Connecting Quantum Bits

Building the Classical Channel between Alice and Bob in Quantum Key Distribution
Quantenbits Verbinden
Umsetzung des Klassischen Kanals zwischen Alice und Bob im Quanten-Schlüsselaustausch
Bachelorarbeit von Nico Alt
Tag der Einreichung: 1. August 2022

1. Gutachten: Prof. Dr. Thomas Walther
2. Gutachten: Maximilian Tippmann, M. Sc.

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Physik

Institut für Angewandte Physik

Laser und Quantenoptik

Connecting Quantum Bits
Building the Classical Channel between Alice and Bob in Quantum Key Distribution

Bachelorarbeit von Nico Alt

Tag der Einreichung: 1. August 2022

Darmstadt

Für alle, die mich einfach mal machen lassen haben.
Para quienes me concedieron la libertad de encontrar mi propio camino.

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Nico Alt, die vorliegende Bachelorarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Darmstadt, 1. August 2022

N. Alt

Contents

Glossary **ix**

1. Introduction **1**

2. Preliminaries **3**
- 2.1. Modern Cryptography . . . 3
  - 2.1.1. Encryption with One-Time Pads . . . 3
  - 2.1.2. Encryption with AES . . . 4
  - 2.1.3. Authentication with RSA . . . 4
  - 2.1.4. Quantum Key Distribution with BBM92 . . . 5
- 2.2. Computer Networks . . . 9
  - 2.2.1. Network Protocols and Layers . . . 9
  - 2.2.2. Network Sockets . . . 9

3. Setup **11**
- 3.1. Experimental Setup . . . 11
- 3.2. Analysis of Measurement Data . . . 14
  - 3.2.1. Offset Calibration . . . 14
  - 3.2.2. Sorting Timestamps into Time Bins . . . 15
  - 3.2.3. Subsequent Analysis . . . 15
- 3.3. Inter-Process Communication . . . 16

4. Details on the Implementation **17**
- 4.1. Modifications to the Previous Software . . . 17
- 4.2. Software Architecture . . . 18
- 4.3. Inter-Process Communication . . . 20
  - 4.3.1. Communication between Alice, Bob, and the Source . . . 20
  - 4.3.2. Communication between Java and Python . . . 23
- 4.4. Timestamp Analysis . . . 24
- 4.5. Key Sifting . . . 25
- 4.6. Error Correction and Privacy Amplification . . . 28

5. Results **31**
- 5.1. Inter-Process Communication . . . 31
  - 5.1.1. Stopping the Memory Leak . . . 31
  - 5.1.2. Security Considerations on Serialization . . . 33
- 5.2. Timestamp Analysis . . . 34
  - 5.2.1. Detector Offsets . . . 34
  - 5.2.2. Security Considerations on Time-Bin Sorting . . . 36

5.3. Sifted and Secure Keys 36

6. Summary and Outlook **41**

A. Setting up the Quantum Network Control Center **43**

B. Configuring the Quantum Network Control Center **45**

C. Code Sections where Untrusted Data is Deserialized **47**

List of Figures **49**

Bibliography **51**

Glossary

AES Advanced Encryption Standard.

BBM92 QKD protocol published by C. Bennett, G. Brassard, and N. Mermin in 1992.

BER Bit Error Rate.

CBC Cipher Block Chaining.

CROSSING Research center at TU Darmstadt focussing on advances in cryptography.

DES Data Encryption Standard.

ECB Electronic Codebook.

FWM Four-Wave Mixing.

IP Internet Protocol.

IPv4 Version 4 of IP, using 32-bit addresses.

IV Initialization Vector.

LDPC Low-Density Parity-Check.

MITM Man-in-the-middle.

network socket Application programming interface for data exchange via the network.

One-Time Pad Encryption algorithm using XOR where keys must only be used once.

OSI Open Systems Interconnection.

QBER Quantum Bit Error Rate.

QKD Quantum Key Distribution.

QNCC Quantum Network Control Center.

RSA Encryption algorithm published by R. Rivest, A. Shamir, and L. Adleman in 1977.

SPDC Spontaneous Parametric Down-Conversion.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

XOR exclusive or.

1. Introduction

Having been used by humans for millennia [1], cryptography has become one of the main building blocks of today's societies. Cryptography is omnipresent: from securing communications over authenticating digital payments to storing data securely, processes of cryptography such as encryption and authentication are constantly used throughout everyone's daily life.

With the advent of usable quantum computers getting closer each day, the security of cryptography used to date is at risk [2]. Algorithms designed to run on quantum computers, like Shor's Algorithm [3], offer polynomial-time solutions to problems that could previously only been solved in exponential time, like prime factorization and discrete logarithms. Thus, they become a serious threat to break all asymmetric cryptography popular today [4]. Research on so-called *post-quantum cryptography* [5] or *quantum-resistant cryptography* [6] has become a huge field where mathematical problems are researched that serve as replacements for existing *trapdoor functions*, i.e., functions that are easy to compute but hard to invert without the knowledge of a secret [7].

An alternative to investigating mathematical problems regarding their security against attacks with quantum computing power is making use of the laws of quantum mechanics not only to build quantum computers but also to secure cryptography, a technique called *quantum cryptography* [8]. Most of this quantum cryptography today is used to agree on a shared secret between two parties and is thereby called Quantum Key Distribution (QKD). In QKD, measurements are done on quantum objects exchanged between two communicating parties, in cryptography usually called *Alice* and *Bob*. Due to the laws of quantum mechanics, a *secret key* can be obtained from the results of these measurements, because an attacker trying to eavesdrop this process would disturb the quanta's states, essentially introducing a high amount of errors in the parties' keys that could be easily detected. If QKD successfully yielded a secret key, it can be used with symmetric-cryptography algorithms like the Advanced Encryption Standard (AES) or One-Time Pads to, e.g., encrypt and sign data.

At TU Darmstadt, research on QKD is done as part of CROSSING, a joint project between computer science and physics, and more specifically as part of the project area *P4 – Quantum Key Hubs*. In contrast to existing 2-party QKD setups, the setup at TU Darmstadt enables scaling up to more than 100 communicating parties in a star-shaped QKD network with demonstrated ranges of up to 100 km under real-world conditions using the so-called BBM92 protocol, which uses entangled pairs of quanta to obtain a shared secret key. In collaboration with *E1 – Secure Integration of Cryptographic Algorithms* and *E3 – Secure Refinement of Cryptographic Algorithms* of CROSSING, the implementation of this QKD setup is analyzed on correct implementation of cryptographic primitives and security against side-channel attacks, respectively.

Research on QKD at TU Darmstadt started many years ago with Oleg Nikiforov building the first working QKD system at TU Darmstadt using BBM92, followed by Erik Fitzke and Maximilian Tippmann currently working on extending this to multiple parties under real-world conditions [9]. Research on BBM92 with the *time-bin coding* used in this setup is important because unlike polarization-based QKD protocols it works even with large distances and harsh environments. The works of most interest for this thesis were done by Lucas Bialowons in his Master's thesis *Completion of a 4-Party Time-bin Entanglement QKD System* [10], by

Jendrik Seip in his Master's thesis *Error Correction and Key Post-Processing for Quantum Key Distribution* [11], and by Hühne, Schumann, Hernandez, Petri, and Dentler on the Quantum Network Control Center (QNCC) program [12]. In these works, they built a QKD setup fully functional in a centralized version, implemented an algorithm that allows to correct errors in the resulting keys, and developed a software that allows the communication between all communicating parties, respectively.

Based on all this work, this Bachelor thesis focusses on implementing the protocols and algorithms of BBM92 for parties on separate devices, allowing them to generate secret keys based on the measurement data produced by the QKD setup. This work gets the QKD system at *P4 – Quantum Key Hubs* a lot closer to a very first real-world usage. The result of this thesis' work is version $0.1.0$ of QNCC that can be found in the GitLab group of *P4 – Quantum Key Hubs*.

2. Preliminaries

Before describing the setup of this Bachelor thesis, details on the implementation of its software, and results, let us shortly revise some fundamental concepts of modern cryptography and computer network technologies.

2.1. Modern Cryptography

While Caesar already used cryptography to try to make his letters confidential [1], this *classical cryptography* was mostly about only this application of cryptography and did not receive formal verification of its secureness. With the advent of *modern cryptography*, algorithms were invented that allowed other usages like authentication and signatures. [13]

Many processes today require some type of cryptography. To ensure confidentiality when communicating with lawyers or authorities, messages can be encrypted. When doing banking transactions via the Internet, browsers use authentication to make sure that a website is run by some trusted organization and not by attackers. Even with quantum computers breaking some of these cryptography algorithms used to date [2], "classical" modern cryptography is still required. The most popular type of quantum cryptography, Quantum Key Distribution (QKD), is about establishing a shared secret between two parties, but in order for it to be useful, classical algorithms like One-Time Pads and Advanced Encryption Standard (AES) need to be used to, e.g., encrypt data using this shared secret.

2.1.1. Encryption with One-Time Pads

One-Time Pad encryption is one of the simplest algorithms to encrypt data, yet it is the only one proven to be *information-theoretically secure* [14]. This means that no matter how much time and computational resources attackers invest, they will never be able to find the plain text belonging to a given cipher text.

Bits can be represented by elements of $\mathbb{F}_2$, the finite field with only two elements 0 and 1. To encrypt a bit string $p \in \mathbb{F}_2^n$ called *plain text* and consisting of n bits, n bits of random data are needed. They form the key $k \in \mathbb{F}_2^n$ and are allowed to be used as a key only once. With the addition on $\mathbb{F}_2^n$ being defined element-wise in table 2.1, the plain text p can be encrypted to a *cipher text* with $c = p + k$. Since $k + k = 0$ for any $k \in \mathbb{F}_2^n$, decrypting c is done with $c + k = p$. Especially in the context of cryptography, One-Time Pads are defined using exclusive or (XOR) which is equal to additions on $\mathbb{F}_2$.

