



UNIVERSITY OF NAIROBI

**IMPLEMENTATION OF QUANTUM KEY DISTRIBUTION
WITH MULTIPLE EAVESDROPPERS**

BY

MUTIA MUEKE LILIAN

I56/40597/2021

**A Project Thesis Submitted in Partial Fulfillment of the Requirements for Award of the Degree
of Master of Science in Physics**

DEPARTMENT OF PHYSICS

FACULTY OF SCIENCE AND TECHNOLOGY

UNIVERSITY OF NAIROBI

July, 2023

UNIVERSITY OF NAIROBI

Declaration of Originality Form

This form must be completed and signed for all works submitted to the University for Examination.

Name of student: Mutia Mueke Lilian

Registration Number: I56/40597/2021

Faculty/School/Institute: Faculty of Science and Technology

Department: Physics

Course Name: SPH 6207: Project

Title of the work

IMPLEMENTATION OF QUANTUM KEY
DISTRIBUTION WITH MULTIPLE
EAVESDROPPERS

DECLARATION


1. I understand what Plagiarism is and I am aware of the University's policy in this regard.
2. I declare that this Thesis (Thesis, project, essay, assignment, paper, report etc) is my original work and has not been submitted elsewhere for examination, award of a degree or publication. Where other people's work, or my own work has been used, this has properly been acknowledged and referenced in accordance with the University of Nairobi's requirements.
3. I have not sought or used the services of any professional agencies to produce this work
4. I have not allowed, and shall not allow anyone to copy my work with the intention of passing it off as his/her own work
5. I understand that any false claim in respect of this work shall result in disciplinary action in accordance with University Plagiarism Policy.

Signature: 

Date: 31/07/2023

DECLARATION

I declare that this project thesis is my own original work and has not been submitted elsewhere for purposes of examination, publication, or award of a degree. Where other people's work or my own work has been used, this has properly been acknowledged and referenced in accordance with the University of Nairobi's requirements.

Signature  Date 27/07/2023

Mutia Mueke Lilian

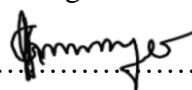
I56/40597/2022

Department of Physics

Faculty of Science and Technology

University of Nairobi

This proposal is submitted for registration with our approval as research supervisors:

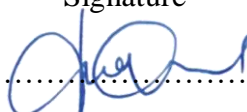
	Signature	Date
Dr. Geoffrey O. Okeng'o		01/08/2023

Department of Physics, University of Nairobi

P. O Box 30197-00100

Nairobi, Kenya

gokengo@uonbi.ac.ke

	Signature	Date
Prof. Andrew. M. Kahonge		04-Aug-2023

Department of Computer Science & Informatics, University of Nairobi

P. O Box 30197-00100

Nairobi, Kenya

andrew.mwaura@uonbi.ac.ke

Signature

Date

Dr. John B. Awuor

.....

.....August 3, 2023

Department of Physics, University of Nairobi

P. O Box 30197-00100

Nairobi, Kenya

buers@uonbi.ac.ke

DEDICATION

This work is dedicated to my parents Boniface and Burnice Mutia, my siblings Dennis Kitili and Davies Mutua, and my friends and colleagues for their support and encouragement during my studies.

ACKNOWLEDGEMENT

I wish to express my heartfelt appreciation to the University of Nairobi's entire administration as well as the Department of Physics for their assistance and availability of the resources required to complete this project. I would also like to thank IBM for the provision of the Qiskit software that was used to generate data for this research. to my supervisors, thank you for continuously encouraging, guiding, and advising me. Finally, I am grateful to God Almighty for keeping me in perfect health throughout the course of this research.

ABSTRACT

Quantum Key Distribution (QKD), a type of quantum cryptography in which two parties generate and share a highly secure secret key, is a novel computational technique that can ensure high data security in modern quantum computers. It is based on the laws of quantum mechanics such as superposition, the Heisenberg Uncertainty Principle (HUP), and the No-Cloning theorem, as opposed to integer factorization and discrete logarithmic problems used in conventional classic computers. Although proposed in the 1970's by Stephen Wiesner at Columbia University via introduction of the idea of quantum conjugate coding, and further improved on by Charles H. Bennett through his concept of secure communication, the very first successful attempt to implement this technique was in the 1980's when Charles H. Bennett proposed the first quantum cryptography protocol – the BB84 based on nonorthogonal states. At around the same time Artur Ekert proposed a QKD method based on the idea of quantum entanglement. Since then, there has been remarkable progress in research aimed at the adoption of technologies to ensure highly secure quantum computers with the most notable work being the 2021 study by Hamish Johnston on implementation of the Beijing-Shanghai QKD network over 4600 km. However, despite the wide ongoing adoption of QKD technologies in commercial applications, research on new technologies to ensure high data rates and data security is still at its infancy. This research work is an effort in that direction. Data obtained through simulations implemented using the publicly available IBM's Qiskit (see <https://www.ibm.com/quantum>) is used to study, test and implement a BB84-based QKD protocol for the case of multiple eavesdroppers, using the Intercept-Resend (IR) attack on a secure QKD system. Associated Quantum Bit Error Rates (QBERs) for the cases of (i) no eavesdropper, (ii) single eavesdropper and (iii) multiple eavesdroppers are computed and presented. Our results show that the QBERs are independent of the number of eavesdroppers, hence providing evidence that our QKD system remains secure even with an increased number of eavesdroppers.

TABLE OF CONTENTS

DECLARATION	ii
DEDICATION	iv
ACKNOWLEDGEMENT	v
ABSTRACT	vi
TABLE OF CONTENTS	vii
LIST OF ABBREVIATIONS	ix
LIST OF FIGURES	x
LIST OF TABLES	xi
CHAPTER ONE	1
INTRODUCTION	1
1.1 Research background	1
1.2 Statement of the problem	4
1.3 Research objectives	4
1.3.1 Main objective	4
1.4 Justification and significance of the study	5
CHAPTER 2	6
LITERATURE REVIEW	6
2.1 Quantum Key Distribution	6
2.2 Simulation of QKD Protocols	7
2.3 Simulation of QKD protocols with multiple eavesdroppers	9
CHAPTER 3	11
THEORETICAL INPUT	11
3.1 The BB84 protocol	11
3.2 Quantum Bits	12
3.3 Measurement Bases	13
3.4 Creating Superposition	14
3.5 Channels	15
3.6 Detection of Eavesdropping in the BB84 Protocol	15
CHAPTER 4	17
RESEARCH METHODOLOGY	17
4.1 Introduction	17
4.2 Research Design	17
4.3 Research and Implementation Approach	18

4.3.1 Research Approach	18
4.3.2 Implementation Approach	18
4.3.2.1 Implementation without an eavesdropper	18
4.3.2.2 Implementation with eavesdroppers	19
CHAPTER 5	20
RESULTS AND DISCUSSIONS	20
5.1 Introduction	20
5.2 Implementation without an eavesdropper	20
5.3 Implementation with an eavesdropper	22
5.3 Implementation with multiple eavesdroppers	23
CHAPTER 6	26
CONCLUSIONS AND RECOMMENDATIONS	26
6.1 Conclusions	26
6.2 Recommendations	26
REFERENCES	27

LIST OF ABBREVIATIONS

QC	Quantum Cryptography
QKD	Quantum Key Distribution
H gate	Hadamard gate
NS-3	Network Simulator-3
IBM	International Business Machines
HUP	Heisenberg Uncertainty Principle
QBER	Quantum Error Bit Rate
FPR	False Positive Ratio
FNR	False Negative Ratio
IR	Intercept-Resend
RSA	Rivest-Shamir-Adleman
QB	Quantum Block
BER	Bit Error Ratio

LIST OF FIGURES

Figure 1: The BB84 protocol.....	11
Figure 2: Polarization bases and polarization states	13
Figure 3a: Creating superposition from the $ 0\rangle$ state. This shows the equal probability of obtain either basis state.....	14
Figure 3b: Creating superposition from the $ 1\rangle$ state. This shows the equal probability of obtaining either basis state.....	15
Figure 4: Simulation results for the BB84 protocol without an eavesdropper	20
Figure 5: Circuit diagrams for qubits 3, 7 and 13 (indexing starts at 0)	21
Figure 6: Randomness of Alice's and Bob's measurement bases	21
Fig. 7: Data obtained from the simulation of the BB84 protocol with an eavesdropper	22
Fig. 8: Circuit diagrams for bits at indices 0, 1, and 2.....	22
Fig. 9: Comparison of the measurement bases of Alice, Eve and Bob.....	23
Fig. 10: Alice's and Bob's sifted keys, and samples of the sifted keys used to calculate the QBER.	23
Fig. 11: The BB84 protocol with ten eavesdroppers.....	24
Fig. 12: A graph of the number of eavesdroppers vs. the average QBER.....	25

LIST OF TABLES

Table 1: An illustration of the BB84 protocol.....	11
Table 2: QBERs obtained for different numbers of eavesdroppers.	25

CHAPTER ONE

INTRODUCTION

1.1 Research background

The inability of classical computers to efficiently simulate some effects of quantum physics was identified by the American theoretical physicist Richard Feynman in 1982 (Feynman, 1982). It was then speculated that computers which employed these effects could be built to enhance computation (Rieffel & Polak, 2000). This led to the invention of quantum computers. They solve issues that are complex substantially quicker than conventional computers by employing quantum physics principles like superposition and entanglement (Rietsche et al., 2022). A common misconception is that quantum computers are a better version of classical computers. However, they do have an advantage in problems that require too much searching or testing for regular computers to do, problems that require secure encryption, as well as problems that involve simulating quantum mechanical systems. An example is quantum communication.

Quantum communication involves transmitting quantum states between two parties (Gisin & Thew, 2007). One fundamental justification is the fact that quantum states encode quantum information, allowing one to conduct operations that are nearly impossible to carry out using classical information (Gisin & Thew, 2007). Quantum communication guarantees unconditional security of quantum information, theoretically (Chen, 2021). This is because it employs Quantum Key Distribution, a major aspect of quantum cryptography, to distribute cryptographic symmetric keys between two distant parties (Mehic et al., 2021).

Cryptography is the technique of protecting information against a third party (Pooja & Renuka, 2016). Classical cryptography is divided into two types: symmetric and asymmetric (Elboukhari et al., 2010). Symmetric cryptography, commonly referred to as secret key cryptography, encrypts and decrypts data using a single secret key (Elboukhari et al., 2010). Although this encryption scheme is faster and uses less resources, it is old and less secure than asymmetric cryptography as multiple people possess the same key. Additionally, it is shared via an insecure classical channel.

A public key and a secret key, related by a mathematical function, are used in asymmetric cryptography, commonly referred to as public key cryptography. Encryption is performed using

the public key, while decryption is performed using the private key (Elboukhari et al., 2010). The security of these schemes is based on mathematical complexity (Elboukhari et al., 2010). The RSA algorithm, which relies on the factorization of a semiprime (product of two prime numbers), is an example of this scheme. However, because of the creation of the Shor algorithm which enables faster factorization using a quantum computer by Peter Shor in 1994, the security of this algorithm is compromised (Elampari & Ramakrishnan, 2016).

Quantum cryptography (QC) is possibly the fastest-growing field in quantum information science (Pirandola et al., 2020). A well-known example of QC is QKD. It involves the creation and distribution of a secure secret key between two parties against an eavesdropper's presence, which is then used to encrypt and decrypt data (Lee et al., 2022). To secure tamper-proof shared keys, remote parties agree on secret shared keys that alert the original parties if an adversary tampers with them during transmission (Adu-Kyere et al., 2022). It manipulates photons using the laws of quantum physics such as superposition, the no-cloning theorem, which states that a polarized photon cannot be duplicated since its quantum state is unknown, quantum indeterminacy and quantum entanglement to create a secure key for safe end-to-end communication (Adu-Kyere et al., 2022; Sidhu et al., 2021; Wootters & Zurek, 1982). There are three approaches to QKD: discrete variable coding, continuous variable coding, and distributed phase difference coding, with the former being the most applied. Additionally, QKD protocols are grouped depending on the principles of quantum mechanics employed. These are the Entanglement-Based protocol and the Prepare-and-Measure protocol (Gyongyosi et al., 2019; Haitjema, 2007).

