Anoop Kumar
M.Tech Computer Science (CS1921)

Classical and Quantum Components of Quantum Key Distribution Protocol

Indian Statistical Institute
203 BT Road, Kolkata - 700108

Supervisors: Prof. Subhamoy Maitra

July 2021

A dissertation submitted in partial fulfilment of the requirement for the degree of Master of Technology in Computer Science

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others.

Anoop Kumar
M.Tech CS 2nd year
Roll no : CS1921
Indian Statistical Institute,
Kolkata - 700108, INDIA

Certificate

This is to certify that the dissertation titled **"Classical and Quantum Components of Quantum Key Distribution Protocol"** submitted by **Anoop Kumar** to Indian Statistical Institute, Kolkata in partial fulfilment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per regulation of this institute and, in my opinion, has reached the standard needed for submission.

Subhamoy Maitra
Professor, Applied Statistical Unit
Indian Statistical Institute,
Kolkata - 700108, INDIA

Acknowledgement

I would like to express my sincere gratitude to my supervisor **Prof. Subhamoy Maitra** for giving me an opportunity to work under him and providing me exciting problems to think upon. Throughout the development of this project, he was always available for discussion and guidance. He never hesitated to discuss with me whenever I asked for. Special thanks to **Chandrasekhar Mukherjee** for helping me to understand many things regarding research work and this project. Finally, I would like to express my special gratitude to my parents, who have always encouraged me in all aspects for igniting the love for knowledge within me.

Anoop Kumar,
M.Tech CS,
CS1921

Abstract

Quantum Key Distribution Protocol (QKD) is a type of quantum cryptography, i.e., it uses the concepts related to quantum mechanics to establish security in the communication channel. It is a key generation protocol and is used for generating a shared secret key between two parties, commonly known as Alice and Bob. Any QKD protocol uses two communication channels for the key generation. The first one is a quantum channel for the transmission of qubits. The other one is a classical channel for the transmission of random classical bits. The two major processes of any QKD protocol involve key generation algorithm and classical post processing phase. For the key generation process, algorithms like BB84 protocol or Eckart E91 protocol is used. These algorithms use polarisation, entanglement, and measurement concepts of quantum mechanics for generating the key. The key generated in the final step of the key generation algorithm of the QKD protocol is made up of classical bits. Because of the inherent noisy nature of quantum channels, the keys generated at both ends after classical reconciliation can still be wrong. This defeats the purpose of key sharing. To overcome this adversities classical post processing is used. In this phase the keys generated at the two ends are processed via an error correcting scheme to potentially reach a common key without any public sharing of the data. To this end we have studied the behavior of various error correction schemes, both block based and convolutional schemes and later used them as part of the post processing phase of the BB84 QKD protocol. Then we studied the efficiency of these coding schemes with and without the interruption of a malicious third party.

Contents

List of Tables

List of Figures

1 Introduction

Quantum cryptography is the study of creating a secure communication system using various encoding and decoding algorithms using quantum mechanics. Quantum mechanics provides various concepts that help quantum cryptography to create a cryptographic system that is impossible to make using classical cryptography only. For example, using the concept of the no cloning theorem[14], the detection of an unauthorized party is done in quantum cryptography.

Quantum Key Distribution (QKD) is one of the primary examples of quantum cryptography. QKD is a key generation algorithm, i.e., it is used to generate secure keys, which will further be used in encryption or decryption of some other cryptographic protocol. The basic idea behind QKD protocol is that it uses both classical and quantum bits to generate the shared secret key, say between Alice and Bob, such that no eavesdropper (Eve) can observe it without disturbing the key generated. Furthermore, if Eve tries to find out about the key, Alice and Bob will get notified of Eve's eavesdropping.

QKD uses a quantum communication channel for the transmission of quantum bits and a classical communication channel for the transmission of classical bits. The QKD protocol uses the concept of entanglement and quantum bits measurement (section 2.1) for the preparation of the shared secret key. The final shared key is a string of classical bits, and can be used further

in a different classical cryptographic algorithm. Due to the noisy nature of quantum channels and the possibility of a third party intervention (Eve), even after the classical reconciliation step, there can still be mismatches between the two keys generated at the two ends.

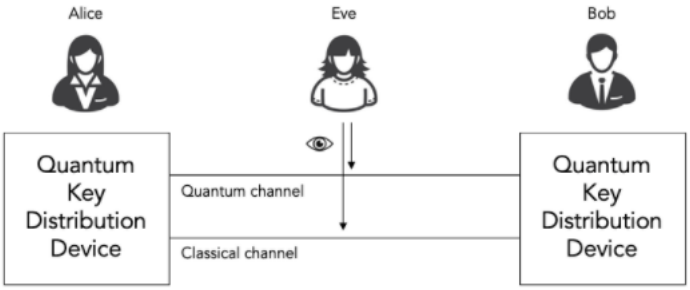

Figure 1.1: QKD Protocol

We are going to use the BB84 QKD protocol [10] in our thesis for generating secret shared keys. Due to the imperfections of hardware, or Eve, there will always be a noise resulting in bit errors among the bits of keys between Alice and Bob, shown in fig1.2. All QKD protocols have some noise level estimation for estimating bit error rates. If the key generated at both ends are same then the error rate will be 0 and if the key generated at one end is the complement of the other then the error rate is 1.

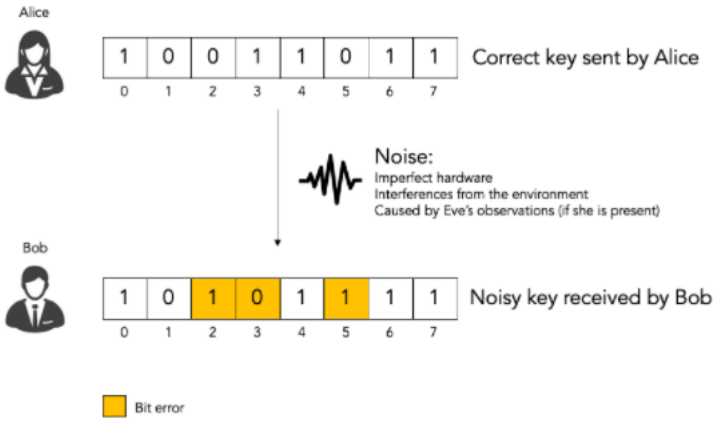

Figure 1.2: Key bit error

We use the classical post processing phase of the QKD protocol if the bit error rate is not more than some threshold value. This classical post processing phase involves the process of information reconciliation and privacy amplification. We use some error correction mechanisms to remove the bit error in the key in the information reconciliation process. In the privacy amplification process, we use some hash functions on the keys generated by the information reconciliation phase for reducing any information gain related to the secret key obtained by Eve during transmission. These two steps of the classical post processing phase of QKD create a final key common to Alice and Bob. In this proposal we only concentrate on the information reconciliation process via error correcting schemes.