Using QKD in combination with One-Time Pads makes the whole process information-theoretically secure. However, due to the requirement of k being of the same size as the plain text p and that k is allowed to be used only once, in practice more sophisticated algorithms like AES are used that allow smaller key sizes and the reuse of keys.

Table 2.1.: Definition of addition on $\mathbb{F}_2$ and XOR.

$+$	0	1
0	0	1
1	1	0

2.1.2. Encryption with AES

As the successor specification to the now broken Data Encryption Standard (DES) [15], the Advanced Encryption Standard (AES) is a *symmetric block cipher* first published by Joan Daemen and Vincent Rijmen in 1998 [16]. AES is a symmetric-key algorithm which means that it uses the same key to encrypt and decrypt data. To establish this key between two communicating parties, QKD can be used. Being a block cipher, AES does not encrypt and decrypt the whole plain text as a single piece, but instead divides it into blocks of a certain size, usually 256 bits.

Ultimately, AES still uses XOR just like One-Time Pads to generate a cipher text from the plain text and the key. However, three different algorithms are used during the process of encryption that allow AES to reuse keys that are of smaller sizes. These algorithms are applied in sequential order for numerous times as part of so-called *rounds*. For a block size of 256 bits, 14 rounds are used [16], i.e., on each 256-bit block of plain text data the three algorithms and a slightly sophisticated One-Time Pad version are applied 14 times. In the last round, however, only two of the three algorithms are applied.

When encrypting with AES, it is important not to simply divide a plain text into pieces of 256 bit and encrypt each piece independently, as this is inherently insecure. Doing so would allow attackers to learn about the structure of a plain text messages since equal plain text blocks get encrypted to equal cipher text blocks. Instead, a different *block cipher mode of operation* needs to be used in place of this simple algorithm called the Electronic Codebook (ECB) mode. One such secure algorithm is given by the Cipher Block Chaining (CBC) mode. With CBC, the plain text of a block is XORed with the cipher text of the previous block before encrypting it with AES. By this, it is ensured that equal plain text blocks do not result in equal cipher text blocks, since the cipher text of one block now depends on all previous blocks. For the first block, an Initialization Vector (IV) is used to XOR it with the first plain text block. This IV must be different for each encryption with AES that uses the same key, since otherwise equal beginnings of plain text messages would result in equal beginnings of cipher text messages. [7]

2.1.3. Authentication with RSA

Symmetric-key algorithms like AES presented in the previous section require parties that want to exchange encrypted messages to establish a secret key that is only known to these parties. Instead of this, asymmetric-key algorithms can be used that drop this requirement of having to exchange secret keys beforehand. With asymmetric-key algorithms, every party has a *public* and a *private key*. The public key can be shared with anyone and is used to encrypt a message for the party. To decrypt this message, the private key is needed that is only known to the party. One example for such an algorithm of *asymmetric cryptography* is RSA [17]. Named after its founders Ron Rivest, Adi Shamir, and Leonard Adleman, RSA is based on the problem of factoring a product of two large prime numbers. Following this idea, the public key in RSA is generated from the product of two large prime numbers, while the private key is generated from the knowledge of these prime numbers. It is safe to publish this public key, since no algorithm is known that factors the product of prime numbers in polynomial time.

For RSA, the same mathematical operation is used to encrypt and decrypt a message. To encrypt a message, the public key is used during this operation, while decrypting the message is done with the private key. By interchanging the keys used during encryption and decryption, RSA can be used to sign messages. When Alice wants to sign a message, first she uses a *hash function* to hash this message. A hash function is an irreversible function that takes a message of arbitrary size and produces a message of fixed size. Alice encrypts the hash result using her private key, not her public key, to obtain the *signature*. Bob verifies the message by decrypting the signature with Alice's public key and comparing the decrypted signature to his own hash result of the message. If they match, the message was signed by Alice, since only she knows her private key.

While there are no known polynomial-time algorithms that break RSA on classical computers, RSA can be broken by quantum computers using Shor's algorithm [3]. Shor's algorithm allows to factor the product of prime numbers in polynomial time and thus break RSA. Until this happens, RSA may be used to authenticate a QKD session by signing and verifying all messages exchanged, since *authenticity* only needs to be assured at the time of running a QKD session. Man-in-the-middle (MITM) attacks — where an attacker pretends to be Alice or Bob by breaking the authentication — cannot happen afterwards, in contrast to attacks on the *confidentiality* that may get broken in any moment in the future. The latter would result in confidential data getting revealed. Nevertheless, it is recommended to already start now using authentication algorithms that are not known to be broken by quantum computers, such as the ones researched in post-quantum cryptography.

2.1.4. Quantum Key Distribution with BBM92

With quantum computers breaking mostly *asymmetric cryptography*, *symmetric cryptography*, like AES, can still be used to, e.g., encrypt data. However, with the lack of secure asymmetric cryptography, some method is needed to establish a shared secret key between two or more parties. One way of doing this is using the Quantum Key Distribution (QKD) protocol BBM92.

First published by Charles H. Bennett, Gilles Brassard, and N. David Mermin in 1992 [18], BBM92 is a variant of QKD protocols that is based on an idea first presented by Artur K. Ekert in 1991 [19]. Ekert noted that nonlinear optical processes like Spontaneous Parametric Down-Conversion (SPDC) or Four-Wave Mixing (FWM) could be used to create two entangled quanta that contain secret information to be obtained by Alice and Bob during measurements. In crystals featuring either SPDC or FWM, there is a non-zero probability that a single photon entering the crystal results in two photons leaving the crystal. These photons are called *entangled* because they can only be described by a single wave function in quantum mechanics, not by two independent ones, and thus measurements with them, even in distinct places, will yield identical results.

The BBM92 setup at *P4 – Quantum Key Hubs* uses imbalanced interferometers to encode secret information via a so-called *time-bin coding* [20, 21] which makes use of the entanglement between photons leaving a crystal featuring SPDC. Figure 2.1 shows a schematic setup containing three Mach-Zehnder interferometers: one interferometer at the photon source and two interferometers on the receivers' side, Alice and Bob, respectively. The interferometers are imbalanced, i.e., the lengths of the two arms of a single interferometer are not equal. By this, a single photon entering an interferometer and taking either the left or the right arm arrives at different times at the interferometer's output. This results in an arrival histogram as shown in figure 2.2. There, photons taking the short paths in both the source's and, e.g., Alice's interferometers will be detected as part of the left so-called *satellite peak*. Similarly, photons taking the long paths in both interferometers will create the right satellite peak. In the *central peak*, photons are detected that either take the short path in the source's interferometer and the long path in Alice's interferometer, or vice versa with the long path at the source and the short path on Alice's side. It is therefore that the central peak is roughly double the size of the satellite peaks.

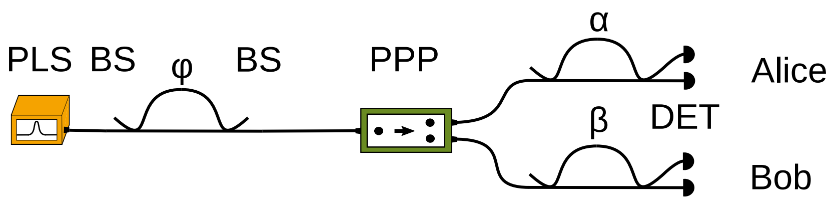

Figure 2.1.: QKD setup showing the source and two receivers. The source, Alice, and Bob have identical imbalanced interferometers with phase delays φ, α, and β, respectively. The photons leaving the *pulsed laser source* (PLS) first hit a *beam splitter* (BS) where they take either the short or the long path of the interferometer. Then they enter the *photon-pair production* (PPP) where two entangled photons travel to the interferometers of Alice and Bob and each get detected by two single-photon detectors (DET) at the interferometers' outputs. [9]

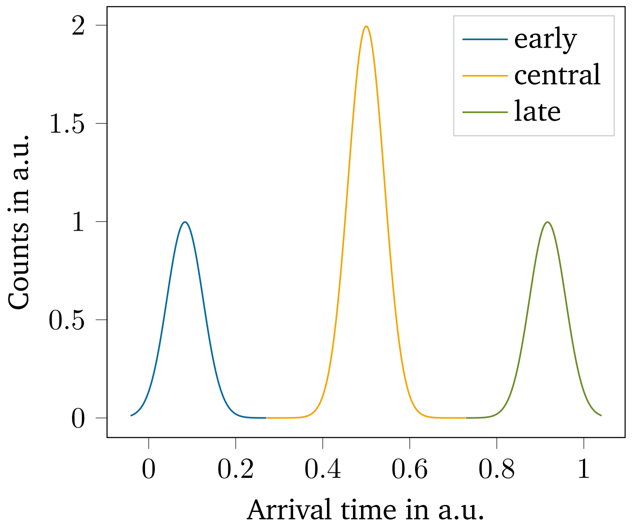

Figure 2.2.: The simplified histogram shows the photon count as a function of their arrival time at the parties' detectors. A major so-called *central peak* can be seen, surrounded by two minor *early* and *late satellite peaks*.

Whenever a photon enters an interferometer, it has a $50\,\%$ probability of taking either the long or the short path. If a photon takes the short path in the source's interferometer and gets split up into an entangled pair of photons, they could take the short path in Alice's interferometer and the long path in Bob's interferometer. Alice and Bob cannot extract secret information from such an event and will therefore discard it in the later key sifting. However, if both photons take the short path in Alice's and Bob's interferometers, both Alice and Bob will detect them as part of the early satellite peak. The same happens when the photons take the long path in all interferometers — Alice and Bob will detect them as part of the late satellite peak.

