, Alice has Z, Bob has X
Same choice for qubit: 4, basis: X
Different choice for qubit: 5, Alice has Z, Bob has X
Same choice for qubit: 6, basis: Z
Different choice for qubit: 7, Alice has Z, Bob has X
Same choice for qubit: 8, basis: Z
Different choice for qubit: 9, Alice has X, Bob has Z
Different choice for qubit: 10, Alice has Z, Bob has X
Same choice for qubit: 11, basis: X
Different choice for qubit: 12, Alice has Z, Bob has X
Different choice for qubit: 13, Alice has Z, Bob has X
Different choice for qubit: 14, Alice has X, Bob has Z
Same choice for qubit: 15, basis: Z
```

[77]:

```
accuracy = 0

for bit in zip(alice_key, bob_key):
    if(bit[0] == bit[1]):
        accuracy += 1

print('Percentage of qubits to be discarded according to table comparison: ',␣
↪len(keep)/n)
print('Measurement convergence by additional chance: ', accuracy/n)
```

```
Percentage of qubits to be discarded according to table comparison:  0.4375
Measurement convergence by additional chance:  0.4375
```

[78]:

```
new_alice_key = [alice_key[qubit] for qubit in keep]
new_bob_key = [bob_key[qubit] for qubit in keep]

accuracy = 0
for bit in zip(new_alice_key, new_bob_key):
    if bit[0] == bit[1]:
        accuracy += 1

percentSim = accuracy/len(new_alice_key) * 100
print('Percentage of similarity between the keys: ', percentSim)

if percentSim  >= 50:
```

```
    print('Key is secure. KEY: ', new_alice_key)

else:
    print('Key has been comprimised')
```

```
Percentage of similarity between the keys:  42.857142857142854
Key has been comprimised
```

[79]:

```
OTP_QSS_KEY = [new_alice_key[0], new_alice_key[1]]

print('The following key was generated using the most significant bits of the␣
 ↪secure key. Please use this key to interact with the OTP-QSS Channel if you␣
 ↪are a user/superuser. KEY: ', OTP_QSS_KEY)
```

```
The following key was generated using the most significant bits of the secure
key. Please use this key to interact with the OTP-QSS Channel if you are a
user/superuser. KEY:  ['0', '1']
```

[85]:

```
q = QuantumRegister(4)
c = ClassicalRegister(4)

OTP_QSS = QuantumCircuit(q, c)

if(OTP_QSS_KEY[0] == '1'):
    OTP_QSS.x(q[0])

if(OTP_QSS_KEY[1] == '1'):
    OTP_QSS.z(q[0])

#GATE TO BE APPLIED BY SERVER
OTP_QSS.t(q[1])

OTP_QSS.h(q[1])

OTP_QSS.cx(q[1], q[2])
OTP_QSS.cx(q[1], q[3])

OTP_QSS.h(q[2])

OTP_QSS.cx(q[0], q[1])

OTP_QSS.h(q[0])

OTP_QSS.measure(q[1], c[1])

if(c[1] == 1):
    OTP_QSS.x(q[3])
```

```python
if(c[2] == 1):
    OTP_QSS.z(q[3])

OTP_QSS.measure(q[2], c[2])

if(OTP_QSS_KEY[0] == '1'):
    OTP_QSS.x(q[3])

if(OTP_QSS_KEY[1] == '1'):
    OTP_QSS.z(q[3])

OTP_QSS.measure(q[3], c[3])

secretStateCharlie = c[3]
OTP_QSS.draw(output='mpl')

print('Secret: ', secretStateCharlie)
```

```
Secret:  Clbit(ClassicalRegister(4, 'c45'), 3)
```