A Simulation of the BB84 Quantum Key Exchange Protocol

by

Andrew Thorp

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

December 2019

APPROVED BY:

Raghuveer Mohan, Ph.D., Thesis Project Director

Chad Waters, M.Sc., Second Reader

TBD, Departmental Honors Director

Rahman Tashakkori, Ph.D., Chair, Computer Science

ABSTRACT

A Simulation of the BB84 Quantum Key Exchange Protocol.

(December 2019)

Andrew Thorp, Appalachian State University

Appalachian State University

Thesis Chairperson: Raghuveer Mohan, Ph.D.

Hello world!

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Personal computers have been developed to a point where those unfamiliar with computer science theory might conclude there is nothing computers cannot do. While this is an understandable conclusion, it has been proven that there is a limit to the types of computation our "classical computers", what we today consider general purpose computers, can perform [3]. In the last few decades however, the field of quantum mechanics and quantum computing have advanced to the point where primitive operations are now possible in the quantum sphere. Just this year Google claimed to achieve "quantum supremacy" in an experiment in which they performed a computation, using a quantum computer, in under five minutes. Google estimated that the same computation would take a state-of-the-art super computer 10,000 years to complete [9]. While quantum computers are not general purpose, they can solve NP-Hard and exponentially complex problems in polynomial time or better [6]. This breakthrough in computability will change the way information is stored, secured, and created.

Currently, encryption protocols ensure the integrity of data and identities. One such protocol is the the widely adopted Diffe-Hellman protocol, which is an asymmetric encryption protocol that relies on the historic difficulty of factoring large prime numbers for security [8]. The protocol works by using a public and private key pair for each participating party. Data encrypted using one of the keys (usually a public key) can then only be decrypted using the private key. A person's public and private keys are mathematically related, but it requires factoring large prime numbers to derive the private key from the public key, which is computationally infeasible with a classical computer. Quantum computers however, are able to do

this in only polynomial time complexity [12]. This development, combined with the growing power of quantum computers, gives rise to security concerns to the Diffe-Hellman protocol in the future.

The BB84 is a quantum key distribution protocol which uses properties of quantum bits to address the growing security concerns of current key exchange protocols (KEP). The BB84 protocol allows two parties to co-generate a disposable, randomly generated, encryption key which can be used to encrypt and decrypt data, and then be discarded after use. This type of a use and throw encryption key is known as a one-time-pad, and it gets its security from being disposable. Common data patterns for breaking encryption keys by a malicious third party cannot be used due to the disposable nature of the one-time-pad. This technique is only susceptible to a brute force attack, which is always possible in principal but often infeasible [10]. The BB84 protocol allows for detection of an eavesdropper during the key generation process, allowing them to abort key generation before any encrypted messages are transmitted [7].

This thesis presents a peer to peer simulation of the BB84 quantum KEP, and serves as an introduction to programming in the quantum computing paradigm using simulaqron and cqc, a quantum network simulator and messaging interface [2]. We show that the BB84s security guarantees hold true in the simulated environment. The simulation allows two parties to co-generate an encryption key and begin exchanging encrypted information using that key. Both parties can simultaneously detect if a third party tried to eavesdrop on the key generation, allowing the users to abort the communication.

The rest of the thesis is organized as follows. Chapter 2 provides background information on encryption protocols, the quantum computation paradigm, and other background information related to the BB84 protocol. Chapter 3 details the BB84 protocol algorithm, as well as discusses its theoretical guarantees. Chapter 4 describes the simulaqron and cqc libraries, as well as the BB84 simulator. This chapter also serves as introductory material for someone getting into quantum simulation or programming in the quantum paradigm. Chapter 5 summarizes this simulator and its potential applications, as well as discusses possible future work.

Chapter 2

Preliminaries

In this chapter, we discuss the preliminaries of encryption and quantum computation, the two ideas key to understanding the BB84 quantum KEP.

2.1 Encryption

Consider two parties, Alice and Bob, trying to communicate through a channel. If the channel is insucure, a malicious third party, Eve, can listen and eavesdrop on the communication. Encryption helps to make the communication more secure, by obscuring data such that an eavesdropper cannot gain any information by intercepting communication [5].

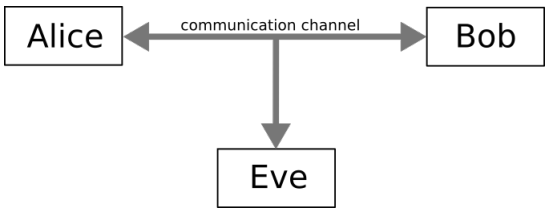

Figure 2.1: Alice and Bob communicate over an insecure channel with an eavesdropper, Eve

In practice, information is encrypted by manipulating the data according to some encryption algorithm. The goal of a good encryption algorithm is simple: allow the sender and receiver to access the information, but make the data useless to anyone else. In order to allow for information to be decrypted by the receiver, the algorithm must be reversible with the use of a secret key, also known as the encryption key. The basic encryption process is as follows:

- A sender, Alice, composes a message, referred to as the plaintext.
- Alice encrypts the plaintext into cyphertext using some encryption algorithm and an encryption key.
- Alice then sends the cyphertext to the receiver, Bob.
- Bob uses the encryption key to decrypt the cyphertext into plaintext.
- Note that the plaintext was never transmitted over the network.