1.1 Our Contribution

As discussed above, the shared key generated in the BB84 QKD protocol may contain bit errors because of the inherent noisy nature of current age quantum channels. This error will affect the reliability of the key generation protocol. These bit errors can be corrected in the classical post processing

phase, which requires a suitable error correction mechanism. For this classical post processing phase, we used error correcting coding schemes. We have simulated various error correcting coding schemes upon different randomly generated inputs on different error rates and determine the efficiency of those error correcting coding schemes. Later, we use these coding schemes as a part of the BB84 protocol's classical post process phase and observe the behaviour of the keys generated by the BB84 QKD protocol.

Xiongfeng Ma [15] summarised the important concepts involved in post processing of QKD protocol. Brassard and Salvail [5] gave an information reconciliation method (cascading), used in the first step of the classical post processing phase. This method is quite simple, but the only problem is that it requires multiple interactions between the involved parties to generate a final shared secret key. We have conducted our experiment by practically implementing the concept of Xiongfeng Ma and used error correcting codes as part of information reconciliation which provide the benefit that interaction for the key generation is done only once thus saving more time.

1.2 Organisation of Thesis

We introduced the basic concept of QKD protocols along with classical post processing requirements. These will help us to understand the motive behind our objective stated in the section1.1.

Chapter 2 consists of the information regarding Quantum Key Distribution and classical post processing procedures. Chapter 3 contains various error correcting coding schemes that we will deploy as a part of classical post processing of the QKD protocols. Using the concept used in these chapters, we have conducted the practical implementation of post processing can be

found in chapter 4. In Chapter 5 we have written our conclusion regarding the experiment as well as the future work that can be done. The references used for this thesis can be found in the reference section at the end.

2

Quantum Key Distribution And Its Different Steps

This chapter contains the basics of quantum computing, QKD protocol, and its classical post processing phase. These concepts are later be used in our experiment in the later chapter.

2.1 Preliminaries

2.1.1 Qubits

The qubit or quantum bit is the basic fundamental unit in quantum computers used for carrying the information. It is similar to the concept of bits in classical computers. A qubit is a 2-D quantum system. A qubit's state may be written as follows:

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.1}$$

where α and β are complex numbers and such that $|\alpha|^2 + |\beta|^2 = 1$. Here $Dirac - notation$ is used where $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ represent the basis state of 2d vector space. The measurement of the state of the qubit in the equation 2.1 will give zero (0) with probability $|\alpha|^2$ or one(1) with probability $|\beta|^2$.

2.1.2 Superposition and entanglement

Any linear combination of 2 quantum states is known as superposition. Even if they are normalised, they will still be a valid quantum state. Using some basis states, any quantum state can be expressed as a linear combination of them like in equation 2.1 using the linear combination of $|0\rangle$ and $|1\rangle$, we can express any state of the qubit. A normalised linear combination of 2^n bit string states can be used to express the state of any n qubit system. Computational basis are those orthonormal basis obtained from the 2^n bit string states.

Generally, a system of n qubits can be expressed as the tensor product of n different single qubit states, but sometimes it is impossible to present it in terms of tensor products. An example of such a state is,

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle) \tag{2.2}$$

Such states are known as entangled states. In quantum computing and quantum information processing, the entangled states are seen as physical facts holding important consequences. In fact, in the absence of such states, the power of quantum computers would be similar to their classical counterparts. [12] Because of Entanglement, it is possible that n physical qubits can be used to create a complete 2^n dimensional complex vector space for doing the computations.

2.1.3 Inner and Outer Products

Let for the two qubit states, $|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ and $|\psi\rangle = \gamma\,|0\rangle + \delta\,|1\rangle$. The inner product between these states is denoted as $\langle\psi|\phi\rangle$ in the $ket - notation$ is given by,

$$\langle\psi|\phi\rangle = \gamma^*\alpha + \delta^*\beta \tag{2.3}$$

Where * denotes the complex conjugate.

The outer product of two states yields a matrix upon two states given as input. The outer product of the two states will be denoted by $|\psi\rangle\langle\phi|$. The outer product of two states is a matrix obtained by multiplication of the column vector of the first state with the complex conjugated row vector of the second state, as shown in the example below.

$$|\psi\rangle\langle\phi| = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}(\gamma^* \;\; \delta^*) = \begin{pmatrix}\alpha\gamma^* & \alpha\delta^*\\ \beta\gamma^* & \beta\delta^*\end{pmatrix} \tag{2.4}$$

2.1.4 Measurements

Measurement can be seen as the process of converting the quantum information (present in a quantum system) into a classical one, like measuring a qubit typically the same as checking a classical bit, i.e., whether it is 0 or 1. The outcomes of measurements are probabilistic according to the quantum mechanics principle.

For a qubit state in Eq. 2.1, the probability of getting $|0\rangle$ after applying measurement is given by $|\langle 0|\phi\rangle|^2$ and the probability of getting $|1\rangle$ after applying measurement is given by $|\langle 1|\phi\rangle|^2$. So the squared absolute values of inner products can be used to express the probability of measurements. In general, the probability of obtaining the bit string $|x_1...x_n\rangle$ after measurement of an n qubit state, $|\phi\rangle$, is $|\langle x_1...x_n|\phi\rangle|^2$.

Now take another example where, for a three qubit state, $|\psi\rangle$, we want that only the first qubit is measured and leaving the other two qubits. In

the first qubit, the probability of observing a $|0\rangle$ will be given by,

$$\sum_{(x_2x_3)\in\{0,1\}^2} |\langle 0x_2x_3|\phi\rangle|^2 \tag{2.5}$$

by normalizing the state we get the state of the system,

$$\sum_{(x_2x_3)\in\{0,1\}^2} \langle 0x_2x_3|\phi\rangle^2 |0x_2x_3\rangle \tag{2.6}$$

Applying this procedure to the state in Eq 2.2 we observe that the probability of obtaining $|0\rangle$ in the first qubit will be 0.5, and we would get a final state $|111\rangle$.

Sometimes we need to perform measurements on a basis entirely different from the computational basis, which is achieved by applying a transformation on the qubit register before measurement. [12]

2.1.5 Unitary Transformation and Gates

By using unitary transformation, we can alter the state of a qubit or a qubit system. A complex matrix U is used to describe a unitary transformation which satisfies the following condition :

$$UU^\dagger = U^\dagger U = I \tag{2.7}$$

where $U^\dagger$ is the transposed, complex conjugate of U (also known as Hermitian conjugate) and I is the identity matrix. A qubit state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ changes after applying the 2 x 2 matrix U as follow :

$$|\phi\rangle \to U|\phi\rangle = \begin{pmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} U_{00}\alpha + U_{01}\beta \\ U_{10}\alpha + U_{11}\beta \end{pmatrix} \tag{2.8}$$

The Kronecker product is used to combine the operators working on different qubits. For example, $U_1 \otimes U_2$ will be used for a combined two qubit system having operators U_1 and U_2 acting on different qubits. Like classical logic gates, basic unitary transformations are used to create complicated n qubit transformations known as *gates*. These are also unitary transformations and satisfies the equation 2.7. They are different from the classical gates as it is required for them to be reversible means they can create the gate's input from the gate's output. We can also say that they are the generalization of classical reversible gates. Table 1 contains some common gates.

