UNIVERSIDAD POLITÉCNICA DE MADRID

ETSI SISTEMAS INFORMÁTICOS

Practical free-space quantum key distribution system for metropolitan links

Jose Manuel Bustarviejo Casado

Contents

CONTENTS I
FIGURES III
GRAPHS V
TABLES VII
ACKNOWLEDGMENTS IX
INTRODUCTION XI
OBJECTIVE XIII
SUMMARY XV
ABSTRACT XVII
CHAPTER 1: SOFTWARE ANALYSIS. COMPARATIVES. 19
1.2 SOFTWARE DEVELOPMENT APPLICATIONS 19
1.2.1 Visual Studio Code *19*
1.2.2 CLion *20*
1.2.3 CodeBlocks *21*
1.2.4 Eclipse *21*
CHAPTER 2: PROJECT GLOBAL DESCRIPTION 23
2.1 QUANTUM COMMUNICATION BETWEEN PARTIES 23
2.2 RECONCILIATION PROTOCOL BETWEEN PARTIES 24
CHAPTER 3: METHODOLOGY 27
3.1 PYTHON IMPLEMENTATION 27
3.2 CLASSICAL COMMUNICATION PROTOCOL 27
3.3 KEY BLOCK ANALYSIS 27
3.4 SENDING BLOCKS 28
CHAPTER 4: RESULTS 29
4.1 ERROR RATE 2% 29
4.2 ERROR RATE 4% 30
4.3 ERROR RATE 6% 30
4.4 ERROR RATE 8% 31
4.5 ERROR RATE 10% 31
4.6 QBER VS EXECUTION TIME 32
4.6 DISCUSSION 33
CHAPTER 5: PROJECT COSTS 35
5.1 TEMPORAL COSTS 35
5.2 HUMAN COSTS 35
CHAPTER 6: SOCIAL, ECONOMIC, ENVIRONMENTAL AND LEGAL ASPECTS 37
APPENDIX 39
APPENDIX I: THE CASCADE PROTOCOL 39
I.I Quantum key distribution (QKD) protocols *39*

I.II Key bit errors (noise)....40
I.III Classical post-processing....41
I.IV The Cascade Protocol....42
I.V Cascade as a client-server protocol....42
I.VI The classical channel....43
I.VII Input and output of the Cascade protocol....45
I.VIII Cascade iterations....46
I.IX Key shuffling....47
I.X Creation of the top-level blocks....48
I.XI Detecting and correcting bit errors in each block....50
I.XII Computing the error parity for each top-level block: even or odd....50
I.XIII Correcting a single bit error in top-level blocks with an odd number of bits....54
I.XIV The Binary algorithm....55
I.XV What about the remaining errors after correcting a single bit error?....59
I.XVI The role of shuffling in error correction....59
I.XVII The Cascade Effect....61
I.XVIII Parallelization and bulking....62
APPENDIX II: VISUAL STUDIO CODE SETUP....65
II.I Installation....65
II.II User setup versus system setup....65
II.III Updates....65
APPENDIX III: C++ COMPILER AND DEBUGGER....67
ACRONYM....69
REFERENCES....71

Figures

Figure 1: Quantum Communications between Alice and Bob 23
Figure 2: Classical communications between Alice and Bob 24
Figure 3: Quantum key distribution protocols 39
Figure 4: Key bit errors 40
Figure 5: Classical post-processing 41
Figure 6: Peer-to-peer and Client-Server scheme 43
Figure 7: Input and output 45
Figure 8: Cascade iterations 47
Figure 9: Key shuffling (I) 47
Figure 10: Key shuffling (II) 48
Figure 11: Creation of the top level blocks 49
Figure 12: Computing the current parity 50
Figure 13: Computing the correct parity (I) 51
Figure 14: Computing the correct parity (II) 52
Figure 15: Computing the correct parity (III) 52
Figure 16: Computing the correct parity (IV) 53
Figure 17: Computing the correct parity (V) 53
Figure 18: Inference the error parity from current parity and the correct parity 54
Figure 19: Split block (I) 56
Figure 20: Split block (II) 56
Figure 21: Split block (III) 56
Figure 22: Recursion 57
Figure 23: Role of the shuffling in error correction(I) 59
Figure 24: Role of the shuffling in error correction (II) 60
Figure 25: Cascade Effect (I) 61
Figure 26: Cascade Effect (II) 61

Graphs

Graph 1: Error rate 2%....29
Graph 2: Error rate 4%....30
Graph 3: Error rate 6%....30
Graph 4: Error rate 8%....31
Graph 5: Error rate 10%....31
Graph 6: QBER vs Execution time (I)....32
Graph 7: QBER vs Execution time (II)....32
Graph 8: Gantt Chart....35

Tables

TABLE 1: SOCIAL, ECONOMIC, ENVIRONMENTAL AND LEGAL ASPECTS 37

Acknowledgments

I want to thank specially CSIC PhD Student, Pablo Arteaga Diaz who has supported me in the whole development of this master thesis. His selfless helping and patience has contributed greatly to the successful of this project. Thank you very much for all, Pablo. I wish you all the best for the future.

I would like also to thank CSIC PhD Veronica Fernandez Marmol for giving me the opportunity to develop this master thesis in CSIC and supporting me when I needed it.

Thank you also to PhD Jesus Garcia Lopez de Lacalle for giving me the chance of developing my master thesis in a scientific institution like CSIC and helping me in the whole master and the development of the thesis.

Thank you to PhD. Giannicola Scarpa for helping me always when I have needed as in the master as in the thesis. Thank you very much for all PhD Giannicola!

Thank you to technical staff Jesus Negrillo from CSIC for helping me in the installations of all programs that I have needed for doing this thesis.

Thank you to my parents for helping and giving me the chance of doing the master which has allowed me to learn more about Quantum Computing.

And finally, thank you to all master's teachers for having the patience and interest of teaching me. I'll try to apply all the knowledge that you have taught me.

Introduction

Quantum mechanics has taken a great importance in the last years. There are a lot of companies and countries that are researching and working in possible applications for this field.

One of these applications is the use of Quantum mechanics laws for carrying out communications in a safe way. This application field is called Quantum communications.

This document will cover the use of the Cascade protocol for error correction in a quantum communications channel connecting two parties (Alice and Bob) to implement the BB84 quantum protocol for key exchange in a secure way.

Objective

The main aim is the error correcting in a secure way of a key which has been exchange between two parties (Alice and Bob) through a quantum channel.

For the quantum communications have been used the BB84 protocol. This protocol utilises QKD for reliable exchange of a key between two parties. For error correcting has been utilized the Cascade protocol which allows find and fix errors in a key. The communication in Cascade protocol are classical communications.

The system works as follow: Suppose there are two parties, Alice and Bob, who want to exchange a key through a quantum channel. Alice has a string of bits which she wants to send Bob in a safety way. For this, Alice codifies the bits string in a key using photons which are polarized through four bases. Once the photons have been polarized, Alice sends the key obtained to Bob. When Bob receives the key, he decodes this using the same bases that Alice. However, the key can have error produced by the noise or because Bob has chosen the incorrect basis for decoding one or more photons of the received key. For resolving it, Bob divides his key in blocks and calculates the parity for each block. For each block, Bob requests the block parity in Alice's key which corresponds with the current block that Bob is analysing in that moment. Alice sends her block parity to Bob, and he compares Alice's parity with his block parity. If they are different an error has found. In this case, Bob searches the position where the error is and fix it. This process for error correcting is repeat until the key has been fixed completely.

For quantum communications have been used a laser beam system which won't be covered in this document. On the other hand, for implementing the Cascade protocol has been utilised Visual Studio Code like IDE and C++ like programming language.

Summary

This master thesis is related to the error processing stage of a practical quantum communication system.

The system allows secure communications between two parties, (Alice and Bob). Initially, Alice has a string of bits that wants to send to Bob in a safe way. In order to do this, Alice encrypts the string of bits using photons that are polarized through applying horizontal, vertical, diagonal left and diagonal right bases. When all photons have been polarized, Alice sends them through a quantum channel to Bob. Once Bob receives them, he applies the same type of bases that Alice uses to encrypt for decrypting but in a random manner. They then compare the bases they used through a public, but authenticated channel, and keep only those in which both use the same bases for encryption and decryption. This string of bits, known as the *sifted* key, can still contain errors (bits that are different for Alice and Bob) which are produced by the noise of the channel or because Bob has applied a wrong basis in one or more photons. To resolve this, Alice and Bob use a reconciliation protocol which allows to fix the errors of the sifted key that Bob has. In this reconciliation protocol, Bob divides his key in blocks. For each block, Bob calculates the parity and asks Alice for the parity of the corresponding block in Alice's key. When Bob receives Alice's parity, he compares it with his parity for the current block. If the parties are not equal, there is an error in the current block. In this case, Bob will look for the error in the block and fix it.

However, there can be an eavesdropper (Eve) in the communications who can try to steal information. In this case, during quantum communications, Eve can intercept the photons that Alice has sent to Bob and can do the following:

1. Decrypt the key that Alice has sent to Bob using the same bases that Alice. Eve has ¼ of probability of success and ¾ probability of failing. If Eve chooses this option, she will modify the state of each photon.
2. Save the photons state without measuring anything and waiting to finish the photon transmissions.

In the first case (1), Alice and Bob can detect Eve if the QBER (Quantum Bits Error Rate) is greater that a certain threshold. To measure the QBER Alice and Bob publicly reveal part of the sequence they share and compare their bit values. These bits will then be discarded. If the QBER is higher than a certain threshold, which is determined by the attacks assumed on the channel by Eve, the transmission is not secure and must be aborted. This error is sufficient to be detected by Alice and Bob, and they must assume that all errors in the transmission are caused by Eve.

In the second case (2), Eve can wait for the error correction protocol. However, the information Bob shares with Alice during the reconciliation protocol gives no indication about the value of the bit as Bob sends Alice the indices of the current block that Bob is analysing and Alice responds with the parity of the block.

Abstract

Esta tesis de máster consiste en realizar la etapa de corrección de errores de una clave depurada de un sistema de comunicaciones cuánticas experimental.

El sistema permite comunicaciones seguras entre dos partes (Alice y Bob). Inicialmente, Alice tiene una cadena de bits que quiere enviar a Bob de forma segura. Para ello, Alice cifra la cadena de bits utilizando fotones que se polarizan aplicando las siguientes bases de manera aleatoria: rectilínea con polarizaciones horizontal y vertical, codificando '1's y '0's, respectivamente; y base diagonal, con polarizaciones diagonal a izquierdas y diagonal a derechas, codificando '1's y '0's, respectivamente. Alice envía los fotones a Bob a través de un canal cuántico (fibra óptica o aire). Una vez que Bob los recibe, aplica bases de medición aleatoriamente, e independientemente de Alice, para medir cada fotón y obtener una cadena de bits. En la siguiente etapa denominada reconciliación de bases, Alice y Bob comparan las bases que utilizaron para codificar y medir cada bit y descartan todos aquellos bits en los que ambas bases no coinciden. Esta cadena de bits en el caso ideal debería ser idéntica, pero en la realidad, puede contener errores producidos por el canal, cuentas oscuras de los detectores o de fuentes de luz solar o artificial que se cuelan en los detectores, o porque Bob ha aplicado una base incorrecta en uno o más fotones. Para resolverlo, Alice y Bob utilizan un protocolo de reconciliación de errores, que permite corregir la clave depurada de Alice y Bob. En este protocolo de reconciliación, Alice y Bob dividen su clave en bloques. Para cada bloque, Bob calcula la paridad y solicita a Alice (por un canal público y autenticado) la paridad del bloque correspondiente en la clave de Alice. Cuando Bob recibe la paridad de Alice, la compara con su paridad para el bloque actual. Si las paridades no son iguales, hay un error en el bloque actual que Bob está analizando en ese momento. En este caso, Bob buscará el error en la clave y lo corregirá, (para una explicación más detallada consultar *Appendix I*).

La explicación anterior trata sobre el funcionamiento del sistema. En caso de que haya un intruso (Eve) en el sistema, durante las comunicaciones cuánticas puede hacer lo siguiente:

1. Interceptar los fotones que Alice envía a Bob. Eve puede introducir las mismas bases que Alice con probabilidad de 1/4.
2. Guardar los fotones que envía Alice por un lado sin aplicar ninguna medida y esperar a que acabe el envío de fotones a Bob.

Para el primer caso (1), Alice y Bob pueden detectar a Eve si el QBER (Quantum Bits Error Rate) es mayor que un cierto umbral. Para medir el QBER Alice y Bob revelan públicamente parte de la secuencia que comparten y comparan sus valores de bits. Estos bits serán descartados. Si el QBER es superior a un cierto umbral, que está determinado por los ataques asumidos en el canal por Eve, la transmisión no es segura y debe ser abortada. Este error es suficiente para ser detectado por Alice y Bob, y deben asumir que todos los errores en la transmisión son causados por Eve.

Para el segundo caso (2), Eve puede esperar al protocolo de corrección de errores. Sin embargo, la información que Bob comparte con Alice durante el protocolo de reconciliación no da indicios acerca del valor del bit ya que Bob envía a Alice los índices del bloque actual que Bob está analizando y Alice responde con la paridad del bloque.

Por lo tanto, Eve no puede obtener suficiente información para identificar los errores que obtuvo cuando mide su clave.

Chapter 1: Software analysis. Comparatives.

In this chapter we will be described the applications selected for this project and their characteristics.

1.2 Software development applications

We have used Visual Studio Code (Microsoft, www), (VSC-Microsoft) for software development.

On the other hand, it was contemplated other applications such as CLion (JetBrains, www), (CLion-JetBrains, www), CodeBlocks (CodeBlocks, www) and Eclipse (Eclipse, www). These applications weren't used by the next reasons:

- For using CLion is necessary to pay a license. In this case, there wasn't enough budget for paying it.
- CodeBlocks and Eclipse weren't used because they weren't inside the preferences.

Next, we are going to explain each one of the last designate IDEs and their characteristics:

1.2.1 Visual Studio Code

Visual Studio Code (VSC-Wikipedia, www) is an integrated development environment made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality.

Visual Studio Code was first announced on April 29, 2015, by Microsoft at the 2015 Build conference. A preview build was released shortly thereafter.

On November 18, 2015, the source of Visual Studio Code was released under the MIT License and made available on GitHub. Extension support was also announced. On April 14, 2016, Visual Studio Code graduated from the public preview stage and was released to the Web.

The advantages of developing with Visual Studio Code are:

- It can be used with a variety of programming languages, including Java, JavaScript, Go, Node.js, Python and C++.

- Users can open one or more directories instead of a project system. Directories can be saved in workspaces for future reuse.
- Visual Studio Code can be extended via extensions, available through a central repository. A notable feature is the ability to create extensions that add support for new languages, themes, and debuggers, perform static code analysis, and add code linters using the Language Server Protocol.
- Visual Studio Code includes multiple extensions for FTP, allowing the software to be used as a free alternative for web development.
- Visual Studio Code allows users to set the code page in which the active document is saved, the newline character, and the programming language of the active document.