The Entanglement-Based protocol relies on quantum entanglement. Within a quantum system, entangled particles correlatively spin in one direction (Adu-Kyere et al., 2022). Measuring one part of an entangled pair affects the other. Therefore, interfering with an entangled system affects the overall system, making eavesdropping easily detectable. To generate a secret shared key, the sender and receiver receive photons from an entangled pair (Ekert, 1991). Published in 1991 by Artur Ekert, the E91 protocol is an example of the Entanglement-Based protocol. It relies heavily on correlation – Alice and Bob should get the same result after measuring photons from an entangled pair. According to Gisin and Thew, this protocol takes place as follows; laser light converted using a crystal is used to produce entangled photon pairs (Gisin & Thew, 2007). Alice and Bob receive a photon from each pair. Upon measurement of these photons, Alice and Bob should get the same result for each pair if they use the same measurement basis. They

discard photons where different measurement bases were used and retain the rest. The remaining photons are then used to produce a shared secret key.

In 1969, Stephen Wiesner asserted that the uncertainty principle could be used in cryptography (Wiesner, 1983). In collaboration with Gilles Brassard and Charles Bennett, they founded the first QKD protocol, the BB84 protocol, in 1984 (C. Bennett & Brassard, 1984). It relies on the HUP which states that one cannot know both states of a conjugate pair in a quantum system with absolute certainty, and quantum indeterminacy, which is the inability to completely describe a physical system (Elboukhari et al., 2010; Haitjema, 2007). In this protocol, two channels are used to connect the sender and the receiver: a unidirectional quantum channel and a bidirectional classical channel (Elboukhari et al., 2010; Gyongyosi et al., 2019; Ruiz-Alba et al., 2010). Alice randomly chooses the bits she wants to transmit to Bob. To encode these bits into qubits, she uses a randomly selected measurement basis – it could be rectilinear or diagonal (Lee et al., 2022). A pre-agreed encoding rule is used to encode the bits. For example, 0 and 1 can be encoded by a qubit with horizontal and vertical polarization in the rectangular basis respectively, and 45° and 135° polarizations in the diagonal basis can represent 0 and 1 respectively (Lee et al., 2022). The polarization state of the photon is prepared by taking into consideration the bit value and the measurement basis chosen. Alice proceeds to send these photons one at a time through the quantum channel to Bob, keeping record of the state, basis, and time that each photon is transmitted. She keeps her measurement bases a secret (Lee et al., 2022). Upon receiving these photons, Bob randomly picks one of the two measurement bases for each photon, and measures the polarization state. He too, keeps record of the time each photon is received, the basis he chooses, and the measurement result. He communicates with Alice over the bidirectional classical channel about the choice of basis for each photon, producing a raw key (Ruiz-Alba et al., 2010). If the measurement bases for a particular qubit are the same, Bob decodes the original binary bit (Lee et al., 2022). Alice and Bob retain bits where the same basis was used and discard the rest. The retained bits form the sifted key (Ruiz-Alba et al., 2010). An IR attack from an eavesdropper introduces quantum bit errors which announce her presence (Lee et al., 2022). To reduce the probability that an eavesdropper, Eve, has gained information from the system, additional calculations and error-correcting techniques are performed on the classical bit string obtained.

1.2 Statement of the problem

Classical encryption schemes rely on the complexity of mathematical problems to safeguard information. However, with the development of quantum computers, these encryption schemes are becoming easier to break. Additionally, detection of an eavesdropper's presence is challenging. QKD, on the other hand, allows easier detection of an eavesdropper since the eavesdropper must measure the quantum state to get information, which is equivalent to her announcing her presence. Although it is considered the most secure means of safeguarding information, some of its aspects remain unstudied, such as the effect of multiple eavesdroppers on a QKD protocol. Implementing one of QKD's protocols for the case of multiple eavesdroppers with limited literature to date is important in providing useful insights into the associated QBERs and hence the security of the BB84-based QKD protocol.

1.3 Research question

What is the effect of multiple eavesdroppers on a QKD system?

1.4 Research objectives

1.4.1 Main objective

To implement and study a BB84-based QKD protocol for the case of multiple eavesdroppers using the Python3 in IBM's Qiskit.

1.4.2 Specific objectives

The specific objectives are:

- i. To model and simulate the a BB84-based QKD protocol without the presence of an eavesdropper.
- ii. To model and simulate the BB84-based protocol with the presence of a single eavesdropper.
- iii. To simulate and model the BB84 protocol with the presence of multiple eavesdroppers.

1.5 Justification and significance of the study

QKD-based protocols form a major important component in ongoing research aimed at designed highly secure quantum communication systems. Not only shall this guarantee increased data speeds and security but shall also drive the future of ongoing efforts to fully realize the benefits of quantum communication. With knowledge in this field being still at its nascent stage, there remains more work to be done towards the implementation, testing and applications of highly secure quantum communication systems. Therefore, this research work is geared towards simulation and implementation of a BB84-based protocol with multiple eavesdroppers and analysing the results obtained. As a sidelobe, this work shall also contribute towards creating awareness on QKD as an alternative to traditional encryption schemes due to its guaranteed higher security, enabling the science community, and the public in general, to gain broader knowledge in the interesting area.

CHAPTER 2

LITERATURE REVIEW

Despite being a relatively new discipline, there has been significant advancement in QKD during the past few years. Simulations of some of the protocols have shown that it is easy to detect an eavesdropper, although that might not be the case during implementation on the ground. These simulations have also proven theory to be true, in terms of the number of times Bob is able to obtain the correct result, and an eavesdropper's effect on the information being transmitted.

2.1 Quantum Key Distribution

Quantum cryptography provides a cryptographic solution that is impenetrable by means of quantum mechanics. Unlike classical cryptography where information is encoded in bits, qubits such as photons are used in quantum cryptography. This has resulted in the development of various QKD protocols, with some being implemented. These include BB84, B92, six-state (SSP), and SARG04. The B92 is a modified version of BB84. Instead of four states, it has two states, 0^0 and 45^0 . SSP uses three diagonal bases to encode the bits. Instead of announcing Alice announcing her bases, she announces a pair of nonorthogonal states in SARG04. Some QKD networks that have been developed and are still in use include DARPA (Massachusetts, USA), Tokyo QKD network, and Hub and Spoke network (Padamvathi et al., 2016).

Lee et al. (2022) analyse the False Positive Ratio (FPR) and False Negative Ratio (FNR) upper bounds utilizing Hoeffding's inequality to ascertain the correlation of quantum resources and the accuracy of eavesdropping detection. They also propose a new version of the BB84 protocol, group-BB84, a new iteration of the BB84 protocol that addresses the issue of quickly changing quantum channel conditions. An optimal combinatory algorithm used in the group-BB84 shows higher accuracy in detecting an eavesdropper compared to the algorithm applied to the classical BB84 protocol. Through several simulations, they show that enough quantum resources are required to improve the accuracy of detection of an eavesdropper.

Elboukhari et al. (2010) compare classical cryptography and quantum cryptography and state that security problems arising in classical cryptography can be solved through quantum cryptography. The authors discuss at length some QKD protocols such as the BB84, the B92

and the EPR protocols. Their advantages and disadvantages are mentioned in addition to their applicability.

V & V (2021) describe a new protocol, the Enhanced BB84 Quantum Cryptography Protocol (EBB84QCP), which is beneficial for data encryption in wireless body sensor networks for medical applications. It is similar to the classical BB84 protocol except it performs a bitwise operation between the matched bits of Alice and Bob and the unmatched bits of Alice to produce a secure secret key, making it attack-proof.

Shor and Preskill (2000) suggest an elementary proof of the BB84 QKD protocol's security. This proof relates the security of BB84 to quantum error correction codes and entanglement purification. Demonstration that a key distribution mechanism whose basis is entanglement purification is secure implies the security of the BB84 protocol. Recent proofs of security of QKD protocols are also discussed mentioning their advantages and disadvantages. (C. Bennett & Brassard, 1984) proof is easy to understand but requires the use of a quantum computer while (Lo & Chau, 1999) and (Mayers, 2001) prove the security of a BB84 based protocol but are complicated.

Ruiz-Alba et al. (2010) propose a new scheme based on subcarrier multiplexing which enables parallel QKD and discuss an experimental demonstration of the same. Parallel QKD increases the bit rate, which is a major challenge of QKD. To show the theoretical ability of this technique to increase the key rate, the subcarrier multiplexing QKD systems which use N subcarriers to encode N parallel keys independently has been discussed. The aim is to make use of available optical communication resources and multiplexing techniques to implement practical QKD systems. A major challenge faced in this proposed protocol is the filtering of radio frequency sidebands in the optical domain at the output of Bob's modulation. However, the system allows Alice to send different bits using two or more independently encoded different radio-frequency tones. The theoretical predictions agree with the experimental results.

2.2 Simulation of QKD Protocols

Sharifi et al. (2007) explain how cryptography can benefit from quantum mechanics, examine a BB84 variation that is allegedly more effective., and simulate BB84 and its improved version to validate the above claim. In the improved version, the two parties choose the measurement bases with different probabilities, and estimate two error rates, as opposed to the classical BB84. Through the simulation, it is evident that any eavesdropping that can be detected in the

BB84 can also be detected in the improved version, and the efficiency of BB84 can also be doubled without affecting its security.

Kohnle & Rizzoli (2017) came up with engaging simulations to demonstrate QKD and its operating principles and support the learning and teaching of QKD. They made assumptions such as true single-photon sources and perfect detectors, and that only the IR attack was used by the eavesdropper.

Chatterjee et al. (2019) designed a QKD simulation toolkit, qkdSim, that considered the imperfections that could be found experimentally. They were able to accurately simulate the B92 protocol using a prototype of qkdSim. However, this toolkit is not applicable to Entanglement-Based protocols. They hope to develop it in a way that can be used with any other QKD protocol.

Shajahan & Nair (2020) explained how to ensure a flawless and secure transfer of data by exploiting physics laws that could be simulated in a classical communication channel. They also implemented the BB84 protocol and provided a hybrid approach, which combined traditional key generation, encryption-decryption and the BB84 protocol.

Jasim et al. (2015) describe how quantum mechanics principles are used to aid in the encryption and decryption process. Furthermore, they simulate the BB84 protocol, and implement it using two modes: with and without interference. Through this, they show that QKD can be paired with different applications and generate a key for said applications. Additionally, they establish that QKD is adversely affected by the higher rate of authentication cost.

Mina & Simion (2021) published a paper entitled where they simulated the BB84 protocol using IBM's Qiskit, both with and without eavesdropping. They were able to prove that Bob's probability of choosing the correct measurement basis is 50% without the presence of an eavesdropper. Therefore, only about half of the original bits produced by Alice are discarded. Additionally, they were able to show with absolute certainty that if enough bits are used, the presence of an eavesdropper can be discovered. However, they assumed perfect sources and channels during the simulation.

Praveen Kumar et al. (2022) analysed the efficiency of QKD by simulating it using NS-3 (Network Simulator-3), which has an inbuilt module for creating the quantum channel. Through this, they were able to show the overall data loss among network nodes. Their research encompasses challenges and important attributes of QKD protocols, and their implementation.

Adu-Kyere et al. (2022) implemented the BB84 protocol without and with an eavesdropper using the IR attack. They used Python3 to conduct their research and were able to determine that an eavesdropper is detectable if the error rate introduced is higher than the error threshold, 0.11. They also designed a communication architecture model that uses QC to ensure communication is secure. This architecture model consists of three quantum blocks (QB), where quantum processes such as base generation, encoding/ decoding, and key generation take place. Alice's and Bob's QBs contain a photon base generator, a photon-based encoder/ decoder, and a key generator. Data filtration and rectification takes place in the key generator. Through the results obtained, they were able to show the randomness of Alice's, Bob's, and Eve's polarization states. They were also able to detect an eavesdropper's presence through comparison of the parameters obtained with and without the presence of an eavesdropper. They recorded an error rate of 0.04296875 in the absence of an eavesdropper. Since it was lower than the error threshold, it was because of system errors. On the other hand, an error rate of 0.125 was obtained in the presence of an eavesdropper, which is greater than the threshold, indicating the presence of an eavesdropper.

2. 3 Simulation of QKD protocols with multiple eavesdroppers

Dehmani et al., (2012) simulates the BB84 protocol with many IR attacks through a depolarizing channel, a model for quantum noise in quantum systems. Through this research, they establish that with an increase in the depolarizing parameter, the QBER decreases. A decrease in the depolarizing parameter is because of an increase in the number of attacks. If the depolarizing parameter is less than 0.165, there exists a threshold below which the information is secure, else it is insecure. If the depolarizing parameter is greater than or equal to 0.165, regardless of the number of attacks, the information is not secure. Additionally, the authors prove that the QBER decreases when the number of attacks and/ or the depolarizing parameter is increased.