One-qubit gates	Multi-qubit gates
Hadamard = H = $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$	CNOT = CX = $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$
I = $\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$, S = $\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$	CZ = $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{pmatrix}$
T = $\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix}$	Controlled-U= $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & U_{00} & U_{01}\\ 0 & 0 & U_{10} & U_{11}\end{pmatrix}$
NOT=X = $\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$	SWAP = $\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$
Y = $\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$ Z = $\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$	Toffoli = $\begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{pmatrix}$
$R(\theta) = P(\theta) = \begin{pmatrix}1 & 0\\ 0 & e^{i\theta}\end{pmatrix}$	

Table 2.1: Common Quantum Gates

2.1.6 Quantum Circuits

Circuits are used to represent the quantum algorithm diagrammatically. Horizontal lines are used to denote qubits in circuit representation. Upon those lines, gates acting on them are drawn sequentially from left to right. At the starting of the line on the left side initial state is shown. One thing to note that when writing expression mathematically, right to left convention is followed for the gates.

In Fig 2.1 represent a circuit to create an entangled two qubit state known as Bell state from $|00\rangle$.

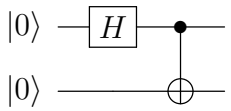

Figure 2.1: Bell State Quantum Circuit

The following equation is encoded by the circuit,

$$CNOT_{12}(H \otimes I)\,|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.9}$$

A meter symbol is used as a special gate to denote the measurement of a qubit as shown in Fig 2.2. This symbol denotes that the measurement is done on a computational basis.

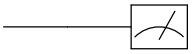

Figure 2.2: Measurement gate

2.2 Quantum Key Distribution

Quantum key distribution (QKD) is a provably secure protocol used over a public channel to generate private key bits between two users. Qubits

can be used over the public channel with the sole requirement that an error rate of the key generation algorithm is below some threshold. The quantum information properties of quantum mechanics guarantee that the resulting key is secure [4] unless the laws of physics are broken!

The basic concept of the QKD is based on the observation that during the transmission of qubits from Alice to Bob, no information gain is acquired by Eve without altering the qubit state. Along with the *no cloning theorem* [14] which proves that Alices qubit cannot be copied by Eve, there is also a proposition which states that information gain implies disturbance. [4] These two concepts are used when the non orthogonal states of qubits are transmitted between Alice and Bob, and also help in establishing an upper bound on eavesdropping and noise occurred in the channel [10].

The basic steps involve in any QKD protocol are the key generation algorithm and the classical post processing phase. The key generation algorithm of the QKD protocol is done using a quantum and a classical communication channel for transmitting qubits and classical bits, respectively. A secret shared key is obtained by both Alice and Bob at the end of the key generation algorithm. Because of the inherent noisy nature of quantum channels, the keys generated at both ends after classical reconciliation can still be wrong. This defeats the purpose of key sharing. To overcome this adversities classical post processing is used. In this phase the keys generated at the two ends are processed via an error correcting scheme to potentially reach a common key without any public sharing of the data.

2.3 BB84 Protocol

This section contains the description of the procedure of the BB84 QKD protocol.[1] The circuit diagram for the simple BB84 QKD protocol can be seen in the fig 2.3.

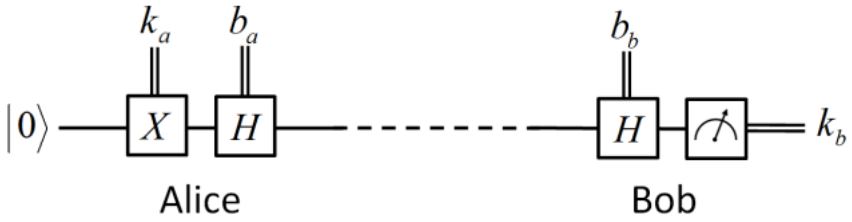

Figure 2.3: Circuit diagram for Simple BB84 QKD protocol

The basic steps for the BB84 QKD protocols are as follow:

1. Alice selects a random data string D of length 4n.
2. Alice selects another random sting B of length 4n.
3. Alice encoded D_i bit of D at i^{th} position in Z basis if B_i is 0 else encode in X basis, for $1 \leq i \leq 4n$
4. Alice sends those encoded qubits to Bob
5. Bob acknowledge publicly that he has received the qubits.
6. Bob measures the qubits randomly on Z or X basis.
7. Alice announces B, and now Bob knows the basis of Alice.
8. Both match their basis and remove the bits of mismatch basis.
9. If the length of the remaining bits is less than $2n$, they abort the protocol.

10. Alice selects n bit position on $2n$ bits and tells the positions to Bob.
11. Both of them check the n bits at those selected n positions.
12. If they match less than a certain threshold, then Eve is trying to intercept, and they abort the protocol else continue.
13. Now, the remaining n bits are sent to the classical post processing phase.
14. A final bit string key of length $m < n$ is obtained.
15. This is Alica and Bob's shared secret key.

Alice's bit	0	1	1	0	1	0	1	1	0	0	1
Alice's basis	+	+	X	+	X	X	X	+	X	X	+
Alice's measurement	↑	→	↖	↑	↖	↗	↖	→	↗	↗	→
Bob's basis	+	X	X	X	+	X	+	+	X	+	+
Bob's measurement	↑	↗	↖	↗	→	↗	→	→	↗	→	→
Shared Secret key	0		1			0		1	0		1

Figure 2.4: Key Generation by Simple BB84

Basis	0	1
Z or +	↑	→
X or ×	↗	↖

Figure 2.5: Label for Basis X and Basis Z

Using fig 2.4, we will see an example of a simple BB84 QKD protocol, which is as follow:

1. Alice chooses random data bits 0 1 1 0 1 0 1 1 0 0 1.
2. Alice's basis are encoded as + + X + X X X + X X +.

3. Measurement results of Alice is given.

4. Bob randomly chooses basis as + X X X + X + + X + +.

5. Measurement results of Bob is given.

6. Resultant key is 0 1 0 1 0 1

2.4 QKD Classical Post Processing

This section discusses the classical post-processing done after the QKD verify that there is no interaction with Eve during transmission, i.e., the check bits do not mismatch up to some error threshold value.