1.2.2 CLion

CLion (CLion-LinuxAdic, www) is an IDE for developing in C and C++ programming languages. It can be used in Windows, Linux and macOS integrated with the compilation system CMake.

CMake is a tools family designed for creating, testing and packing since this manages the software compilation process using simple platforms and setting files which are independent from compiler.

The first version of CLion (CLion-ReleaseDate, www) was released in 2015.

Between the main characteristics of CLion (CLion-MainPage, www) can be found:

- Code assistance: Read and write code effectively with an editor that deeply understand C and C++.
- Code generation: Generate tons of boilerplate code instantly.
- Safe refactoring: Rename symbols; inline a function, variable, or macro; move members through the hierarchy; change function signatures; and extract functions, variables, parameters, or a typedef.
- Quick Documentation: Inspect the code under the caret to learn just about anything: function signature details, review comments, preview Doxygen-style documentation, check out the inferred type for symbols lacking explicit types, and even see properly formatted final macro replacements.
- Code analysis: Potential code issues are detected instantly and can be fixed at the touch of a button, while the IDE correctly handles the changes. CLion runs its code analysis, Data Flow Analysis, other Clangd-based checks, and Clang-Tidy to detect unused and unreachable code, dangling pointers, missing type casts, no matching function overload, and many other issues.
- Integrated debugger: Set breakpoints, Evaluate expressions and View values inline.
- Fully Integrated C/C++ Development Environment.

1.2.3 CodeBlocks

CodeBlocks is a free C/C++ and Fortran IDE built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable.

Built around a plugin framework, CodeBlocks can be extended with plugins. Any kind of functionality can be added by installing/coding a plugin. For instance, event compiling and debugging functionality is provided by plugins.

The first stable version was released on February 28, 2008, with version number 8.02 (CodeBlocks-Wikipedia, www).

Next, it will be showed some characteristic about CodeBlocks:

- CodeBlocks includes wxSmith which is a complement that gives functionality for creation and visual edition user graphical interface.
- Support for compilers such as GCC, Microsoft Visual Studio Toolkit, etc.
- GNU GDB interface.
- Workspace for combing different projects
- Automatic Tabulation
- Task lists (ToDo)
- Classes Generation

1.2.4 Eclipse

Eclipse (Eclipse-Wikipedia, www) is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins, including Ada, ABAP, C, C++, C#, Clojure, COBOL, D, Erlang, Fortran, Groovy, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Rust, Scala, and Scheme. It can also be used to develop documents with LaTeX (via a TeXlipse plug-in) and packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++, and Eclipse PDT for PHP, among others.

Eclipse software development kit (SDK) is free and open-source software, released under the terms of the Eclipse Public License, although it is incompatible with the GNU General Public License. It was one of the first IDEs to run under GNU Classpath and it runs without problems under IcedTea.

The first version was released on December 7, 2001.

Next, some characteristic about Eclipse will be indicated (Eclipse-Characteristics, www):

- Almost everything in Eclipse is a plugin.
- Functionality of Eclipse IDE can be extended by adding plugin to the IDE.
- Supports various source knowledge tools like folding and hyperlink navigation, grading, etc.
- Provides excellent visual code debugging tool to debug the code.
- Eclipse has a wonderful user interface with drag and drop facility for UI designing.
- Supports project development and administered framework for different toolchains, classic make framework, and source navigation.
- Java Eclipse IDE has a JavaDoc facility using which we can automatically create documentation for classes in our application.

Chapter 2: Project global description

In this chapter we will carry out a global description from the different parts of the project.

2.1 Quantum communication between parties

The quantum communications in this project are between two parties, (Alice and Bob), using QKD for sending a key through a quantum channel. The quantum communication protocol used for quantum communications is BB84. The next picture shows the communication scheme between two parties, Alice and Bob, using the BB84 protocol:

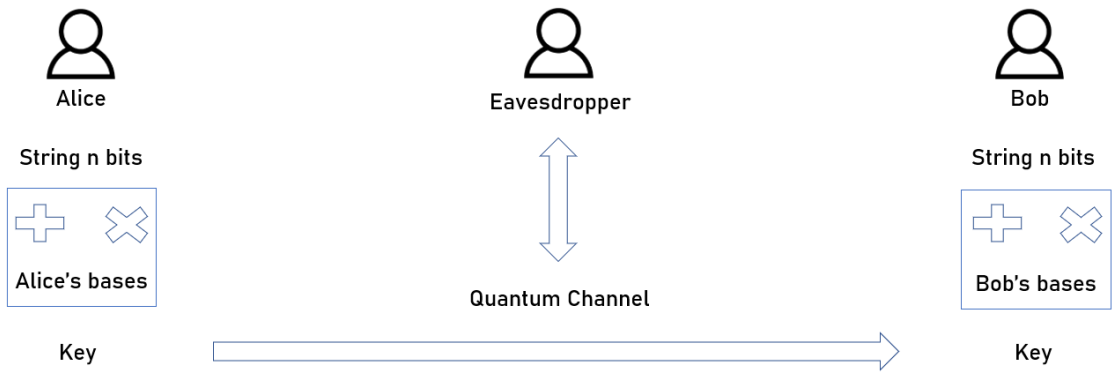


Figure 1: Quantum Communications between Alice and Bob

The communication between Alice and Bob follows the next steps:

1. Alice sends to Bob a string of n bits which has been encrypted in a key using polarized photons.
2. However, the key sends by Alice can be intercepted by an eavesdropper that can be listening in the channel. Eavesdropper doesn't know in which bases Alice has polarized each photon, so an eavesdropper has 1/4 of successful chance. By non-cloning theorem, the eavesdropper can't clone the states of each photon. Thus, the eavesdropper can do the next:
    a. Saving the photons without measuring them. At this point, the eavesdropper waits until the communications have finished to measure the key.
3. Bob measures the photons applying his bases to photons and gets the string of n bits.
4. To know if there is a possible eavesdropper listening in the quantum channel, Alice and Bob calculate the QBER (Quantum Bit Error Rate) on a certain number of bits from Alice and Bob's sifted keys though a public channel. If the QBER is greater than a certain threshold, they know an eavesdropper is in the channel and must discard the key.

5. If QBER isn't greater than a certain threshold, still there isn't a eavesdropper in the channel.

2.2 Reconciliation protocol between parties

In this section we will explain the steps of reconciliation protocol used by Alice and Bob for error correcting. The communications, in this case will be held through a classical authenticated channel. The protocol that has been used is Cascade (the explanation of this protocol is in the *Appendix I*).

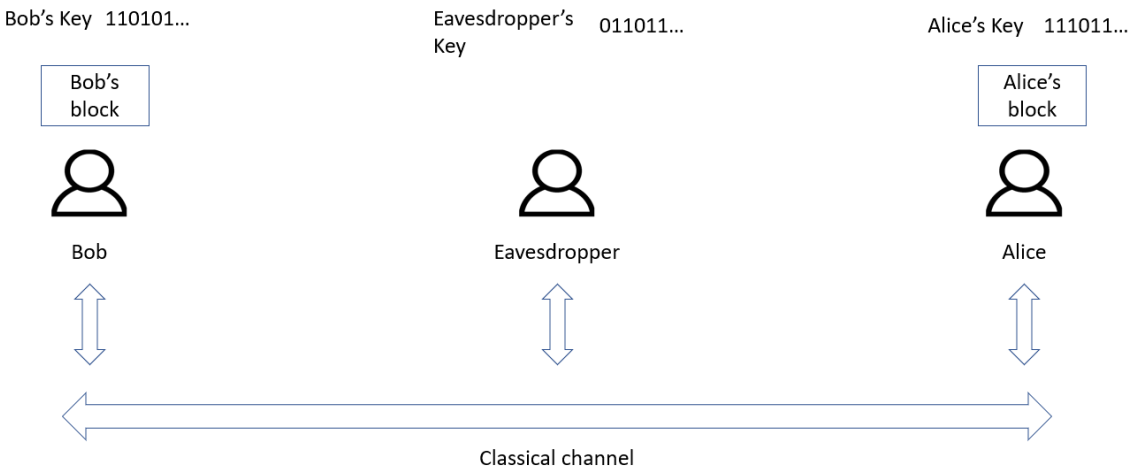


Figure 2: Classical communications between Alice and Bob

The steps followed by the Cascade protocol will be summarized below:

1. Bob gets the indexes of his sifted key. These indexes are the position which occupies each bit in the key.
2. Bob builds a matrix where each row is an iteration of the Cascade protocol. Each row contents a shuffle of indexes from the original key. This shuffle is a permutation of the bits of the key. Thus, for each row a permutation about the bits of the key is done.
3. For each iteration of Cascade protocol, Bob selects the row which corresponds to this iteration which is divided in blocks. For each block, Bob calculates the parity and requests the parity of Alice's corresponding block in her key.
4. If Bob's parity is different to Alice's parity, the correction is carried out. This correction is made using Binary algorithm (this algorithm is explained in the *Appendix I*) which returns an index where the error has been committed.

Please note that at this stage, if there is an eavesdropper in the channel, the only information he or she can get from the channel is the block of indexes that Bob

requests from Alice and the parity sent by Alice. This does not reveal any information of the key.

Chapter 3: Methodology

In this chapter will be explained the different problems that have been found and their solutions.

3.1 Python implementation

The code used for implementing the classical communication between Alice and Bob is based in the work of Andre Reis (AndreReisWork, Thesis). Firstly, we tried to implement the code using Python, but it gave a problem concretely in *do-while loop* of CascadeEffect algorithm. Thus, it was changed from Python to C++ because the Andre Reis's document explained that the code was implemented in C so it deciding to program the code in C++ for this reason. On the other hand, we chose C++ because it is faster than Python.

Once the code was translated from Python to C++, the problems were fixed, and this worked perfectly. Thus, we believe it is recommendable to use C/C++ for programming the Cascade protocol.

3.2 Classical communication protocol

Initially, the protocol used for classical communication was TCP. This protocol was used because the classical channel that Alice and Bob used for sending information isn't safe and TCP guarantees the security and reliability in the communications. However, TCP protocol is slow, and this is a problem since there is a continuous exchange of information between Alice and Bob that requires speed. This exchange increases as a function of the key size and the blocks number which the key is divided in.

Thus, for increasing the exchange speed, the TCP protocol has been substituted by the UDP protocol. The UDP protocol increases the velocity of the communication, but it is not as safe as TCP.

3.3 Key block analysis

The way that Bob analyses each block of the key is by a *for loop* which takes the blocks one by one for analysing. For increasing the number of blocks that are analyses at the same time, the for loop has been parallelized using OpenMP (OpenMP, www).

OpenMP manages the threads which are set by the user. Therefore, OpenMP will assign to each thread a block for its analysis, so the execution speed should increase and the execution time decrease.

3.4 Sending blocks

Each block from the key is sent to Alice to calculate the parity. For sending each block through the classical channel, it is necessary to convert an integer array to a string. It supposes a computational cost which implies more execution time. To reduce this, instead of sending the full block of the key, the first and last element from the block are sent.

However, Alice needs to have the matrix, which contains the indexes of the key for each iteration of Cascade protocol so it will be necessary to share with Alice the matrix at the beginning of the Cascade algorithm execution.

Chapter 4: Results

In this chapter we will show the results obtained when the Cascade protocol is applied for error correcting with classical communications.

Each section will present the graphs Key length versus Execution time for 2%, 4%, 6%, 8% and 10% of error rate in the key.

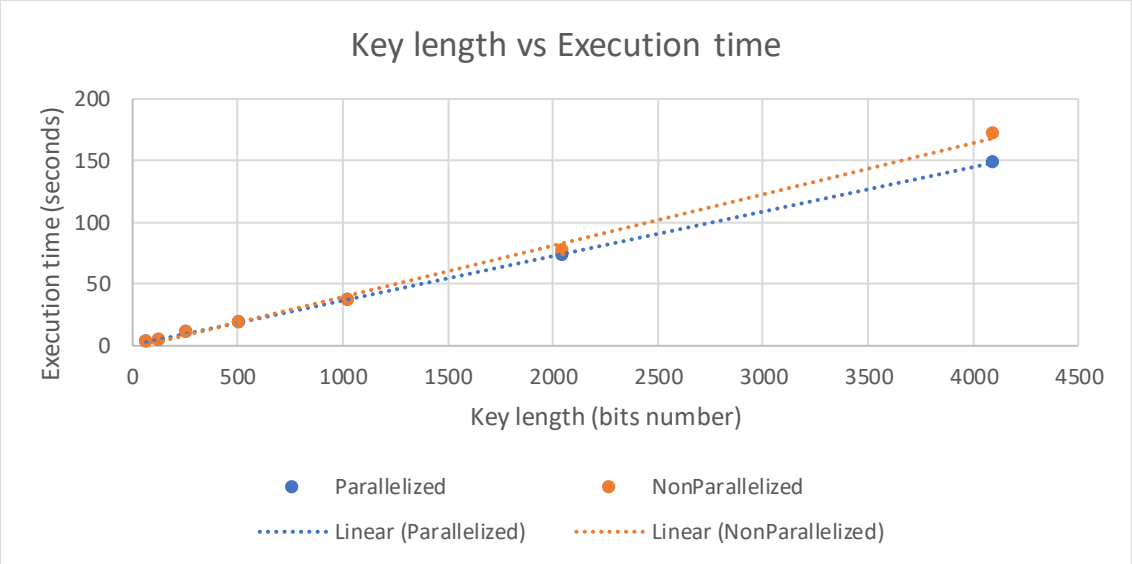
The communications in every test are in **localhost**.

The error rate is the percentage of errors that have been produced in the key.

For making these errors, we have used a script in Python, which produces the errors in the key.

4.1 Error rate 2%

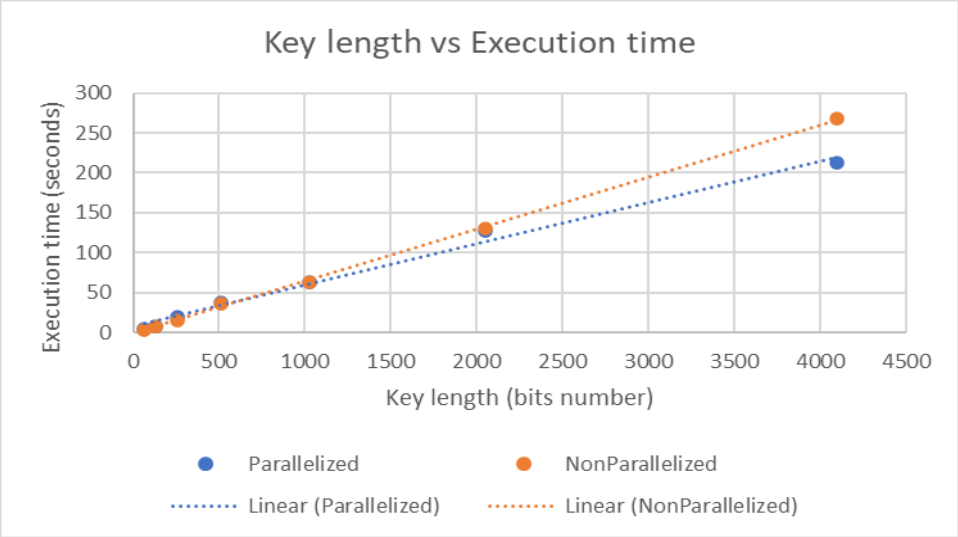
In this section the key length vs execution time will be plotted for the Cascade protocol with parallelization and without parallelization for a 2% of error rate.