The case left is when the photons take different paths in the source's and receivers' interferometers. Due to a quantum effect called *Franson interference* [22], the probability for such photons to reach the same output at both Alice's and Bob's interferometers depends on the phase delays φ, α, and β introduced in the source's and receivers' interferometers (see fig. 2.1) [23]:

$$P(A_i, B_i) \propto \sin^2\left(\frac{\alpha+\beta-\varphi}{2}\right) , \tag{2.1}$$

where A_i is Alice's detector and B_i is Bob's detector at same outputs. If entangled photons always reach same outputs in both interferometers, the probability for photons reaching different outputs is 0. Consequently, this latter probability is given by

$$P(A_i, B_j) \propto \cos^2\left(\frac{\alpha+\beta-\varphi}{2}\right) . \tag{2.2}$$

Here, different indices i and j in A_i and B_j indicate that the photons reach detectors at different outputs. To maximize the probability for photons to always reach different outputs when detected as part of the central peak, the phase delays in the interferometers may add up to:

$$\alpha+\beta-\varphi = 2\pi n; n \in \mathbb{N} . \tag{2.3}$$

The interferometers are built in such a way that the phase delays add up to a value close to $2\pi n$. Nevertheless, the sum slightly deviates from $2\pi n$ since the interferometers cannot be constructed with perfect precision. Therefore, the interferometers are heated to further adapt the phase delays with thermal expansion. The quantity used to control the temperature is the correlation coefficient [10]

$$C = \frac{N_{\text{corr}} - N_{\text{cross-corr}}}{N_{\text{corr}} + N_{\text{cross-corr}}} = P(A_i, B_i) - P(A_i, B_j) . \tag{2.4}$$

It consists of the two aforementioned probabilities $P(A_i, B_i)$ and $P(A_i, B_j)$, and the event counts N_{corr} and $N_{\text{cross-corr}}$ where both Alice and Bob measure an event in the central peak at the same output of their interferometers or at different outputs, respectively. By adjusting the temperature of one party's interferometer, a value close to $C = -1$ is achieved which means that events in the central peak are fully anti-correlated. For counts in the central peak, bit values 0 and 1 are assigned depending on which detector clicked. Therefore, a correlation coefficient of $C = -1$ is preferred over $C = 1$ because like this the ratio of 0 and 1 bits in the sifted key does not depend on the relative detection rates of the detectors.

To derive a shared secret key from Alice's and Bob's measurements, each detected photon is matched to a *state* of a certain *basis*. Two bases are defined containing in total four different states. One basis is the *time basis*, containing the states *short-short* and *long-long*. These two states are *orthogonal*, i.e., they can be measured simultaneously, and ultimately a quantum object measured in the time basis is either in the *short-short* state or the *long-long* state. Alice and Bob assign values 0 and 1 to these states, which are used in the later key.

The other basis is the *phase basis*, containing the *short-long* and *long-short* states. Unlike in the time basis, Alice and Bob cannot assign bit values depending on the peak they measured the photon in. Instead, when measuring in this basis, they assign values 0 and 1 to these events depending on which detector clicked. Due to a correlation coefficient of -1 and Franson interference, Alice's left detector clicks when Bob's right detector clicks, and vice versa. Therefore, when measuring in the phase basis, Alice assigns 0 to the events measured with her left detector and 1 to the events of her right detector, while Bob assigns opposite bit values 1 for his left-detector events and 0 for his right-detector ones. The states from one basis are not orthogonal to the ones from another basis, meaning that only one basis can be measured at a time.

A crucial point to understanding the security of QKD protocols lies in the *no-cloning theorem* in quantum mechanics [24, 25, 26] and the usage of *non-orthogonal states* to obtain shared secret information. The no-cloning theorem states that quantum objects cannot be cloned. From this it follows that an attacker, usually named *Eve*, cannot intercept the quanta to create and forward a copy of them while obtaining the secret information from her quantum copies. Due to using non-orthogonal states, an attacker can neither measure the quanta, obtain information about their states, and then forward those quanta, since measuring a quantum object inevitably results in disturbing its state.

When intercepting quanta, Eve has a $50\,\%$ probability of making the measurement results of Alice or Bob unpredictable by picking "the wrong" pair of states compared to the ones Alice or Bob are about to measure. Let us assume Eve measures a quantum in a certain basis and then forwards it to Bob. Alice and Bob then both happen to measure the quantum in the other basis. Even though the quanta were once entangled, Alice and Bob will get uncorrelated results, because by having Eve measuring the quantum in a different basis before Bob measures it, there is no longer a correlation between Alice's and Bob's results, but Bob's result will be random with a $50\,\%$ probability each.

Key Sifting and Post-Processing

Once Alice and Bob matched each event to a certain basis and assigned bit values accordingly, they can publicly announce in which basis they measured for each photon they detected. They only keep the values where they measured in the same basis and discard all other values. This list of values is called the *sifted key*. Theoretically, the sifted key is equal for both Alice and Bob, but due to various different reasons, like, e.g., imperfect setups or attackers, there are still some values that differ between Alice and Bob. This is fixed as part of the post-processing.

When announcing their list of bases, the classical channel via, e.g., the Internet does not need to be encrypted. However, it must be authenticated, i.e., Alice and Bob must be able to verify that a received message was sent by their communication partner and not by an attacker Eve. If it were not authenticated, Eve could run an Man-in-the-middle (MITM) attack where essentially Alice and Bob would each run their own BBM92 protocol with Eve without noticing it. The quantum channel where entangled quanta are exchanged between Alice and Bob is secured by the aforementioned laws of quantum mechanics, even against a malicious source, as shown in reference [18], and therefore does not have any security requirements at all.

Analogously to the classical Bit Error Rate (BER), a Quantum Bit Error Rate (QBER) is introduced as a measure of error occurrence. For the QKD setup at *P4 – Quantum Key Hubs*, QBER values around $2.5\,\%$ are found for a total fiber length of 76.9 km [9], meaning that around 2.5 % of the sifted key's bits are flipped between Alice and Bob.

In his Master thesis [11], Jendrik Seip implemented an error correction algorithm based on Low-Density Parity-Check (LDPC) codes in a Java program designed to correct errors during QKD sessions. Since error correction algorithms need to expose information about the key to be corrected, the error-corrected key is privacy-amplified, essentially by generating a shorter key that is more secure as the key with errors before error correction. It got implemented in Java in order to allow *E1 – Secure Integration of Cryptographic Algorithms* to analyze the implementation of QKD algorithms on correct implementation of cryptographic primitives. The limit for being able to obtain a secure key is a maximum QBER value of 11.5 %, as shown by Lütkenhaus in ref. [27]. If the QBER value is higher than this, the post-processing code does no return a secret key. This thesis makes use of Seip's work to let Alice and Bob obtain an identical secret key.

2.2. Computer Networks

With the invention of the *World Wide Web* (WWW) by Tim Berners-Lee in 1989 [28], the Internet and computer networks in general started to rapidly spread around the globe and even into space. Its underlying network technologies, the Transmission Control Protocol (TCP) and the Internet Protocol (IP), however, were already published as soon as 1974 [29].

2.2.1. Network Protocols and Layers

To let two computers in distinct places communicate with each other via the Internet, a set of different technologies and algorithms is used that is split into four different *layers* in the TCP/IP model. These four layers are derived from seven more general Open Systems Interconnection (OSI) layers. [30]

On the lowest layer, the *Link Layer*, the communication on the hardware-level between two neighbouring computer systems is handled by using, e.g., lasers and fibers. On top of this, the *Internet Layer* handles the routing of *packets* of data that is to be exchanged between two computers by using IP. With IPv4, each computer is assigned a 32-bit IP address like, e.g., 130.83.47.181, that is used by IP to route *packets* from their source to their destination system. On top of this *Internet Layer*, the *Transport Layer* assures that data sent from the *Application Layer* is transported to its destination. Protocols like TCP do so by offering applications an interface that allows them to send and receive continuous streams of data. In most programming languages, these interfaces are implemented as so-called *network sockets*. In addition to the IPv4 address, *Transport Layer* protocols like TCP allow to further specify a *port number* which is an unsigned 16-bit integer, allowing up to 65 535 processes to send and receive data. [30]

2.2.2. Network Sockets

To let one process on one computer communicate with another process on either the same or another computer, network sockets are commonly used. They come in at least two different flavours: *connection-oriented* and *connectionless*. For connectionless network sockets, most often the User Datagram Protocol (UDP) is used. They allow to send packets of data without previously having to establish a connection on the Transport Layer. However, packets may get lost during their transmission and also the order of the packets is not guaranteed.

As it is important for QKD to receive every single bit and process the bits in the correct order, connection-oriented network sockets are used together with TCP. Two processes $\mathcal{A}$ and $\mathcal{B}$ that want to communicate with each other start by letting one process, e.g., $\mathcal{A}$, wait for incoming connections on a certain port. $\mathcal{B}$ then tries to connect to $\mathcal{A}$ by indicating the IP address and port number of $\mathcal{A}$. Once $\mathcal{A}$ has accepted this connection, they can start exchanging data by simply writing bytes of data into an *input stream* and reading them from an *output stream*. They do not have to worry about packaging this stream of data into packets, nor to resend packets in case of losses nor to reorder packets when receiving them, since this is handled by TCP. Both network sockets of $\mathcal{A}$ and $\mathcal{B}$ offer an input and an output stream.

3. Setup

Due to the work done by Lucas Bialowons et al. [9, 10], the QKD setup using BBM92 at *P4 – Quantum Key Hubs* has been operational at the beginning of this thesis. Here, all measurement devices need to be connected to a single computer on which all of the calculations and analyses are done inside a single program. While it sorts detected photons into time bins and generates a sifted key from the knowledge of all detectors' measurement results, no secure key is derived but instead only estimations on key lengths are given. This setup and the program that will be used to separate the receivers and source, the Quantum Network Control Center (QNCC), are presented in this chapter.