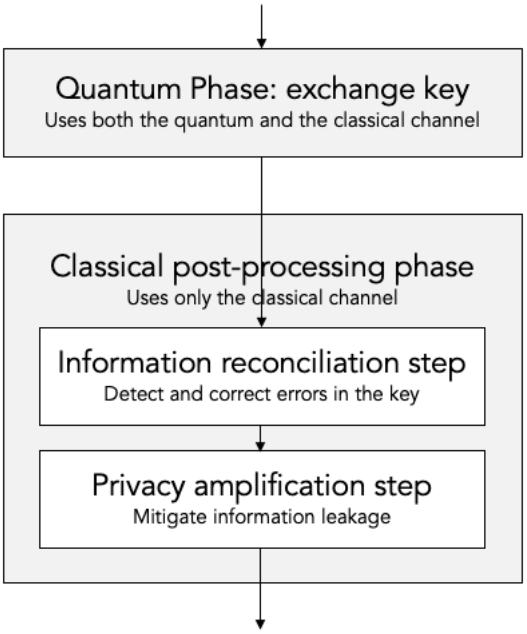

Figure 2.6: Classical Post Processing Phase

Information reconciliation and *privacy amplification* are the two steps used in the classical post processing phase of QKD protocol, as shown in fig-

ure 2.6. These steps will increase the correlation between Alice and Bob keys and reduce the mutual information possessed by Eve during transmission.

Let Alice and Bob have shared random key strings X and Y and the upper bound on mutual information related to X and Y possessed by Eve, then :

Information reconciliation is the process of error correction done to mitigate errors between X and Y on a public communication channel. This step will yield a shared key string W. At the end of this step, suppose Eve got a random variable Z having a partial correlation with W. Privacy amplification will then help Alice and Bob for the distillation from W to get a smaller bits set S whose correlation with Z will be less than the desired threshold. [10]

To accomplish privacy amplification one can use the class of *universal hash functions* G that maps the set of n bit strings A to the set of m bit strings B, such that when g is chosen uniformly at random from G and for any distinct a_1 and $a_2 \in A$ then the probability $g(a_1) = g(a_2)$ is the most $1/|B|$. [2]

We can perform both information reconciliation using classical codes to correct up to t errors. Consider an example where A random string X of length n is chosen by Alice and send to Bob. Bob gets Y such that Y = X + e, where e is some bit error. Let us assume that we know the error rate of a communication channel below some value t. Then we can employ the decoding mechanism of that classical code on X and Y, which will yield another string W and V respectively, such that W = V. The same procedure is used in information reconciliation for error correction.

3

Error Correcting Code

An error correcting code or ECC is widely used in the field of computer science, coding theory, telecommunication, and information theory to detect or correct errors in the data sent over noisy communication channels.[7] The basic concept is that the message being sent is encoded along with some redundant data specified by the ECC. Using the redundant data, up to a certain amount of errors can be detected and corrected by the receiver in the message during transmission.

This will provide the advantage of saving time for retransmission of the message. It provides benefits in the situation where retransmission is not only expensive but also sometimes impossible, such as multicasting among large users. It differs from error detection in that it can also correct errors along with detection. There is a disadvantage of adding extra information to the message, which is an overhead and requires higher bandwidth for forwarding the message. Error correcting code is widely used in satellite communication, storage devices, and primary memory.

The number of errors detected or corrected in a message depends upon the ECC code's design. Thus the effectiveness of various ECC codes varies upon different conditions. If we add more redundant information to the message, it can correct more errors but will require more bandwidth. Claude Shannon's coding theorem [11] over a noisy channel is used to determine

the maximum bandwidth that can be achieved for a given error probability, which also provides a theoretical bound for maximum transfer rate over a channel for some given noise level.

The procedure of adding redundancy to a message is dependent upon the ECC algorithm. The message's bits may also be used to design a function to produce redundant bits. The encoded information may contain the message bit without any changes or in some form of hidden message.

The ECC codes the mainly divided into the block and convolutional codes.

Block Codes are of fixed size, also called packets which are predetermined by the ECC algorithm. Some of its examples include RS codes widely used in CDs, DVDs, and HDDs. There are also BCH, Golay, and Hamming Codes.

Convolutional codes can be seen as bitstream of arbitrary length, but they can also be used as block codes. Some examples include Viterbi, BCJR, or MAP, which are used in analog signal processing.

Code rate can be defined as the ratio of the number of message bits to the number of encoded bits, i.e., message and redundant bits. Strong codes have a low code rate, i.e., approximate to zero, due to large redundancy and better efficiency, while weak codes have high code rates approximate to 1. One must compromise between data rate [3] and reliability during transmission due to the fact that redundant bits and message bits use the same resources over the communication channel. Strong codes have better reliability but at the cost of fewer data rates while opposite for weak codes.

3.1 (3,1) Repetition Code

It is an Error Correcting Code in which each message bit is transmitted three times. On receiver side we get an encoded message of 8 possible combinations

of 3 bits, as shown in the table below :

Codeword	Message
000	0
001	0
010	0
100	0
110	1
101	1
011	1
111	1

Table 3.1: (3,1) Repetition Code

For decoding, majority voting is used, i.e., if the majority is of 1's, it will be encoded as one; otherwise, it will be encoded as zero. Even though it is simple and can be implemented easily, it is relatively considered an inefficient error correcting code.

3.2 Binary BCH

BCH codes are those cyclic codes that are constructed by selecting the roots of their generator polynomials:

A BCH code of $d_{min} \leq 2t_d + 1$ is a cyclic code with generator polynomial $\bar{g}(x)$ having $2t_d$ consecutive roots $\alpha^b, \alpha^{b+1}, .., \alpha^{b+2t_d-1}$ [9]

Therefore, a binary BCH (n, k, d_{min}) code has a generator polynomial

$$\bar{g}(x) = LCM[\phi_b(x), \phi_{b+1}(x), ..., \phi_{b+2t_d-1}(x)],$$

where ϕ_i is a minimal polynomial, length $n = LCM[n_b, n_{b+1}, ..., n_{b+2td1}]$, and dimension $k = n - deg[\bar{g}(x)]$. A binary BCH code has a predetermined minimum distance of $2t_d + 1$.

Example 3.2.1 With $GF(2^3)$, $p(x) = x3 + x + 1$, $t_d = 1$ and $b = 1$, the polynomial

$$\bar{g}(x) = LCM[\phi_1(x), \phi_2(x)] = x^3 + x + 1,$$

generates a binary BCH (7,4,3) code where $GF(2^3)$ is a *Golay Field Function*. Observe that the hamming weight of $\bar{g}(x)$ is equal to 3.