Graph 1: Error rate 2%

4.2 Error rate 4%

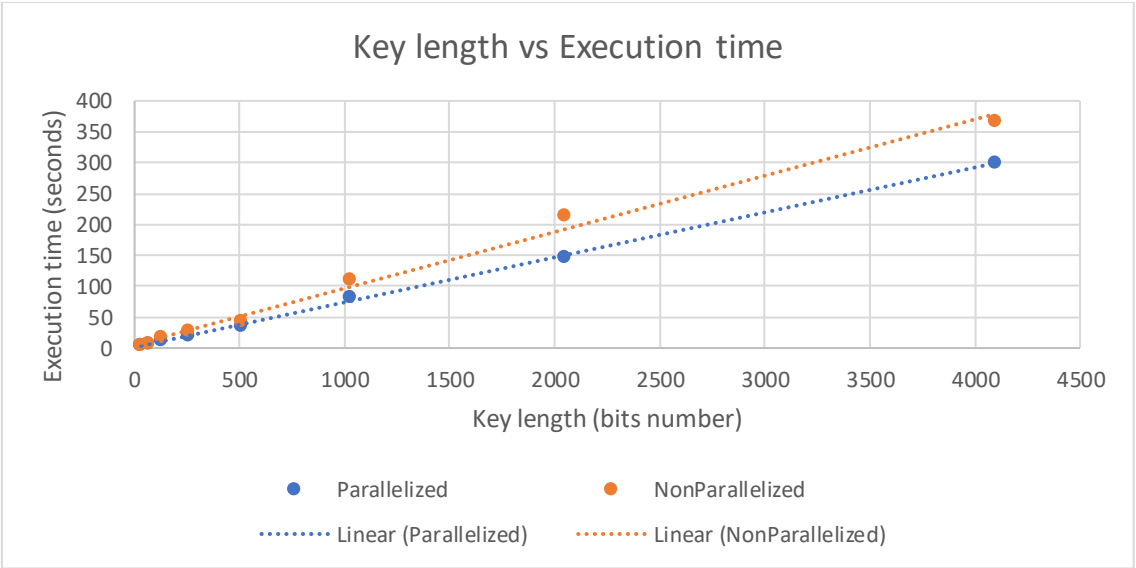
In this section will be plotted the Key length versus Execution time for the Cascade protocol with parallelization and without parallelization for a 4% of error rate.



Graph 2: Error rate 4%

4.3 Error rate 6%

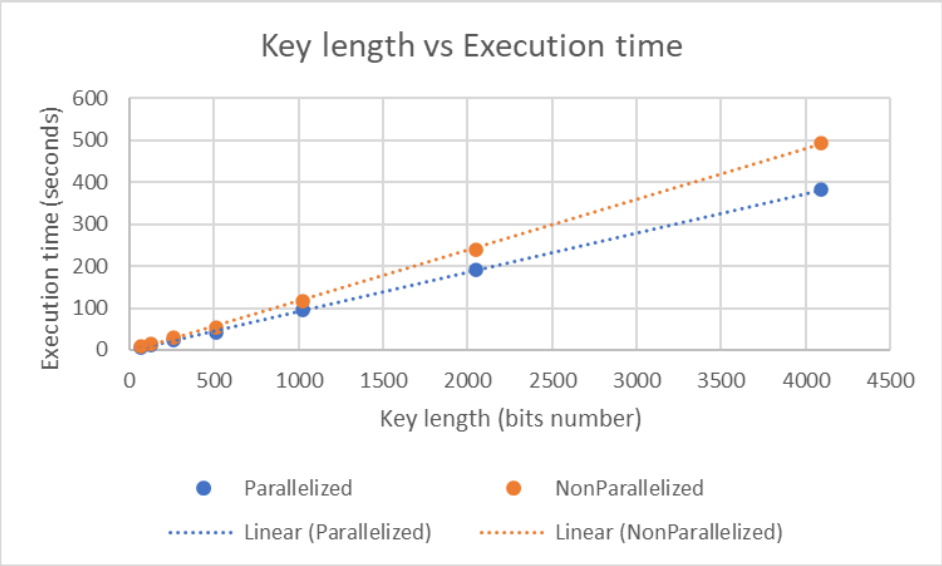
In this section will be plotted the Key length versus Execution time for the Cascade protocol with parallelization and without parallelization for a 6% of error rate.



Graph 3: Error rate 6%

4.4 Error rate 8%

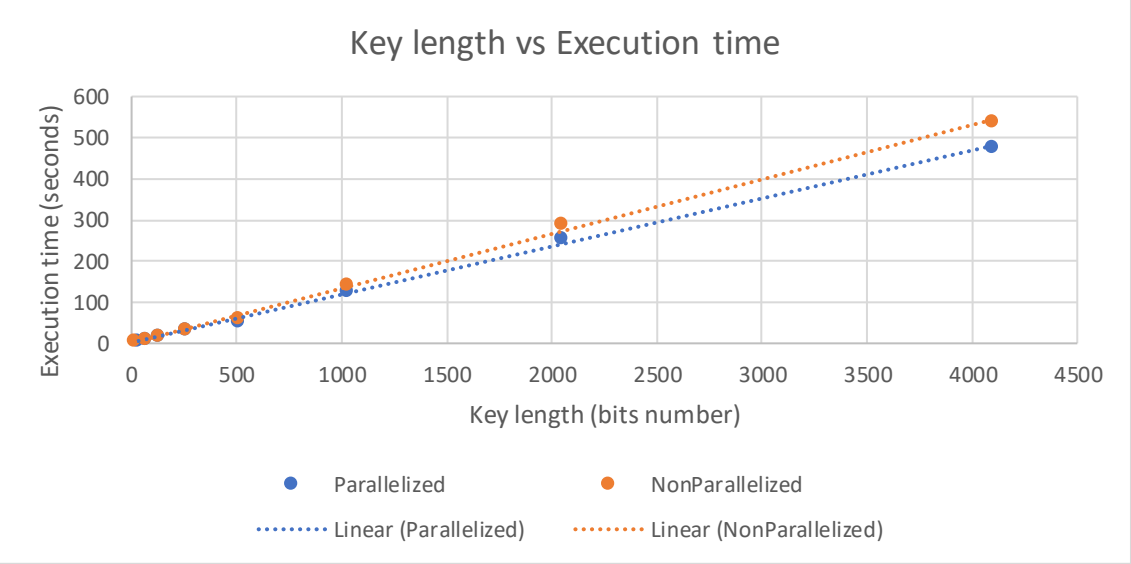
In this section will be plotted the Key length versus Execution time for the Cascade protocol with parallelization and without parallelization for an 8% of error rate.



Graph 4: Error rate 8%

4.5 Error rate 10%

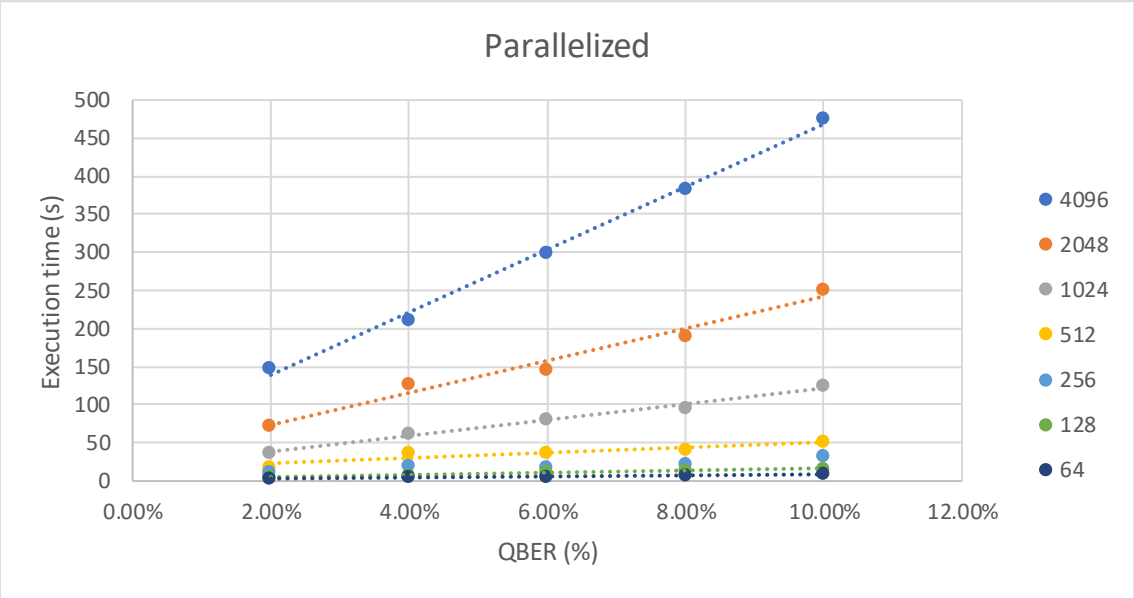
In this section will be plotted the Key length versus Execution time for the Cascade protocol with parallelization and without parallelization for a 10% of error rate.



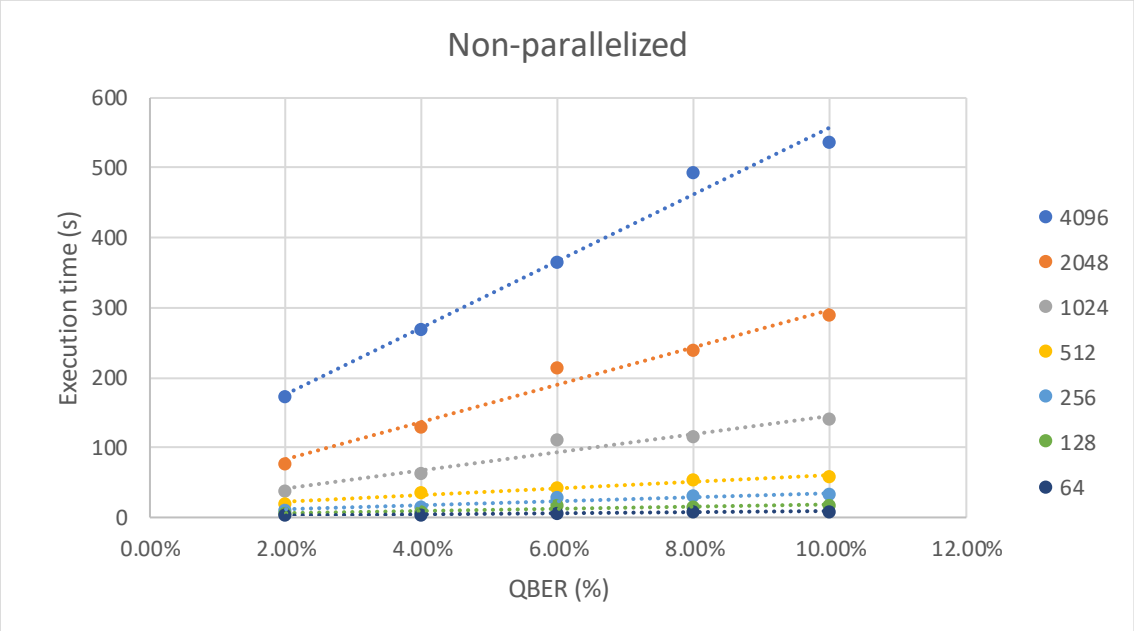
Graph 5: Error rate 10%

4.6 QBER vs Execution time

In this section will be represented QBER (Quantum Bit Error Rate) vs Execution time. There are two graphs: the first one for parallelized algorithm and the second one for non-parallelized algorithm.



Graph 6: QBER vs Execution time (I)



Graph 7: QBER vs Execution time (II)

4.6 Discussion

The first five graphs show the execution time for key lengths between 2^4-2^{12} with 2%, 4%, 6%, 8% and 10% of error rate. In each graph can be appreciated that for key lengths minor than 2^{11} the execution time of code parallelized and non-parallelized are equal or close each other. Parallelized advantage seems to have an effect for key length greater or equal than 2^{11} bits. Thus, if the key is long enough, the parallelized protocol will be faster than non-parallelized protocol. Besides, as error rate increase, the difference between parallelization and non-parallelization are greater.

In the last two graphics can be appreciated the algorithm behaviour in a global way for the parallelized and non-parallelized algorithm. When algorithm isn't parallelized, the execution time is a little scattered, but if the algorithm is parallelized the execution time tends to be linear. Thus, there is an improvement on results when the code is parallelized.

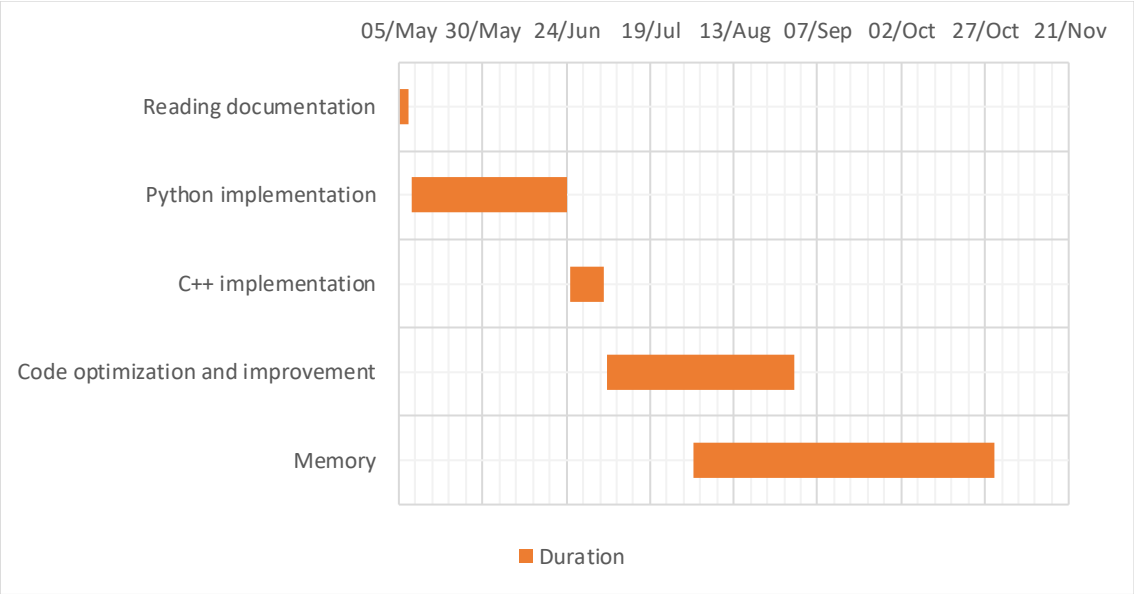
On the other hand, the complexity for each graph is O(n), (more and less), which implies that the execution time will go grow linearly with the number of elements of the key. Maybe this complexity would be less if the algorithm was implemented in a quantum computer but for the moment this is not possible.