Encryption appears in two general forms: symmetric and asymmetric, each with their own benefits and drawbacks.

Symmetric Encryption

Symmetric encryption is an encryption method in which both the sender and receiver, Alice and Bob respectively, use the same key. While symmetric encryption is the oldest form of encryption,

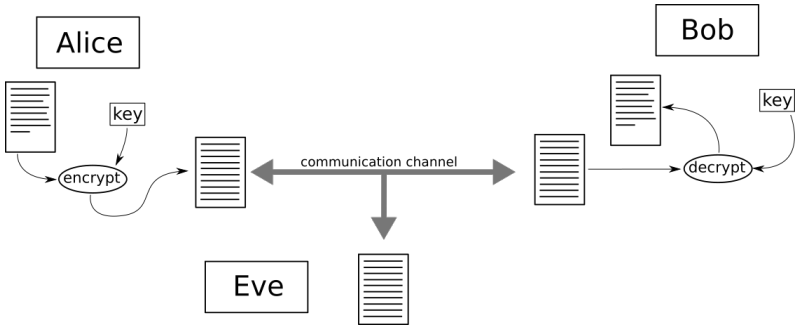

Figure 2.2: Alice and Bob symmetrically encrypt and decrypt a message

being used for over 4000 years, it does have a major drawback – the key distribution problem: if Alice and Bob are separate parties (as in the case of secure message transmission), they must exchange their key beforehand using a secure channel [10]. If an eavesdropper, Eve, were to intercept the key in transit, then the encryption is compromised and there is not necessarily any way for Alice or Bob to know.

A One-Time-Pad (OTP) is a symmetric key that is used only once. The OTP, as the name suggests, is discarded after a singe message has been sent, and the OTP is usually the

same length as the message it encrypts. This allows bit-wise encryption techniques, such as parity manipulation, to add the the security of the key and ensures that common frequency hacking techniques are not possible [10]. In order to make the OTP truly secure, the bits used must be generated using a true-random number generator to avoid a malicious party guessing the pad. Data is encrypted using a OTP with the following bitwise operation:

$$\begin{aligned} \text{Plaintext:} \quad & X = \{x_0, x_1, ..., x_n\} \\ \text{OTP:} \quad & P = \{p_0, p_1, ..., p_n\} \\ \text{Cyphertext:} \quad & Y = \{y_0, y_1, ..., y_n\} \\ \text{where} \quad & y_i = (x_i + p_i) \mod 2 \end{aligned}$$

For example, in binary:

Plaintext:	0 1 0 0 1 0 1 0
OTP:	0 1 1 0 1 0 0 1
Cyphertext:	0 0 1 0 0 0 1 1

To decrypt the cyphertext using the OTP, you simply apply the same opperation:

Cyphertext:	0 0 1 0 0 0 1 1
OTP:	0 1 1 0 1 0 0 1
Plaintext:	0 1 0 0 1 0 1 0

Under the proper conditions, a OTP is considered to be "perfect encryption", and is only susceptible to brute force attacks, where one has to try every possible combination of the secret key, which is computationally infeasible [10]. However OTPs suffer from the same issue as all symmetric encryption protocols, the key distribution problem. In the case of the OTP, not only does one encryption key have to be distributed, a unique key must be distributed for each message, making it highly impractical in most cases.

Asymmetric Encryption

Asymmetric encryption, also known as public-key cryptography, on a classical computer, is the process of encrypting data using one key in such a way that it can be decrypted using a different key. In effect, Alice can encrypt a message using a key that is publicly accessible, and

the message can only be decrypted by Bob, who has the secret key corresponding to the public key used. This method requires a key to be compromised of two parts, a public and private key-pair: $k = (k_{pub}, k_{priv})$ where $k_{pub} = f(k_{priv})$ [10].

The Diffie-Hellman KEP was the first asymmetric KEP to be proposed, and is still widely used in various forms today. It is performed as follows:

1. The sender and receiver, Alice and Bob respectively, establish a line of communication.
2. Some large prime p and an integer $\alpha \in \{2, 3, 4, ..., p-2\}$ are shared between Alice and Bob.
3. Alice and Bob each compute their own private keys, a and b, respectively.

$$k_{priv,A} = a \in \{2, 3, 4, ..., p-2\}$$

4. Alice sends Bob her public key $A = \alpha^a \mod p$.
5. Bob sends Alice his public key $B = \alpha^b \mod p$.
6. Alice computes a session key $k_{AB} = B^a \mod p$.
7. Bob computes a session key $k_{AB} = A^b \mod p$. Because $(\alpha^a)^b = (\alpha^b)^a = \alpha^{ab}$, both Alice and Bob's computations result in the same session key despite never knowing each other's private keys.

This method is cryptographically secure due to the Discreet Logarithm Problem (DLP), finding x such that $B = \alpha^x \mod p$. This problem is computationally not feasible to solve using classical computers; although it has never been shown to be NP-Hard, nor has it been shown to be polynomial time solvable [11]. Although these techniques are currently secure, they become easy to solve in the quantum space.