3.1. Experimental Setup

The QKD setup is fully automated using a set of *measurement* and *analysis scripts* implemented in Python. Python is used because it features useful libraries for communication with measurement devices and for calculations with large amounts of data. The measurement scripts handle the initial configuration and startup of all devices used as part of the source and receivers, while the analysis scripts use the measurement results to obtain sifted keys and statistics like QBER values. In contrast to most existing QKD setups, the setup at *P4 – Quantum Key Hubs* enables key distribution between more than two parties. As presented in ref. [9], dense wavelength-division multiplexing is used together with a broad SPDC spectrum which enables connecting more than 34 parties. In the current setup, besides Alice and Bob, there are two more parties called *Charlie* and *Diana* that participate in the key generation process using the same photon-pair source. Currently, all four parties can be freely connected pair-wise to form two *subnets*, but within these subnets the situation is identical to a 2-party setup. That is why in this thesis the names Alice and Bob merely refer to *roles*, but not to actual parties within a potential 34-party network.

Figure 3.1 shows the setup of the receiver of a party. Each receiver has in total two single-photon detectors, one for each of the two outputs of the Michelson interferometer. By using Michelson interferometers with Faraday mirrors instead of Mach-Zehnder interferometers, the receivers can be used for BBM92 time-bin coding despite polarization changes in the fiber connecting two parties. To be able to detect photons on both outputs of the interferometer, an optical circulator is used. This optical circulator transmits light into one direction, but reflects light into a second output if entered from the other side. Due to losses of the optical circulator, one detector detects more photons than the other. Every time a photon is detected, the *event* is logged in form of a 64-bit timestamp using a time tagger[1]. The measurement results consist of two lists of timestamps per party, whereas one list contains the timestamps of one detector. Since the clock speed of Alice's and Bob's time tagger is not exactly equal, clock recovery from ref. [9] is used to correct the obtained timestamps afterwards.

[1]*ID900 Time Controller Series* by ID Quantique

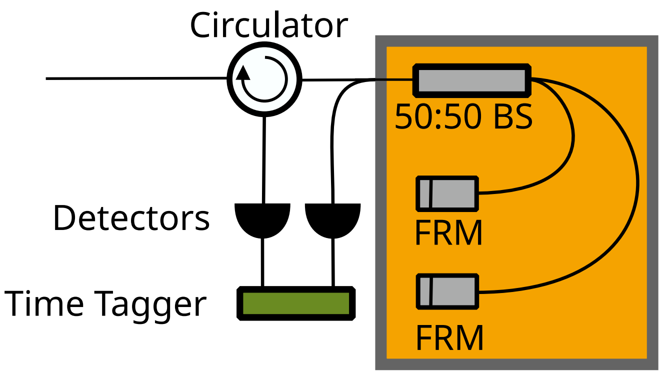

Figure 3.1.: This setup scheme adapted from ref. [31] shows one receiver. Light entering from the left passes through an optical circulator. Inside the temperature-controlled box in yellow the Michelson interferometer is shown, consisting of a 50:50 beam splitter (BS), two arms of different lengths, and two Faraday mirrors (FRM). One output of the interferometer leads to the right single-photon detector, while light from the other output first passes through the optical circulator before hitting the left detector. For all detections, timestamps get logged with the time tagger.

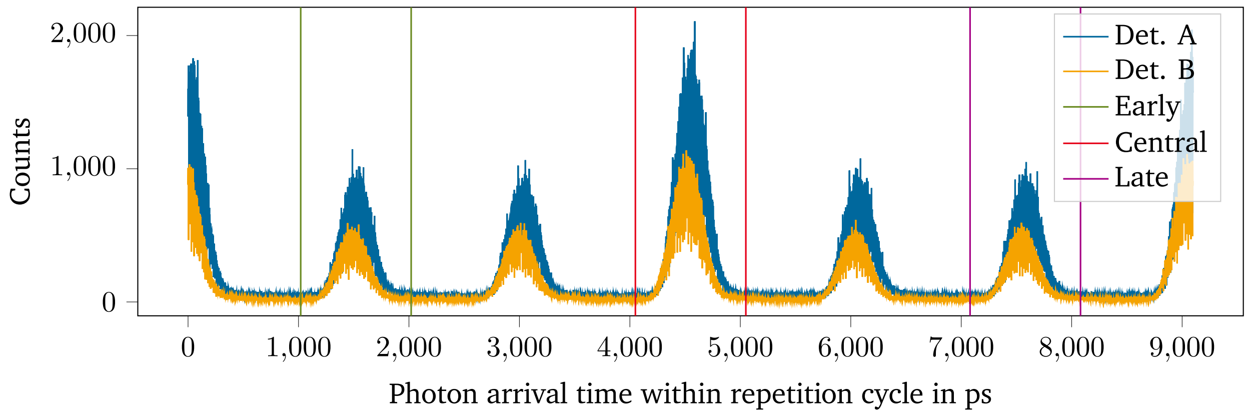

Figure 3.2.: The arrival histogram of data produced by the QKD setup in ref. [9] shows two interleaved three-peak structures.

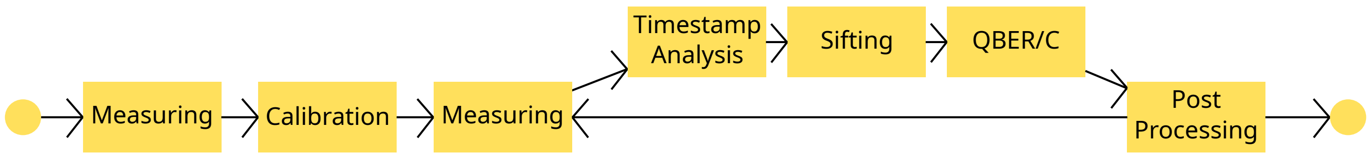

Figure 3.3.: This diagram shows the various steps during measurement and analysis. Starting with a first measurement, an initial calibration is done. Timestamps from subsequent measurements are recalibrated and sorted into time bins during timestamp analysis, a sifted key is obtained, the QBER and correlation coefficient C are calculated, phases are adapted, if necessary, and the sifted key is error-corrected and privacy-amplified during post-processing.

With clock recovery, knowing the clock periodicity allows the timestamp list of Alice and Bob to be separately analyzed by each party on its own to retrieve the clock speed of the source and correct the timestamps accordingly. Without clock recovery, the time taggers must be analyzed using an electrical connection.

While the source is continuously producing photon pairs, Alice and Bob each measure for periods of $90\,\mathrm{s}$. Afterwards, they immediately start a new 90-second measurement, but simultaneously analyze the obtained timestamps and, e.g., use the calculated correlation coefficient C from eq. (2.4) to correct an interferometer's phase delay, if necessary. This means that the analysis of the measurement's results must not take longer than $90\,\mathrm{s}$, because otherwise another measurement iteration may be started with sub-optimal phase delays. The imbalancement of the interferometer in this setup is $3.03\,\mathrm{ns}$ [10]. To produce the three-peak structure shown in figure 2.2, the pulsed laser source would need to produce photons with a pulse period of $3 \cdot 3.03\,\mathrm{ns} \approx 9.1\,\mathrm{ns}$. In this setup, however, the pulsed laser source is producing photons with a period of $1.5 \cdot 3.03\,\mathrm{ns} \approx 4.55\,\mathrm{ns}$ [10] which leads to an interleaved peak structure as shown in figure 3.2. With this interleaved peak structure, the space between a single structure's peaks is used to fill in another peak structure, leading to a *nesting level* of 2.

An overview of the various steps necessary during measurement and analysis is given in figure 3.3. After measuring for the first time for $90\,\mathrm{s}$, an initial calibration is done in which offsets for Alice's and Bob's detectors are obtained. After measuring again for the same period of time, the analysis is started by performing a recalibration of the timestamps and sorting the timestamps into time bins for key sifting. The recalibration is only necessary when analyzing timestamps without clock recovery, otherwise the tiny drifts are already compensated by the clock recovery algorithms. Alice and Bob then do the key sifting and based on comparing the sifted key between Alice and Bob, values for QBER and correlation coefficient C are obtained that are used to confirm the security of the key exchange session and adapt the phase delays of the interferometers. The sifted key is then post-processed by doing an error correction and privacy amplification. All this is described in section 3.2. At this point, a new 90-second interval of measurements can be analyzed.

3.2. Analysis of Measurement Data

The first step of the analysis of a 90-second iteration is the timestamp analysis. As part of it, the offsets between the parties' detectors get calculated and each timestamp gets sorted into a time bin, if possible.

3.2.1. Offset Calibration

Alice and Bob measure single photons with their two detectors at their interferometers' outputs. When starting a measurement for a 90-second period, the time taggers of both parties start with their clocks at zero. However, the fibers connecting the source with Alice and Bob are of different lengths which results in one party detecting photons later than the other one. In addition to this, Alice and Bob never start their measurements at exactly the same point in time, adding yet another offset between their timestamps. Finally, even between the two detectors of a single party there is an offset due to different lengths from the interferometer exits to the detectors. In the first iteration, an initial calibration is done in which offsets for all detectors are obtained while choosing one of the parties' detectors as the reference detector. In this thesis, this is Bob's left detector with its timestamp list B_1. Alice calibrates her set of timestamp lists $\{A_1, A_2\}$ by cross-correlating each list with the timestamp list of Bob's left detector. This is possible since both detectors of Alice detect photons that are entangled with photons measured at Bob's detector. This results in two offsets $O_{A_1,B_1} = A_1 - B_1$ and $O_{A_2,B_1} = A_2 - B_1$.

Bob cannot do this cross-correlation between his left and right detectors since the pairs of entangled photon are split among Alice and Bob, i.e., there are no two entangled photons going both to a single party — besides invalid events that happen due to imperfect wavelength-division multiplexing. Thus, the timestamp list of his left detector is not correlated to the one of his right detector. Instead, he uses the timestamp list A_1 of Alice's left detector to obtain the offset between his two detectors:

$$O_{B_2,B_1} = B_2 - B_1 = B_2 - A_1 + A_1 - B_1 = O_{B_2,A_1} - O_{B_1,A_1} \,. \tag{3.5}$$

This means that in order to let Alice and Bob calibrate their detectors, they both need to know one of the timestamp lists of the other party. The offsets obtained in this initial calibration are called *relative offsets* and stored throughout the whole QKD session's lifetime. Since Bob's left detector is chosen as the reference detector, its relative offset is always 0.