Chapter 5: Project Costs

In this chapter will be indicated the project costs. They are temporal and human costs.

5.1 Temporal Costs

In this section will be showed the temporal costs of the project. These costs will be represented by a Gantt Chart.



Graph 8: Gantt Chart

5.2 Human Costs

For this project was necessary the support of different experts in distinct matters. Next, will be detailed the cost of each expert:

- Software Engineer: 1700 €/month
- Electronic physical: 2500 €/month
- Quantum physical: 1700 €/month

Chapter 6: Social, economic, environmental and legal aspects

In this chapter will be described Social, economic, environmental and legal aspects.

This project is thought for carrying out quantum communication between two parties and correcting the errors that can be produced.

The main problem is the execution time for error correcting protocol which increases with the key's size.

All devices used in this project fulfil with environmental regulations for energy saving as well as the using of non-polluting batteries. The cost of the product development isn't very high, only it's necessary to buy the devices involved in the quantum communication between Alice and Bob. The software applications are free.

The life cycle of the product depends only on the devices used in the project.

	Ethical aspects	Social aspects	Environmental aspects
Design, software develop, simulation and production of the tests	Having software license or hardware of the product	Comply with the data protection regulations of each country	Management of energy resources and use of biodegradable materials
Physical implementation of the system	Mention of the contribution of third parties	Analysis of safety, risks and energy resources involved	Comply with the environmental and current regulations of each country
System use and maintenance	Privacy Rights	Planned obsolescence	Management of energy resources of low consumption
Reuse, recycling or disposal	Intellectual Property Rights	Integration and acceptance of society in the face of possible innovations	Reuse of the devices used in the prototyping part

Table 1: Social, economic, environmental and legal aspects

Appendix

Appendix I: The Cascade protocol

In this appendix will be explained Cascade protocol (Cascade-protocol, www).

I.I Quantum key distribution (QKD) protocols

All quantum key distribution (QKD) protocols involve using a combination of quantum communications (qubits) and classical communications (classical bits) to allow two parties Alice and Bob to agree on a secret key in such a way that our nefarious eavesdropper Eve cannot observe what the secret key is without being detected by Alice and Bob.

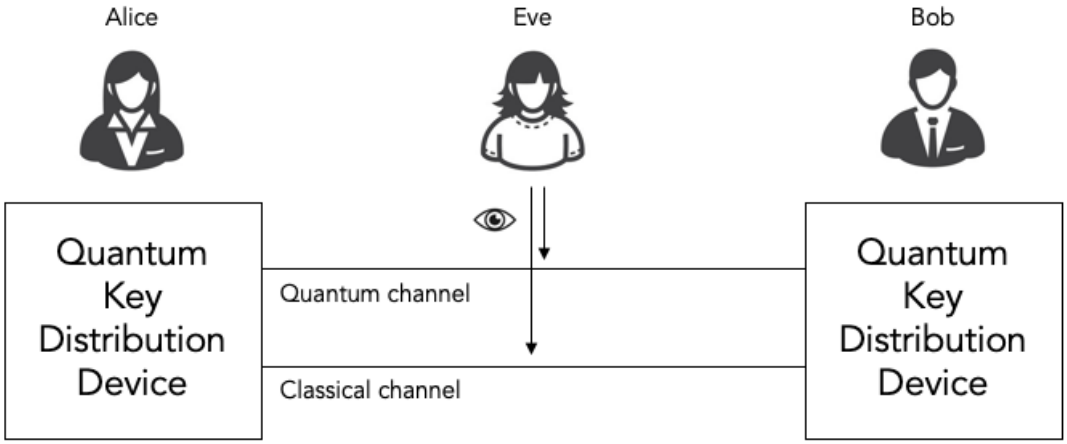


Figure 3: Quantum key distribution protocols

There are multiple quantum key distribution protocols, including for example BB84 and B92. All these protocols consist of both a quantum phase and a classical post-processing phase.

The quantum phase uses both the quantum channel and the classical channel to exchange the key.

The classical post-processing phase only uses the classical channel. The classical post-processing phase is further sub-divided into two parts:

- Information reconciliation, which is responsible for detecting and correcting inevitable bit errors (noise) in the key that was exchanged during the quantum phase.
- Privacy enhancement, which is responsible for mitigating the information leakage during the information reconciliation step.

In this document we only discuss one specific information reconciliation protocol, namely the Cascade protocol.

We won't discuss privacy enhancement nor the quantum phase. Those interested in more details on the quantum phase can have a look at our simulaqron-bb84-python GitHub repository that contains our Python implementation of the quantum phase in the BB84 quantum key distribution protocol.

I.II Key bit errors (noise)

Key distribution protocols always introduce some noise in the key. The key that Bob receives contains some noise (i.e. bit errors) as compared to the key that Alice sent. For that reason, we refer to the key that Alice sent as the correct key and to the key that Bob received as the noisy key.

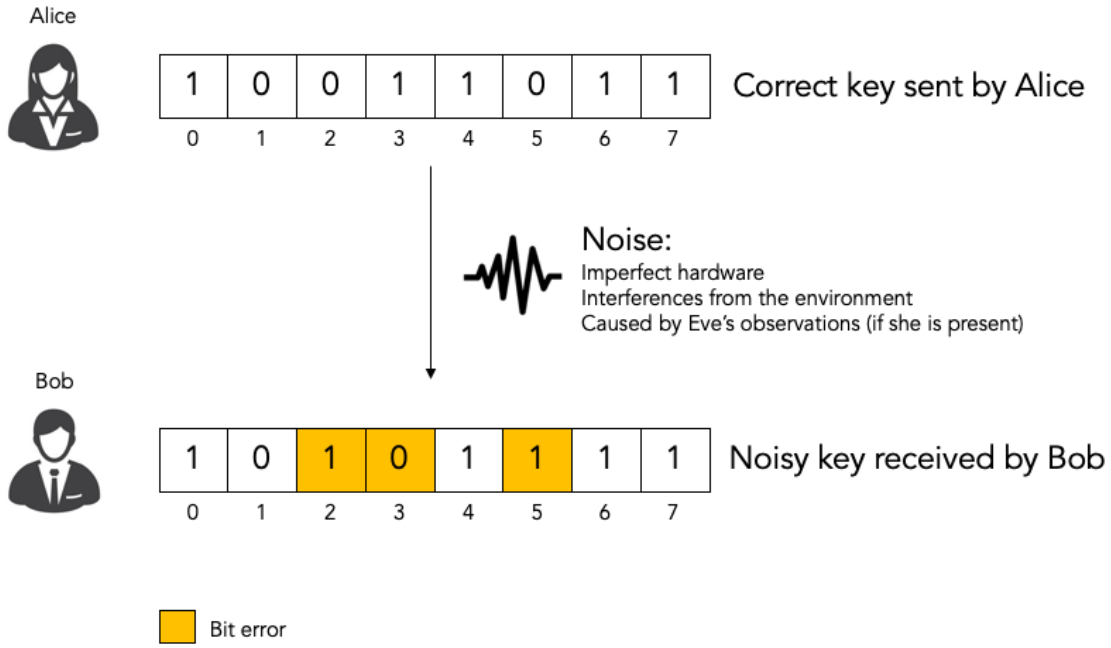


Figure 4: Key bit errors

The noise can be introduced by imperfections in the hardware and by random fluctuations in the environment. Or the noise can be introduced by eavesdropper Eve observing traffic. Remember: in quantum mechanics observing a photon causes the photon to change and hence introduces detectable noise.

All quantum key distribution protocols provide an estimate of the noise level in the form of an estimated bit error rate. Bit error rate 0.0 means that no key bits have been flipped and bit error rate 1.0 means that all key bits have been flipped.

I.III Classical post-processing

If the estimated bit error rate is above some threshold, we conclude that Eve is observing the traffic trying to determine the secret key. In that case, we abandon the key distribution attempt.

If the estimated bit error rate is below the threshold we perform classical post-processing, which consist of the two steps that we mentioned earlier. Both steps are classical protocols in the sense that they only involve classical communications and not any quantum communications.

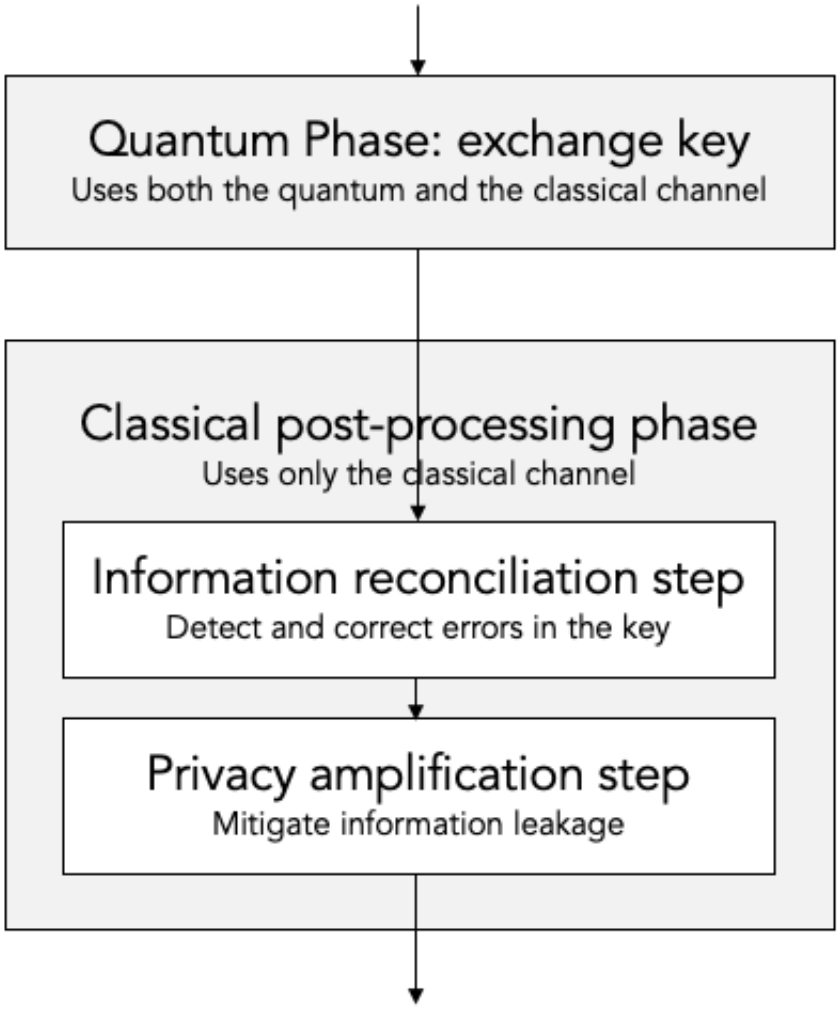


Figure 5: Classical post-processing

I.III.I Information reconciliation

The first classical post-processing step is information reconciliation. Even if the bit error rate is below the threshold, it is not zero. There is still some noise: there are still bit errors in the noisy key that Bob received as compared to the correct key that Alice sent. The purpose of the information reconciliation step is to detect and correct these remaining bit errors.

There are multiple information reconciliation protocols. In this document we discuss only one specific protocol, namely the Cascade protocol.

The tricky part to information reconciliation is to avoid leaking (i.e. exposing) too much information about the key. Eve, the eavesdropper, can learn any information that we leak during the information reconciliation step. Even if she does not learn the entire key, learning any leaked partial information about the key simplifies her task of decrypting the encrypted traffic. Every bit of leaked key information halves the number of keys that Eve has to try during a brute force attack.

That said, it is unavoidable that the information reconciliation protocol leaks some limited amount of information. This is okay if the amount of leaked information is bounded and known, so that we can compensate for it.

I.III.II Privacy amplification

The second classical post-processing step is privacy amplification. The purpose of privacy amplification is to compensate for the information leakage in the information reconciliation step. Privacy amplification introduces extra randomness at the cost of reducing the effective key size.

In this document we do not discuss privacy amplification any further.

I.IV The Cascade Protocol

The Cascade protocol is one example of an information reconciliation protocol. The purpose of the Cascade protocol is to detect and to correct any remaining bit errors in the noisy key that one peer (Bob) received relative to the correct key that other peer (Alice) hast sent.

Let's say that Alice and Bob are the two parties that have just run the quantum phase of a quantum key distribution such as BB84. Alice has the correct key and Bob has a noisy key, which is similar to Alice's correct key, but which has some limited number of bit errors (i.e. noise). Alice and Bob will now run the Cascade protocol to detect and correct any remaining bit errors in the Bob's noisy key.

I.V Cascade as a client-server protocol

From a protocol point of view, it makes sense to describe the Cascade protocol in terms of a client-server protocol.

Bob takes the role of the client. As far as Cascade is concerned, he is the active party. He decides what needs to happen when. He does most of the computation. And he sends messages to Alice when he needs her to do something.

Alice takes the role of the server. As far as Cascade is concerned, her role is mostly passive. She waits for Bob to ask her simple questions and she provides the answers.

We will describe the Cascade protocol from the perspective of the client, i.e. from the perspective of Bob.

An interesting observation is that the Cascade protocol puts most of the complexity and most of the computational burden on the client. The server doesn't do much except compute simple parities when asked to do so by the client. This is a very nice property for a client-server protocol. The server could have many (thousands) of sessions to clients, so it is very desirable that each session is simple and lightweight. The client will typically have only a few sessions to a few servers, so it is okay if the sessions are more complex and heavy weight.

It is fair to say that quantum key distribution is currently often (almost always, perhaps) used to secure point-to-point links with a quantum key distribution device on either end of the link. From that perspective it is natural to think of Cascade as a peer-to-peer protocol.

That said, quantum key distribution in general and Cascade could very well be deployed in true client-server scenarios. One example scenario is secure web traffic where a web server (e.g. Apache) has many session to many different web clients (e.g. Chrome web browsers) using the HTTPS protocol.

