Capstone Project: Encryption in Python – Quantum Key Distribution (QKD)

Leonardo de Oliveira Lopes

Computer Engineering, Universidade Virtual do Estado de São Paulo (UNIVESP)

Qubit by Qubit's Introduction to Quantum Computing - Spring 2024

Professors Dr. Coonie Hsueh and Dr. Derrick Boone Jr.

April 21, 2024

Introduction

When working with quantum cryptography, the Quantum Key Distribution (QKD) is often a popular application for simulating and understanding it's concepts with quantum computing without using too much resources or a quantum processing unit (QPU).

Consequently, researchers, professionals, students and enthusiasts prefer to pick a secure protocol for key generation and encryption/decryption of a message considering that it has an established process which can be followed.

The work done in this capstone project was based on the BB84 protocol (Bennett & Brassard, 2014) and inspired by the code and theory presented during the two-semester quantum computing course. The BB84, which was developed by Bennett & Brassard in 1984, has been studied over the weeks 9, 10 and 11.

The workflow of this project has six main steps: message input, key size definition, encryption of message, decryption comparison and message output. The author developed three functions: key_size(message), encryption() and decryption().

After importing the Python libraries cirq, math, numpy, random, binascii and the function binary_labels, the message can be inserted as a string data type. At first, it was considered that foreign letters, special characters and accents wouldn't be interpreted correctly by key_size(message). After testing, there were no problems, so it supports string messages with any character and number, in any language.

key_size(message) uses message as argument to define the size of Alice (sender) and Bob's (receiver) secret keys. That is, how many bits it will have and how many qubits will be used in each circuit. It takes the message, converts it to binary based on the length, calculates the quantity of 0s and 1s, then applies a mathematical operation on the result with rounding up.

Applying this function is better than defining the size arbitrarily without any logic behind it because it directly states how many resources (bits and qubits) will be used. Here, the key size is influenced by factors that evens out the use of said resources. Size can be checked by calling the function. Then, just define the qubits.

```
def key_size(message):
  binary_string = " ".join(f"{ord(i):08b}" for i in message)

  count_ones = binary_string.count("1")
  count_zeros = binary_string.count("0")
  key_size.total_values = count_ones + count_zeros

  key_size.num_bits = key_size.total_values /
math.exp(math.log10(key_size.total_values))

  key_size.num_bits = math.ceil(key_size.num_bits)

  print('\nTotal of numbers coded in binary:', key_size.total_values)
  print('\nThe generated key will have', key_size.num_bits, 'bits')

key_size(message)

key_size.num_bits

qubits = cirq.NamedQubit.range(key_size.num_bits, prefix = 'q')
qubits
```

The function encryption() works as Alice of BB84, but Alice is treated as the “sender”. It encrypts the message by generation of a secret key and a circuit by applying random choices on the dictionaries of bits and bases. It needs to use the attribute *key_size.num_bits* to be the range. It doesn't take any arguments. The randomness guarantees that many different keys can be generated as needed during the communication. The circuit generated has qubits in many states, varying from pure like |0> and |1> (encoded in Z basis) to superpositions |+> and |–> (encoded in X basis).

```
def encryption():
  encryption.sender_key = choices([0, 1], k = key_size.num_bits)
  encryption.sender_bases = choices(['Z', 'X'], k = key_size.num_bits)
  encryption.sender_circuit = cirq.Circuit()

  for bit in range(key_size.num_bits):
    encode_value = encryption.sender_key[bit]
    encode_gate = encode_gates[encode_value]

    basis_value = encryption.sender_bases[bit]
    basis_gate = basis_gates[basis_value]

    qubit = qubits[bit]

    encryption.sender_circuit.append(encode_gate(qubit))
    encryption.sender_circuit.append(basis_gate(qubit))



  print('\nAlice\'s randomly chosen bases: ', encryption.sender_bases)
  print('\nAlice\'s initial key: ', encryption.sender_key)
  print('\nAlice\'s Phase 1 circuit:\n', encryption.sender_circuit)
```

The decryption() function is used by Bob, the receiver. It decrypts the message by interacting with the qubits that Alice produced in her circuit. They are in states like |0>, |1> or a superpositions like |+> and |–>. The function randomly picks bases X or Z from the dictionary, applies them to the qubits and generates a new sequence of states. These new states must be measured to generate a random secret key formed by a sequence of bits. The variable bb84_circuit combines those steps, including the measurement, to generate Bob's key.