2.2 Quantum Computation

Just as classical computation involves bits, quantum computation is computation using quantum bits, qubits, which are usually denoted using "bra-ket" notation as $|\psi\rangle$ [8]. A ket is simply

a representation of a vector, and in this case the vector is the "state-vector" of the qubit, as further explained. Similar to classical bits having the value 0 or 1, qubits' state can be the analogous $|0\rangle$ or $|1\rangle$. These states will be used as binary, just like 0 and 1, for computation. It is commonly said that while a classical bit exists in either the state 0 or 1, qubits can exist in both $|0\rangle$ and $|1\rangle$ at once. There is an element of truth in this, but a more accurate description would be that as a qubits state-vector exists in a 3-dimensional space, and it can point somewhere in between the states $|0\rangle$ and $|1\rangle$. This state-vector is described as a linear combination $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $|\alpha|^2 + |\beta|^2 = 1$, and α and β are referred to as the amplitude of probability for their respective kets.

Just as a computer can read the value of a classical bit, we can read, or"measure", a qubit. The measurement is performed against two states, $|0\rangle$ and $|1\rangle$, and upon measurement the qubits state collapses to one of the two values. The probability with which a vector will collapse into one of two states is the square of the amplitude for that state. It is unknown exactly what causes the superposition to collapse, but it has been derived from empirical observations and is the single most important property of quantum mechanics [8]. For example, consider the qubit $|\psi\rangle = \frac{1}{\sqrt{3}}\,|0\rangle + \sqrt{\frac{2}{3}}\,|1\rangle$. The probability that $|\psi\rangle$ will be $|0\rangle$ when measured is $(\frac{1}{\sqrt{3}})^2$, or $\frac{1}{3}$. Any qubit or state-vector is defined using this vector formula. If $|\psi\rangle$ is a linear combination of $|0\rangle$ and $|1\rangle$, and neither amplitudes are zero, the qubit is said to be in a superposition of $|0\rangle$ and $|1\rangle$. Superposition is one of the fundamental properties of quantum computation [8].

We use quantum operators called gates to manipulate α and β probabilities. For example, the Z gate inverts the qubit:

$$|\psi\rangle = |0\rangle \rightarrow \boxed{\text{Z}} \rightarrow |\psi\rangle = |1\rangle$$

$$|\psi\rangle = |1\rangle \rightarrow \boxed{\text{Z}} \rightarrow |\psi\rangle = |0\rangle$$

Similarly, the Hadamard Gate, or H gate, performs what is called a "quarter turn"; it maps

$$|\psi\rangle = |0\rangle \rightarrow \boxed{\text{H}} \rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle$$

$$|\psi\rangle = |1\rangle \rightarrow \boxed{\text{H}} \rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

[8]. Since after the H gate is applied α and β both equal $\frac{1}{\sqrt{2}}$, and $(\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$, if we measure $|\psi\rangle$ in this state it would collapse to $|0\rangle$ and $|1\rangle$ with equal probability. It is worth noting this is one way to create a true-random number generator.

Because the state vector exists in a 3D state space, we do not necessarily have to measure against $|0\rangle$ and $|1\rangle$, in fact we can measure against any two vector values that are opposite each other on the unit circle. A set of vectors to measure against is known as a basis, and there are many possible bases. The set $\{|0\rangle, |1\rangle\}$ is known as the standard basis or computational basis, as it is analogous to classical bits [7]. Another common basis is the Hadamard Basis, which is denoted $|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. The H gate will put $|0\rangle \rightarrow |+\rangle$ and $|1\rangle \rightarrow |-\rangle$, but it will also revert the Hadamard basis into the standard basis: $|+\rangle \rightarrow |0\rangle$ and $|-\rangle \rightarrow |1\rangle$.

Because the standard basis and the Hadamard basis are perpendicular to each other they are referred to as orthonormal bases. That is to say if a $|\psi\rangle = |+\rangle$ or $|\psi\rangle = |-\rangle$ then it will have a 50% change of being $|0\rangle$ or $|1\rangle$ if measured in the standard basis, and vice-versa [8].

		Measured value			
		$\|0\rangle$	$\|1\rangle$	$\|+\rangle$	$\|-\rangle$
Qubit state	$\|0\rangle$	100%	0%	50%	50%
	$\|1\rangle$	0%	100%	50%	50%
	$\|+\rangle$	50%	50%	100%	0%
	$\|-\rangle$	50%	50%	0%	100%

Figure 2.3: The probability of measuring a qubit from one basis state into another

In effect, if you have a vector in one orthonormal basis it is useless if measured in another orthonormal basis. Further, since the vector state changes when measured, if a value is encoded in one orthonormal basis, that information is destroyed by measuring in another orthonormal basis [7].

Another crucial property of qubits is that they cannot be cloned. It is impossible to

have an operator that clones the state of an input qubit into an output qubit without knowing the basis of the input [8]. This property proves crucial in the BB84 protocol, as explained in Chapter 3.