3.2.1 General Decoding of BCH codes [9]

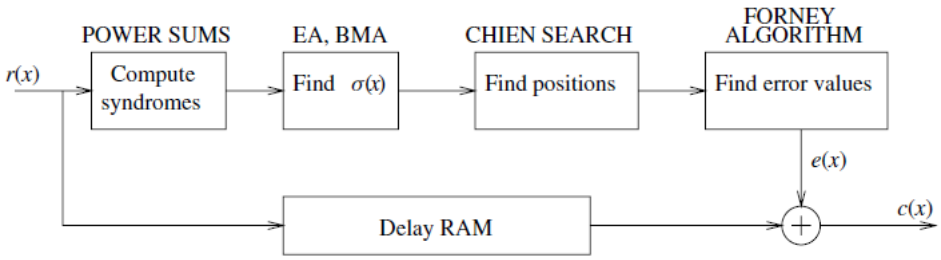

Figure 3.1: Binary BCH Decoder

Fig 3.1 is the block diagram of a decoder for BCH codes. The decoder follows the following tasks:

- Compute the syndromes, by evaluating the received polynomial at the zeros of the code

$$S_i =\triangleq \bar{r}(\alpha^i), \qquad i = b, b+1, ..., b+2t_d-1 \tag{3.1}$$

- Find the coefficients of the error-locator polynomial $\sigma(x)$.
- Find the coefficients of the error-locator polynomial $\sigma(x)$.
- Find the values of the errors $e_{j_1}, ..., e_{j_v}$.
- Correct the received word with the error locations, and values found.

One of the advantages of introducing $GF(2^m)$ arithmetic is that the decoding operations can be implemented with relatively simple circuit elements. Fig 3.2 shows the hardware implementation of how the syndrome S_j can be computed. The multiplier is over $GF(2^m)$, and by using simple combinatorial logic, we can construct it.

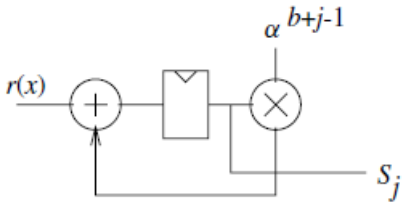

Figure 3.2: Circuit for computing Syndrome

3.3 Golay(23,12,7) code

Golay used

$$\sum_{i=0}^{3} \binom{23}{i} = 2^{11} \tag{3.2}$$

This equation 3.2 shows a binary code (23,12,7) can exist with t=3, i.e., it can correct at most all three errors that occurred in any 23 bit positions. He also gave a generator matrix for this code. Due to relatively small in size can use look-up tables (LUTs) for encoding and decoding purpose.

3.3.1 Encoding

Golay code uses LUT based encoding, which contains a list of all the $2^{12} = 4096$ code words, which are indexed using the data. Let $\bar{u}$ denote a 12-bit binary data that need to be encoded, and let $\bar{v}$ denote the corresponding

23-bit code word. The construction of encoder LUT involves the generation of all the 4096 12-bit binary data and computation of the syndrome of a pattern for which the 12 MSB equal to the data bits and the 11 LSB equal to zero. The 11-bit syndrome is then taken as LSB of the code word. The LUT follows one-to-one mapping from $\bar{u}$ onto $\bar{v}$, that can be written as

$$\bar{v} = LUT(\bar{u}) = (\bar{u}, get\_syndrome(\bar{u}, \bar{0})). \tag{3.3}$$

By taking the advantage of the cyclic nature of the Golay code for the constructing encoder LUT . Its generator polynomial1 is

$$g(x) = x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + 1$$

This polynomial generates the syndrome in the procedure get syndrome in equation 3.3 above.

3.3.2 Decoding

The decoders task is to estimate the most likely error vector $\bar{e}$ from the received vector $\bar{r}$. The decoder for the Golay code is based on an LUT that accepts as input the syndrome $\bar{s}$ of the received vector $\bar{r}$, and outputs the error vector $\bar{e}$.

The procedure to construct the decoder LUT is as follows:

1. Generate all possible error patterns $\bar{e}$ of Hamming weight less than or equal to three

2. For each error pattern, compute the corresponding syndrome $\bar{s}$ = get syndrome($\bar{e}$)

3. Store at location $\bar{s}$ of the LUT, the error vector $\bar{e}$,

$$LUT(\bar{s}) = \bar{e}$$

With the decoder LUT, upon reception of a corrupted received word r, the correction of up to three bit errors is accomplished by the following:

$$\hat{v} = \bar{r} \oplus LUT(get\_syndrome(\bar{r}))$$

where $\hat{v}$ denotes the corrected word.

3.4 Viterbi Code

As a dynamic programming algorithm, the Viterbi algorithm [13] is used to get the maximum posterior probability estimate of the Viterbi path, giving the sequence of observed events by utilising Hidden Markov models (HMM).

This algorithm is widely used in decoding convolutional codes used in various technology related to cellular communication, modems, satellite communication, Wi-fi, speech recognition and synthesis, bioinformatics, and linguistics.

It generalises an algorithm known as max-sum is used to search a subset of most likely hidden variables in various models like Markov random fields and Bayesian network. These hidden variables are connected in the same way to an HMM but with less connection among variables with some linear structure. It generalises algorithm is similar to the forward-backward algorithm.

There also exists the Lazy Viterbi algorithm [8] which much faster than the original decoder. It maintained a priority list of nodes for evaluation.

Compared to check every node in the original algorithm, this requires fewer checks. However, it isn't easy to implement in hardware to support parallelism.

3.4.1 Decoding

Let $S_i^{(k)}$ denotes a state at stage i with each stage having a *metric*, $M(S_i^{(k)})$ and a path $\bar{y}^{(k)}$, where k is total possible combination of memory m.

Let the coded bits be denoted by

$$\bar{v}[i] = (v_0[i]v_1[i]...v_{n-1}[i])$$

and let denote the output by

$$\bar{r}[i] = (r_0[i]r_1[i]...r_{n-1}[i])$$

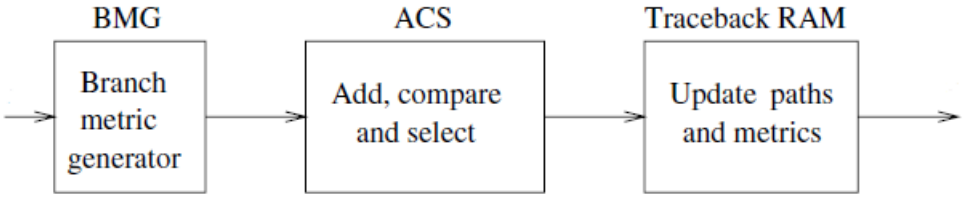

Figure 3.3: Viterbi Decoder Block Diagram

The figure 3.3 is a simple block diagram for a viterbi decoder. Basic decoding steps are as follows :

Initialisation : Set stage i metric state of stage i to zero and its path empty.

1. Branch Metric Computation : Calculate the partial branch metrics at stage i

2. Add, Compare and Select : For each $S_i^{(k)}$, k = 0,1,...,$2^m - 1$ and its incoming branch states, compare the branch metrics and winning branch have smallest path metric, then update the metric.