Elampari & Ramakrishnan, (2016) use the quantum package for Mathematica to analyze the bit error ratio (BER) of a QKD system with two eavesdroppers. They highlight challenges faced during QKD implementation, putting an emphasis on the lack of a true single photon source. During the simulation, they assume that the attack used is opaque eavesdropping, an example of which is the Trojan horse attack. According to their research, if Alice's source does not

produce single photons, the state that altered by one eavesdropper may be modified back by the second eavesdropper, without Bob's knowledge. Simulation results produce a lower BER in the presence of multiple eavesdroppers as compared to a single eavesdropper, or no eavesdropper. Therefore, a lower BER indicates the possibility of multiple eavesdroppers. The BER is the total number of mismatched bits over by the number of bits transmitted during that time. This research is contradictory as the BER generated in the absence of an eavesdropper should be the lowest.

CHAPTER 3

THEORETICAL INPUT

3.1 The BB84 protocol

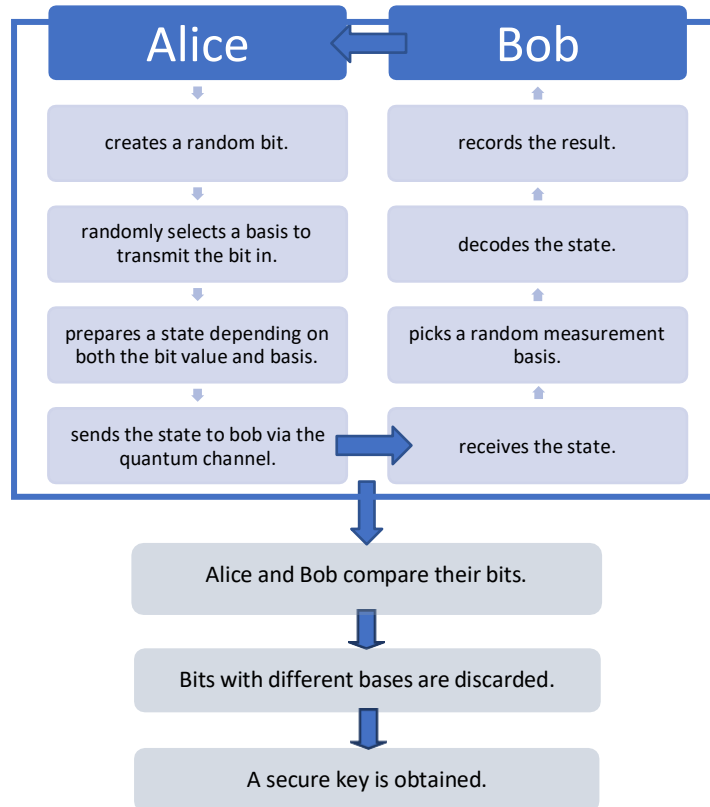


Figure 1: The BB84 protocol.

Information security in the BB84 protocol is ensured via HUP and the no-cloning theorem. (Elboukhari et al., 2010; Haitjema, 2007). The parties agree on how the classical bits will be encoded using qubits beforehand. In the protocol, Alice randomly generates a string of classical bits and picks measurement bases. The measurement bases used in this protocol are the rectilinear and diagonal bases. She encodes the classical bits using this polarization bases into qubits and transmits them to the receiver through the quantum channel. Upon receiving the qubits, Bob picks his own measurement bases at random, with no knowledge of Alice's encoding bases, and performs measurement on the qubits. Alice and Bob keep record of the bits they encode and decode, the measurement basis chosen for each bit, and the time the bit is send and received (Mina & Simion, 2021).

In the reconciliation stage, the parties liase over the classical channel the bases chosen for the raw key. They discard bits encoded and decoded using different bases and retain the rest to form the sifted key (Mina & Simion, 2021). *Table 1* shows an illustration of the protocol.

Alice's bit string	1	1	1	1	0	0	1	0	1
Alice's bases	+	x	x	x	+	x	x	x	+
Alice's polarization states	t	↖	↖	↖	→	↗	↖	↗	t
Bob's bases	X	x	x	x	+	+	x	+	x
Bob's polarization states	↖	↖	↖	↖	→	t	↖	→	t
Raw key	1	1	1	1	0	1	1	0	1
Comparison of bases	N	Y	Y	Y	Y	N	Y	N	N
Sifted key		1	1	1	0		1		

Table 1: An illustration of the BB84 protocol.

In the illustration, Alice and Bob chose the same measurement bases five times.

3.2 Quantum Bits

In QKD, information is encoded in the quantum bit, which is the physical state of a particle. (Lee et al., 2022). A quantum bit, also referred to as a qubit, is the quantum equivalent of a classical bit. It is the fundamental unit of QC (Padamvathi et al., 2016). Unlike a classical bit whose bit value is either 0 or 1, a qubit can exist in a state of superposition. The basis states for qubits are $|0\rangle$ and $|1\rangle$. A superposition is a linear combination of $|0\rangle$ and $|1\rangle$ (Padamvathi et al., 2016) and is expressed as

$$a|0\rangle + b|1\rangle$$

where a and b are amplitudes, which are complex numbers. The modulus squared of a and b gives the probability of getting the $|0\rangle$ and $|1\rangle$ states, respectively. That is, the probability of getting $|0\rangle$ is $|a|^2$, and the probability of getting state $|1\rangle$ is $|b|^2$, upon measurement (Padamvathi et al., 2016). The sum of the square of these magnitudes should be equal to 1. Although qubits exist in a superposition of states, each qubit only contains a classical bit's worth of information. This is because after measurement, the qubit collapses to classical bit.

During practical implementations of QKD systems, a qubit may experience errors such as generation of multiple photons in a pulse, attenuation in a fibre, and dark counts at a photon detector (Inoue, 2006; Pirandola et al., 2020).

3.3 Measurement Bases

Generally, there are three measurement bases – the rectilinear basis (0° and 90°), the diagonal basis (45° and 135°), and the circular basis (left and right-handedness) (C. H. Bennett & Brassard, 2014; Wiesner, 1983). In the BB84 protocol, the bases used are the rectilinear and diagonal bases. The two bases are conjugate to each other, meaning that if a state is measured in a basis different from the basis it was encoded in, the result of the measurement could return either state equally (Mina & Simion, 2021). The two states within each basis are orthogonal.

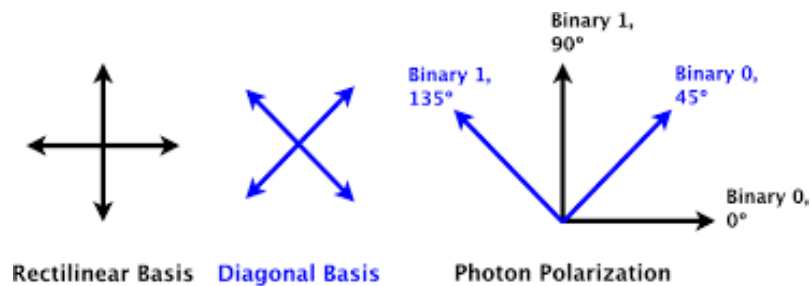


Figure 2: Polarization bases and polarization states.

In the implementation of this protocol using Qiskit, a Hadamard gate is applied to random qubits to create superposition. This implies that some qubits will be in a state of superposition ($|+\rangle, |-\rangle$) (diagonal basis) with (Elboukhari et al., 2010),

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad (1)$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (2)$$

The rest of the qubits will be in states of the standard measurement basis ($|0\rangle, |1\rangle$) (rectilinear basis). If Bob applies a Hadamard gate to a qubit that was prepared in a superposition state, he takes it out of superposition and upon measurement gets the correct result. Otherwise, he will get either state with a fifty percent probability.

3.4 Creating Superposition

As per Qiskit's settings, the qubits are always in state $|0\rangle$. To create superposition in the $|+\rangle$ state, the Hadamard gate (H gate) is applied to the $|0\rangle$ state. In matrix notation,

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = |+\rangle \quad (3)$$

For state $|1\rangle$, application of the H-gate creates superposition in the $|-\rangle$ state.

To create superposition in the $|-\rangle$ state, an X gate is applied followed by an H gate, to convert the qubit from the $|0\rangle$ state to the $|1\rangle$ state and finally to the $|-\rangle$. In matrix notation (Padamvathi et al., 2016),

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = |-\rangle \quad (4)$$

where H-gate = $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

Upon measurement, the outcome is as shown in Figure 3. The outcome being either $|0\rangle$ or $|1\rangle$ is almost equal showing that superposition has been created.

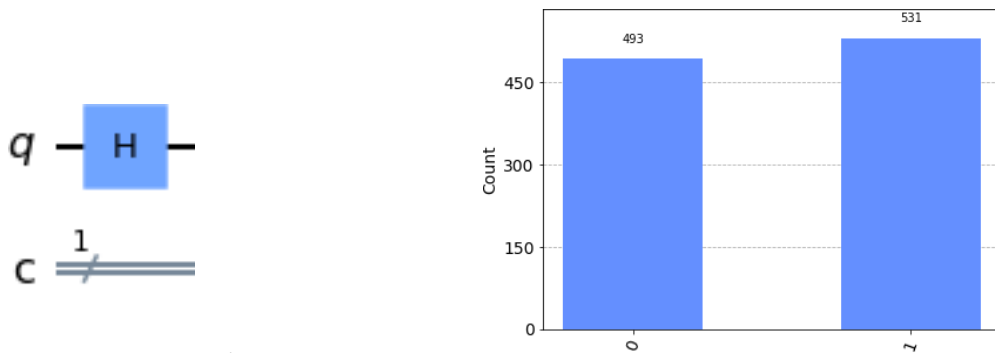


Figure 3a: Creating superposition from the $|0\rangle$ state. This shows the equal probability of obtain either basis state.

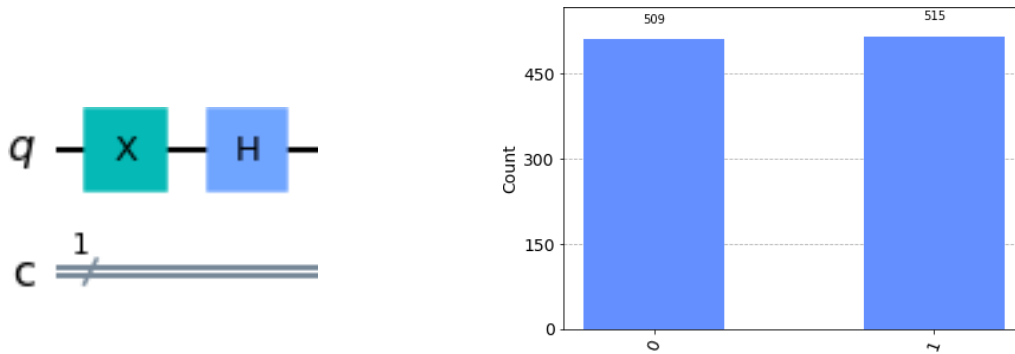


Figure 3b: Creating superposition from the $|1\rangle$ state. This shows the equal probability of obtaining either basis state.

3.5 Channels

The BB84 protocol makes use of two channels - a quantum channel and a public classical channel (Elboukhari et al., 2010; Lee et al., 2022; Ruiz-Alba et al., 2010). Qubits are shared using the quantum channel, which is susceptible to manipulation by a third party (Ruiz-Alba et al., 2010). The classical channel is used after the transmission process by the sender and receiver to compare the measurement bases chosen (Lee et al., 2022). The classical channel needs to be authenticated. That is, the message should not be modified during transit and the receiver needs to verify the message. Broadcast radio or the internet are examples of classical communication channels (Elboukhari et al., 2010). A quantum channel could either be an optical fibre or free space. Both optical fibres and free space have their advantages and disadvantages. Optical fibres are used mostly because their cost of implementation is low compared to free space. However, they are not scalable and have a high photon loss rate. Nine out of ten photons are lost for every fifty kilometres. On the other hand, although free space as a channel is scalable and offers a longer communication distance, its implementation is expensive. In the implementation through Qiskit, an ideal quantum channel where the only source of error is eavesdropping is assumed.