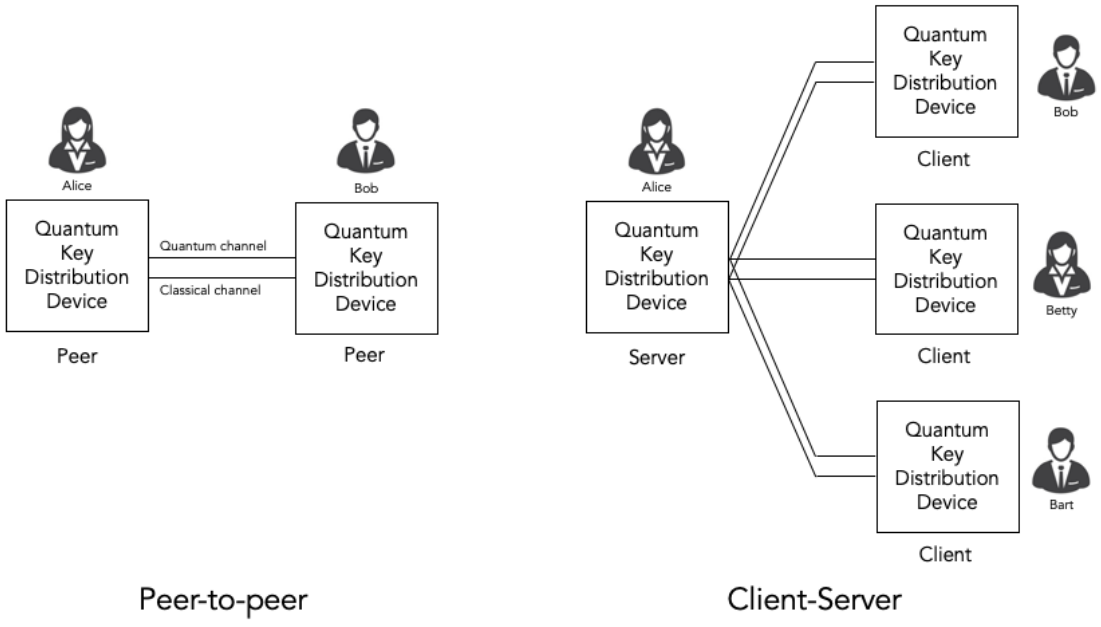


Figure 6: Peer-to-peer and Client-Server scheme

I.VI The classical channel

Cascade is a fully classical protocol. It only involves the exchange of classical messages. It does not involve any quantum communications.

We assume that there is a classical channel between Alice and Bob that allows Alice and Bob to exchange classical messages as part of the Cascade protocol. We rely on classical techniques to provide reliability, flow-control, etc. (for example, we could use TCP/IP).

We do not require that the classical channel is encrypted: we assume that eavesdropper Eve can observe all classical messages in the clear.

Any requirement that the classical channel be encrypted would introduce a chicken-and-egg problem: we would need a quantum key distribution protocol to encrypt the classical channel, but the quantum key distribution protocol would need an encrypted classical channel.

We do, however, require that the classical channel provides authentication and integrity. We assume that there is a mechanism that allows Alice and Bob to verify that all classical messages were actually sent by Bob and Alice and have not been forged or tampered with by Eve.

This is needed to avoid woman-in-the-middle attacks by Eve, where Eve intercepts all classical traffic and pretends to be Bob to Alice and pretends to Alice to Bob.

We do not discuss how the authentication and integrity are implemented nor does the code in this repository contain any authentication or integrity mechanisms.

This is consistent with most of the literature on quantum key distribution. Most literature barely mentions the need for an authentication and integrity on the classical channel. Details on how to do it are even less forthcoming. This might give you the impression that it is a trivial matter not worth discussing. Nothing could be further from the truth!

Yes, it is true that authentication and integrity are considered to be well-solved problems for classical protocols. For authentication, classical protocols typically use either public key infrastructure (PKI) or pre-shared keys. For integrity, classical protocols typically use hash-based message authentication codes (HMAC) in combination with Diffie-Hellman or pre-shared keys to agree on the message authentication key.

But none of those options (pre-shared keys, public key infrastructure, Diffie-Hellman) are attractive options for quantum key distribution.

Public-key infrastructure and Diffie-Hellman are problematic because they are not quantum-safe: they rely on the assumption that factorization or modular logarithms are computationally difficult.

Pre-shared keys are somewhat acceptable for point-to-point connections, but they are problematic in client-server scenarios where the server does not know a-priori which clients will connect to it. But more importantly, using pre-shared keys defeats the whole purpose of running a quantum key distribution protocol.

In summary: while the topic of authenticating the classical channel is usually glossed over, it is not at all obvious how to achieve it in the context of quantum key distribution.

I.VII Input and output of the Cascade protocol

Let's start by looking at the Cascade protocol as a black box algorithm, and let's consider what the input and the output of the Cascade protocol are.

Figure 7: Input and output

I.VII.I Input: noisy key and estimated quantum bit error rate (QBER)

Bob initiates the Cascade protocol after the quantum phase of the quantum key distribution has been completed.

At this point, Bob has the following information available to him, which is the input to the Cascade protocol.

Bob has the noisy key that he has received from Alice. Although a quantum key distribution protocol was used to agree on this key, there is nothing quantum about the key at this point. It is just a string of classical bits of a certain length (the key size).

As we described earlier, the quantum key distribution protocol introduces some noise when it delivers this key to Bob. Thus, Bob has a noisy key which has some bit errors compared to Alice's correct key.

Bob does not know exactly how many bit errors there are or which bits are in error, but the quantum key distribution protocol does provide an estimate of the bit error rate, which also known as the quantum bit error rate (QBER).

Thus, we have two inputs to the Cascade protocol: the noisy key and the estimated quantum bit error rate (QBER).

I.VII.II Output: reconciliated key and amount of leaked information

It is the job of the Cascade protocol to determine which bits exactly are in error and to fix them.

It is important to understand that Cascade does not guarantee that all bit errors are corrected. In other words, Bob's reconciliated key is still not guaranteed to be the same

as Alice's correct key. Even after the reconciliation is complete, there is still a remaining bit error rate. The remaining bit error rate is orders of magnitude smaller than the original bit error rate before Cascade was run. But it is not zero. That is why we prefer to use the term reconciliated key and not corrected key, although the latter is also often used.

Cascade per-se does not contain any mechanism to detect and report whether the reconciliation was successful. It will neither detect nor report that there are any remaining bit errors after reconciliation. Some mechanism outside of Cascade is needed to validate whether the reconciliated key is correct or not.

The Cascade protocol can also keep track of exactly how much information was leaked. Specifically, Cascade running at Bob can keep track of which parities he asked Alice to compute. We must assume that Eve will also know about those parities. We can express the amount of leaked information in terms of leaked key bits (this is a logical abstraction - it does not indicate which specific key bits were leaked; it only provides a measure of how much information was leaked).

The amount of leaked information may be used by the privacy amplification phase that runs after the information reconciliation phase to determine how much amplification is needed.

Thus, the output of Cascade is the reconciliated key and the amount of leaked information.

I.VIII Cascade iterations

Now we are ready to start describing the guts of the Cascade protocol, i.e. to describe in detail how it actually works.

Let's define a single run of the Cascade protocol as Alice and Bob reconciliating (i.e. attempting to correct) a single key.

A single Cascade run consists of multiple iterations (these are also known as passes). Different variations of the Cascade protocol use different numbers of iterations. But we start by describing the original version of the Cascade protocol which uses four iterations.

Each Cascade iteration corrects some of the bit errors in the key. It is very probable (but not entirely certain) that all bit errors will have been corrected by the end of the last iteration.

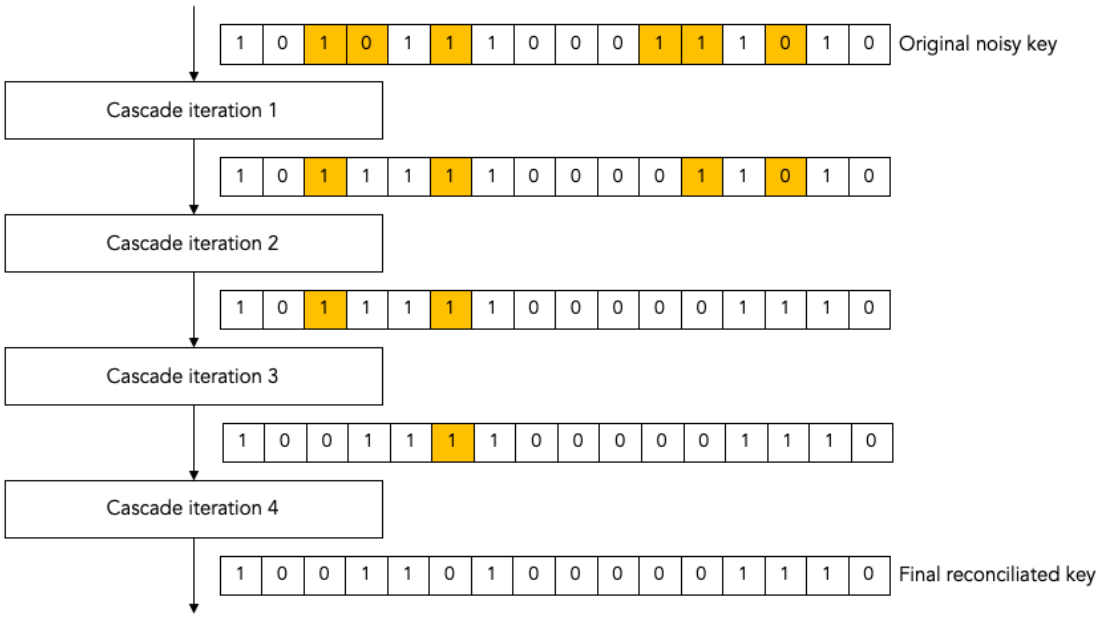


Figure 8: Cascade iterations

Note: for the sake of clarity, all of our diagrams show very small keys. In the above diagram, for example, we use 16-bit keys. In later diagrams we will use even smaller keys to make them fit in the diagram. In real life the keys can be much larger: tens of thousands or even hundreds of thousands of bits.

I.IX Key shuffling

At the beginning of each iteration, except the first one, Bob randomly shuffles the bits in the noisy key. Shuffling means randomly reordering the bits in the key.

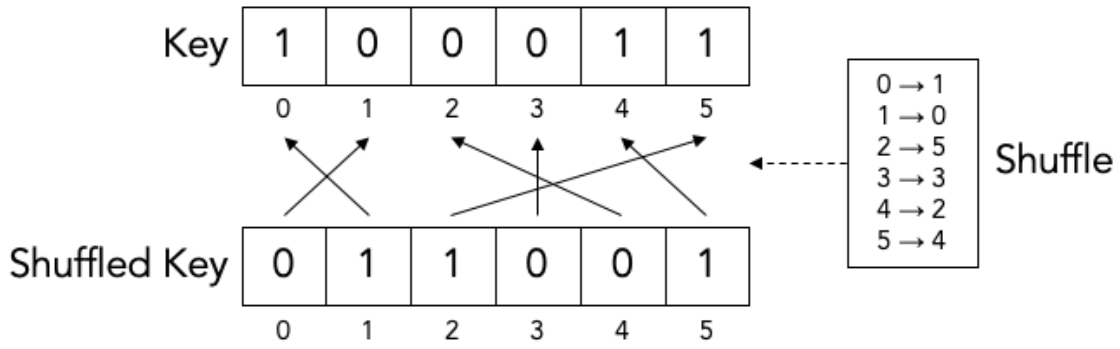


Figure 9: Key shuffling (I)

Later we will find out what the purpose of shuffling the key is. For now, we just point out that the shuffling is not intended to obfuscate the key for Eve. It is perfectly okay if the shuffling is only pseudo-random or even deterministic.

It is even okay if Eve knows what the shuffling permutation is (shown as the "Shuffle" in the above diagram) as long as the actual key values before ("Key") or after the

shuffling ("Shuffled key") are not divulged. In fact, Bob needs to inform Alice what the shuffle permutation for each Cascade iteration is. It is no problem if the information about the shuffle permutation is sent in the clear and Eve can observe it.

As we mentioned, Bob re-shuffles his noisy key at the beginning of each iteration except the first one:

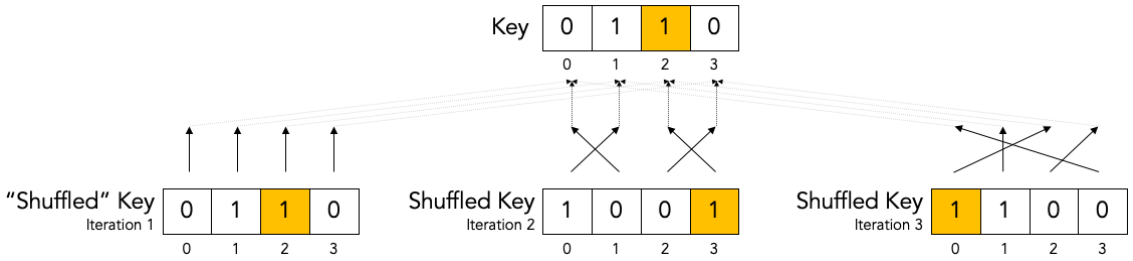


Figure 10: Key shuffling (II)

We put the word "shuffled" in quotes for the first iteration because they key is not really shuffled for the first iteration.

The important thing to observe is that any given bit in the original unshuffled key (for example bit number 2 which is marked in yellow) ends up in a different position in the shuffled key during each iteration.

I.X Creation of the top-level blocks

During each iteration, right after shuffling the key, Bob divides the shuffled key into equally sized blocks (the last block may be a smaller size if the key is not an exact multiple of the block size).

We will call these blocks top-level blocks to distinguish them from other types of blocks (the so-called sub-blocks) that will appear later in the protocol as a result of block splitting.

The size of the top-level blocks depends on two things:

- The iteration number i. Early iterations have smaller block sizes (and hence more blocks) than later iterations.
- The estimated quantum bit error rate Q. The higher the quantum bitter error rate, the smaller the block size

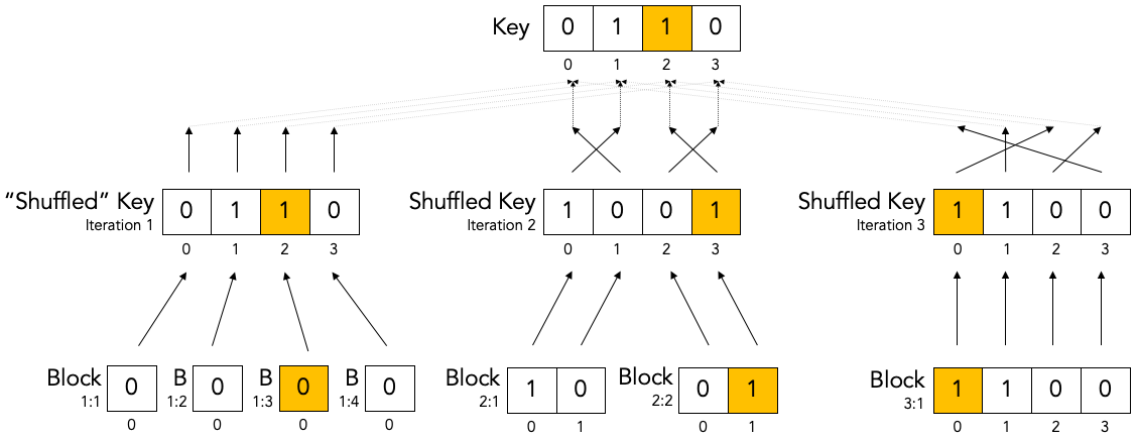


Figure 11: Creation of the top level blocks

Note: to make things fit on a page the block sizes are extremely small in this diagram. In real life, top-level blocks are much larger. Specifically, we would never see a single-bit top-level block.