Using these properties and others, it has been shown that quantum computers can solve the DLP on an n-bit number in only $O(n^2 \log n \log\log n)$ time [6]. Therefore, the popularity of quantum computers poses a serious threat to the securities of the Diffie Hellman KEP and asymmetric encryption. The BB84 protocol is a quantum key distribution (QKD) protocol that allows two parties to co-create a shared key, using a verifiably secure channel, that can then be used to symmetrically encrypt messages.

Chapter 3

Quantum Key Distribution

Quantum computers have been shown to solve the mathematical problems that make our encryption secure. Although symmetric encryption is much less effected by this, it is not commonly used due to the key distribution problem [10]. One potential solution to this issue is the use of Quantum Key Distribution (QKD) which is the application of quantum mechanics and quantum computing properties in a key distribution protocol that is provably secure [6]. It allows for two parties to securely generate a symmetric key. In this chapter, we look at the BB84 quantum key exchange protocol that uses principles of quantum mechanics to exchange encryption keys between two parties, Alice and Bob. The encryption key is exchanged through a quantum channel in the form of qubits. An eavesdropper listening in on this communication cannot obtain any information without disturbing the qubits and measuring them, which introduces noise to the signal since the basis is unknown. This means the security of the BB84 protocol comes from the fundamental laws of quantum physics [6].

3.1 The BB84 Protocol

In 1984 Charles Bennett and Gilles Brassard proposed the first quantum key distribution protocol, the BB84 [8]. The protocol allows two parties communicating over a public classical channel, such as the internet, and a public quantum channel to securely generate a shared encryption key for symmetric encryption.

BB84

1. Alice and Bob connect via quantum network simulator.
2. Alice generates N random bits, where $N \geq 3k$, k = desired key size
3. Alice encodes the bits into qubits, randomly choosing to encode using one of two orthonormal bases.
4. Alice sends the qubits to Bob.
5. Bob measures each qubit, choosing to use one of the two selected bases at random.
6. Bob sends a bitvector of his chosen bases to Alice over a classical channel.
7. Alice responds with a bitvector showing which of Bob's bases were correct.
8. Alice and Bob discard all qubits that were measured in the wrong basis.
9. Bob sends Alice some number of the measured values to ensure the key was received without any interference.
10. Alice responds with whether or not the exchanged values were correct.
11. If there were no errors in the compared bits, the bits that were exchanged are discarded by both parties, and the rest of the bits are used as the key.

Figure 3.1: A formal definition of the BB84 Quantum Key Distribution Protocol

To start the quantum key exchange, Alice randomly generates two bit sequences, a and b, of length at least $N = 4l$ bits each, where l is the intended key length. Alice then encodes a into a block of N qubits, $|\psi\rangle$. This is done using two bases; in our case, the standard basis and Hadamard basis. The basis chosen for encoding each bit in a is determined by the corresponding bit in b, with $b_i = \{0 or 1\}$ where $0 \Rightarrow$ standard basis and $1 \Rightarrow$ Hadamard basis. Thus, each qubit is in either the standard basis or the Hadamard basis, and is in one of four states shown in figure 3.2. Now Alice has the qubits encoding a, $|\psi\rangle$, the bases of which are determined by

a_i	b_i	$\|\psi_i\rangle$
0	0	$\|0\rangle$
1	0	$\|1\rangle$
0	1	$\|+\rangle$
1	1	$\|-\rangle$

Figure 3.2: The four possible states of a qubit in a BB84 encoded string

b. Alice sends each qubit to Bob using a public quantum channel. When Bob has received each qubit he can assemble the full qubit block $|\psi\rangle'$. Assuming a perfect quantum channel and no eavesdropping, there should not be any disturbance or noise in the communication: $|\psi\rangle = |\psi\rangle'$. Once Bob has received all qubits, he measures each qubit in $|\psi\rangle'$ into a bit sequence a' by randomly choosing a basis of measurement for each bit. The bases chosen are stored in a bit sequence b'. Bob then informs Alice that he has measured all the received qubits. Because there is a 50% chance that Bob will choose an incorrect basis for measurement for each qubit, and a 50% probability of measuring the correct value using the wrong basis, Bob has measured 75% of the qubits correctly, on average (figure 3.3). While 75% of the bits are measured correct

Encoded value	0	0	0	0	1	1	1	1
Encoded basis	0	0	1	1	0	0	1	1
Measured basis	0	1	0	1	0	1	0	1
Measured value	0	0/1	0/1	0	1	0/1	0/1	1

Figure 3.3: Exhaustive list of encoding and measurement of a single qubit between Alice and Bob

on average, an average of 50% are measured in the correct basis; those qubits are guaranteed to be measured correctly. Note that Alice has (a, b) and Bob has(a' b'), but they each do

not know the others a and a'. Alice and Bob now exchange the bases they each used, b and b', respectively. Alice and Bob both discard every bit that was encoded and measured using different bases: bit i is discarded from a and a' if $b_i \neq b'_k$. They store the remaining bits from a and a' into a new bit sequence, k and k', respectively. Alice and Bob now each have the same key, $k = k'$. They each exchange some number of bits from k to verify there was no error in their key generation. If the exchanged bits are identical, then Alice and Bob can confirm with high probability that there was no eavesdropper, and that they key is secure [6]. They can now use the key for symmetrically encrypted communication, or even as a OTP.