3.6 Detection of Eavesdropping in the BB84 Protocol

To detect eavesdropping, we need to calculate the QBER. If it is greater than a given threshold (0.11), eavesdropping has occurred (Lee et al., 2022). Under an ideal quantum channel where eavesdropping does not occur, the QBER must be measured as 0 (C. H. Bennett & Brassard, 2014; Elboukhari et al., 2010). To calculate the QBER, a sample of a specific length from the

sifted key is selected. As an example, let's assume Alice encoded N binary bits and sent them to Bob and upon comparison of the measurement bases and discarding bits with mismatched bases, the number of bits remaining is M . From these M bits, Bob shares the results of K bits. After comparing these bits with her own, Alice realizes that some bits do not match. The QBER is calculated as follows:

$$QBER = \frac{\text{No. of mismatched bits in } K}{K} \quad (5)$$

Even though the errors could be due to transmission processes or eavesdropping, they are all attributed to eavesdropping because it is impossible to distinguish between the two (Lee et al., 2022). An average QBER of 25% is measured under eavesdropping (Elboukhari et al., 2010; Lee et al., 2022), because errors occur when Bob and Eve's measurement bases differ. This implies that the probability of successful eavesdropping is 75% (Sharifi et al., 2007). Half the time, she chooses the wrong basis, and measuring the qubit with the wrong basis causes it to collapse to the wrong state half the time (Lee et al., 2022). The probability of detecting an eavesdropper after comparing K of their bits is:

$$1 - \left(\frac{3}{4}\right)^K \quad (6)$$

To guarantee a 99% accuracy of detecting an eavesdropper, equation (3) is used to determine the length of the sample, K to be chosen (Mina & Simion, 2021).

$$K = \frac{|M|}{3} \approx \frac{N}{6} \quad (7)$$

CHAPTER 4

RESEARCH METHODOLOGY

4.1 Introduction

The procedure and methodology used in this research work are presented in this chapter. The chapter is divided into two sections: research design, and research and implementation approach, which is further divided into two subsections; implementation without an eavesdropper, and implementation with *eavesdroppers*.

4.2 Research Design

The experimental research design was used in this research project. This helped establish a cause-and-effect relationship between the number of eavesdroppers and the QBER, which was used to determine the effect of multiple eavesdroppers on a QKD system.

Step 1: Defining variables.

The variables in this project are the number of eavesdroppers and the key produced. The number of eavesdroppers is the independent variable, while the QBER is the dependent variable.

Step 2: Hypothesis.

Null hypothesis: The QBER of an even number of eavesdroppers is lower as compared to that of an odd number of eavesdroppers.

Alternate hypothesis: An increase in the number of eavesdroppers has no effect on the QBER.

Step 3: Designing experimental treatments.

The maximum number of eavesdroppers used was 10. The control group consisted of only a single eavesdropper.

Step 4: Assigning independent variables to treatment groups.

After obtaining the QBER from the control group, we gradually increased the number of eavesdroppers. We then compared the QBER obtained from an even number of eavesdroppers and that from an odd number of eavesdroppers.

Step 5: Measuring the dependent variable.

The dependent variable is the QBER calculated from the raw key produced. To measure the effects of the eavesdroppers, we compared the QBER of the keys produced in each experimental group.

Since this is a controlled experiment, we were able to manipulate the independent variable (number of eavesdroppers) and measure the dependent variable (QBER).

4.3 Research and Implementation Approach

Although there exists other means of simulation and implementation of QKD protocols such as NS-3, implementation took place through IBM's *Qiskit*. It is a great avenue for making quantum simulations and the results are easily interpreted.

4.3.1 Research Approach

The simulation and implementation of the protocol both with and without the presence of an eavesdropper was done on IBM's *Qiskit* platform by using the QASM simulator, an IBM high-performance cloud simulator. The implementation made use of quasi-randomly generated bits instead of true random bits. However, the implementation showed that the generation of the key was practically the same and was able to prove that the key obtained in the absence of an eavesdropper was about half the number of original bits, which is the case theoretically. Furthermore, we tested the key's security and detect the presence of an eavesdropper.

4.3.2 Implementation Approach

To simulate the BB84 protocol in *Qiskit*, some of the classical bits were put into a state of superposition through the application of the Hadamard gate, or a combination of the X gate and the Hadamard gate, depending on the initial value of the bit.

4.3.2.1 Implementation without an eavesdropper

Implementation in the absence of an eavesdropper took place as follows:

Step 1: Alice generated a random string of classical bits.

Using the *random* module, Alice generated classical bits randomly and stored them in the variable *alice_bits*. Since bits are preset to $|0\rangle$ in IBM's *Qiskit*, she applied the *X gate* to bits with a bit value of '1' to convert them to state $|1\rangle$.

Step 2: Alice picked a random measurement basis for each bit.

To encode the qubits, Alice chose to either leave some of them in the standard basis or create superposition randomly by using the *random* module. To create superposition, she applied an *H gate* to the qubit. The measurement bases chosen were stored in the *alice_choices* string. Alice then sent the qubits to Bob keeping the measurement basis chosen for each qubit secret.

Step 3: Bob picked a measurement basis for each qubit.

With no knowledge of Alice's measurement basis, Bob picked a measurement basis for each qubit randomly using the *random* module and stored them in the *bob_choices* string.

Step 4: Bob measured the qubits.

To decode the qubits, Bob measured them using the *measure(q, c)* method where the qubit's measurement result *q*, was stored in a classical bit *c*. In cases where Bob used a different measurement basis to Alice for a particular bit, he had a 50% probability of decoding the correct bit value. He stored the measured values in the string *bob_bits*.

Step 5: Alice and Bob generate a key.

Alice and Bob compared their measurement bases. To generate a key, they discarded bits with different measurement bases. The sifted key, which is usually around half the number of bits in the original bitstring, is formed by the remaining bits.

To determine whether the key was secure, the QBER was calculated using a sample of the sifted key. In the absence of an eavesdropper and assuming perfect sources and channel, $QBER = 0$.

4.3.2.2 Implementation with eavesdroppers

Herein, Alice followed the first two steps as in the implementation without an eavesdropper: generating a string of classical bits randomly and picking a random measurement basis for each. If an eavesdropper intercepted the qubits, she measured each qubit using a randomly chosen measurement basis, stored in the variable *eve_choices*. Thereafter, she sent the measured bits, *eve_bits*, to the intended receiver, Bob, who then proceeded with steps 3-5 above.

In the presence of multiple eavesdroppers, the above procedure was repeated. In this study, the assumptions that the attacks occurred sequentially and that only the IR attack was used, are made.

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 Introduction

This chapter entails the results and discussions of this study. It is divided into three sections: (i) implementation of the QKD system without eavesdropping, (ii) implementation with one eavesdropper, and (iii) implementation with multiple eavesdroppers.

5.2 Implementation without an eavesdropper

The results for one of the simulations without an eavesdropper are shown in *Fig. 4*.

```
Alice's bits: 1111001010100111010001611010011010100110101011010001110000001110011011100101010001111001111100110110
Alice's choices: +xxx+xxx+x+xx+xx+x+x+x+xxxx+xxx+x+++++xx+xxxx+xx+x+xxx-x+xxxxxxx+++++xx+xx+x+++++xxx+x+xx+xxx+xx
Bob's choices: xx+xx++xx+xxxx+xx+xx+x+xxx+xxxxxxx+x++x+xx+xxx+x+xx++x+xxx++x+xx+x+xxx++x+xx++x+xx++x+xx++x+xx++x
A-B bases: -Y-Y----Y--YYYYYY-YY----YY----YYY---YY-YY-Y-Y-Y--Y--YY--YY--YY--YY--YYY-Y--Y-Y--YYY-Y-YY--Y-YYY
Bob's bits: 0111011010010111010000000101010100001111000110111010000100111011101110010110101100100011110111110
Alice's key: 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 1 1 1 1 1 0
Key match: Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
Bob's key: 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 1 1 1 1 1 0
Key: 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 0 1 1 1 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 1 1 1 1 1 0
Key length: 48
Basis match: 48
QBER = 0.0 %
```

Figure 4: Simulation results for the BB84 protocol without an eavesdropper.

The output *Alice's bits* shows the randomness with which Alice generates the bits, which is required for the protocol. *Alice's choices* and *Bob's choices* show the bases chosen for each bit. “+” was used to represent the rectilinear basis while “x” was used to represent the diagonal basis. *Bob's bits* shows the measurement results obtained for the above protocol.

To determine the authenticity of the protocol, we compared Alice and Bob's measurement bases. According to theory, Alice and Bob's measurement bases match about half the time. In this simulation, Alice and Bob chose the same basis 48 times for 100 bits, shown in Figure 1 as *Basis match*.

Additionally, circuit diagrams were drawn for randomly chosen bits. This is shown in Figure 5.

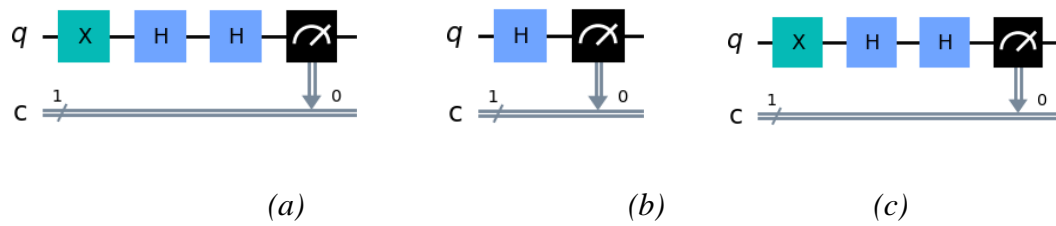


Figure 5: Circuit diagrams for qubits 3, 7 and 13 (indexing starts at 0).

In Fig. 5(a), the classical bit 1 is converted to a qubit by applying the X gate. Since Alice and Bob chose the diagonal basis, an H gate is applied in both instances. Upon measurement, Bob gets bit 1, which is the same as Alice’s bit. This is as expected since he chose the same measurement basis as Alice. In Fig. 5(b), Alice and Bob chose different measurement basis. Therefore, the probability of Bob’s measurement result being similar to Alice’s is 50%, as per the Heisenberg’s uncertainty principle.

To better understand the randomness of Alice’s and Bob’s choice of measurement bases, graphs were plotted as shown in Fig. 6. Polarization states (0^0 , 90^0) are as a result of the rectilinear basis (+) while polarization states (45^0 , 135^0) are as a result of the diagonal basis (x).

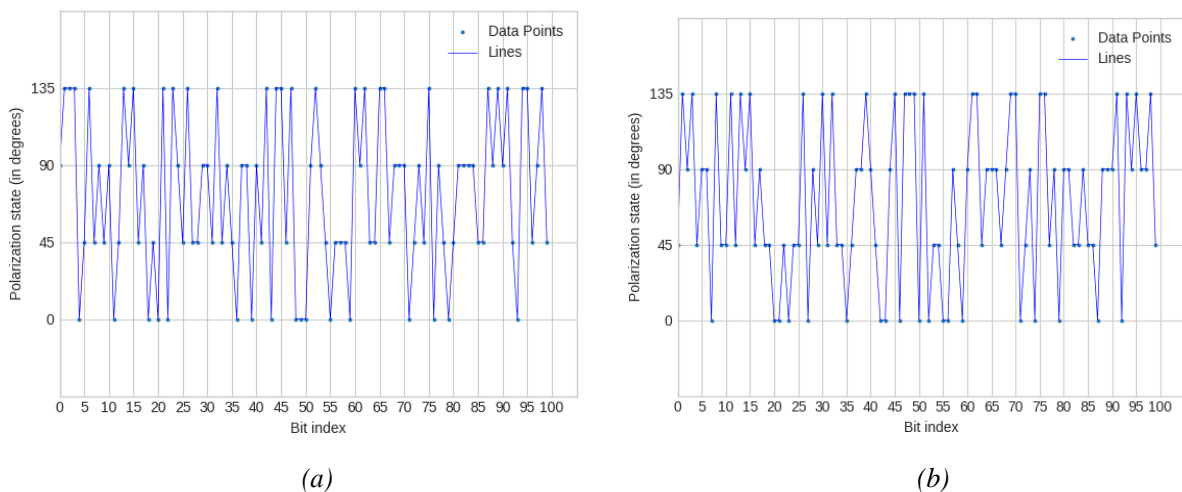


Figure 6: Randomness of Alice’s and Bob’s measurement bases.

After comparison of measurement basis and discarding bits where different basis were used, the sifted keys, represented by Alice’s key and Bob’s key in Fig. 1, were generated. A comparison

of the two keys showed a perfect match. The QBER, calculated using equation (5), was equal to 0. This is below the QBER threshold which is usually 11%, meaning that there was no eavesdropping activity. Therefore, the key generated was secure.