There are many variations of the Cascade protocol, and one of the main differences between these variations is the exact formula for the block size ki as a function of the iteration number i and the quantum bit error rate Q.

For the original version of the Cascade protocol the formula is as follows:

k1= 0.73 / Q

k2= 2 * k1

k3= 2 * k2

k4= 2 * k3

Without getting into the mathematical details behind this formula, we can build up some intuition about the reasons behind it.

Later, we will see that Cascade is able to correct a single bit error in a block but is not able to correct a double bit error in a block.

If we pick a block size 1/Q for the first iteration, then each block will be expected to contain a single bit error on average. That is just the definition of bit error rate. If the bit error rate is 1 error per 100 bits, then a block of 100 bits will contain on average one error.

Now, if we use 0.73/Q instead of 1/Q then we will have slightly smaller blocks than that. As a result, we will have more blocks with zero errors (which are harmless) and fewer blocks with two errors (which are bad because they cannot be corrected).

On the other hand, we don't want to make the blocks too small, because the smaller we make the blocks, the more information is leaked to Eve. Knowing the parity over more smaller blocks allows Eve to know more about the key.

So, that explains the formula 0.73/Q for the first iteration. What about the doubling of the block size in each iteration?

Well, during each iteration Cascade corrects some number of errors. Thus, the remaining quantum bit error rate for the next iteration is lower (i.e. fewer error bits). This allows us to use a bigger block size for the next iteration, and still have a low probability of two (incorrigible) errors in a single block.

I.XI Detecting and correcting bit errors in each block

After having shuffled the key and after having split the key into blocks for a given iteration, Bob sets out on the task of determining, for each block, whether or not there are any bit errors in that block and, if so, to correct those bit errors.

The process of doing so is a bit complex because Bob needs to do it in such a way that he leaks a minimum of information to eavesdropper Eve who is watching his every move.

I.XII Computing the error parity for each top-level block: even or odd

I.XII.I Computing the current parity

Bob locally computes the current parity of each top-level block. This is a parity over some subset of bits in the shuffled noisy key that Bob has received from Alice. In the following example, Bob computes the parity of the 2nd top-level block in the 2nd iteration. That block has value 01 so its current parity is 1.

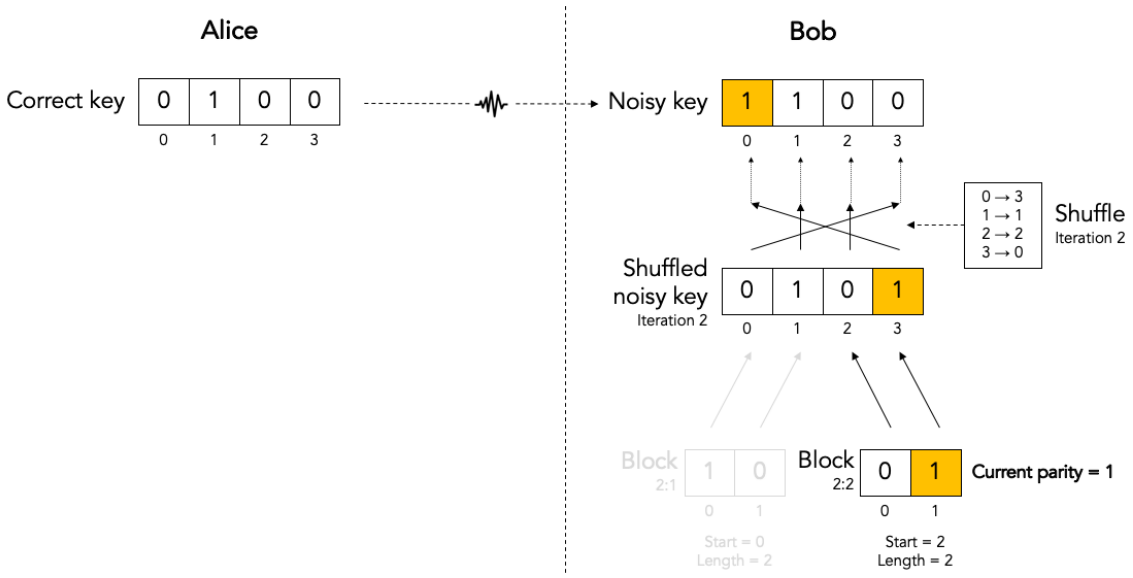


Figure 12: Computing the current parity

I.XII.II Computing the correct parity

Next, Bob wants to know Alice's perspective on the block parity. He already knows the "current parity" of the block in his own noisy key, but now he wants to know the "correct parity" of the same block in the Alice's correct key.

There is no way for Bob to compute the correct parity himself. Bob does not have access to the correct key, only Alice does. That statement is a little bit too strong. It turns out that there is an exception to this statement. Hold on until we discuss block parity inference (BPI) near the end of this tutorial.

The solution is simple: Bob simply sends an ask parity message to Alice. The purpose of this message is to ask Alice to compute the correct parity.

Let's first look at a very naive way of implementing the ask parity message, which is very inefficient, but which makes the concept very clear:

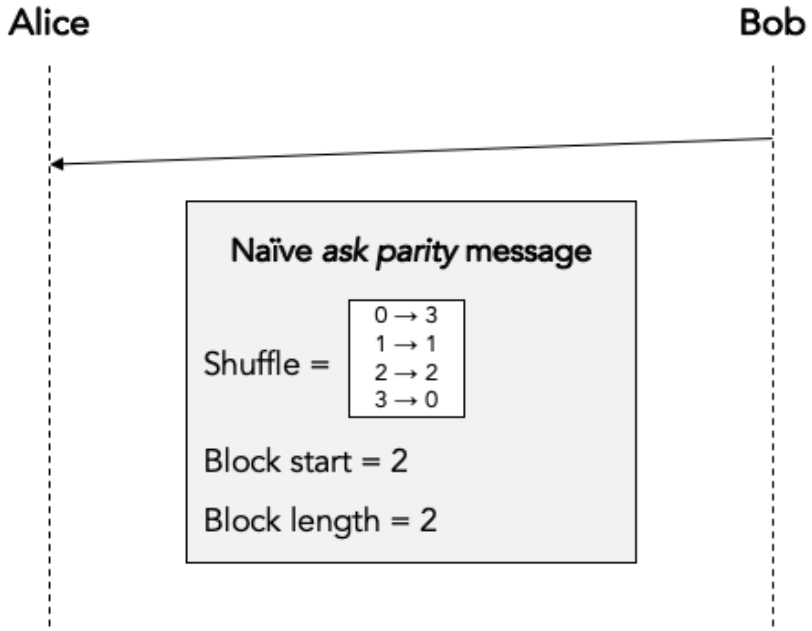


Figure 13: Computing the correct parity (I)

In this implementation Bob literally provides all the information that Alice needs to reconstruct the block and compute the correct parity:

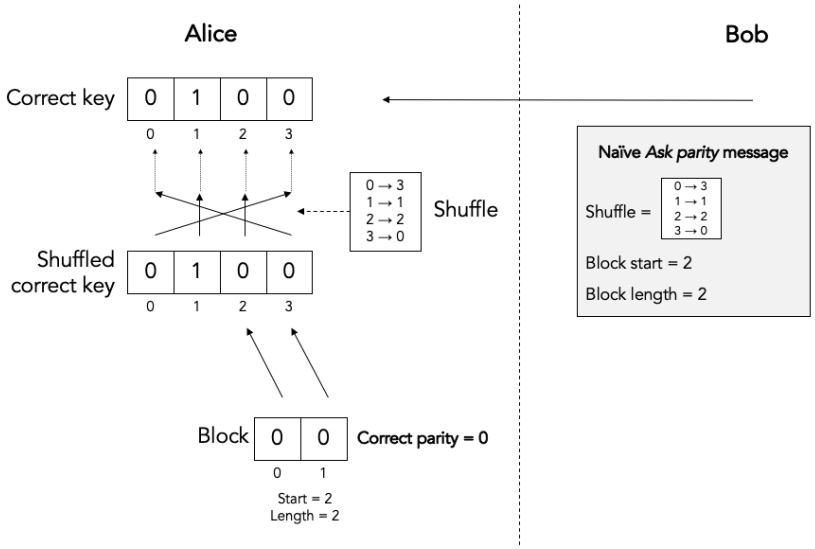


Figure 14: Computing the correct parity (II)

This is an inefficient way of computing the correct the parity. For one, the ask parity message can get very large because the shuffle permutation can get very large: here it is N numbers, where N is the key size (but it is easy to see that we could reduce N to the block size). Secondly, it requires Alice to spend processing time on reconstructing the shuffled key and the block.

An obvious optimization is for Bob to just cut to the chase and list the actual unshuffled key indexes over which Alice must compute the parity:

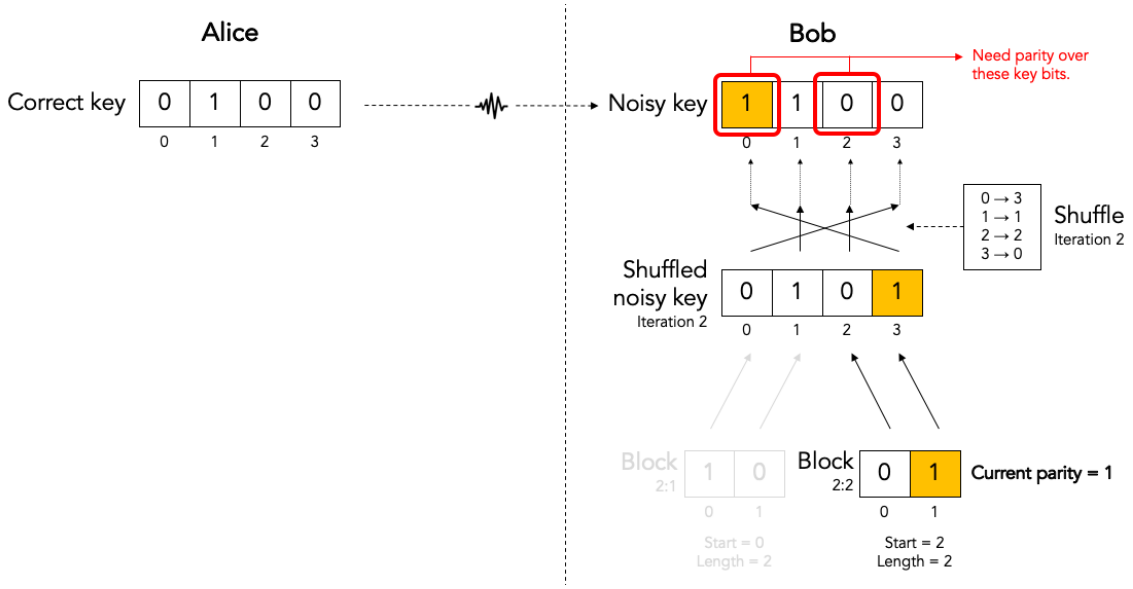


Figure 15: Computing the correct parity (III)

This allows Alice to just compute the correct parity without wasting CPU cycles on reconstructing the shuffled key and block:

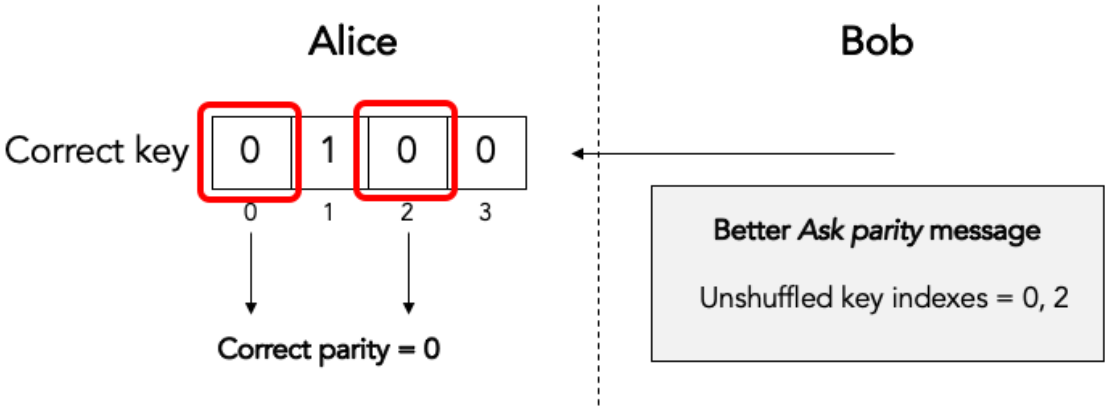


Figure 16: Computing the correct parity (IV)

In both cases the ask parity message does not leak any information about the key (yet): it does not contain the value of any key bit or any other information about the key bits themselves.

It turns out that there are even more efficient ways of implementing the ask parity message. These rely on the fact that the key is only shuffled once per iteration and we ask for block parities many times per iteration.

The only thing left to do is for Alice to send the correct parity back to Bob in a reply parity message:

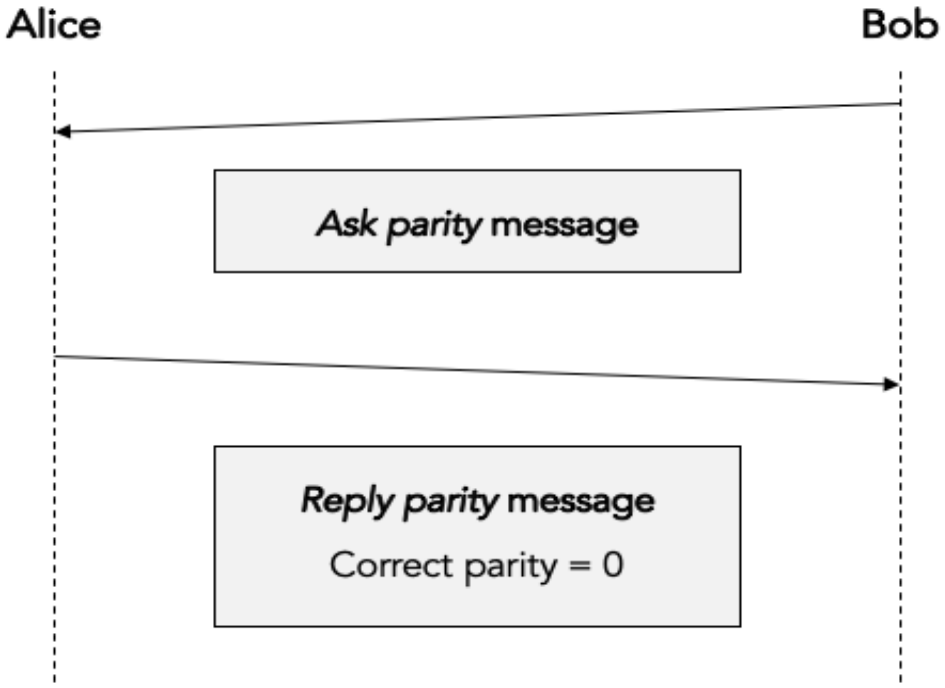


Figure 17: Computing the correct parity (V)

In any real implementation there would be additional fields in the reply message to associate the reply to parity message with the corresponding ask parity message, but we gloss over those details here.