If an eavesdropper, Eve, were to attempt to listen in on the conversation, not only would they not be able to gain any useful information from the qubits, since she does not know the basis at which the qubits are encoded, nor can she duplicate the qubits. Therefore the only strategy she has is to perform a "man-in-the-middle" attack, in which Eve impersonates Bob to Alice and Alice to Bob [8].

To eavesdrop on the communication, Eve listens on the quantum channel and waits for Alice to transmit qubits. As Alice sends Bob the qubits over the quantum channel Eve intercepts each qubit, forming her own qubit block $|\psi\rangle'$. With the qubits now in Eve's possession, she attempt to measure the qubits or clone them. However, as previously shown, this would introduce noise to the signal, reducing Bob's correct average measurement percentage from 75% to 62.5%, leaving only 25% of the qubits measured in the same basis used during encoding. As

Basis_{Alice}	Basis_{Eve}	Basis_{Bob}	Percent Correct	In Final Key
0	0	0	100%	1
0	0	1	50%	0
0	1	0	50%	1
0	1	1	50%	0
1	0	0	50%	0
1	0	1	50%	1
1	1	0	50%	0
1	1	1	100%	1

Figure 3.4: Exhaustive list of encoding and measurement of a single qubit between Alice and Bob with an eavesdropper

can be seen in figure 3.4, when there is an eavesdropper, half of the bits kept in the generated key are measured in an incorrect basis by at least one party. This means, on average, 25% of the

bits in k' are incorrect. When Alice and Bob exchange some bits to verify their correctness, even if only four bits are compared they both will, on average, detect that the qubits were maliciously measured during transmission, at which point Alice and Bob can abort communication [6].

In practice this protocol can be implemented using polarized photons as qubits, which can be sent between Alice and Bob using fiber optics. The data is encoded into the photons using the angle of the polarization since polarization can act as a qubit [8]. In this case the bases for encoding data are the standard basis, $(|0\rangle, |1\rangle) = (|\rightarrow\rangle, |\uparrow\rangle)$, and the Hadamard basis, $(|+\rangle, |-\rangle) = (|\nearrow\rangle, |\nwarrow\rangle)$. However all that is required to perform any QKD protocol is the ability to communicate qubits over a public channel with a very low error rate [6].

Chapter 4

The BB84 Simulator

There are already several quantum computers available to the public in the form of APIs. These APIs allow users to simulate and perform quantum computation by sending requests to a quantum processing unit (QPU) [1]. O'Riely has even recently released a textbook on Qiskit, IBM's quantum programming library[4]. Qiskit allows programmers to simulate and test quantum programs locally before running the code against IBM's cloud quantum computer [1]. Other platforms, such as D-Wave's quantum annealing API, let programmers explore other quantum computation paradigms, not just the imperative quantum paradigm we have discussed here.

These advancements make quantum computation ever more accessible to the general public. However, there are still many limitations; compute-time access must be shared, the number of qubits a user can control is limited, and networking is currently not possible using public quantum computing APIs. This leads to the further development of efficient quantum simulation libraries such as Simulaqron, a quantum networking library, and qrack, an efficient quantum simulation library [13]. Due to the number of qubits and networking required to run the BB84 protocol, all applications presented by this thesis have been written using quantum computation simulators such as those aforementioned.

4.1 Simulaqron Quantum Network Simulator

Simulaqron is a quantum network simulation library that allows the programmer to define networked clients and have let client programs manipulate and exchange simulated qubits [2]. An instance of Simulaqron acts as a quantum server in the simulator. It serves as both the public quantum channel host and the quantum computer that creates and distributes the qubits to each party.

Simulaqron is architected as a simulated QPU singleton, that is to say it stores all qubits in the network. Simularon acts as an interface and adapter to the simulated QPU backend, which can be configured. This allows the user to chose a backend that optimizes the sorts of operations they will perform. Though there are currently no backends for simulaqron that perform quantum computations on a true QPU, one could be written [2].

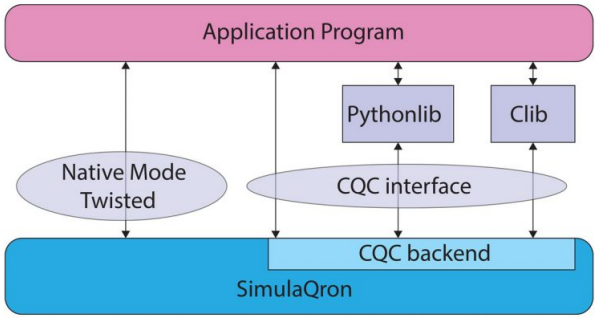

Figure 4.1: Schematic of the simulaqron stack.

Source: http://www.simulaqron.org

The quantum network connected to simulaqron is also configurable. The programmer may specify a network topology graph, client nodes and connections, and a client can connect by identifying itself as a node in the topology. Clients connected to a simulaqron server can request and manipulate qubits kept on the server, and the server maintains qubit information such as state and ownership. This network can be accessed through several methods: Native Mode, which is designed for building interfaces to simulaqron, and the CQC interface, which is what was used in the BB84 simulator.