5.3 Implementation with an eavesdropper

The BB84 protocol was implemented in the presence of an eavesdropper. The results obtained are shown in Fig. 7.

```

Alice's bits: 1011010011010110011001001011101011000001001001001100010101101010110011000000011111110010001001011010
Alice's bases: +xxx+xxx+xxx+++x+xxx+++xxx+xx+xxxxx+xxxx+xx+++++x+xx+x+x+xxxxx+x+++xxxx+xxxx+xx+xxx+
Eve's bases:  +x+xx+xxx+++xxx+x+++x+++++xxxx+xx+xx+++xxx+xx+xxx+x+x+xxxxxx+xxxxxxxx+xxxx+xxx+xx+x+++x+
Eve's bits: 1001100010010000000010100101010000101001000011011000001111010001000100000001101111110001101110000
Bob's bases:  x+xxxx+x+x+xxxxx+++x+xxxx+xx+++xxxxx+++x+x+++++x+x+xx+x+xxx+x+++x+x+xx+xxx+xxxxx+++x+x
Bob's bits: 101011011101001000001101110001001111100101000101010001010100110111010101000011111110101010111110001

```

Figure 7: Data obtained from the simulation of the BB84 protocol with an eavesdropper.

To confirm that the system was effective, circuit diagrams were drawn for bits 0, 1, 2(bits are indexed from zero). The results are shown in Fig. 8. Two circuits are drawn for each bit. Alice and Eve’s circuit, and Eve and Bob’s circuit. This is because a qubit cannot be measured twice without collapsing it to a classical bit. Therefore, after Eve performs her measurement, she encodes it again before sending it to Bob.

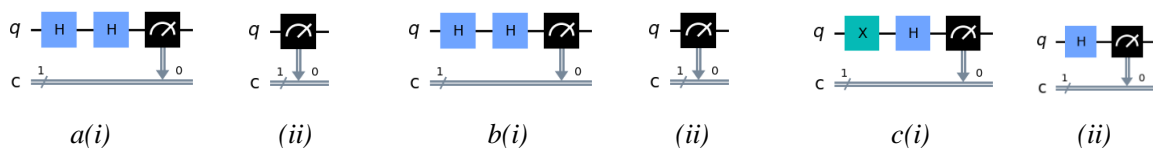


Figure 8: Circuit diagrams for bits at indices 0, 1, and 2.

A comparison was made between Alice’s and Eve’s bases, Eve’ and Bob’s bases, and Alice and Bob’s bases to show that the probability of one party picking the same measurement basis with respect to the second party is ≈ 0.5 . This is shown in Fig. 9. *A-E bases* represents the comparison of Alice and Eve’s bases, *E-B bases* is the comparison of Eve and Bob’s bases, and *A-B bases* is the comparison of Alice and Bob’s bases. Alice and Eve’s measurement bases matched a total of 42 times, Eve and Bob’s 47 times, and Alice and Bob’s 51 times.

```

Alice's bases: +xxx+xxx+xxx+++x+xxx+++xxx+xx+xxxxx+xxx+xx+++++x+xx+x+x+xxxxx+x+++xx+++x+++++xx+xxxx+xxx+xx+xxx+
A-E bases:     YY-Y--YY-----Y--Y--Y---Y-Y-Y---YYY--YYYYY-Y--Y--Y--Y-----YYY-YYYY-YY---Y-YYY-YY-Y-Y-Y
Eve's bases:  +x+xx+xxx+++xxx+x+++x+++++xxxxxxxx+xx+x+x+++++xxx+xx+xxx+x+x+xxxxx+xxxxxxxx+xxxxxxxx+xxx+xx+x+++x++
A-E basis match: 42

Eve's bases:  +x+xx+xxx+++xxx+x+++x+++++xxxxxxxx+xx+x+x+++++xxx+xx+xxx+x+x+xxxxx+xxxxxxxx+xxxxxxxx+xxx+xx+x+++x++
E-B bases:    ---YY--Y--Y-YYY--YY-----YY--YYY-YY--Y---YY---Y--YY---YYY-Y-YY-Y-YY--YY--Y-Y-Y--YY--YYYYY-
Bob's bases:  x+xxxx+x+x+xxxxx+++x+xxxx+xx+++xxxxx+xxx+x+++++x+xx+x+x+xxxxx+x+++x+xx+x+xxx+x+xxx+x+xx+xx+xxxxx+++x+x
E-B basis match: 47

Alice's bases: +xxx+xxx+xxx+++x+xxx+++xxx+xx+xxxxx+xxx+xx+++++x+xx+x+x+xxxxx+x+++xx+++x+++++xx+xxxx+xxx+xx+xxx+
A-B bases:    --YY-Y-YYY-Y---YY--YYY-YYYYY-Y---YY--Y--YYYYY-YY-Y--Y--YYY-Y-Y-Y-YY--Y--Y--Y---Y-YYY-YYY-Y-Y-Y--
Bob's bases:  x+xxxx+x+x+xxxxx+++x+xxxx+xx+++xxxxx+xxx+x+++++x+xx+x+x+xxxxx+x+++x+xx+x+xxx+x+xxx+x+xx+xx+xxxxx+++x+x
A-B basis match: 51

```

Figure 9: Comparison of the measurement bases of Alice, Eve and Bob.

To identify an eavesdropper, a sample was selected from the sifted keys to calculate the QBER. A sample size of 17 bits was selected based on equation (6). Using this sample size, the probability for detecting an eavesdropper was calculated using equation (7). The QBER for this simulation was 29.41%. This indicates the presence of an eavesdropper since it is greater than the QBER threshold.

```

Alice key: 1 1 1 0 1 1 1 0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 1 0 1 0 0 1 0 1 1 0
Bob key:   1 0 1 1 1 1 1 0 0 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0 1 0 0 1 1 1 1 0
Length of the sifted key: 51

Alice's key sample: 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0
Mismatched bits:   Y Y Y Y Y Y ! Y ! Y ! ! ! Y Y Y Y
Bob's key sample:  0 0 0 1 0 1 1 1 0 1 1 1 1 0 1 0 0
Number of mismatched bits: 5
QBER = 29.41 %

```

Fig. 10: Alice's and Bob's sifted keys, and samples of the sifted keys used to calculate the QBER.

5.3 Implementation with multiple eavesdroppers

This simulation involved a total of ten eavesdroppers attacking sequentially. The results obtained are shown in Fig. 11. The simulation was implemented for one eavesdropper, then

QBER2	43.75%
QBER3	37.5%
QBER4	37.5%
QBER5	18.75%
QBER6	43.75%
QBER7	50.0%
QBER8	56.25%
QBER9	56.25%
QBER10	50.0%

Table 2: QBERs obtained for different numbers of eavesdroppers.

From the data obtained, it was concluded that the QBER is independent of the number of eavesdroppers, due to the independence of each eavesdropper's measurement basis. Ten simulations were a run and a graph of the number of eavesdroppers against the average QBER plotted.

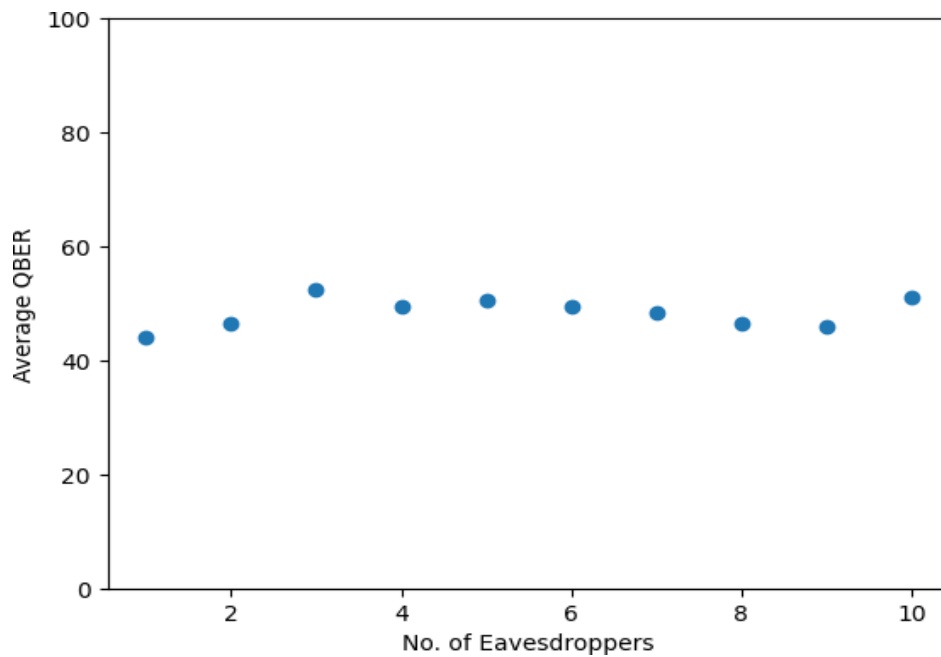


Figure 12: A graph of the number of eavesdroppers vs. the average QBER.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

In this research project, IBM's Qiskit was used to generate data to study the effect of eavesdroppers on a QKD system, by calculating the QBER. We determined that assuming the absence of source, channel, and detector side errors, the QBER was equal to zero when the BB84 protocol was simulated without an eavesdropper. Since the $QBER < 0.11$, the system is secure. We were also able to show the randomness with which Alice and Bob choose their measurement bases.

In the presence of a single eavesdropper, the QBER obtained was 29.41%, which is greater than the QBER threshold, indicating an eavesdropper's presence. In practical implementations, the key would be discarded and a new one generated.

Analysis of the data obtained showed that in the case of multiple eavesdroppers, there was no correlation between the QBER and the number of eavesdroppers. This implies that an increase in the number of eavesdroppers does not affect the QKD system, proving the security of the system. This contrasts with previous research where an increase in the number of eavesdroppers and/ or the depolarizing parameter led to a decrease in the QBER. This previous work took into consideration the presence of a depolarizing channel, which introduces quantum noise in a quantum system. In a different research work, opaque eavesdropping was used to determine the effect of multiple eavesdroppers on a QKD system. Analysis of the BER showed that an increase in the number of eavesdroppers led to a reduced BER. However, a high BER was produced in the absence of eavesdroppers, contradicting theory.

6.2 Recommendations

We recommend that future research focus on implementing the QKD protocol with multiple eavesdroppers over a noisy quantum system. Additionally, research on mitigating technological limitations to the implementation of QKD systems, such as the development of true single photon sources should be considered.

REFERENCES

- Adu-Kyere, A., Nigussie, E., & Isoaho, J. (2022). Quantum Key Distribution: Modeling and Simulation through BB84 Protocol Using Python3. *Sensors*, 22(16), Article 16. <https://doi.org/10.3390/s22166284>
- Bennett, C., & Brassard, G. (1984). *WITHDRAWN: Quantum cryptography: Public key distribution and coin tossing*. 560, 175–179. <https://doi.org/10.1016/j.tcs.2011.08.039>
- Bennett, C. H., & Brassard, G. (2014). Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560, 7–11. <https://doi.org/10.1016/j.tcs.2014.05.025>
- Chatterjee, R., Joarder, K., Chatterjee, S., Sanders, B. C., & Sinha, U. (2019). *qkdSim: An experimenter's simulation toolkit for QKD with imperfections, and its performance analysis with a demonstration of the B92 protocol using heralded photon*. <https://doi.org/10.1103/PhysRevApplied.14.024036>
- Chen, J. (2021). Review on Quantum Communication and Quantum Computation. *Journal of Physics: Conference Series*, 1865(2), 022008. <https://doi.org/10.1088/1742-6596/1865/2/022008>
- Dehmani, M., Errahmani, M., Ez-Zahraouy, H., & Benyoussef, A. (2012). Quantum key distribution with several intercept–resend attacks via a depolarizing channel. *Physica Scripta*, 86(1), 015803. <https://doi.org/10.1088/0031-8949/86/01/015803>
- Ekert, A. K. (1991). Quantum cryptography based on Bell's theorem. *Physical Review Letters*, 67(6), 661–663. <https://doi.org/10.1103/PhysRevLett.67.661>
- Elampari, K., & Ramakrishnan, B. (n.d.). *BIT ERROR RATIO ANALYSIS OF A QKD SYSTEM HAVING MULTIPLE EAVESDROPPERS – REALIZATION USING QUANTUM PACKAGE FOR MATHEMATICA*.
- Elboukhari, M., Azizi, M., & Azizi, A. (2010). *Quantum Key Distribution Protocols: A Survey*.
- Feynman, R. P. (1982). Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6), 467–488. <https://doi.org/10.1007/BF02650179>
- Gisin, N., & Thew, R. (2007). Quantum communication. *Nature Photonics*, 1(3), Article 3. <https://doi.org/10.1038/nphoton.2007.22>
- Gyongyosi, L., Bacsardi, L., & Imre, S. (2019). A Survey on Quantum Key Distribution. *Infocommunications Journal*, 14–21. <https://doi.org/10.36244/ICJ.2019.2.2>
- Haitjema, M. (n.d.). *Quantum Key Distribution—QKD*.
- Inoue, K. (2006). Quantum key distribution technologies. *IEEE Journal of Selected Topics in Quantum Electronics*, 12(4), 888–896. <https://doi.org/10.1109/JSTQE.2006.876606>