3. Update path memory: For each $S_i^{(k)}$, k = 0,1,...,$2^m - 1$, update paths with previous state path and winning branch.

4. Decode: If i is less than decoding depth, then output the path sequence of State $S_i^{(k)}$ having smallest metric. Increment i by 1 and repeat from step 1.

4

Our Experiments On The Classical Post Processing Phase

This chapter contains the experiment done for the proposed problem definition in the section 1.1.

As we discussed in the earlier chapter that the QKD classical post processing phase involves information reconciliation and privacy amplification. In simple terms, information reconciliation involves an error correction mechanism and privacy amplification involves hash functions to mitigate bit errors. We will consider information reconciliation part in our experiment and use error correcting coding schemes to implement it.

We have performed the following two experiments :

- Determining the efficiency of various error correcting coding schemes on different given error rates.
- Observing the behaviour of coding schemes when used as a part of classical post processing of QKD protocol.

For the first part, we have used four block code error correcting coding schemes and one convolutional error correcting scheme which are as follow :

- (3,1) Repetition Coding Scheme
- (31,28,5) Binary BCH Coding Scheme

- (48,36,5) Binary BCH Coding Scheme
- (23,12,7) Golay Coding Scheme
- (3,7,5) Viterbi Codin Scheme

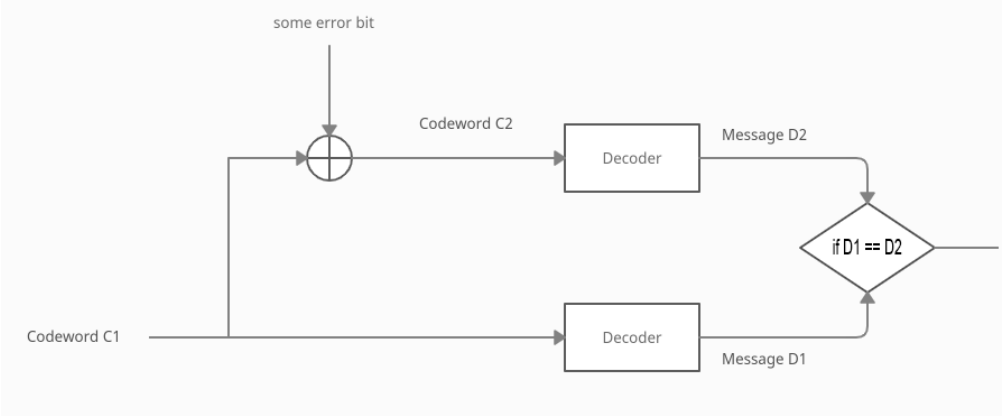

Figure 4.1: Block Diagram for a Single Step Process

The fig 4.1 represents the basic block diagram for a single step done in experiment 1. We introduce some error bits for a randomly generated codeword $C1$ according to the given error rate getting codeword $C2$. Both of them are passed as an input to the decoder to get messages $D1$ and $D2$, respectively. If both messages are the same, then we count it as success or else failure.

In our experiment, we use hundred (100) codewords and flip some bits among them according to predefined error rates to mimic the bit error introduced during transmission. Then we check all codewords' messages with new codewords' messages after decoding (using a particular error correcting code decoding scheme) and count the number of matches. This process is repeated over one thousand times(1000). Finally, we get the efficiency of that error correcting code decoding scheme for a given error rate by taking the average number of matches.

Let's take an example that we take Golay Coding Scheme. Its codeword length is 23 bits. For 100 codewords, it requires 2300 bits. Suppose for an error rate of 0.05, we flip 115 bits among those codewords and get 100 new codewords. Then we decode both old and new codewords and finally get the count of how many messages match. This process is repeated 1000 times, and the average matching rate is given, which is 45.887% in our example.

In our experiment, we have worked on the error rate of [0.05 , 0.08, 0.1, 0.15, 0.20]. This experiment is done on the c++ programming language. The codes for random generation of codewords and introducing some errors according to the given above error rate can be found in the appendix section.

	0.05	0.08	0.1	0.15	0.2
(3,1) Repetition	93.19	92.45	86.70	82.64	74.558
(31,28,5) Binary BCH	27.915	13.571	8.405	2.54	0.945
(48,36,5) Binary BCH	11.012	3.296	1.471	0.175	0.017
(23,12,7) Golay	45.887	30.383	23.43	12.345	6.383
(3,7,5) Viterbi	85.163	77.462	73.31	63.138	55.439

Table 4.1: Efficiency of different coding schemes on different error rate

Table 4.1 shows efficiency of error correcting coding schemes metioned above, after performing the experiment. The graph below will help us to understand it even better.

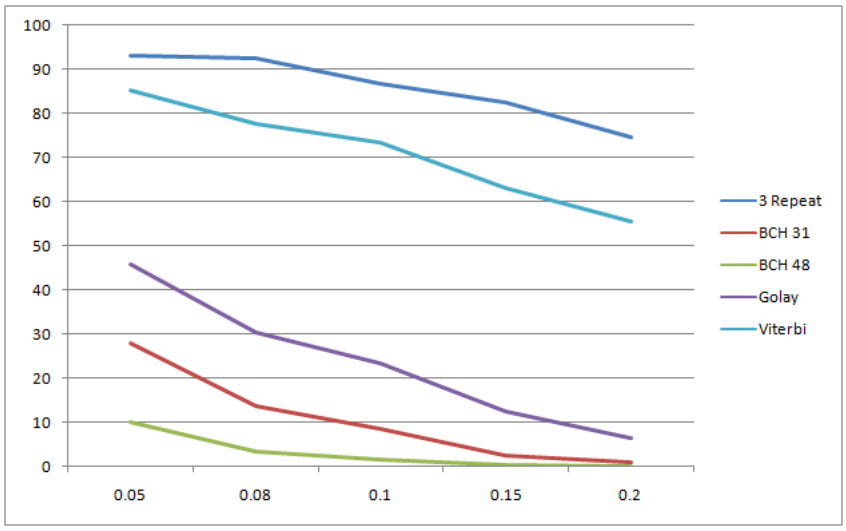

Figure 4.2: Error Coding Scheme Efficiency

For the second experiment, we have used the quantum qiskit library to implement the BB84 protocol. This experiment used the keys generated in the BB84 protocol as codeword $C1$ for Alice key and codeword $C2$ for Bob key. These codewords are then given as input to the experiment conducted above. This second experiment is different from the above experiment as here codewords are generated from the output of BB84 protocol, whereas they were random the earlier. The error rate is fixed and specified in the earlier experiment, but it is entirely random here.

To implement this experiment, we have followed the following procedure :

1. Implement the BB84 using the qiskit library in the python programming language (available in the Program section of the appendix).