The CQC Interface

The CQC interface library is the main method with which a program interfaces with a simulaqron server. The interface itself is a client and server side implementation of the command pattern, with friendly libraries available in Python, C, and Rust [2]. For the purposes of this thesis, the Python library will be the subject of discussion. The client side library behaves in code like a local QPU object, allowing users to create qubits and manipulate them, making it very accessible.

To use the CQC in a client there must be a simulaqron already running locally or at a known address. A connection object must first be instantiated, since all quantum operations are performed server-side. This can be done with `cqc = CQCConnection("Client_name", socket_address=("1.1.1.1", 8801))` or, preferably, using the following syntax: `with CQCConnection("Client_name") as cqc:`. If the socket address argument is omitted then the connection targets localhost [2]. With a CQC connection instantiated, a client can request qubits from the server, `q = Qubit(cqc)`. All new qubits are initiated to the state $|0\rangle$. Once the server has responded with a new qubit, the client can apply quantum gates to it For example, they can use `q.H()` to apply the Hadamard gate and `q.Z()` to apply the Z gate. Once the qubit is in the desired state, it can be measured with ‘result = q.measure()‘, at which point the function returns a 1 or 0 and the qubit is discarded by the server. It is notable that the qubit is not discarded if the user passes the flag `inplace=True` to `measure()` , but this flag is false by default.

The interface allows for communicating classical information as well. The functions `sendClassical()` and `recvClassical()` will send and receive an array of bytes between two client applications. This is especially useful when communicating information about quantum protocols.

A programmer can send qubits between clients as well using this interface. Consider a client, Alice, that wants to send the client Bob a single qubit. This can be accomplished as shown in listing 4.1. Although, when using the interface, it is impossible to know the state of a qubit if you do not measure it yourself, it is noteworthy that since the connection object requests the measurement from the server, and the server responds with the result, a malicious

party could use use packet sniffing to obtain the value of a qubit measurement.

Listing 4.1: An example usage of sending a qubit between two clients in CQC.

```
from cqc.pythonLib import CQCConnection, qubit

# Encode and measure a qubit
with CQCConnection("Alice") as Alice:
    q = qubit(Alice)

    # Encode 1 in qubit
    q.Z()

    # Send qubit to Bob
    Alice.sendQubit(q, "Bob")

# Recieve and decode qubit
with CQCConnection("Bob") as Bob:
    # Recieve qubit from server
    q = Bob.recvQubit()

    decoded = q.measure()
```

As a point of verification, CQC was used to test the randomness of simulaqron. While it is not true-random, it did provide satisfactory results. In a trial of measuring encoding 100 qubits into the state $|+\rangle$, and measuring them in the standard basis, as a 0 or 1, 10,000 times, we found that 50% of the qubits were 1, on average, with a standard deviation of 4%. The same experiment was repeated using python's standard library random number generator, which produced the same mean and standard deviation. This lead us to conclude that the two forms of random number generation were comparable. If the measurements were performed by an actual QPU on real qubits, they would, of course, be true-random.

4.2 BB84 Simulator Library

The BB84 QKD protocol can be implemented using CQC, as it has all the necessary quantum functionality. During the key exchange all quantum and classical communication is done using `cqc.sendQubit()` and `cqc.sendClassical`, respectively. As previously mentioned, in this form, the quantum channel is still vulnerable to packet sniffing, but once the key is generated the symmetric encryption cannot be broken, if used correctly [10]. The BB84 can be broken into into Alice and Bob's respective responsibilities. Since each client will have to operate independently, we will need to examine them each individually.

Alice

Alice first has to connect to the server. As previously shown, this can be done with `with CQC Connection("Alice") as Alice:`. Having connected to simulaqron, Alice must generate N random bits, where $N \geq 3k$, k = desired key size. This is done twice, once for the key's bit values and once for the bases. Since the qubits can be used a random number generator, this is done as shown in listing 4.2. Next, Alice encodes the bits into qubits, choosing the basis to

Listing 4.2: Randomly generating N bits using the CQC library.

```
1  from BitVector import BitVector
2  from cqc.pythonLib import CQCConnection, qubit
3
4  N = 32
5  bits = BitVector(intVal=0, size=N)
6
7  with CQCConnection("Alice") as Alice:
8      for i in range(N):
9          q = qubit(Alice)
10         # Put qubit in superposition
11         q.H()
12         bits[i] = q.measure()
```

use according to that index on the basis bitvector as shown in listing 4.3. Alice then sends all

Listing 4.3: Alice encodes a list of bits into qubits.

```
1  with CQCConnection("Alice") as Alice:
2      qubits = [None] * N
3      for i in range(bits):
4          qubits[i] = qubit(Alice)
5
6          # Encode value
7          if bits[i] == 1:
8              qubits[i].Z()
9
10         # Change basis
11         if bases[i] == 1:
12             qubits[i].H()
```

the qubits to Bob. Once Alice receives Bob's chosen bases for measurement, she can perform a NXOR between her and Bob's bases. The result of this operation is a bitvector representing those qubits which were both encoded and measured using the same basis: `correct_bases = ~bobs_bases ^alices_bases`. This new bitvector of correct bases is sent back to Bob, and Alice discards all qubits that were measured in the wrong basis: `key = [bits[i] for i`

`in range(N) if correct_bases[i] == 1]`. Alice then receives some of the bit values from Bob's final key. In this implementation all bits except the last k are exchanged. Alice then XORs the two verification bitvectors together; if the result of the XOR is non-zero then Alice knows there has been interference [8]. Alice notifies Bob of whether or not interference was detected. If there were no errors in the compared bits, Bob is sent an OK and the compared bits are discarded from the key. If there was an nerror between the compared bits, Bob is notified that there were differences, the process is aborted, and an exception is raised. If the process was not aborted, the remaining bits are the final key to be used in encryption.