- Jasim, O. K., Abbas, S., El-Horbarty, E.-S., & M.Salem, A.-B. (2015). Quantum Key Distribution: Simulation and Characterizations. *Procedia Computer Science*, 65, 701–710. <https://doi.org/10.1016/j.procs.2015.09.014>
- Kohnle, A., & Rizzoli, A. (2017). Interactive simulations for quantum key distribution. *European Journal of Physics*, 38(3), 035403. <https://doi.org/10.1088/1361-6404/aa62c8>
- Lee, C., Sohn, I., & Lee, W. (2022). Eavesdropping Detection in BB84 Quantum Key Distribution Protocols. *IEEE Transactions on Network and Service Management*, 19(3), 2689–2701. <https://doi.org/10.1109/TNSM.2022.3165202>
- Lo, H.-K., & Chau, H. F. (1999). Unconditional Security Of Quantum Key Distribution Over Arbitrarily Long Distances. *Science*, 283(5410), 2050–2056. <https://doi.org/10.1126/science.283.5410.2050>
- Mayers, D. (2001). Unconditional security in quantum cryptography. *Journal of the ACM*, 48(3), 351–406. <https://doi.org/10.1145/382780.382781>
- Mehic, M., Niemiec, M., Rass, S., Ma, J., Peev, M., Aguado, A., Martin, V., Schauer, S., Poppe, A., Pacher, C., & Voznak, M. (2021). Quantum Key Distribution: A Networking Perspective. *ACM Computing Surveys*, 53(5), 1–41. <https://doi.org/10.1145/3402192>
- Mina, M.-Z., & Simion, E. (2021). A Scalable Simulation of the BB84 Protocol Involving Eavesdropping. In D. Maimut, A.-G. Oprina, & D. Sauveron (Eds.), *Innovative Security Solutions for Information Technology and Communications* (pp. 91–109). Springer International Publishing. https://doi.org/10.1007/978-3-030-69255-1_7
- Padamvathi, V., Vardhan, B. V., & Krishna, A. V. N. (2016). Quantum Cryptography and Quantum Key Distribution Protocols: A Survey. *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, 556–562. <https://doi.org/10.1109/IACC.2016.109>
- Pirandola, S., Pirandola, S., Andersen, U. L., Banchi, L., Berta, M., Bunandar, D., Colbeck, R., Englund, D., Gehring, T., Lupo, C., Ottaviani, C., Pereira, J. L., Razavi, M., Shaari, J. S., Shaari, J. S., Tomamichel, M., Tomamichel, M., Usenko, V. C., Vallone, G., ... Wallden, P. (2020). Advances in quantum cryptography. *Advances in Optics and Photonics*, 12(4), 1012–1236. <https://doi.org/10.1364/AOP.361502>
- Praveen Kumar, S., Jaya, T., & Rajalingam, P. (2022). *Implementation of Quantum Key Distribution network simulation in Quantum Channel*. 2335, 012056. <https://doi.org/10.1088/1742-6596/2335/1/012056>
- Rieffel, E., & Polak, W. (2000). An introduction to quantum computing for non-physicists. *ACM Computing Surveys*, 32(3), 300–335. <https://doi.org/10.1145/367701.367709>
- Rietsche, R., Dremel, C., Bosch, S., Steinacker, L., Meckel, M., & Leimeister, J.-M. (2022). Quantum computing. *Springer Berlin Heidelberg*. <https://dspace.mit.edu/handle/1721.1/144261>
- Ruiz-Alba, A., Calvo, D., Garcia-Muñoz, V., Martinez, A., Amaya, W., Rozo, J. G., Mora, J., & Capmany, J. (2010). *Practical Quantum Key Distribution based on the BB84 protocol*.

Shajahan, R., & Nair, S. S. (2020). *Simulation of BB84 Protocol over Classical Cryptography Channel for File Transfer*. 07(09).

Shor, P. W., & Preskill, J. (2000). Simple Proof of Security of the BB84 Quantum Key Distribution Protocol. *Physical Review Letters*, 85(2), 441–444.
<https://doi.org/10.1103/PhysRevLett.85.441>

Sidhu, J. S., Joshi, S. K., Gündoğan, M., Brougham, T., Lowndes, D., Mazzarella, L., Krutzik, M., Mohapatra, S., Dequal, D., Vallone, G., Villoresi, P., Ling, A., Jennewein, T., Mohageg, M., Rarity, J. G., Fuentes, I., Pirandola, S., & Oi, D. K. L. (2021). Advances in space quantum communications. *IET Quantum Communication*, 2(4), 182–217.
<https://doi.org/10.1049/qtc2.12015>

V, A. D., & V, K. (2021). Enhanced BB84 quantum cryptography protocol for secure communication in wireless body sensor networks for medical applications. *Personal and Ubiquitous Computing*, 1–11. <https://doi.org/10.1007/s00779-021-01546-z>

Wiesner, S. (1983). Conjugate coding. *ACM SIGACT News*, 15(1), 78–88.
<https://doi.org/10.1145/1008908.1008920>

Wootters, W. K., & Zurek, W. H. (1982). A single quantum cannot be cloned. *Nature*, 299(5886), Article 5886. <https://doi.org/10.1038/299802a0>

Appendix 1: Simulation in the absence of an eavesdropper

```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, IBMQ, assemble, Aer,
execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit_aer import AerSimulator
#from qiskit.visualization import plot_histogram

import numpy as np
import random
import math

import warnings
#import.filterwarnings('ignore')

# qiskit-ibmq-provider has been deprecated.
# Please the migration guidelines in
https://ibm.biz/provider\_migration\_guide for more detail.
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler,
Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive inside a session. For more details see
https://qiskit.org/documentation/partners/qiskit\_ibm\_runtime/tutorials.html
# with Session(backend=service.backend("ibm_qasm_simulator")):
#     result = Sampler().run(circuits).result()

#=====
# STEP 1
#=====
# Alice generates a random string of bits

n= 100

alice_bits = []

for i in range(n):
    #bits = [0, 1]
    alice_bit = random.randint(0, 1)

    alice_bits.append(alice_bit)

print("Alice's bits:      ", *alice_bits, sep='')
print("")
#=====
```

```

# STEP 2
#=====
# Alice chooses her measurement bases.
# She sends the qubits to Bob.
# Bob picks his measurement bases. He has no idea of Alice's
choices.

alice_choices = []
bob_choices = []
circuits = []

for i in range(n):
    alice_choice = random.choice(['x', '+'])
    bob_choice = random.choice(['x', '+'])

    circuit = QuantumCircuit(1, 1)

    if alice_bits[i] == 1:
        circuit.x(0)

    if alice_choice == 'x':
        circuit.h(0)

    if bob_choice == 'x':
        circuit.h(0)

    alice_choices.append(alice_choice)
    bob_choices.append(bob_choice)
    circuits.append(circuit)

print("Alice's choices: ", *alice_choices, sep='')
print("")
print("Bob's choices:   ", *bob_choices, sep='')
print("")

#=====
# COMPARISON OF ALICE AND BOB'S MEASUREMENT BASES
#=====
same_basis = []
basis_match = 0

for i in range(len(alice_choices)):
    if alice_choices[i] == bob_choices[i]:
        same_basis.append("Y")
        basis_match += 1
    else:
        same_basis.append("-")

print("A-B bases:      ", *same_basis, sep='')
print("")

```

```

#=====
# STEP 3
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(len(circuits)):
    circuits[i].measure(0, 0)
    job = execute(circuits[i], backend = backend, shots = 1, memory
= True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("Bob's bits:      ", *bob_bits, sep='')
print("")

#=====
# STEP 4
#=====
# Alice and Bob compare their measurement bases.
# They discard bits with different bases.
# The remaining bits form the key.

alice_key = []
bob_key = []

for i in range(len(circuits)):
    if alice_choices[i] == bob_choices[i]:
        alice_key.append(alice_bits[i])
        bob_key.append(bob_bits[i])

if alice_key == bob_key:
    key = alice_key

key_len = len(key)

#=====
# COMPARISON OF ALICE AND BOB'S KEY
#=====
key_match = []
for i in range(len(alice_key)):
    if alice_key[i] == bob_key[i]:
        key_match.append("Y")
    else:
        key_match.append("!")

```

```

print("Alice's key: ", *alice_key)
print("")
print("Key match:   ", *key_match)
print("")
print("Bob's key:    ", *bob_key)
print("")

print("Key: ", *key)
print("")
print("Key length: ", key_len)
print("")
print("Basis match: ", basis_match)
print("")
#=====
#STEP 7
#=====
#Calculating the QBER

k = random.sample(key, 17)
mismatched_bits = []

for i in range(len(key)):
    if alice_key[i] != bob_key[i]:
        mismatched_bits.append(alice_key[i])

qber = round((len(mismatched_bits) / len(k) * 100), 2)

print("QBER = ", qber, "%")

```

Appendix 2: Simulation with a single eavesdropper

```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, IBMQ, assemble, Aer,
execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit_aer import AerSimulator
#from qiskit.visualization import plot_histogram

import numpy as np
import random
import math

import warnings
#import.filterwarnings('ignore')

# qiskit-ibmq-provider has been deprecated.
# Please the migration guidelines in
https://ibm.biz/provider\_migration\_guide for more detail.
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler,
Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive inside a session. For more details see
https://qiskit.org/documentation/partners/qiskit\_ibm\_runtime/tutorials.html
# with Session(backend=service.backend("ibmqasm_simulator")):
#     result = Sampler().run(circuits).result()

# STEP 1
# Alice prepares a random bitstring.

n = 100

alice_bits = []

for i in range(n):
    alice_bit = random.randint(0, 1)

    alice_bits.append(alice_bit)

# STEP 2
# Alice chooses measurement bases.
# She creates circuits.
# Eve intercepts.
```

```

# She chooses her measurement bases.

alice_choices = []
eve_choices = []
circuits = []

for i in range(n):
    alice_choice = random.choice(['x', '+'])
    eve_choice = random.choice(['x', '+'])

    circuit = QuantumCircuit(1, 1)

    if alice_bits[i] == 1:
        circuit.x(0)

    if alice_choice == 'x':
        circuit.h(0)

    if eve_choice == 'x':
        circuit.h(0)

    alice_choices.append(alice_choice)
    eve_choices.append(eve_choice)
    circuits.append(circuit)

#print("Alice's choices: ", *alice_choices)
#print("Eve's choices: ", *eve_choices)

# Eve measures the qubits.

backend = Aer.get_backend('qasm_simulator')

eve_bits = []

for i in range(n):
    circuits[i].measure(0, 0)

    job = execute(circuits[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve_bit = int(result.get_memory()[0])

    eve_bits.append(eve_bit)

# Eve prepares her bits to send to Bob.
# STEP 4
# Bob chooses his measurement bases.

new_circuits = []

bob_choices = []

```

```

for i in range(n):
    new_circuit = QuantumCircuit(1, 1)

    bob_choice = random.choice(['x', '+'])

    if eve_bits[i] == 1:
        new_circuit.x(0)

    if bob_choice == 'x':
        new_circuit.h(0)

    bob_choices.append(bob_choice)
    new_circuits.append(new_circuit)

#print("Bob's choices: ", *bob_choices)

# Bob measures the bits.
backend = Aer.get_backend('qasm_simulator')

bob_bits = []

for i in range(n):
    new_circuits[i].measure(0, 0)

    job = execute(new_circuits[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

ae_choices = []
ae_counts = 0
eb_choices = []
eb_counts = 0
ab_choices = []
ab_counts = 0
alice_key = []
bob_key = []

for i in range(n):
    if alice_choices[i] == eve_choices[i]:
        ae_choices.append('Y')

    else:
        ae_choices.append('-')

    if ae_choices[i] == 'Y':