2. Run the algorithm up to a certain number of times, say 10000 and save the keys generated by Alice and Bob.

3. Calculate the Quantum Bit Error Rate (QBER).

4. Use the saved keys as input to error correcting coding schemes and get

the final keys.

This experiment is done on a local machine and yet to be done on an IBMQ machine. But what we found that there is an average of 0.47 to 0.53 QBER when Eve is acting in between protocol, no QBER in the absence of Eve when no noise model is employed, and less than 0.30 or sometimes 0.50 QBER when a noise model is employed in the protocol.

5

Conclusions and future work

In this thesis, we have performed two experiments. The first one is related to the error correcting codes, and the second is related to BB84 classical post processing.

From the table 4.1, we can see that even though (3,1) repetition error correcting code is showing the best result. The resultant message bit of (3,1) repetition error correcting code depends upon 1 out of 8 codewords of (3,1) repetition error correcting code and the error rate in (3,1) repetition error correcting code. The low efficiency in the Binary BCH codes is because the parity bit in the BCH codes is non zero, since we are considering the codeword to be random, so any slight change in parity bit due to noise will decrease the chances of codewords referring to the same message after correction. Golay has performed better than BCH as it can correct up to 3 bits compared to 2 bits in BCH. Viterbi being a convolutional code has performed better than most of code block error correcting code.

We can also see that those codes having smaller message bits size perform better than those with larger message bit lengths. Further study of more complex error correcting codes like CSS, Turbo code, and LDPC codes can be done. They are said to better in terms of accuracy. [16][6]

In the second experiment, further study needs to be done upon the random behavior of Quantum Bit Error.

References

[1] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. *Proceedings of IEEE International Conference on Computers, Systems,and Signal Processing*, 1984.

[2] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal Of Computer And System Sciences*, 1978.

[3] Pramod David; Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, UK, 2005.

[4] Christopher A. Fuchs. Information gain vs. state disturbance in quantum theory. *PhysComp96*, 1996.

[5] Louis Salvail Gilles Brassard. Secret-key reconciliation by public discussion. *Eurocrypt*, 1993.

[6] Predrag Rapajic Grace Oletu. The performance of turbo codes for wireless communication systems. *International Conference on Computer Research and Development*, 2011.

[7] Richard Wesley Hamming. Error detecting and error correcting code. *Bell System Technical Journal*, 1950.

[8] Matteo Frigo Jon Feldman, Ibrahim Abou-Faycal. A fast maximum-likelihood decoder for convolutional codes. *Vehicular Technology Conference*, 2002.

[9] Robert H. Morelos-Zaragoza. *The Art of Error Correcting Coding 2nd Edition*. Wiley, 2006.

[10] Michael Nielsen and Isaac Chuang. *Quantum Computing and Quantum Information*. Cambridge University Press, 2000.

[11] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[12] Guifr Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 2003.

[13] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 1967.

[14] Wojciech Wootters, William; Zurek. A single quantum cannot be cloned. *Nature*, 1982.

[15] H F Chau Xiongfeng Ma, Jean-Christian Boileau. Practical post-processing for quantum-key-distribution experiments. *Computers and Security Volume 30, Issue 4*, 2011.

[16] Shiyong Zhang Zongjie Tu. Overview of ldpc codes. *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, 2007.

Appendix

Programs

Listing 1: BB84 protocol (Uncomment Eve interception to include Eve)

```
#Note: Uncomment print function to see the keys in the process
#importing the required libraries

from qiskit import QuantumCircuit, execute, Aer
from numpy.random import randint
import time
import numpy as np

#intiliasing the seed for randomisation process
np.random.seed((int)(time.time()))
#setting length of qubits
n = 100

#function to encode data bits according to bases
def encode(bits, bases):
    msg = []
    for i in range(n):
        Quan_Cir = QuantumCircuit(1,1)
        #Define Qiskit Circuit input line
        if bases[i] == 0: # Z-basis is used
            if bits[i] == 0:
                pass
            else:
                Quan_Cir.x(0)  # Applying X gate
        else: # X-basis
            if bits[i] == 0:
                Quan_Cir.h(0)
                # Applying Hadamard gate
            else:
                Quan_Cir.x(0)
                Quan_Cir.h(0)
        Quan_Cir.barrier()
        msg.append(Quan_Cir)
```

```
    return msg     #returning the codeword or encoded message

#Function for measurment given message bits and bases
def measure_msg(msg, bases):
    backend = Aer.get_backend('qasm_simulator')
    #selecting which simulator to use
    meter = []
    for i in range(n):
        if bases[i] == 0: # Z-basis measurement
            msg[i].measure(0,0)
        if bases[i] == 1: # X-basis measurement
            msg[i].h(0)
            msg[i].measure(0,0)
        result = execute(msg[i], backend, shots=1, memory=True)
            .result()
        measured_bit = int(result.get_memory()[0])
        meter.append(measured_bit)
    return meter   #return measurement results

#function to remove mismatch bits
def rem_extra(a_b, b_b, bits):
    g_b = []
    for q in range(n):
        if a_b[q] == b_b[q]:
            g_b.append(bits[q])
    return g_b

#function to compare check bits
def bit_sample(bits, select):
    smp = []
    for j in select:
        j = np.mod(j, len(bits))
        smp.append(bits.pop(j))
    return smp

#The files will store Alice and Bob keys along with their
   length
f1 = open("Alice.txt", "w")
f2 = open("Bob.txt", "w")

#Loop for getting Ten thousand keys
for loop in range(10000):

    # Alice bits selected randomly
    Ali_B = randint(2, size=n)
    #print(Ali_B)

    #Alice Base slected randomly
    A_base = randint(2, size=n)
    #print(A_base)
```

```
    msg = encode(Ali_B, A_base)

    ##uncomment for eve interception
    #eve_bases = randint(2, size=n)
    #eve_msg = measure_msg(msg, eve_bases)
    #print(eve_msg)

    #Bob bases selected randomly
    B_base = randint(2, size=n)
    #print(B_base)

    #Measurement is done by Bob
    bob_res = measure_msg(msg, B_base)
    #print(bob_res)

    # Process of removing mismatch bit is done here
    a_key = rem_extra(A_base, B_base, Ali_B)
    #print(a_key)

    b_key = rem_extra(A_base, B_base, bob_res)
    #print(b_key)

    f1.write(str(len(a_key))+ " ")
    #storing the keys in file for further process
    for i in a_key:
        #print(i,end ="")
        f1.write(str(i))

    f1.write("\n")

    f2.write(str(len(b_key))+ " ")

    for i in b_key:
       # print(i,end ="")
        f2.write(str(i))

    f2.write("\n")

    #process of comparing check bits is done here
    smp_sz = 15
    select_bit = randint(n, size=smp_sz)
    B_sample = bit_sample(b_key, select_bit)
    A_sample = bit_sample(a_key, select_bit)

f1.close()
f2.close()
# priting whether eve is present or not.
if B_sample != A_sample:
    print("Eve is detected ")
else:
```

```
    print("No Eve is present")
```