Bob

Bob must first connect to the simulaqron server. Bob listens on the quantum channel for Alice to start sending qubits. Bob then receives N qubits from Alice and measures each qubit in a randomly chosen basis, as shown in listing 4.4. With the bases chosen and qubits measured, Bob sends the bases to Alice in the form of a bitlist: `Bob.sendClassical(Alice, bases[:])`. Next Bob receives the list of correct bases from Alice: `correct_bases = BitVector(bitlist = Bob.recvClassical())`. Bob, like Alice, drops all bits from his key if it was not measured in the correct basis: `key = [bits[i] for i in range(N) if correct_bases[i] == 1]`. They remaining bits should be the same for both Alice and Bob. To verify this, Bob sends all the bits except for the final k to Alice: `Bob.sendClassical("Alice",`

Listing 4.4: Bob measures received qubits in randomly chosen bases.

```
measured_num = BitVector(size=N, intVal=0)
bases = BitVector(size=N)

for i in range(N):
    # Randomly apply a quarter spin to use the Hadamard basis
    with CQCConnection("Bob") as Bob:
        received_qubit = Bob.recvQubit()
        rand = qubit(Bob)
        rand.H()

        if rand.measure() == 1:
            received_qubit.H()
            # update bitvector of bases
            bases[i] = 1

        # Measure the bit and insert it into the decoded number
        measured_num[i] = received_qubit.measure()
```

`key[:len(key)-k])`. Bob then waits to receive the verification confirmation from Alice. If the verification is an OK, then Bob sets the key to the last k bits of the key with `key = key[k:]`, otherwise the process is aborted and an exception is raised.

Both Alice and Bob's functionalities have been abstracted into two functions, `initiate_keygen()` and `recv_keygen()`, respectively. These functions, and all helper functions, have been published as Open Source Software under the MIT license [14].

Eve

Adding an eavesdropper into the mix is not technically possible using the simulaqron library [2]. This is because messages are send directly to the target recipient, and the library does not have functionality to go through a third party silently. However, the eavesdropping behavior can be achieved by having both Alice and Bob use Eve as a proxy.

If Eve does not manipulate the qubits en route, acting as an unmalicious proxy, she can easy accomplish this by having both Alice and Bob list her as the recipient for qubits, while she runs code shown in listing 4.6. If, however, Eve chooses to maliciously attempt to measure the

Listing 4.5: Eve acting as an unmalicious proxy between Alice and Bob.

```
for _ in range(N):
    # Receive qubits from Alice
    qubit = conn.recvQubit()
    # Forward qubits to Bob
    conn.sendQubit(qubit, "Bob")
```

qubits, this can be accomplished as shown in listing **??**. Although, in theory, there is a small

Listing 4.6: Eve acting as an malicious proxy between Alice and Bob.

```
measured_values = [None] * N
for i in range(N):
    # Intercept qubits from Alice
    qubit = conn.recvQubit()
    # Measure all qubits in standard basis
    measured_values[i] = qubit.measure(inplace=True)
    # Forward qubits to Bob
    conn.sendQubit(qubit, "Bob")
```

chance that Eve is not detected, in 1000 empirical tests Eve's measurements were detected in ever instance.

4.3 BBChat Messaging Software

As a proof of concept, BBChat, a peer-to-peer and end to end encrypted messaging application, was written to use the BB84 simulator to establish a symmetric encryption key [15]. It was preferentially written as a terminal application, with text-based user interface (TUI). BBChat requires both clients to specify whether or not they are the initiator, "Alice", or the receiver, "Bob", in the key generation process. This is done with by starting the program with the command line flag `-i`, for "initiator".

Upon starting, the program immediately calls `initiate_keygen()` or `recv_keygen()`, depending on if the program is the initiator or not. Throughout the process of key generation, the BB84 library outputs log information to a "quantum log", which is presented in a pane of the TUI. This allows the user to view the BB84 in real time, and gain better insight into its intricacies. If too few bits are in the final key, too few bases were correct between Alice and Bob, then the key generation is restarted. If Alice and Bob exchange verification bits and there is a difference between the bits, the user is notified in the quantum log, and the key exchange is aborted [15]. If the key generation is successful, both clients open a direct TCP connection to each other. The users can now send symmetrically encrypted messages to each other.