```



```

    ae_counts += 1

    if eve_choices[i] == bob_choices[i]:
        eb_choices.append('Y')

    else:
        eb_choices.append('-')

    if eb_choices[i] == 'Y':
        eb_counts += 1

    if alice_choices[i] == bob_choices[i]:
        ab_choices.append('Y')

    else:
        ab_choices.append('-')

    if ab_choices[i] == 'Y':
        ab_counts += 1

    if alice_choices[i] == bob_choices[i]:
        alice_key.append(alice_bits[i])
        bob_key.append(bob_bits[i])

err = []

num_err = 0

for i in range(len(alice_key)):
    if alice_key[i] != bob_key[i]:
        err.append('!')
        num_err += 1

    else:
        err.append('Y')

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_size = 17

# Select a random starting index.
start_index = random.randint(0, len(alice_key) - sample_size)

# Select a consecutive sample of corresponding bits from both lists.
sample_alice_key = alice_key[start_index:start_index + sample_size]
sample_bob_key = bob_key[start_index:start_index + sample_size]

mis_bits = []

num_mis_bits = 0

```

```

for i in range(len(sample_alice_key)):
    if sample_alice_key[i] != sample_bob_key[i]:
        mis_bits.append('!')
        num_mis_bits += 1

    else:
        mis_bits.append('Y')

qber = round((num_mis_bits / len(sample_alice_key)) * 100, 2)

print("Alice's bits: ", *alice_bits, sep='')
print("")
print("Alice's bases: ", *alice_choices, sep='')
print("")
print("Eve's bases:   ", *eve_choices, sep='')
print("")
print("Eve's bits:     ", *eve_bits, sep='')
print("")
print("Bob's bases:    ", *bob_choices, sep='')
print("")
print("Bob's bits:     ", *bob_bits, sep='')
print("")
print("")
print("")
print("Alice's bases: ", *alice_choices, sep='')
print("")
print("A-E bases:      ", *ae_choices, sep='')
print("")
print("Eve's bases:    ", *eve_choices, sep='')
print("")
print("A-E basis match: ", ae_counts)
print("")
print("Eve's bases:    ", *eve_choices, sep='')
print("")
print("E-B bases:      ", *eb_choices, sep='')
print("")
print("Bob's bases:    ", *bob_choices, sep='')
print("")
print("E-B basis match: ", eb_counts)
print("")
print("Alice's bases: ", *alice_choices, sep='')
print("")
print("A-B bases:      ", *ab_choices, sep='')
print("")
print("Bob's bases:    ", *bob_choices, sep='')
print("")
print("A-B basis match: ", ab_counts)
print("")
print("")
print("")

```

```
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the sifted key: ", len(alice_key))
#print("Errors occur at: ", *err)
#print("")
#print("Number of errors: ", num_err)
print("")
print("")
print("")
print("Alice's key sample: ", *sample_alice_key)
print("")
print("Mismatched bits:     ", *mis_bits)
print("")
print("Bob's key sample:    ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER = ", qber, "%")
```

Appendix 3: Simulation with multiple eavesdroppers

```
# Import standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, IBMQ, assemble, Aer,
execute
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit_aer import AerSimulator

import numpy as np
import random
import math

import warnings

# qiskit-ibmq-provider has been deprecated.
# Please the migration guidelines in
https://ibm.biz/provider\_migration\_guide for more detail.
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler,
Estimator, Session, Options

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive inside a session. For more details see
https://qiskit.org/documentation/partners/qiskit\_ibm\_runtime/tutorials.html
# with Session(backend=service.backend("ibm_qasm_simulator")):
#     result = Sampler().run(circuits).result()

#=====
# DEFINING FUNCTIONS
#=====
# Generating the sifted key.

def generate_key(alice_choices, bob_choices, alice_bits, bob_bits):
    alice_key = []
    bob_key = []

    for i in range(len(alice_choices)):
        if alice_choices[i] == bob_choices[i]:
            alice_key.append(alice_bits[i])
            bob_key.append(bob_bits[i])

    return alice_key, bob_key

# Calculating the QBER

def calculate_qber(alice_key, bob_key, sample_size):
    start_index = random.randint(0, len(alice_key) - sample_size)
    sample_alice_key = alice_key[start_index:start_index +
```

```

sample_size]
    sample_bob_key = bob_key[start_index:start_index + sample_size]

    mis_bits = []
    num_mis_bits = 0

    for i in range(len(sample_alice_key)):
        if sample_alice_key[i] != sample_bob_key[i]:
            mis_bits.append('!')
            num_mis_bits += 1
        else:
            mis_bits.append('Y')

    qber = round((num_mis_bits / len(sample_alice_key)) * 100, 2)

    return sample_alice_key, mis_bits, sample_bob_key, num_mis_bits,
qber

#=====
# STEP 1
#=====
# Alice prepares a random bitstring.

n = 100

alice_bits = []

for i in range(n):
    alice_bit = random.randint(0, 1)

    alice_bits.append(alice_bit)

print("Alice's bits: ", *alice_bits, sep='')

#=====
# STEP 2
#=====
# Alice chooses measurement bases.
# She creates circuits.
# Eve intercepts.
# She chooses her measurement bases.

alice_choices = []
eve_choices = []
circuits = []

for i in range(n):
    alice_choice = random.choice(['x', '+'])
    eve_choice = random.choice(['x', '+'])

    circuit = QuantumCircuit(1, 1)

```

```

    if alice_bits[i] == 1:
        circuit.x(0)

    if alice_choice == 'x':
        circuit.h(0)

    if evel_choice == 'x':
        circuit.h(0)

    alice_choices.append(alice_choice)
    evel_choices.append(evel_choice)
    circuits.append(circuit)

print("")
print("Alice's bases: ", *alice_choices, sep='')
print("")
print("Evel's bases: ", *evel_choices, sep='')

#=====
# EAVESDROPPING!
#=====
# Evel measures the qubits.

backend = Aer.get_backend('qasm_simulator')

evel_bits = []

for i in range(n):
    circuits[i].measure(0, 0)

    job = execute(circuits[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    evel_bit = int(result.get_memory()[0])

    evel_bits.append(evel_bit)

print("")
print("Evel's bits:  ", *evel_bits, sep='')

# Evel prepares her bits to send to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits1 = []

bob_choices = []

```

```

for i in range(n):
    new_circuit1 = QuantumCircuit(1, 1)

    bob_choice = random.choice(['x', '+'])

    if evel_bits[i] == 1:
        new_circuit1.x(0)

    if bob_choice == 'x':
        new_circuit1.h(0)

    bob_choices.append(bob_choice)
    new_circuits1.append(new_circuit1)

print("")
print("Bob's bases: ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

backend = Aer.get_backend('qasm_simulator')

bob_bits = []

for i in range(n):
    new_circuits1[i].measure(0, 0)

    job = execute(new_circuits1[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits: ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key: ", *bob_key)

```

```

print("")
print("Length of the key:", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====

sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:      ", *mis_bits)
print("Bob's key sample", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 1 = ", qber, "%")

#=====
# EAVESDROPPER 2
#=====
# Eve2 intercepts.

new_circuits1 = []
eve2_choices = []

for i in range(n):
    new_circuit1 = QuantumCircuit(1, 1)

    eve2_choice = random.choice(['x', '+'])

    if eve1_bits[i] == 1:
        new_circuit1.x(0)

    if eve2_choice == 'x':
        new_circuit1.h(0)

    eve2_choices.append(eve2_choice)
    new_circuits1.append(new_circuit1)

print("")
print("Eve2's bases: ", *eve2_choices, sep='')

# Eve2 measures the qubits.

backend = Aer.get_backend('qasm_simulator')

eve2_bits = []

for i in range(n):

```



```

    new_circuits1[i].measure(0, 0)

    job = execute(new_circuits1[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve2_bit = int(result.get_memory()[0])

    eve2_bits.append(eve2_bit)

print("")
print("Eve2's bits:  ", *eve2_bits, sep='')

# Eve2 prepares the bits to send them to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits2 = []

for i in range(n):
    new_circuit2 = QuantumCircuit(1, 1)

    if eve2_bits[i] == 1:
        new_circuit2.x(0)

    if bob_choices[i] == 'x':
        new_circuit2.h(0)

    new_circuits2.append(new_circuit2)

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits

backend = Aer.get_backend('qasm_simulator')

bob_bits = []

for i in range(n):
    new_circuits2[i].measure(0, 0)

    job = execute(new_circuits2[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

```

```

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:      ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:     ", *mis_bits)
print("Bob's key sample:     ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 2 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 3!
#=====
# Eve3 intercepts
# She chooses her measurement bases

new_circuits2 = []
eve3_choices = []

for i in range(n):
    new_circuit2 = QuantumCircuit(1, 1)

    eve3_choice = random.choice(['x', '+'])

    if eve2_bits[i] == 1:

```

```

        new_circuit2.x(0)

    if eve3_choice == 'x':
        new_circuit2.h(0)

    eve3_choices.append(eve3_choice)
    new_circuits2.append(new_circuit2)

print("Eve3's bases: ", *eve3_choices, sep='')

# Eve3 measures the qubits

eve3_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits2[i].measure(0, 0)

    job = execute(new_circuits2[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve3_bit = int(result.get_memory()[0])

    eve3_bits.append(eve3_bit)

print("")
print("Eve3's bits: ", *eve3_bits, sep='')

# Eve sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob picks his measurement bases.

new_circuits3 = []

for i in range(n):
    new_circuit3 = QuantumCircuit(1, 1)

    if eve3_bits[i] == 1:
        new_circuit3.x(0)

    if bob_choices[i] == 'x':
        new_circuit3.h(0)

    new_circuits3.append(new_circuit3)

print("")
print("Bob's bases: ", *bob_choices, sep='')

```

```

=====
# STEP 4
=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits3[i].measure(0, 0)

    job = execute(new_circuits3[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:      ", *bob_bits, sep='')

=====
# STEP 5
=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:      ", *mis_bits)
print("Bob's key sample:     ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 3 = ", qber, "%")

```

```

print("")

#=====
# EAVESDROPPER 4!
#=====
# Eve4 intercepts

new_circuits3 = []
eve4_choices = []

for i in range(n):
    new_circuit3 = QuantumCircuit(1, 1)
    eve4_choice = random.choice(['x', '+'])

    if eve3_bits[i] == 1:
        new_circuit3.x(0)

    if eve4_choice == 'x':
        new_circuit3.h(0)

    new_circuits3.append(new_circuit3)
    eve4_choices.append(eve4_choice)

print("Eve4's bases: ", *eve4_choices, sep='')

# Eve4 measures the qubits.

eve4_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits3[i].measure(0, 0)

    job = execute(new_circuits3[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve4_bit = int(result.get_memory()[0])

    eve4_bits.append(eve4_bit)

print("")
print("Eve4's bits: ", *eve4_bits, sep='')

# Eve4 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits4 = []

```

```

for i in range(n):
    new_circuit4 = QuantumCircuit(1, 1)

    if eve4_bits[i] == 1:
        new_circuit4.x(0)

    if bob_choice == 'x':
        new_circuit4.h(0)

    new_circuits4.append(new_circuit4)

print("")
print("Bob's bases: ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits4[i].measure(0, 0)

    job = execute(new_circuits4[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits: ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key: ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))

```

```

print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:      ", *mis_bits)
print("Bob's key sample:     ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 4 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 5!
#=====
# Eve5 intercepts

new_circuits4 = []
eve5_choices = []

for i in range(n):
    new_circuit4 = QuantumCircuit(1, 1)
    eve5_choice = random.choice(['x', '+'])

    if eve4_bits[i] == 1:
        new_circuit4.x(0)

    if eve5_choice == 'x':
        new_circuit4.h(0)

    new_circuits4.append(new_circuit4)
    eve5_choices.append(eve5_choice)

print("Eve5's bases: ", *eve5_choices, sep='')

# Eve5 measures the qubits.

eve5_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits4[i].measure(0, 0)

    job = execute(new_circuits4[i], backend=backend, shots=1,
memory=True)