To allow sorting events into time bins, it is important to have the central peak exactly at the center of a detector's arrival histogram. This is handled by clock recovery. However, the software developed in this thesis does not yet feature clock recovery, so recalibration is described as it is done without clock recovery. As a first step towards obtaining the *central peak offset* of each detector, the central peak offset of the reference detector is obtained as the *reference offset* using a peak finding tool[2]. When obtaining the central peak offsets of the other detectors, the timestamps are first corrected by their relative offsets and the iteration's current reference offset. Then, the peak finding tool is used to obtain the final *total offset* of each detector. For the reference detector, this is just the reference offset, but for all other detectors it is given by

$$O_\mathrm{t} = O_\mathrm{rel} + O_\mathrm{ref} + O_\mathrm{peak} \,, \tag{3.6}$$

[2]`scipy.signal.find_peaks`

where O_{rel} is the relative offset of a detector to the reference detector as obtained during cross-correlation in the first iteration, O_{ref} is the central peak offset of the reference detector in the current iteration, and O_{peak} is the central peak offset of a single detector after correcting the timestamps by O_{rel} and O_{ref}. It is important to recalibrate the detectors this way, because otherwise a timestamp list of one detector may get corrected into a different direction than another, resulting in a drop of the sifted key rate. With the clock recovery from ref. [9], these recalibration steps are no longer necessary and only the relative offsets from the initial calibration are needed.

3.2.2. Sorting Timestamps into Time Bins

Once the timestamp lists are corrected by their respective offsets, the timestamps get sorted into time bins. There are three time bins containing the early and late satellite peaks and the central peak. As stated in the beginning of this chapter, the imbalancement of the interferometers is $3.03\,\text{ns}$ while using a repetition rate of $1.5 \cdot 3.03\,\text{ns} \approx 4.55\,\text{ns}$ to achieve pulse interleaving as described in ref. [10]. This results in two interleaved three-peak structures as shown in figure 3.2. These two interleaved peak structures can be independently analyzed by using a repetition cycle length of $2 \cdot 1.5 \cdot 3.03\,\text{ns} \approx 9.1\,\text{ns}$. In the first iteration, one of the three-peak structures is analyzed by choosing time bins that surround peaks of only one the structures. To analyze the other interleaved three-peak structure, the timestamps are shifted by $1/2 \cdot 9.1\,\text{ns} = 4.55\,\text{ns}$ and analyzed using the same algorithms and time bins as in the first analysis run.

In the scripts developed in ref. [10], time-bin sorting is implemented using the Python library *numba*[3]. Each timestamp list contains up to 2 million timestamps, meaning that time-bin sorting cannot be implemented in pure Python code since it would be too slow to finish before the 90-second limit. numba allows implementing algorithms in Python code, but translates it to machine code before running it. Thereby, performances comparable to native code in, e.g., *C* or *C++* can be achieved. By combining the early-time-bin lists of both detectors, all repetition cycles are obtained that correspond to measuring the short-short state in the time basis. Similarly, the repetition cycles corresponding to the long-long states of the time basis are obtained by combining the late-time-bin lists of both detectors. To gather the two states of the phase basis, the combination of the central-time-bin lists of both detectors is used.

3.2.3. Subsequent Analysis

At this point, Alice and Bob possess the information about the measurement basis and the measured state for all photons that could be matched to a time bin. As the analysis scripts work with a centralized setup, all this information is available in a single place. By comparing the bases and states for Alice and Bob, the analysis scripts calculate various statistics that are useful to estimate sifted and secure key rates. For example, the amount of repetition cycles where they measured a photon in the same basis is equal to the length of the sifted key. By comparing the measured states, the analysis scripts obtain the exact value for the QBER and use this to estimate a secure key length. Since the setup has four parties connected to the source at the same time, the analysis scripts do these analyses for both subnetworks consisting each of two connected parties.

[3]`https://numba.pydata.org/`

3.3. Inter-Process Communication

For the QKD setup to be useful, the source's and receivers' setups must be separated and split into distinct units, connected only by an optical fiber and a classical network connection. The first step towards separating the source and the four parties Alice, Bob, Charlie, and Diana was made by Hühne et al. in ref. [12] where a program called Quantum Network Control Center (QNCC) was built. QNCC is responsible for building connections between the parties and the source via the Internet or the local network, allowing them to exchange messages during key generation, and saving secure keys obtained from QKD. QNCC features a graphical user interface that allows storing several different contacts with their properties, like IP address, port number, and public keys, and also allows exchanging files and chatting with those contacts end-to-end-encrypted by using secure keys from QKD. Just like the post-processing software, QNCC got implemented in Java to allow *E1 – Secure Integration of Cryptographic Algorithms* to analyze cryptographic algorithms on correct implementation. In addition to that, having QNCC and the post-processing software implemented in the same language allows easy integration of the latter into the key generation process of QNCC.

Since Alice and Bob normally are in distinct places when running QKD, some type of classical communication channel is needed to exchange information like the list of bases between them. This channel is realized using network sockets. Network sockets offer a stream-like interface that send and receive bytes without knowing anything about the content or structure sent via this stream. Therefore, some kind of *encoding* is needed that converts data in form of integers, strings, etc. to plain bytes and allows to *decode* these bytes back to the original data. For the communication between Alice, Bob, and the source, this has been implemented in ref. [12] using Java's `ObjectStream` classes `ObjectInputStream`[4] and `ObjectOutputStream`[5]. Each network socket offers an input stream, where data from the other side is received, and an output stream, where data is sent to the other side. These input and output streams of the network sockets are plugged into an `ObjectInputStream` and an `ObjectOutputStream`, respectively. The `ObjectOutputStream` allows to send primitive data like boolean or integer arrays, but also more complex objects, and handles the serialization of this data to its byte representation. It makes sure that the corresponding `ObjectInputStream` on the other side can rebuild the data from this stream of bytes.

The objects exchanged between the parties and the source are of the type `NetworkPackage`, a class implemented in ref. [12] that stores, besides other metadata, the content of the message in form of a byte array. It also stores a signature that allows, e.g., Bob to confirm a message's authenticity he received from Alice. This is important for QKD, since otherwise Alice and Bob could suffer from an MITM attack.

By default, `ObjectInputStream` and `ObjectOutputStream` remember all objects sent via these streams. This is done to update objects on the receiver side that have already been received previously. For QNCC, however, this is not necessary since sent messages are independent from each other, so it is essentially just a log of all messages ever sent.

At the beginning of this thesis, QNCC was a program written purely in Java that was designed to work together with eventual Python code by communicating via files written to and read from a local folder. QNCC would wait for a file named *out.txt* written by the Python code. This file was read by QNCC and sent to the other party's QNCC where the content would be written to a filed named *in.txt*. Thereby, the other party's Python code was able to receive data. To prevent reading partial files, two files *out.txt.lock* and *in.txt.lock* were used as an indication that a file was still being written to.

[4]`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ObjectInputStream.html`
[5]`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ObjectOutputStream.html`

4. Details on the Implementation

As stated in the introduction, this thesis is concerned with separating Alice, Bob, and the source to allow them to run a QKD session using the BBM92 protocol based on measurement data from the centralized setup. The measurement and analysis scripts developed in ref. [10] only work in a centralized setup, where all measurement devices are connected to a single computer. Also, they do not do the post-processing required to obtain secret keys, but just estimate secure key rates by using the full knowledge about all parties' measurement results. As part of this thesis, all analysis steps required by the BBM92 protocol are implemented in a decentralized manner so that Alice and Bob no longer need to be in the same place. To distinguish this new implementation from the already existing scripts, the latter are referred to as the *legacy* measurement and analysis scripts.

4.1. Modifications to the Previous Software

In this thesis, the Quantum Network Control Center (QNCC) is modified and extended to match the project's needs when managing the key generation process and handling the communication between Alice and Bob. In particular, the newly developed Python code of QNCC is responsible for receiving lists of timestamps from the legacy measurement scripts. It uses them to obtain a sifted key, which is then handed over to the Java side of QNCC where the code from ref. [11] does the post-processing. Finally, once the secret key is obtained, it is saved to the key storage of QNCC and the correlation coefficient C — that is obtained from the sifted key — is sent to the Python code. In the future, this value of C can be used to adapt the interferometers' temperatures, if necessary.

When the development of QNCC started in ref. [12], a user interface for the command line was written. In parallel, a graphical user interface was established that quickly superseded the command-line interface. As a first step of cleaning up the code basis, the code belonging to the command-line interface is removed. Nevertheless, the code is still accessible in the project's version history[1], if in the future there is interest for a command-line interface. One of the first bugs encountered in QNCC is that the Python script started by Java is not terminated when QNCC is closed. This is fixed by stopping the `Process`[2] that runs the Python script from Java when the Java program itself is stopped. Running multiple instances of QNCC for Alice, Bob, and the source is difficult since configuration files are created in the same directory where the program is executed. This is improved by storing all user data in a separate directory, allowing several instances of QNCC to be started at the same time.

Inspecting the classes of QNCC that implement encryption with AES[3] reveals that the CBC mode of operation is used together with a hard-coded Initialization Vector (IV). The IV must be different for every encryption

[1]Merge request 3 in the GitLab project removed the command-line code, from where it can be easily recovered.
[2]`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Process.html`
[3]`AES256` in the package `encryptionDecryption`

when using the same key [7], since otherwise plain texts with equal beginning would result in identical cipher text beginning. In QNCC, however, a key is never reused when encrypting with AES because QKD offers a continuous stream of secure bits that get used as secret keys. Therefore, it is fine to use the same IV for every encryption.