Because this is a function, it was necessary to define variables as attributes like in *decryption.receiver_key* so they can be used outside the function. This is the best approach instead of using Python's "global" keyword.

```
def decryption():
  decryption.receiver_bases = choices(['Z', 'X'], k = key_size.num_bits)
  receiver_circuit = cirq.Circuit()
```

```
for bit in range(key_size.num_bits):
  basis_value = decryption.receiver_bases[bit]
  basis_gate = basis_gates[basis_value]

  qubit = qubits[bit]

  receiver_circuit.append(basis_gate(qubit))

receiver_circuit.append(cirq.measure(qubits, key = 'receiver key'))

bb84_circuit = encryption.sender_circuit + receiver_circuit
sim = cirq.Simulator()
results = sim.run(bb84_circuit)

decryption.receiver_key = results.measurements['receiver key'][0]

print('Bob\'s randomly chosen bases: ', decryption.receiver_bases)
print('Bob\'s initial key: ', decryption.receiver_key)
print('Bob\'s Phase 2 circuit:\n', receiver_circuit)
```

Finally, the last step is a comparison of random bases and bits & output of message. Alice and Bob won't compare their entire keys because it ruins the purpose of them being secret, but only some bits. First, bases are the compared to remove qubits from both keys if they are not in the same basis. The remaining qubits are the ones used for bit comparison. If the bits match, the message is revealed as an output. Else, the comparison works as a verification, therefore, Eve could have interacted with the process and the communication couldn't be secure.

If an eavesdropper Eve wants to try to intercept the information, she can perform the simplest active strategy which is the Measurement Attack. She would apply a measurement on the attribute *encryption.sender_circuit*, right after the key was generated, get the results as pure states, then send the circuit with altered states to Bob without interrupting the communication process. Measurement attacks are not very effective to decrypt the message, but they're good as a way to disrupt both encryption and decryption. Alice encryption steps become useless and Bob's decryption will generate a wrong key. This is noticeable during comparison.

Even if Eve tried other strategies like Intercept and Resend or Entanglement Attack, she wouldn't be successful in virtue of the No Cloning Theorem, which rules that it's impossible to make a perfect copy of a pure quantum state. Any attack to get information would imply measurements on the qubits, collapsing and altering their original states. This leaves no way for recreating the initial states.

Conclusion

While classical cryptography systems as RSA (Rivest-Shamir-Adleman) and HFEv- (Hidden Field Equations) are robust and very difficult to break today, quantum algorithms like Shor's and Grover's are expected to be proven as capable of defeating them in the future when fault tolerant quantum computing with thousands of qubits is achieved (Bernstein & Lange, 2017).

Today, there are various QKD protocols available for use with photons as Bell pairs through different ways, like wire, detectors or low-Earth-orbit satellites. These protocols do quantum communication, thus, they're susceptible to sophisticated quantum attacks, to the effects of noise and decoherence due to physical distance and to vulnerabilities of quantum networks and repeaters (Pirandola et al., 2020). Besides that, some part of the communication process may need classical post-processing which may be vulnerable to hacking by several methods of exploitation.

Therefore, quantum cryptography and encryption are areas that probably may be best seem not as enemies, but as allies to secure communication, because classical computing will continue to be the main tool for many technologies that will exist in the future. Global stakeholders from academia, industry, military and governments need to maintain a safe environment for the evolution of cyber and information security.

References

Bennett, C. H., & Brassard, G. (2014). Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, *560*(1), 7–11. https://doi.org/10.1016/j.tcs.2014.05.025

Bernstein, D. J., & Lange, T. (2017). Post-quantum cryptography. *Nature*, *549*, 188–194. https://doi.org/10.1038/nature23461

Pirandola, S. et al. (2020). Advances in Quantum Cryptography. *Advances in Optics and Photonics*, *12*(4), 1012-1236. https://doi.org/10.1364/AOP.361502