```

```

    result = job.result()
    eve5_bit = int(result.get_memory()[0])

    eve5_bits.append(eve5_bit)

print("")
print("Eve5's bits:  ", *eve5_bits, sep='')

# Eve5 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits5 = []

for i in range(n):
    new_circuit5 = QuantumCircuit(1, 1)

    if eve5_bits[i] == 1:
        new_circuit5.x(0)

    if bob_choice == 'x':
        new_circuit5.h(0)

    new_circuits5.append(new_circuit5)

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits5[i].measure(0, 0)

    job = execute(new_circuits5[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:  ", *bob_bits, sep='')

```



```

=====
# STEP 5
=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:    ", *mis_bits)
print("Bob's key sample:   ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 5 = ", qber, "%")
print("")

=====
# EAVESDROPPER 6!
=====
# Eve6 intercepts

new_circuits5 = []
eve6_choices = []

for i in range(n):
    new_circuit5 = QuantumCircuit(1, 1)
    eve6_choice = random.choice(['x', '+'])

    if eve5_bits[i] == 1:
        new_circuit5.x(0)

    if eve6_choice == 'x':
        new_circuit5.h(0)

    new_circuits5.append(new_circuit5)

```

```

    eve6_choices.append(eve6_choice)

print("Eve6's bases: ", *eve6_choices, sep='')

# Eve6 measures the qubits.

eve6_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits5[i].measure(0, 0)

    job = execute(new_circuits5[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve6_bit = int(result.get_memory()[0])

    eve6_bits.append(eve6_bit)

print("")
print("Eve6's bits: ", *eve6_bits, sep='')

# Eve6 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits6 = []

for i in range(n):
    new_circuit6 = QuantumCircuit(1, 1)

    if eve6_bits[i] == 1:
        new_circuit6.x(0)

    if bob_choice == 'x':
        new_circuit6.h(0)

    new_circuits6.append(new_circuit6)

print("")
print("Bob's bases: ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

```

```

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits6[i].measure(0, 0)

    job = execute(new_circuits6[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:      ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:      ", *mis_bits)
print("Bob's key sample:     ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 6 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 7!
#=====
# Eve7 intercepts

```

```

new_circuits6 = []
eve7_choices = []

for i in range(n):
    new_circuit6 = QuantumCircuit(1, 1)
    eve7_choice = random.choice(['x', '+'])

    if eve6_bits[i] == 1:
        new_circuit6.x(0)

    if eve7_choice == 'x':
        new_circuit6.h(0)

    new_circuits6.append(new_circuit6)
    eve7_choices.append(eve7_choice)

print("Eve7's bases: ", *eve7_choices, sep='')

# Eve7 measures the qubits.

eve7_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits6[i].measure(0, 0)

    job = execute(new_circuits6[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve7_bit = int(result.get_memory()[0])

    eve7_bits.append(eve7_bit)

print("")
print("Eve7's bits: ", *eve7_bits, sep='')

# Eve7 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits7 = []

for i in range(n):
    new_circuit7 = QuantumCircuit(1, 1)

    if eve7_bits[i] == 1:
        new_circuit7.x(0)

```

```

    if bob_choice == 'x':
        new_circuit7.h(0)

    new_circuits7.append(new_circuit7)

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits7[i].measure(0, 0)

    job = execute(new_circuits7[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:  ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:  ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =

```

```

calculate_qber(alice_key, bob_key, sample_size)

print("Alice key sample: ", *sample_alice_key)
print("Mismatched bits: ", *mis_bits)
print("Bob key sample:   ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 7 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 8!
#=====
# Eve8 intercepts

new_circuits7 = []
eve8_choices = []

for i in range(n):
    new_circuit7 = QuantumCircuit(1, 1)
    eve8_choice = random.choice(['x', '+'])

    if eve7_bits[i] == 1:
        new_circuit7.x(0)

    if eve8_choice == 'x':
        new_circuit7.h(0)

    new_circuits7.append(new_circuit7)
    eve8_choices.append(eve8_choice)

print("Eve8's bases: ", *eve8_choices, sep='')

# Eve8 measures the qubits.

eve8_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits7[i].measure(0, 0)

    job = execute(new_circuits7[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve8_bit = int(result.get_memory()[0])

    eve8_bits.append(eve8_bit)

print("")

```

```

print("Eve8's bits:  ", *eve8_bits, sep='')

# Eve8 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits8 = []

for i in range(n):
    new_circuit8 = QuantumCircuit(1, 1)

    if eve8_bits[i] == 1:
        new_circuit8.x(0)

    if bob_choice == 'x':
        new_circuit8.h(0)

    new_circuits8.append(new_circuit8)

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits8[i].measure(0, 0)

    job = execute(new_circuits8[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:  ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

```

```

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:     ", *mis_bits)
print("Bob's key sample:    ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 8 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 9
#=====
# Eve9 intercepts

new_circuits8 = []
eve9_choices = []

for i in range(n):
    new_circuit8 = QuantumCircuit(1, 1)
    eve9_choice = random.choice(['x', '+'])

    if eve8_bits[i] == 1:
        new_circuit8.x(0)

    if eve9_choice == 'x':
        new_circuit8.h(0)

    new_circuits8.append(new_circuit8)
    eve9_choices.append(eve9_choice)

print("Eve9's bases: ", *eve9_choices, sep='')

# Eve9 measures the qubits.

```



```

eve9_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits8[i].measure(0, 0)

    job = execute(new_circuits8[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve9_bit = int(result.get_memory()[0])

    eve9_bits.append(eve9_bit)

print("")
print("Eve9's bits:  ", *eve9_bits, sep='')

# Eve9 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits9 = []

for i in range(n):
    new_circuit9 = QuantumCircuit(1, 1)

    if eve9_bits[i] == 1:
        new_circuit9.x(0)

    if bob_choice == 'x':
        new_circuit9.h(0)

    new_circuits9.append(new_circuit9)

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits9[i].measure(0, 0)

```

```

    job = execute(new_circuits9[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:      ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:   ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))
print("")

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:     ", *mis_bits)
print("Bob's key sample:     ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)
print("")
print("QBER 9 = ", qber, "%")
print("")

#=====
# EAVESDROPPER 10!
#=====
# Eve10 intercepts

new_circuits9 = []
eve10_choices = []

for i in range(n):
    new_circuit9 = QuantumCircuit(1, 1)

```

```

eve10_choice = random.choice(['x', '+'])

if eve9_bits[i] == 1:
    new_circuit9.x(0)

if eve10_choice == 'x':
    new_circuit9.h(0)

new_circuits9.append(new_circuit9)
eve10_choices.append(eve10_choice)

print("Eve10's bases: ", *eve10_choices, sep='')

# Eve10 measures the qubits.

eve10_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits9[i].measure(0, 0)

    job = execute(new_circuits9[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    eve10_bit = int(result.get_memory()[0])

    eve10_bits.append(eve10_bit)

print("")
print("Eve10's bits: ", *eve10_bits, sep='')

# Eve10 prepares and sends the bits to Bob.
#=====
# STEP 3
#=====
# Bob chooses measurement bases.

new_circuits10 = []

for i in range(n):
    new_circuit10 = QuantumCircuit(1, 1)

    if eve10_bits[i] == 1:
        new_circuit10.x(0)

    if bob_choice == 'x':
        new_circuit10.h(0)

    new_circuits10.append(new_circuit10)

```

```

print("")
print("Bob's bases:  ", *bob_choices, sep='')

#=====
# STEP 4
#=====
# Bob measures the qubits.

bob_bits = []

backend = Aer.get_backend('qasm_simulator')

for i in range(n):
    new_circuits10[i].measure(0, 0)

    job = execute(new_circuits10[i], backend=backend, shots=1,
memory=True)
    result = job.result()
    bob_bit = int(result.get_memory()[0])

    bob_bits.append(bob_bit)

print("")
print("Bob's bits:  ", *bob_bits, sep='')

#=====
# STEP 5
#=====
# Generating the sifted key

alice_key, bob_key = generate_key(alice_choices, bob_choices,
alice_bits, bob_bits)

print("")
print("Alice key: ", *alice_key)
print("")
print("Bob key:  ", *bob_key)
print("")
print("Length of the key: ", len(alice_key))

#=====
# SELECTING RANDOM SAMPLES FOR POST-PROCESSING
#=====
sample_alice_key, mis_bits, sample_bob_key, num_mis_bits, qber =
calculate_qber(alice_key, bob_key, sample_size)

print("Alice's key sample: ", *sample_alice_key)
print("Mismatched bits:  ", *mis_bits)
print("Bob's key sample:  ", *sample_bob_key)
print("")
print("Number of mismatched bits: ", num_mis_bits)

```

```
print("")  
print("QBER 10 = ", qber, "%")
```

ORIGINALITY REPORT

5%

SIMILARITY INDEX

3%

INTERNET SOURCES

4%

PUBLICATIONS

1%

STUDENT PAPERS

PRIMARY SOURCES

1	Anusuya Devi V, Kalaivani V. "Enhanced BB84 quantum cryptography protocol for secure communication in wireless body sensor networks for medical applications", Personal and Ubiquitous Computing, 2021 Publication	<1%
2	Submitted to De Montfort University Student Paper	<1%
3	en.wikipedia.org Internet Source	<1%
4	Hong-xia Zhao, Li Huang. "Effects of Noise on Joint Remote State Preparation of an Arbitrary Equatorial Two-Qubit State", International Journal of Theoretical Physics, 2016 Publication	<1%
5	"Innovative Security Solutions for Information Technology and Communications", Springer Science and Business Media LLC, 2021 Publication	<1%
6	www.dtic.mil Internet Source	<1%

7	Trent Graham, Christopher Zeitler, Joseph Chapman, Paul Kwiat, Hamid Javadi, Herbert Bernstein. "Superdense teleportation and quantum key distribution for space applications", 2015 IEEE International Conference on Space Optical Systems and Applications (ICSOS), 2015 Publication	<1%
8	Submitted to University of Wales Institute, Cardiff Student Paper	<1%
9	Zeng. "Quantum Key Distribution", Quantum Private Communication, 2010 Publication	<1%
10	Lecture Notes in Computer Science, 2010. Publication	<1%
11	digitalcommons.lsu.edu Internet Source	<1%
12	Ch. Seshu. "Quantum Key Distribution", Communications in Computer and Information Science, 2008 Publication	<1%
13	Takang William Ako, Dobgima Walter Pishoh, Nguemaim Flore, Kwangfis Richard Nemline et al. "The Prevalence Outcome and Associated Factors of Teenage Pregnancy in	<1%

the Bamenda Health District", Open Journal of Obstetrics and Gynecology, 2023

Publication

-
- 14 Yong-Min Li, Xu-Yang Wang, Zeng-Liang Bai, Wen-Yuan Liu, Shen-Shen Yang, Kun-Chi Peng. "Continuous variable quantum key distribution", Chinese Physics B, 2017
Publication <1%
-
- 15 core.ac.uk
Internet Source <1%
-
- 16 repository.up.ac.za
Internet Source <1%
-
- 17 udsspace.uds.edu.gh
Internet Source <1%
-
- 18 publish.illinois.edu
Internet Source <1%
-
- 19 www.biorxiv.org
Internet Source <1%
-
- 20 "Next Generation Intelligent Optical Networks", Springer Science and Business Media LLC, 2008
Publication <1%
-
- 21 Magdalena Krzyszkowska-Pytel. "Quantum Cryptography - The Issue of Security in Selected Quantum Protocols and the Issue of

Data Credibility", Theoretical and Applied Informatics, 01/01/2010

Publication

22 RENATO RENNER. "SECURITY OF QUANTUM KEY DISTRIBUTION", International Journal of Quantum Information, 2011 <1 %

Publication

23 Rupesh Kumar, Marco Lucamarini, Giovanni Di Giuseppe, Riccardo Natali, Giorgio Mancini, Paolo Tombesi. "Two-way quantum key distribution at telecommunication wavelength", Physical Review A, 2008 <1 %

Publication

24 www.accenture.com <1 %

Internet Source

25 Hitoshi Inamori. Journal of Physics A Mathematical and General, 09/07/2001 <1 %

Publication

26 Submitted to Trident University International <1 %

Student Paper

27 Yeong Cherng Liang, Dagomir Kaszlikowski, Berthold-Georg Englert, Leong Chuan Kwek, C. H. Oh. "Tomographic quantum cryptography", Physical Review A, 2003 <1 %

Publication

28 link.springer.com <1 %

Internet Source

29

ntv.ifmo.ru

Internet Source

<1%

30

www.grin.com

Internet Source

<1%

Exclude quotes On

Exclude matches < 5 words

Exclude bibliography On