Finally, one bug that becomes apparent only after running QNCC over several iterations of measurement data is that the memory usage of QNCC increases in time. This is due to QNCC remembering every single packet ever sent between Alice and Bob. The problem and its solution are discussed in detail in section 5.1.1. QNCC now forgets data immediately after sending it, thus making memory usage constant in time.

4.2. Software Architecture

With QNCC having many different responsibilities that are already partly implemented in Python and Java code, it is important to design a clean software architecture for the Python code developed in this thesis that allows for easy maintenance and that is modular so that new features can be easily integrated in the future. In particular, this thesis focusses on working with data sets from historical measurements done in recent years. In the future, however, QNCC will also be responsible for controlling the experimental setups of the parties and the source. Therefore, a *historical key generation mode* is implemented in this thesis, while the *default* key generation mode that also controls the measurement devices will eventually be implemented by future students. While working on this historical key generation mode, the implementation of the BBM92 logic is done in a way as modular as possible to allow most parts to be reused in the future default key generation mode.

To achieve this modularity, a set of so-called *abstract classes* is used in the Python code[4]. They often have two *concrete* classes *inheriting* from these abstract classes. The abstract classes are called *abstract* because they cannot be used directly. Instead, a class needs to inherit from this abstract class by implementing all abstract methods defined in the abstract class and by providing attributes of concrete type for attributes with abstract types. These classes are then called *concrete* because they can be directly used to create objects. An example is shown in figure 4.1. There, the abstract class for the task of key generation is `AbstractKeyGeneration`. It has two methods besides the constructor function `__init__()` and a set of five *attributes*: `measuring`, `timestamp_analysis`, `sifting`, `post_processing`, and `phase_adjustments`. The five attributes have the type of their respective abstract classes and point to the implementations of the various different steps of a QKD session.

Note that the methods `start()` and `evaluate_iteration()` are not abstract. The logic of these methods is already implemented in `AbstractKeyGeneration`. Thus, when calling, e.g., `evaluate_iteration()` on objects of the type `HistoricalKeyGeneration`, the method of the class that `HistoricalKeyGeneration` inherits from, `AbstractKeyGeneration`, is called. By this, all that is needed in the classes `KeyGeneration` and `HistoricalKeyGeneration` is the initialization of the five attributes which happens in their `__init__()` function. Nevertheless, if there is the need in the future, both methods can be overwritten in the concrete classes to provide their own implementation.

Similar to `AbstractKeyGeneration`, all steps of QNCC are managed using a set of abstract classes. For some of them, like `AbstractTimestampAnalysis`, `AbstractSifting`, and `AbstractPostProcessing`, there currently only exists a single class inheriting from these classes, which could make the abstract classes look like unnecessary overhead. Nevertheless, it is useful to have the structure of these classes —

[4]Abstract classes are implemented in Python in the *abc* library: `https://docs.python.org/3/library/abc.html`

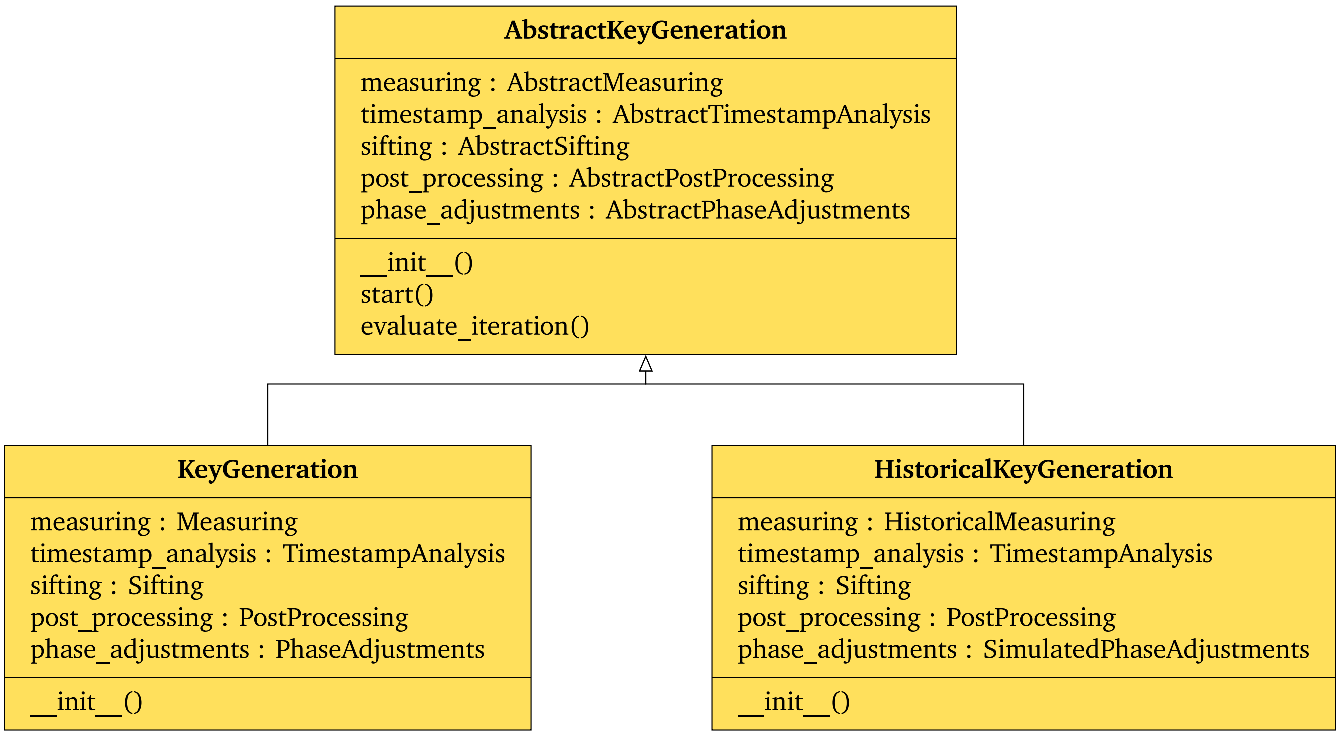
AbstractKeyGeneration
measuring : AbstractMeasuring
timestamp_analysis : AbstractTimestampAnalysis
sifting : AbstractSifting
post_processing : AbstractPostProcessing
phase_adjustments : AbstractPhaseAdjustments
__init__()
start()
evaluate_iteration()
KeyGeneration
measuring : Measuring
timestamp_analysis : TimestampAnalysis
sifting : Sifting
post_processing : PostProcessing
phase_adjustments : PhaseAdjustments
__init__()
HistoricalKeyGeneration
measuring : HistoricalMeasuring
timestamp_analysis : TimestampAnalysis
sifting : Sifting
post_processing : PostProcessing
phase_adjustments : SimulatedPhaseAdjustments
__init__()

Figure 4.1.: This class diagram shows the entry point into the Python code of QNCC. Depending on the configuration, QNCC either starts `KeyGeneration` or `HistoricalKeyGeneration`.

usually called *interface* in software engineering — decoupled from their implementations, as it, e.g., allows to easily add different implementations in the future. For example, when implementing clock recovery in `TimestampAnalysis`, there might be interest in keeping the current implementation without clock recovery. This could be achieved by renaming the current `TimestampAnalysis` class to something like `SynchronizedTimestampAnalysis` — because the time taggers are synchronized using an electrical connection — and implementing clock recovery in a new `TimestampAnalysis` class. Similarly, post-processing could be implemented using a different error correction algorithm in another class inheriting from `AbstractPostProcessing`, or a third key generation mode `OmniscientKeyGeneration` could be introduced that always exchanges all timestamps and other information between Alice, Bob, and the source to produce similar statistics like the legacy analysis scripts do. This `OmniscientKeyGeneration` could then be used during research and development on the physics of the QKD setup, since switching between QNCC and the legacy measurement and analysis scripts is cumbersome due to having to change all measurement devices' connections from the separate-computers setup back to the legacy single-computer setup. Thanks to a common interface declared in the abstract class, all classes inheriting from an abstract class can be easily exchanged with each other without the need to change other code.

Utilizing abstract classes everywhere also allows for easier automated testing of code. When testing, e.g., `HistoricalKeyGeneration`, it is important to only test the code of this particular class and not of other classes called by `HistoricalKeyGeneration`. In software engineering, this is usually done by *mocking* these dependencies, i.e., creating special implementations of, e.g., `AbstractPostProcessing` that are designed just for these testing purposes. This hypothetical `MockedPostProcessing` class would not run the real post-processing logic. Instead, it would offer methods of `AbstractPostProcessing` that return reasonable values, in this case a correlation coefficient C, without requiring the post-processing code and without doing any post-processing at all.

When a key generation is started in Python by calling the method `start()`, a connection is created to the Java side of QNCC via a network socket on the local device and the measurement is started. The former is described in section 4.3.2. The latter is currently only implemented in `HistoricalMeasuring` where measurement data from the recent years is analyzed. These measurements were made with the legacy measurement scripts and their data consists of two timestamp lists per party and iteration. `HistoricalMeasuring` allows to dynamically read this data from any measurement set recorded in the past and hand it over to `HistoricalKeyGeneration` where it is analyzed iteration-wise with `evaluate_iteration()`.

4.3. Inter-Process Communication

Alice, Bob, and the source are in distinct places when running the QKD session and therefore require some sort of communication channel. For this, the software developed in ref. [12] is used, QNCC. As the software implementing the BBM92 protocol is developed in Python to eventually work together with the legacy measurement scripts, another communication channel is needed that connects Java with Python. Both communication channels are described in this section.

4.3.1. Communication between Alice, Bob, and the Source

All communication that happens between Alice, Bob, and the source from beginning a QKD session to obtaining the secure key is shown in figures 4.2 and 4.3. It starts with Alice requesting a QKD session from Bob. Once Bob accepts this QKD session, Alice requests from the source a QKD session with Alice and Bob. In the current implementation with historical data, the source simply accepts this QKD session, but in the future key generation implementation the source may start sending entangled photon pairs to Alice and Bob at this point. Once the source informed Alice that it is now sending entangled photon pairs to Alice and Bob, Alice can inform Bob in the future implementation that both may start with their measurement. This gets acknowledged by Bob.