Listing 2: Program to Calculate QBER from BB84 keys

```
//Header files
#include<bits/stdc++.h>

using namespace std;

int main()
{
 //opening files containg keys generated by QKD Functions
 ifstream f1,f2;
    f1.open("Alice.txt");
 f2.open("Bob.txt");

 //For storing the length of keys
 int l1=0,l2=0;
 char a,b;

 //counter
 unsigned long counter = 0;
 //For error rate
 float rate=0;
 while(f1 || f2)
 {
   counter =0;
   f1>> l1;
   f2>> l2;
  for(int i=0;i<l1;i++)
  {
   f1>>a;
   f2>>b;
   if(a!=b)
    counter++;
   //cout<<a<<" ";
  }
  rate = (float)counter/l1;
  cout<<counter<<" : "<<rate<<endl;
 }
 f1.close();
 f2.close();
 return 0;
}
```

Listing 3: 3 bit Repetition Code

```
//Heder files
#include<bits/stdc++.h>

using namespace std;

//encoding
vector<int> encode(int A)
{
  A = A%2;
  vector<int> B={A,A,A};
  return B;
}

//decoding
int decode(vector<int> A)
{

  if(A[0]+A[1]+A[2]>1)
    return 0;
  else
   return 1;
}


int main(){


 int m_l = 100;
 int c_l = 300;

 //error rates
 vector<float> err = {0.05,0.08,0.1,0.15,0.2};

 //message bits
 std::vector<int> m(100);
 std::vector<int> m_d(100);

 //codeword bits
 std::vector<int> c(300);
 std::vector<int> c_d(300);


 std::vector<int> v;
 for(int i=0;i<c_l;i++)
```

```
 v.push_back(i);
std::mt19937 g(std::time(nullptr));

//Runiing for different error rate
for(int k=0;k<err.size();k++)
{
 unsigned long long count =0,frame=0;
 int err_len = ceil(c_l*err[k]);

 for(unsigned long long loop=0;loop<1000;loop++)
 {

 //genrating random codewords
 for(int i=0;i<c_l;i++)
 {
  c[i]= (rand()%2);
 }

  c_d = c;

  std::shuffle(v.begin(), v.end(), g);

  //introducing error
  for(int i=0;i<err_len;i++)
  {
   c_d[v[i]] = c_d[v[i]] ^ 1;
  }

  //decoding both the codeword
  for(int i=0;i<m_l;i++)
  {
   vector<int> ans1(3),ans(3);
   ans[0]=c[i*3];
   ans[1]=c[(i*3)+1];
   ans[2]=c[(i*3)+2];
   ans1[0]=c_d[i*3];
   ans1[1]=c_d[(i*3)+1];
   ans1[2]=c_d[(i*3)+2];
   m[i]=decode(ans);
   m_d[i]=decode(ans1);
   if(m[i]==m_d[i])
    frame++;

  }
  //uncomment to see the messages
  /*cout<<"loop "<<loop<<"\n";
  for(int i=0;i<m_l;i++)
   cout<<m[i];
  cout<<endl;
  for(int i=0;i<m_l;i++)
```

```
    cout<<m_d[i];
   cout<<endl;*/
   if(m==m_d)
    count++;
  }
  cout<<err[k]*100<<" : count = "<<count<<" , frame = "<<(float
     )frame/1000<<endl;
 }
 return 0;
}
```

Listing 4: Generate Codewords having different error rate of length 7

```
//file used to genreate randomy codewords along with error
   induced codewords as per given error rate and length
//header files
#include <math.h>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <random>
#include <ctime>

using namespace std;

int length = 7; //<<--- change it for different length

int main()
{
 int i;

 srandom(std::time(nullptr));

 //files to save Codewords
 ofstream f1,f2;
 f1.open("0.2A.txt"); //<<-- for original codeword
 f2.open("0.2B.txt");//<<-- for error induced codewords

 vector<int> ecn(100); //error code no
 vector <vector<int>> epos(100);
 //error rates
 vector<float> per_error = {0.05,0.08,0.1,0.15,0.2};
 int el =4;// <-- which error rate to use

 long long frame =0;
 long long avg_frame =0;
```

```
std::vector<int> v;
for(int i=0;i<700;i++) // length of codewords * number of
    codewords
 v.push_back(i);
std::mt19937 g(std::time(nullptr));

 unsigned long long count =0;

// for repeating the process 1000 times
 for(long long lp =0;lp<1000;lp++)
      {

std::shuffle(v.begin(), v.end(), g);

for(int i=0;i<100;i++)
 epos[i] = vector<int>(0);

for(int e_pos=0;e_pos<100;e_pos++)
  ecn[e_pos]=0;

unsigned long err_len = ceil(700*per_error[el]);

//Choosing the position to introduce the error among 100
    codewords
for(unsigned long lim=0;lim<err_len;lim++)
{

   int tn = (v[lim])/7;
   int rloc = (v[lim])%7;
  // cout<<rloc<<" ";
   ecn[tn]++;
   epos[tn].push_back(rloc);
}

for(int loop=0;loop<100;loop++){

int cd[7];//vector<int> cd(48);
int cd2[7];//vector<int> cd2(48);

/* Randomly generate DATA */
for (i = 0; i < length; i++)
{
 cd[i] = (random()%2);
 cd2[i] = cd[i];
}

//introducing the error
numerr=ecn[loop];
//cout<<loop<<" : "<<numerr<<" ";
```

```
 for (i = 0; i < numerr; i++)
 {
  errpos[i] = epos[loop][i];
 // cout<<errpos[i]<<" ";
  cd2[errpos[i]] ^= 1;
 }
 //cout<<endl;

 for (i = 0; i < length; i++)
 {
  f1<<cd[i];
  f2<<cd2[i];
 }

 f1<<endl;
 f2<<endl;
 }
}
 f1.close();
 f2.close();
    return 0;
}
```

Listing 5: Compare decoded message given by Viterbi

```
//header files
#include <math.h>
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <random>
#include <ctime>

using namespace std;

//command line argument to read file names for comparing keys
int main(int argc, char** argv)
{
 int i;

 srandom(std::time(nullptr));

 ifstream f1,f2;
 f1.open(argv[1]);
 f2.open(argv[2]);
```

```
 int n1 ,n2;
 unsigned long count=0;
 while(f1)
 {
  f1>>n1;
  f2>>n2;
  if(n1 == n2) //<<--comparing the keys
   count++;
 }
 cout<< ((float)(count-1)/1000); //<<-- change accoring to
    number of times experiment is done
 f1.close();
 f2.close();
    return 0;
}
```