Although neither Alice nor Bob ever divulges any actual key bits, the divulgence of the correct parity in the reply parity message does leak a little bit of information to Eve. This is easy to understand if we look at the number of values Eve has to try out in a brute force attack. If Eve knows nothing about N bits, she has to try out 2N values in a brute force attack. But if she knows the parity of those N bits, she only has to try out 2N-1 values.

I.XII.III Inference the error parity from current parity and the correct parity

At this point Bob knows both the correct parity and the current parity of the block.

Can Bob determine which bits in the block are in error? Well, no, he cannot. Can Bob at least determine whether there are any errors in the block or not? Well, no, he cannot determine even that.

What can Bob determine then? Well, Bob can determine whether there are an even or an odd number of errors in the block (the so-called error parity), by using the following table:

Current parity (Parity of block in Bob's noisy key)	Correct parity (Parity of block in Alice's correct key)	Error parity (Odd or even number of errors in block)
0	0	Even
0	1	Odd
1	0	Odd
1	1	Even

Figure 18: Inference the error parity from current parity and the correct parity

If the error parity is odd, then Bob knows that there is at least bit one error in the block. He doesn't know exactly how many bit errors there are: it could be 1 or 3 or 5 or 7 etc. And he certainly doesn't which key bits are in error.

If the error parity is even, then Bob knows even less. Remember that zero is an even number. So, there could be no (zero) errors, or there could be some (2, 4, 6, etc.) errors.

I.XIII Correcting a single bit error in top-level blocks with an odd number of bits

When Bob finds a block with an even number of errors, Bob does nothing with that block (for now).

But when Bob finds a block with an odd number of errors, Bob knows that there is at least one remaining bit error in the block. Bob doesn't know whether there is 1 or 3 or

5 etc. bit errors, but he does now there is at least one bit error and that the number is odd. For such a block, Bob executes the Binary algorithm. We will describe the Binary algorithm in the next section. For now, suffice it to say that the Binary algorithm finds and corrects exactly one bit error in the block.

Let's summarize what we have done so far.

In each iteration (except the first) Bob first shuffles the noisy key. Then he takes the shuffled key and breaks it up into blocks. Then he visits every block and determines the error parity for that block. If the error parity is even, he does nothing. If the error parity is odd, then he runs the Binary algorithm to correct exactly one-bit errors.

So, at the end of the iteration, Bob ends up with a list of blocks that all have an even error parity.

Some blocks already had an even error parity at the beginning of the iteration and Bob did not touch them.

Some blocks had an odd error parity at the beginning of the iteration and Bob ran the Binary algorithm to correct exactly one bit error. If you start with a block with an odd number of bit errors, and you correct exactly one bit error, then you end up with a block with an even number of bit errors.

Does this mean that we have removed all errors during this iteration? No, it doesn't. We only know that each block now contains an even number of errors. It could be zero errors. But it could also be 2, 4, 6, etc. errors.

During these iterations there is nothing more Bob can do to find or correct those remaining errors. But that doesn't mean those remaining error won't get corrected. Later we will see how a combination of reshuffling in later iterations and the so-called cascading effect will (with high probability) find and correct those remaining errors.

I.XIV The Binary algorithm

The Binary algorithm takes as input a block that has an odd number of errors. It finds and corrects exactly one bit error.

Bob is only allowed to run the Binary algorithm on blocks that have an odd number of errors. Bob is not allowed to run the Binary algorithm on a block that has an even number of errors (it is a useful exercise to figure out why not).

I.XIV.I Split block

The first thing Binary does is to split the block into two sub-blocks of equal size. We call these sub-blocks the left sub-block and the right sub-block. And we call the block that was split the parent block. If the parent block has an odd size, then the left sub-block is one bit bigger than the right sub-block.

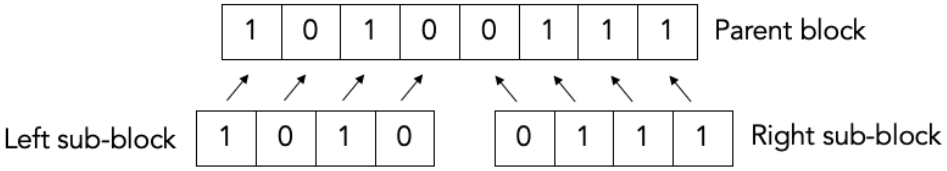


Figure 19: Split block (I)

Given the fact that we know for certain that the parent block has an odd number of errors, there are only two possibilities for the sub-blocks.

Either the left sub-block has an odd number of errors, and the right sub-block has an even number of errors, as in the following examples:

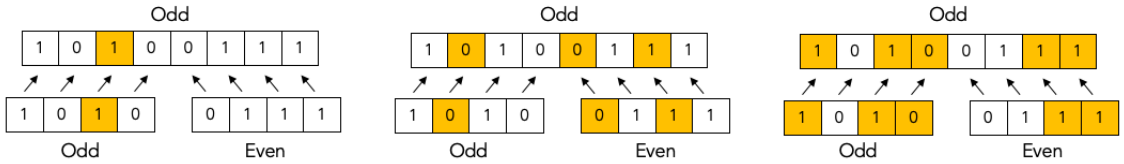


Figure 20: Split block (II)

Or the left sub-block has an even number of errors, and the right sub-block has an odd number of errors, as in the following examples:

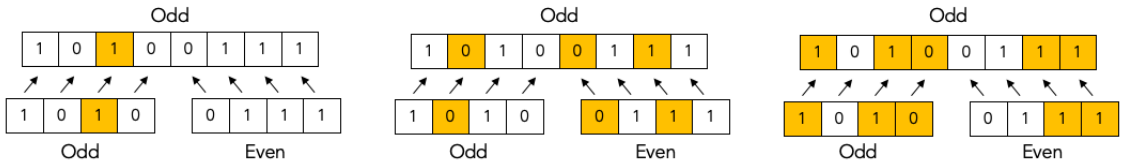


Figure 21: Split block (III)

It is simply not possible that both sub-blocks have an even number of errors and it is also not possible that both sub-blocks have an odd number of errors.

I.XIV.II Determine error parity of sub-blocks

Bob doesn't know which it is: Bob doesn't know whether the left sub-block or the right sub-block has an odd number of errors. All Bob knows at this point is that the parent block has an odd number of errors.

In order to find out, Bon sends an ask parity message to Alice to ask the correct parity for the left sub-block (only the left sub-block, not the right sub-block). When Alice responds with the correct parity for the left sub-block, Bob can compute the error parity (odd or even) for the left sub-block: he just needs to combine the locally computed current parity with the correct parity provided by Alice.

If the error parity of the left sub-block turns out to be odd, then Bob immediately knows that the error parity of the right sub-block must be even.

On the other hand, if the error parity of the left sub-block turns out to be even, then Bob immediately knows that the error parity of the right sub-block must be odd.

Either way, Bob knows the error parity of both the left sub-block and the right sub-block. Bob only asked Alice to give the correct parity for the left sub-block. Bob never asked Alice to provide the correct parity for the right sub-block. Bob can infer the correct parity and hence error parity for the right sub-block (and so can Eve, by the way).

By the way, this inference trick only works if Bob knows for a fact that the error parity of the parent block is odd. That is why Bob is not allowed to run the Binary protocol on a block with even error parity.

I.XIV.III Recursion

Once Bob has determined whether the left sub-block or the right sub-block contains an odd number of errors, Bob can recursively apply the Binary algorithm to that sub-block.

The Binary algorithm will keep recursing into smaller and smaller sub-blocks, until it finally reaches a sub-block that has a size of only a single bit.

If we have a sub-block whose size is one single bit and we also know that that same sub-block has an odd number of errors, then we can conclude that the single bit must be in error. We have found our single error bit that we can correct!

Let's look at a detailed example to get a better feel for how this works in practice:

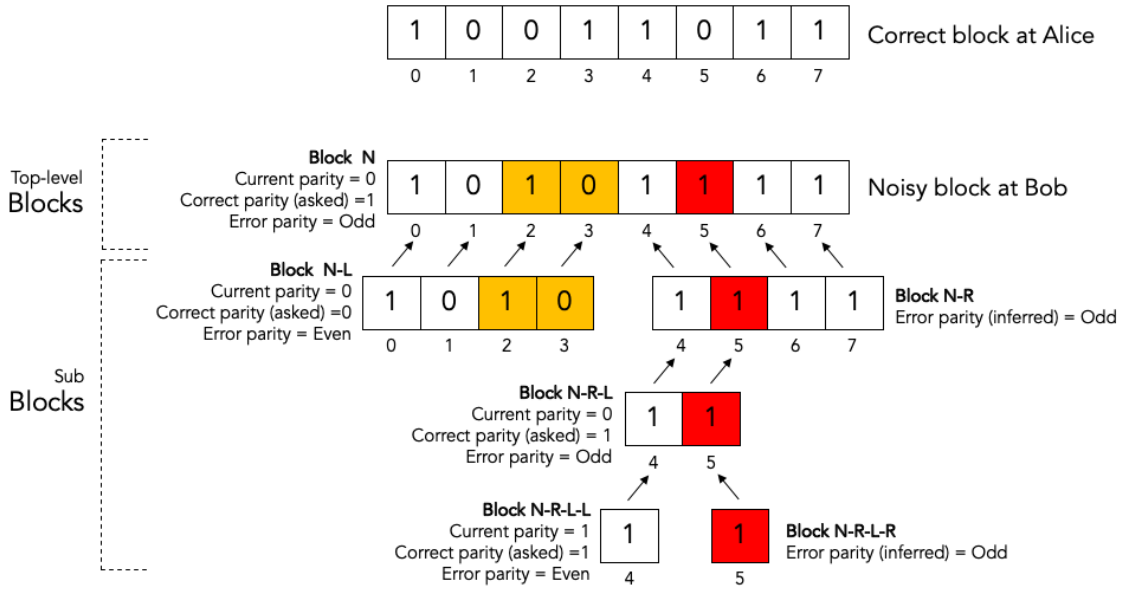


Figure 22: Recursion

Bob has received a noisy key from Alice, he has shuffled that key, and he has split the shuffled key into top-level blocks. The block labelled "noisy block at Bob" is one of those blocks. Let's just call it block N.

For the sake of clarity, we have shown corresponding block in the correct key at Alice as well. This is the block labelled "correct block at Alice".

As we can see, there are three-bit errors in the noisy top-level block, namely the coloured blocks at block indexes 2, 3 and 5.

We will now show how the Binary algorithm will detect and correct exactly one of those errors, namely the red one at block index 5.

The other two errors, the orange once at block index 2 and 3, will neither be detected nor corrected by the Binary algorithm.

Here are the steps:

1. Bob splits top-level block N into two sub-blocks: the left sub-block N-L, and the right sub-block N-R.
2. Bob determines the error parity for the blocks N-L and N-R as follows:
    a. Bob computes the current parity over block N-L and finds that it is 0.
    b. Bob asks Alice for the correct parity over block N-L and gets the answer that it is 0.
    c. Since the current parity and the correct parity for block N-L are the same, Bob concludes that the error parity must be even.
    d. Bob infers that block N-R must have odd error parity.
3. Bob recurses in the sub-block with odd error parity, which is block N-R.
4. Bob splits sub-block N-R into two sub-sub-blocks: the left sub-sub-block N-R-L, and the right sub-sub-block N-R-R.
5. Bob determines the error parity for the blocks N-R-L and N-R-R as follows:
    a. Bob computes the current parity over block N-R-L and finds that it is 0.
    b. Bob asks Alice for the correct parity over block N-R-L and gets the answer that it is 1.
    c. Since the current parity and the correct parity for block N-R-L are the different, Bob concludes that the error parity must be odd.
    d. Bob doesn't care about block N-R-R because he has already found his block to recurse into.
6. Bob recurses in the sub-sub-block with odd error parity, which is block N-R-L.
7. Bob splits sub-sub-block N-R-L into two sub-sub-sub-blocks: the left sub-sub-sub-block N-R-L-L, and the right sub-sub-sub-block N-R-L-R.
8. Bob determines the error parity for the blocks N-R-L-L and N-R-L-R as follows:
    a. Bob computes the current parity over block N-R-L-L and finds that it is 1.
    b. Bob asks Alice for the correct parity over block N-R-L-L and gets the answer that it is 1.
    c. Since the current parity and the correct parity for block N-R-L-L are the same, Bob concludes that the error parity must be even.
    d. Bob infers that block N-R-L-R must have odd error parity.
9. Bob notices that block N-R-L-R has a size of only one bit. Bob has found an error and corrects that error by flipping the bit!

I.XV What about the remaining errors after correcting a single bit error?

Now consider what happens after Bob has used the Binary protocol to correct a single bit error in a block.

Before the correction the block had an odd number of errors, which means that after the correction the block will contain an even number of errors. It may be error-free (have 0 remaining errors), or it may not yet be error-free (have 2, 4, 6, etc. remaining errors).

There is no way for Bob to know whether there are any errors left, and even if he did, Bob could not run the Binary algorithm on the same block again since the Binary algorithm can only be run on blocks with odd error parity. There is nothing left for Bob to do with the block, at least not during this iteration.

Thus, what about the remaining errors in the block (if any)? How will they get corrected? There are two mechanisms:

1. Reshuffling in later iterations.
2. The cascading effect.

We will now discuss each of these mechanisms in turn.