Alice and Bob now both measure incoming photons on their fibers with their BBM92 receiver setups from figure 3.1 for a period of $90\,\mathrm{s}$ and store the timestamps of the detected photons in two lists, one for each detector. To calibrate their timestamp lists, Alice and Bob each send to the other party one list of timestamps. They use this to obtain an initial offset between their detectors' timestamp lists. This initial calibration is described in section 3.2.1.

At this point, Alice and Bob discard the timestamps and do not use them for a further key generation, since they disclosed the timestamps' content by sending them via a public channel to their communication partner. Having obtained the offsets between their detectors, they can analyze the results of the next 90-second measurement to obtain a secret key. When QNCC eventually controls the measurement devices in the future, Alice and Bob start a new 90-second measurement immediately after finishing the previous one. In this thesis, only the key generation with historical data is implemented, so Alice and Bob simply analyze one iteration after another. They recalibrate their timestamp lists as described in section 3.2.1 and start with the key sifting process described in section 4.5. Essentially, Alice and Bob send each other the list of bases they measured in for each photon they detected and eliminate all timestamps where they measured in a different basis than their communication partner.

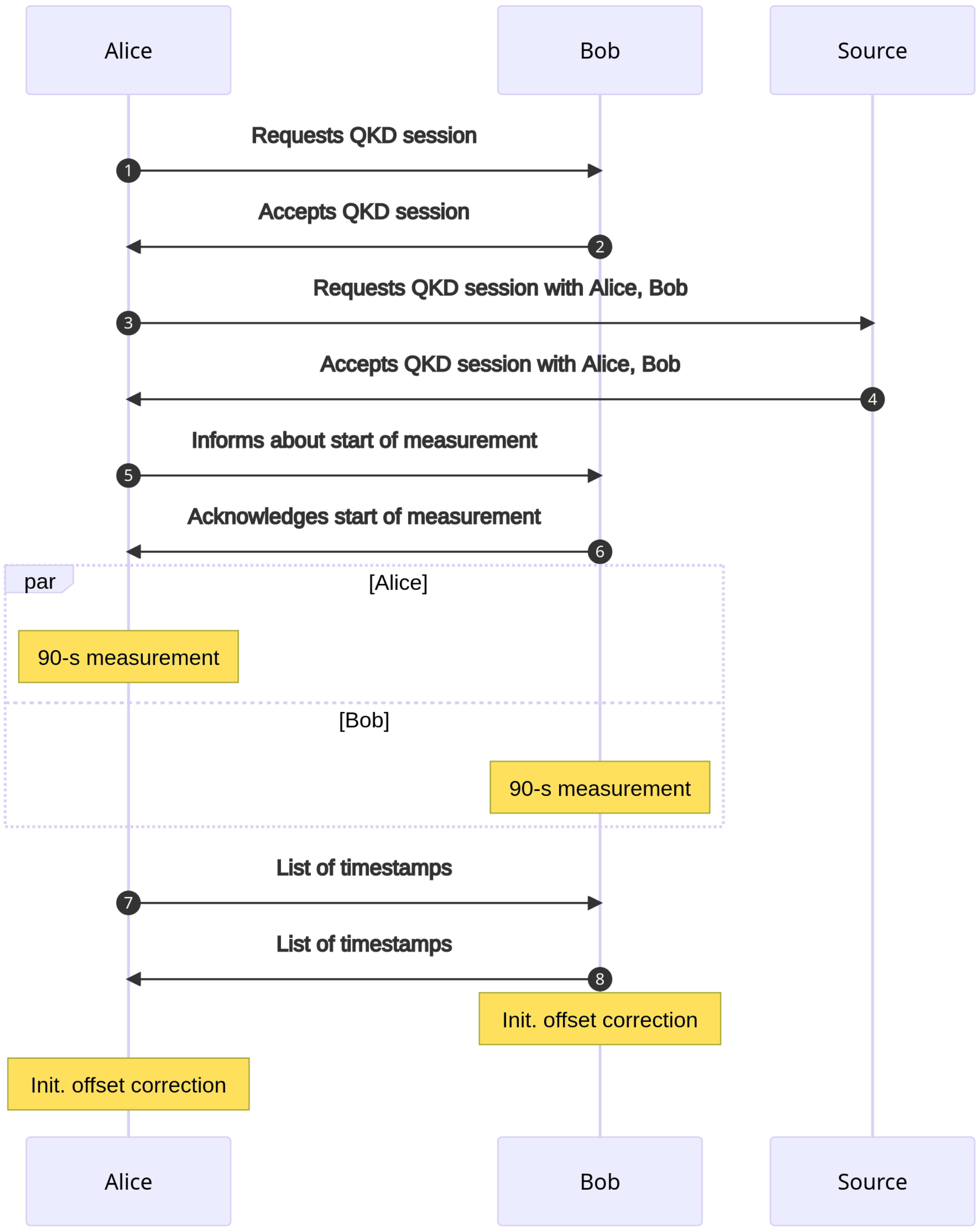

Figure 4.2.: The sequence diagram shows the information exchanged between Alice, Bob, and the source in the first iteration. The box denoted by *par* means that the steps surrounded by it happen in parallel. The diagram is discussed in section 4.3.1.

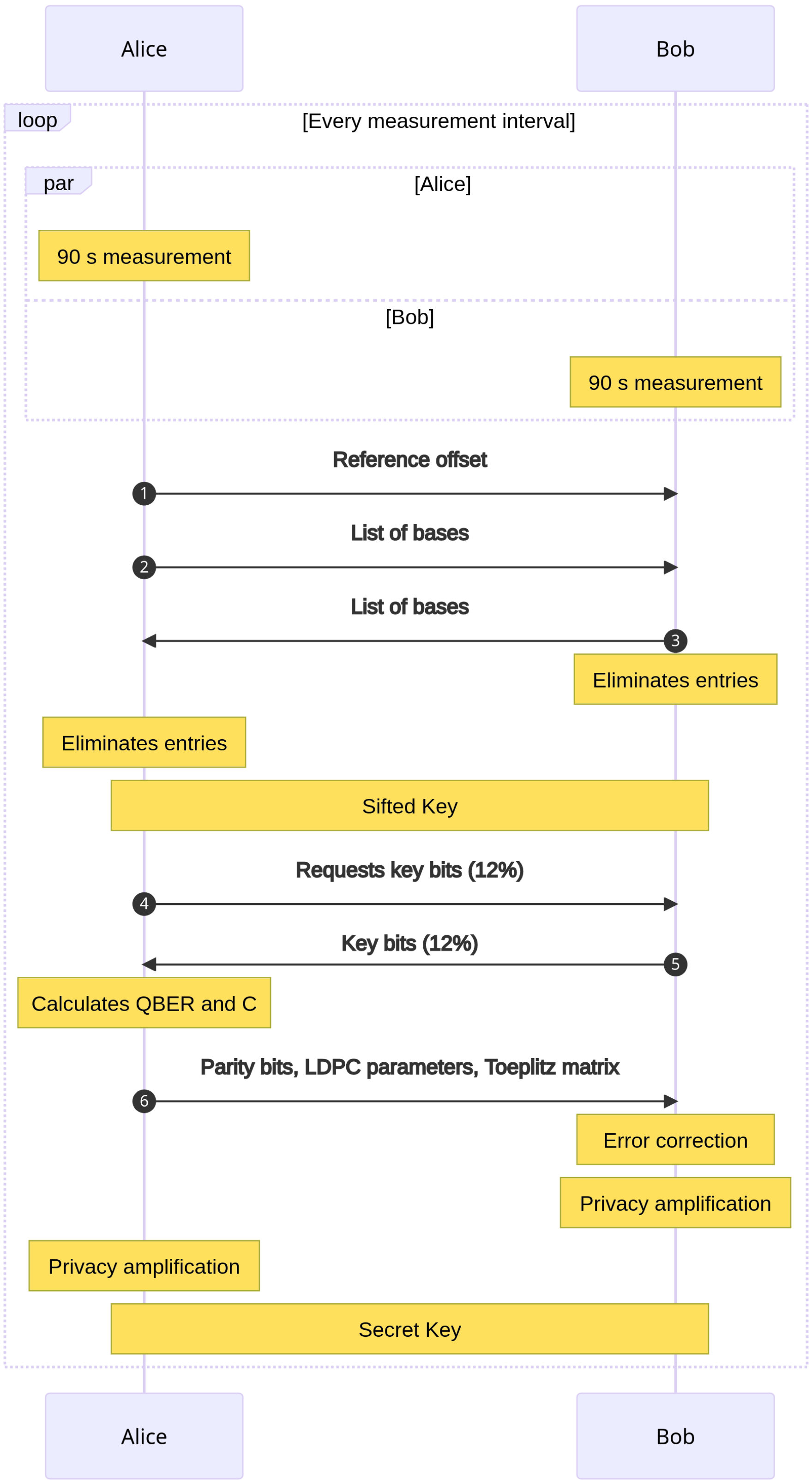

Figure 4.3.: The sequence diagram shows the information exchanged between Alice, Bob, and the source in subsequent iterations after the first one. The box denoted by *loop* indicates that the steps surrounded by it are executed various times. The other box denoted by *par* means that the steps are executed in parallel. The diagram is discussed in section 4.3.1.

After obtaining the sifted key through this procedure, Alice and Bob start with the post-processing described in section 4.6. During this post-processing, Alice and Bob exchange three messages in total. The first two messages are about exchanging a subset of sifted key bits that allow Alice to generate a third message. With this third message, Bob can correct his key to be equal to Alice's key using an LDPC code algorithm from ref. [11]. As part of this post-processing, Alice estimates values for the QBER and correlation coefficient C that she can use to adapt the temperature and thus the phase delay of her interferometer, if necessary. At this point it becomes apparent that an analysis of each iteration may only take $90\,\text{s}$ at maximum, since otherwise a new iteration might get started with suboptimal phase synchronization. At the end of this post-processing, Alice and Bob obtain an equal secret key and are ready for analyzing the next iteration.