Chapter 5

Conclusion

5.1 Future Work

5.2 Summary

Bibliography

[1] Héctor Abraham, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, Gadi Alexandrowics, Eli Arbel, Abraham Asfaw, Carlos Azaustre, Panagiotis Barkoutsos, George Barron, Luciano Bello, Yael Ben-Haim, Lev S. Bishop, Samuel Bosch, David Bucher, CZ, Fran Cabrera, Padraic Calpin, Lauren Capelluto, Jorge Carballo, Chun-Fu Chen, Adrian Chen, Richard Chen, Jerry M. Chow, Christian Claus, Andrew W. Cross, Abigail J. Cross, Juan Cruz-Benito, Chris Culver, Antonio D. Córcoles-Gonzales, Sean Dague, Matthieu Dartiailh, Abdón Rodríguez Davila, Delton Ding, Eugene Dumitrescu, Karel Dumon, Ivan Duran, Pieter Eendebak, Daniel Egger, Mark Everitt, Paco Martín Fernández, Albert Frisch, Andreas Fuhrer, Julien Gacon, Gadi, Borja Godoy Gago, Jay M. Gambetta, Luis Garcia, Shelly Garion, Gawel-Kus, Leron Gil, Juan Gomez-Mosquera, Salvador de la Puente González, Donny Greenberg, John A. Gunnels, Isabel Haide, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Connor Howington, Wei Hu, Shaohan Hu, Haruki Imai, Takashi Imamichi, Raban Iten, Toshinari Itoko, Ali Javadi-Abhari, Jessica, Kiran Johns, Naoki Kanazawa, Anton Karazeev, Paul Kassebaum, Vivek Krishnan, Kevin Krsulich, Gawel Kus, Ryan LaRose, Raphaël Lambert, Joe Latone, Scott Lawrence, Peng Liu, Panagiotis Barkoutsos ZRL Mac, Yunho Maeng, Aleksei Malyshev, Jakub Marecek, Manoel Marques, Dolph Mathews, Atsushi Matsuo, Douglas T. McClure, Cameron McGarry, David McKay, Srujan Meesala, Antonio Mezzacapo, Rohit Midha, Zlatko Minev, Prakash Murali, Jan Müggenburg, David Nadlinger, Giacomo Nannicini, Paul Nation, Yehuda Naveh, Nick-Singstock, Pradeep Niroula, Hassi Norlen, Lee James O'Riordan, Steven Oud, Dan Padilha, Hanhee Paik, Simone Perriello, Anna Phan, Marco Pistoia, Alejandro Pozas-iKerstjens, Viktor Prutyanov, Jesús Pérez, Quintiii, Rudy Raymond, Rafael Martín-Cuevas Redondo, Max Reuter, Diego M. Rodríguez, Mingi Ryu, Martin Sandberg, Ninad Sathaye, Bruno Schmitt, Chris Schnabel, Travis L. Scholten, Eddie Schoute, Ismael Faro Sertage, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Dominik Steenken, Matt Stypulkoski, Hitomi Takahashi, Charles Taylor, Pete Taylour, Soolu Thomas, Mathieu Tillet, Maddy Tod, Enrique de la Torre, Kenso Trabing, Matthew Treinish, TrishaPe, Wes Turner, Yotam Vaknin, Carmen Recio Valcarce, Francois Varchon, Desiree Vogt-Lee, Christophe Vuillot, James Weaver, Rafal Wieczorek, Jonathan A. Wildstrom, Robert Wille, Erick Winston, Jack J. Woehr, Stefan Woerner, Ryan Woo, Christopher J. Wood, Ryan Wood, Stephen Wood, James Wootton, Daniyar Yeralin, Jessie Yu, Laura Zdanski, Zoufalc, azulehner, drholmie, fanizzamarco, kanejess, klinvill, merav aharoni, ordmoj, tigerjack, yang.luh, and yotamvakninibm. Qiskit: An open-source framework for quantum computing, 2019.

[2] Axel Dahlberg and Stephanie Wehner. SimulaQron—a simulator for developing quantum internet software. *Quantum Science and Technology*, 4(1):015001, sep 2018.

[3] Peter Linz. *An Introduction to Formal Language and Automata.* Jones and Barlett Learning, Sudbury, MA, US, fifth edition, 2012.

[4] Eric R. Johnston Mercedes Gimeno-Segovia, Nic Harrigan. *Programming Quantum Computers.* O'Reilly Media, Inc., 2019.

[5] Merriam-Webster. encrypt.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[7] Riley Tipton Perry. *Quantum Computing from the Ground Up.* World Scientific Publishing Co. Pte. Ltd, Hackensack, NJ, US, 2012.

[8] Eleanor Rieffel and Wolfgang Polak. *Quantum Computing: A Gentle Introduction.* The MIT Press, Cambridge, MA, US, 2011.

[9] Eleanor G. Rieffel. Quantum supremacy using a programmable superconducting processor. Paper describing experiment performed by NASA and Google to achieve Quantum Supremacy, 2019.

[10] Simon Robinson. *Understanding Cryptography*, pages 519–551. Apress, Berkeley, CA, 2004.

[11] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):14841509, Oct 1997.

[12] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[13] Daniel Strano and Benn Bollay. vm6502q/qrack, may 2018.

[14] Andrew Thorp. Bb84 quantum simulation library. `https://github.com/athorp96/bb84`, 2019.

[15] Andrew Thorp. Bbchat. `https://github.com/athorp96/bbchat`, 2019.