I.XVI The role of shuffling in error correction

The following diagram show the situation that Bob might find himself in at the end of some iteration, say iteration number N:

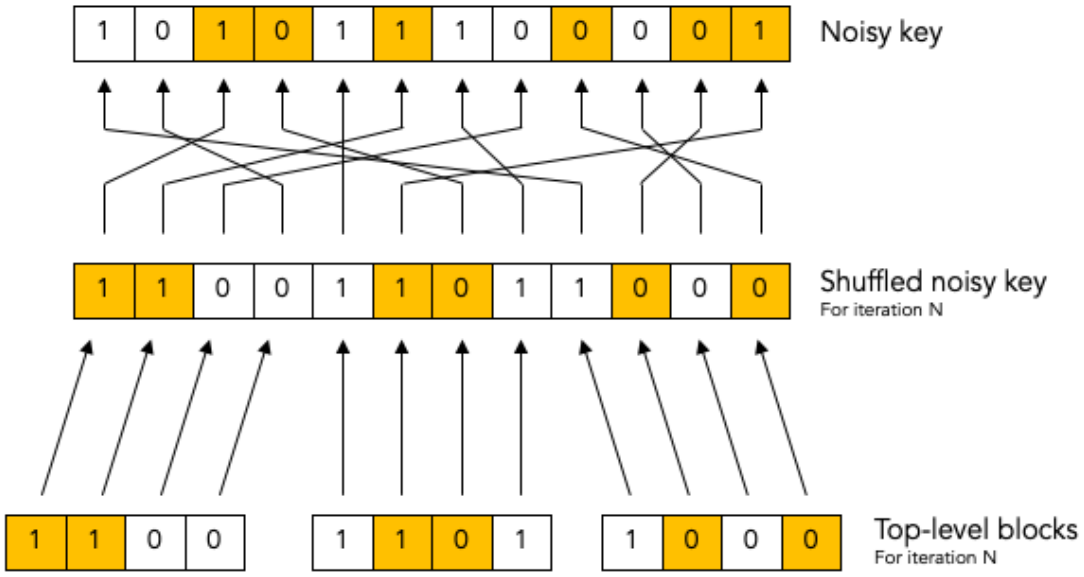


Figure 23: Role of the shuffling in error correction(I)

Maybe Bob already corrected a bunch of errors, but there are still six remaining errors left to correct.

Unfortunately, every top-level block contains an even number of remaining errors, so Bob is not able to make any progress during this iteration.

Bob has no choice but to move on to the next iteration N+1. In the next iteration, Bon reshuffles the keys (using a different shuffling order). Then he breaks up the reshuffled key into top-level blocks again but using bigger blocks this time.

We might up with something like this at the beginning of iteration N+1:

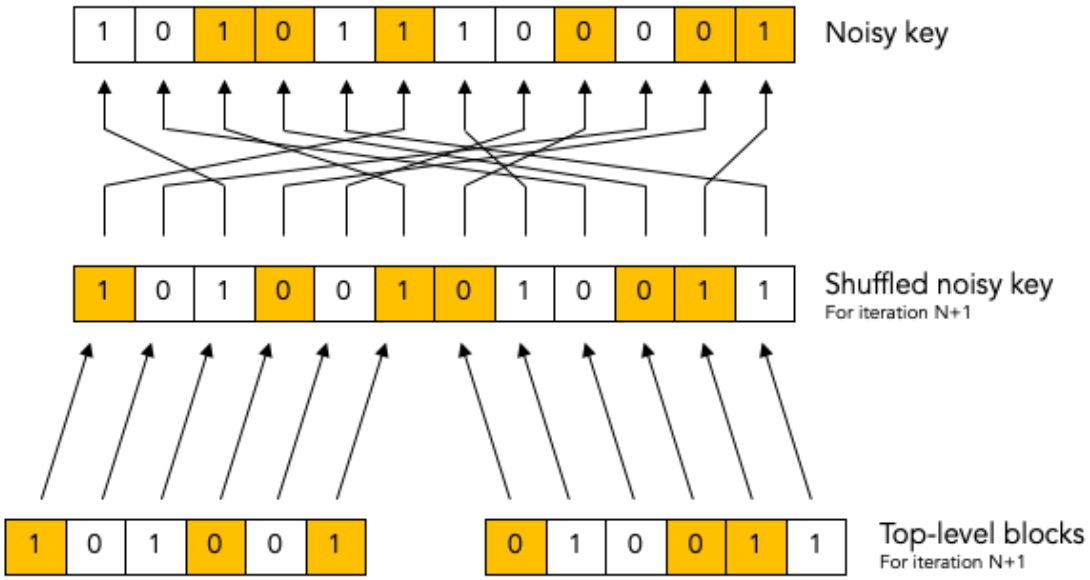


Figure 24: Role of the shuffling in error correction (II)

It is possible that at the beginning of iteration N+1 Bob ends up with some blocks that have an odd number of errors. Indeed, in this example Bob is quite lucky both remaining blocks have an odd number of errors (3 errors in each block).

Now Bob can make progress again: he can run the Binary algorithm on each block and remove exactly one error in each block. At the end of iteration N+1 there will be 4 errors remaining (2 in each block).

I.XVII The Cascade Effect

Let's keep going with same example a little bit more.

If you go through the steps of the Binary protocol, you will see that Bob will end up correcting the two bits that are marked in red during iteration N+1:

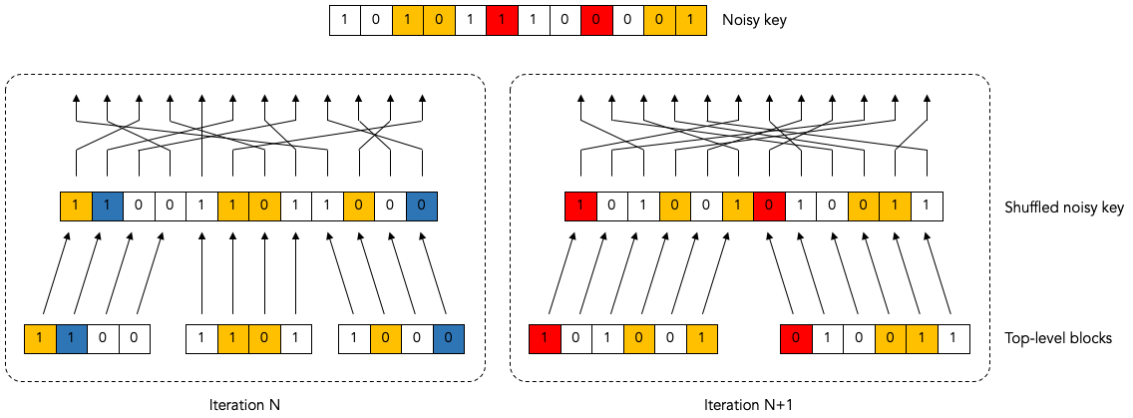


Figure 25: Cascade Effect (I)

If you follow the arrows from the red corrected bit in the shuffled blocks back to the top you can see which bits in the underlying unshuffled noisy key will end up being corrected (these are also marked in red).

But wait! If are going to be flipping bits (correcting errors) on de underlying unshuffled noisy key, then this is going to have a ripple effect on the shuffled blocks from earlier iterations.

In this example, we can see that flipping the red bits in iteration N+1 will cause the blue bits in iteration N to be flipped as a side-effect.

After all the red and blue bit flipping is done, we end up with the following situation:

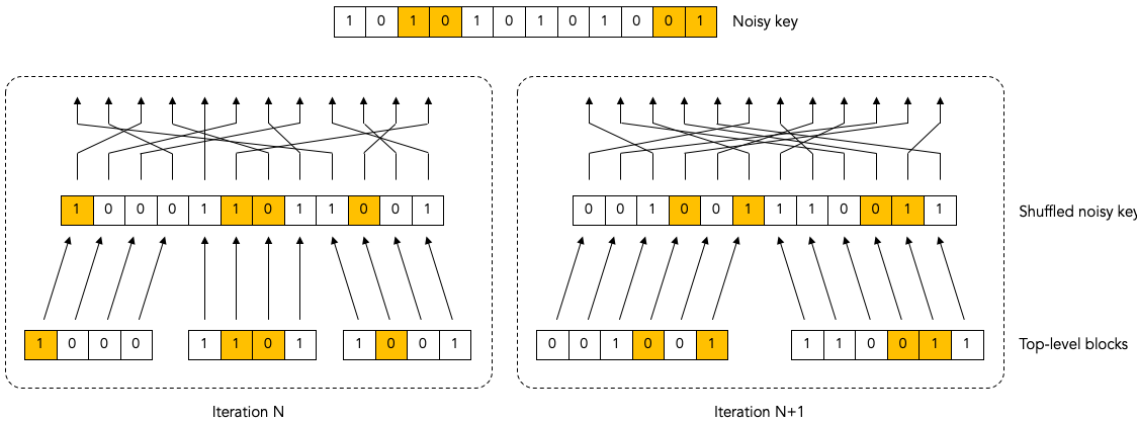


Figure 26: Cascade Effect (II)

As we discussed before, we have corrected two-bit errors in iteration N+1, and now there are 4-bit errors remaining.

And, as expected, we are now stuck as far as iteration N+1 is concerned. We can make no further progress in iteration N+1 because each block has an even number of errors.

But look! Because of the ripple effect on the previous iteration N, we now have two blocks in iteration N that now have an odd number of errors! Bob can go back to those iteration N blocks and re-apply the Binary protocol to correct one more error in them.

This ripple effect is what is called the cascade effect that gives the Cascade protocol its name.

The cascade effect is very profound and much stronger than it seems from our simple example.

Firstly, fixing an error in iteration N does not only affect iteration N, but also iterations N-1, N-2, …, 1.

Secondly, consider what happens when the cascade effect causes us to go back and revisit a block in an earlier iteration and fix another error there. Fixing that error in the earlier block with cause yet another cascade effect in other blocks. Thus, when we correct a single error, the cascade effect can cause a veritable avalanche of other cascaded error corrections.

I.XVIII Parallelization and bulking

Every time Bob asks Alice to provide the correct parity for a block, he sends an ask parity message and then waits for the reply to parity response message.

If Alice is in Amsterdam and Bob is in Boston, they are 5,500 km apart. The round-trip delay of the ask parity and reply to parity messages will be 110 milliseconds (the speed of light in fiber is 200,000 km/sec) plus whatever time Alice needs to process the message.

During the reconciliation of a single large key Bob can ask Alice for many parities (hundreds of ask parities for a 10,000-bit key, for example).

If Bob processes all blocks serially, i.e. if Bob doesn't start working on the next block until he has completely finished the Binary algorithm for the previous block, then the total delay will be very long. If we assume 200 ask parity messages, it will add up to at least a whopping 22 seconds. That is clearly too slow.

5,500 km was a bit extreme, just to make a point. But even for more realistic distances for quantum key distribution, say 50 km, the round-trip time delay is significant.

Luckily Bob does not have to process all blocks sequentially; he can do some parallel processing of blocks.

The lowest hanging fruit to parallelize the processing of the top-level blocks. At the beginning of each iteration, Bob shuffles the key and splits it up into top-level blocks. Bob can then send a single "bulked" ask parities (plural) message asking Alice for all the parities of all the top-level blocks in that iteration. Alice sends a single reply to

parities (plural) message with all correct parities. Then Bob can start processing all the top-level blocks.

But to get the full effect of parallelization Bob must do more. When Bob, in the process of running the Binary algorithm, gets to the point that he needs to ask Alice for the parity of a sub-block, Bob should not block and do nothing, waiting for the answer from Alice. Instead, Bob should send the ask parity message and then go work on some other block that has an odd number of errors "in parallel" while waiting for the answer to the first message. Working on multiple sub-blocks "in parallel" greatly reduces the total latency for an iteration.

If, in addition to reducing the latency, Bob also wants to reduce the number of ask parity messages, Bob can do "bulking" of messages. When Bob needs to ask Alice for the parity for some block B1, Bob can already start working on some other block B2. But instead of immediately sending the ask parity message for block B1, Bob can hold off for some time in anticipation of probably having to ask to parity for some other parities as well.

Note that the bulking of messages reduces the number of messages bit it does very little to reduce the volume (i.e. total number of bytes) of messages.

In the extreme case, Bob can hold off sending any ask parities message until he can positively not make any more progress before he gets an answer. But that would increase the latency again because it would force Bob to sit idle not doing anything.

The sweet spot is probably to hold off sending ask parity messages for only a fixed delay (similar to what the Nagle algorithm does in TCP/IP).

Appendix II: Visual Studio Code Setup

In this appendix will be shown the setup for Visual Studio Code on Windows, (VSC-Setup, www).

II.I Installation

For installation of Visual Studio Code is necessary following the next steps:

1. Download the Visual Studio Code installer for Windows.
2. Once it is downloaded, run the installer (VSCodeUserSetup-{version}.exe). This will only take a minute.
3. By default, VS Code is installed under C:\users\{username}\AppData\Local\Programs\Microsoft VS Code.

Alternatively, you can also download a Zip archive, extract it and run Code from there.

Note: .NET Framework 4.5.2 or higher is required for VS Code. If you are using Windows 7, make sure you have at least .NET Framework 4.5.2 installed. You can check your version of .NET Framework using this command, reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\full" /v version from a command prompt.

Tip: Setup will add Visual Studio Code to your %PATH%, so from the console you can type 'code .' to open VS Code on that folder. You will need to restart your console after the installation for the change to the %PATH% environmental variable to take effect.

II.II User setup versus system setup

VS Code provides both Windows user and system level setups. Installing the user setup does not require Administrator privileges as the location will be under your user Local AppData (LOCALAPPDATA) folder. User setup also provides a smoother background update experience.

The system setup requires elevation to Administrator privileges and will place the installation under Program Files. This also means that VS Code will be available to all users in the system.

II.III Updates

VS Code ships monthly releases and supports auto-update when a new release is available. If a window is prompted by VS Code, accept the newest update and it will be installed (it won't need to do anything else to get the latest bits).

Appendix III: C++ Compiler and Debugger

For compiling and debugging in C++ have been utilised GCC C++ compiler (g++) and GDB debugger from mingw-w64 (Mingw-w64, www).

For installing mingw-w64, it's necessary to download the installer from this page:

https://sourceforge.net/projects/mingw-w64/

Once the installer has been downloaded, follow the steps for install it.

For configuring Visual Studio Code with mingw-w64, check the next tutorial:

https://code.visualstudio.com/docs/cpp/config-mingw

Acronym

IDE: Integrated Development Environment

QKD: Quantum Key Distribution

References

(Microsoft, www) https://www.microsoft.com/es-es

(VSC-Microsoft) https://code.visualstudio.com/

(JetBrains, www) https://www.jetbrains.com/

(CLion-JetBrains, www) https://www.jetbrains.com/clion/

(CodeBlocks, www) https://www.codeblocks.org/

(Eclipse, www) https://www.eclipse.org/downloads/

(VSC-Wikipedia, www) https://en.wikipedia.org/wiki/Visual_Studio_Code

(CLion-LinuxAdic, www) https://www.linuxadictos.com/clion-un-ide-multiplataforma-para-c-y-c.html

(CLion-ReleaseDate, www) https://blog.jetbrains.com/clion/2015/04/clion-1-0-has-finally-arrived/

(CLion-MainPage,www) https://www.jetbrains.com/clion/

(CodeBlocks-Wikipedia, www) https://es.wikipedia.org/wiki/Code::Blocks

(Eclipse-Characteristics, www) https://www.softwaretestinghelp.com/java/java-development-using-eclipse-ide/

(OpenMP, www) https://www.openmp.org/

(AndreReisWork, Thesis) *Quantum Key Distribution Post Processing - A study on the Information Reconciliation Cascade Protocol*

(Cascade-protocol, www). https://cascade-python.readthedocs.io/en/latest/protocol.html

(VSC-Setup, www) https://code.visualstudio.com/docs/setup/windows

(Mingw-w64, www) https://www.mingw-w64.org/