4.3.2. Communication between Java and Python

The approach presented in section 3.3 of an inter-process communication between Java and Python using the file system has several weaknesses and limitations. For example, communicating this way is slow because data has to leave the memory and get written to the solid-state drive (SSD) or hard-disk drive (HDD). Additionally, it is inherently insecure since other programs on a computer usually are allowed to read and write files in any of the user's directories. Finally, communicating this way allows only one message to be sent at a time since QNCC does not implement queues where messages would wait until a previous message has been read. In the first versions of this thesis' code, this resulted in race conditions where Alice would send two messages before Bob was able to read the first message. Thus, he missed the content of the first message and the QKD session failed at this point. To prevent such race conditions, the code in this thesis waited for $10\,\text{s}$ after sending a message before sending the next one. This made the key generation process unnecessary slow.

To overcome these drawbacks, communication channels between two processes on a single computer can be implemented using network sockets as well. Due to their stream-like nature, several messages can be sent in a row while the underlying transport protocol ensures that all messages are received and read in the correct order. By this, no queue has to be implemented in QNCC, all messages can be sent directly one after another, and the key generation process becomes much faster. Like the communication between Alice, Bob, and the source that is handled by the Java code from ref. [12], some kind of protocol is needed that encodes and decodes information into byte form. `ObjectStream` cannot be used for this because it is only implemented in Java, and reimplementing it in Python would be too much of an effort. So-called *websockets* are built on top of network sockets and allow sending information in packets with metadata in form of *headers*, but they require an HTTP server and thus are not used either due to their overhead. Instead, a simple protocol is defined that allows to encode and decode information with the sockets' streams.

In this protocol, whenever data is sent in its byte representation, it is prefixed by a header consisting of 8 byte. The first 4 byte represent the size of the message's content. The subsequent 2 byte are interpreted as an enumeration of the message type, followed by the final 2 byte of the header that specify the *nesting branch*. The latter is useful to distinguish between the two list of bases sent with a nesting level of 2. All information in the header is encoded as unsigned integers with big-endian byte order.

Following this 8-byte header, the message's content is sent via the stream. By reading the header from the stream first, the receiver knows how much data to request from the stream to obtain the complete message. As part of this thesis, this protocol has been implemented in Python[5] and Java[6] together with a set of functions in the class `JavaCommunicator` in Python that allow to send integers, floats, and arrays containing integers.

[5]see the classes `JavaMessage` and `MessageType` in the module `networking`
[6]see the classes `PythonMessage` and `MessageType` in the package `keyGeneration`

Java most of the time does not interpret the messages' content, but handles them based on the message type. For example, all messages sent during key sifting are simply forwarded to the other party. Only when receiving the sifted key and sending the correlation coefficient C, Java works with the content of these messages.

Security Considerations on Java-Python Communication

Processes of users without root or administrator permissions usually are not able to access other processes' memory content. Connecting two processes via network sockets like shown in the previous section opens up potential attack vectors that need to be carefully considered. An attacker could try to get information about secret keys by starting an MITM attack between the Java and the Python process, or simply block key generation by doing a denial-of-service attack against one party. This is prevented by deploying various security measures.

When the Python process is started by Java, a random port number and an authentication token consisting of a random 32-bit integer are passed to Python. Both values get generated when starting a new key generation. Python is required to send this authentication token as the first message when opening the network socket to Java, otherwise Java refuses and closes the connection. Python is configured to only connect to IP addresses on the same device (*localhost*), while Java also only allows incoming connections from the same device. After receiving one connection, no new connections are allowed by Java. This ensures that only the Python process started by Java connects to QNCC.

4.4. Timestamp Analysis

Like in the legacy implementation, the first step of a QKD session is the analysis of the timestamps, in particular the offset calibration and recalibration, and the time-bin sorting. For the offset calibration where the initial relative offsets get calculated, the code from the legacy analysis scripts gets reused that implements the cross-correlation using the Python library numba. The offset recalibration also uses the code from the legacy scripts, where a peak finding tool is used to center the peak structures of all detectors. The time-bin sorting, however, is implemented from scratch because the code in the legacy scripts only works with all detectors connected to a single computer.

When sorting timestamps into time bins, it is important not to implement too much of the logic in plain Python code, since Python code is significantly slower than native implementations in, e.g., C or Fortran. It is therefore that the software developed in this thesis uses the libraries *numpy*[7] and *sortednp*[8] that allow fast calculations on large arrays containing the timestamps in form of 64-bit integers. Each timestamp list contains up to two million entries and iterating over them with Python `for` loops is not possible within the 90-second-analysis limit of an iteration.

The timestamps given to the algorithm presented below must have the same clock speed as the timestamps of the other party — either by using an electrical synchronization between the parties' time taggers or by using clock recovery — and the timestamps must already be corrected by their respective offsets. Depending on the nesting level, the algorithm is run various times on each timestamp list. In this thesis, it is run twice, but the algorithm supports arbitrary nesting levels. First it analyzes one three-peak structure and then shifts all timestamps by the *nesting offset* of $4.55\,\mathrm{ns}$ to analyze the other interleaved three-peak structure. For a single

[7] `https://numpy.org/`
[8] `https://pypi.org/project/sortednp/`

party with one analyzed detector and three-peak structure, this results in three lists containing repetition cycle numbers for each time bin. The repetition cycle numbers are the quotient of a modulo operation on the timestamps with the repetition cycle time of $9100\,\mathrm{ps}$ and are equal for events of Alice and Bob with entangled photons.

The first calculation done on the timestamps is the modulo operation while keeping the quotient and remainder of each timestamp:

```
rep_numbers, remainders = np.divmod(timestamps, rep_time)
```

Here, `np` is the numpy library, `timestamps` is the array containing the timestamps of one detector in ps, `rep_time` is the repetition cycle time, `rep_numbers` is the resulting repetition cycle number for each timestamp, and `remainders` is the arrival time of each measured photon within a repetition cycle.

Next, we want to classify the remainders according to whether the photons arrived in the early, central, or late time bin. For the early time bin, this is done with

```
(remainders >= 1020) & (remainders < 2020)
```

Here, 1020 and 2020 represent the lower and upper boundaries of the early time bin in ps. For the central time bin, the boundaries are $4050\,\mathrm{ps}$ and $5050\,\mathrm{ps}$, while for the late time bin they are given by $7080\,\mathrm{ps}$ and $8080\,\mathrm{ps}$. These values are empirically found to work with the fiber lengths and setup configuration used in ref. [10], but can be freely adapted by changing the configuration of QNCC as described in appendix B.

By comparing all remainders in the array `remainders` with these lower and upper boundaries of the early time bin, a boolean array of the same size as the remainders array is obtained that contains values `True` for all timestamps that got sorted into the early time bin, and `False` for all other timestamps. This information is used to filter the array containing the repetition cycle numbers. Ultimately, this leads to three arrays containing repetition cycle numbers that got matched to the early, central, and late time bin. By running the same algorithm on the timestamp list of the second detector, another three arrays are obtained. To analyze the other nesting branch, the timestamps are shifted by the nesting offset to obtain yet another six arrays for the two detectors.

These two times six arrays are now processed for the latter key sifting. Like in the time-bin sorting presented above, the data from the two nesting branches can be processed independently. In the key sifting, the necessary information is the state of each measured photon. By combining the two arrays containing the early-time-bin events of the left and right detector of one nesting branch, an array is obtained containing all repetition cycle numbers where the short-short state got measured. Similarly, this is done for the late-time-bin events to obtain an array with all long-long-state measurements. The central-time-bin events of the left and right detector contain the short-long and long-short states of the phase basis. Analyzing both nesting branches results in $2 \cdot 4 = 8$ arrays.

4.5. Key Sifting

Now that Alice and Bob sorted their timestamps into time bins and thus determined the states of these events, they can start deriving a sifted key from this information. By announcing to each other the basis for each photon they measured, Alice and Bob can *sift*, i.e., filter, their four arrays of each nesting branch containing the repetition cycle numbers per state. Due to the 50/50 beam splitter in their interferometers, Alice and Bob discard approximately $50\,\%$ of their timestamps where they measured in a different basis. For the remaining

$50\,\%$ of the events, Alice and Bob measured in the same basis, and the entanglement of the photons ensures that they measured the same state. Again, this is only true for perfect setups and without attackers. When running the key sifting algorithm on the data used in this thesis, in around $2.5\,\%$ of the events Alice and Bob measure different states even though they measure in the same basis. It is therefore that their sifted key is not $100\,\%$ equal, which will be corrected during the post-processing described in section 4.6. Like the timestamp analysis, the key sifting algorithm is implemented using the libraries numpy and sortednp for performance reasons, and does not need numba as the legacy scripts do.

When announcing their list of bases, Alice and Bob must not reveal any information about the states. Still, this information must be preserved to eventually obtain the sifted key. To gather the information about the time basis, the two arrays containing the repetition cycle numbers of the early and late time bin are merged using the function `merge` of sortednp. When doing so, the information about the states gets lost, i.e., whether the photon was detected in the early or late time bin, so it is safe to publicly exchange this array between Alice and Bob. To gather the information about the phase basis, the two remaining arrays are merged that contain the timestamps of the left and right detector that got sorted into the central time bin. Again, the information about the states gets lost, which is the detector that clicked, i.e., that detected a photon. Thus, it is safe to also exchange this array between Alice and Bob.

Alice and Bob each send these two arrays containing the repetition cycle numbers of the time and phase basis to the other party. They use these two arrays together with the function `intersect` of sortednp to obtain all repetition cycle numbers where they measured in the same bases. When using the function `intersect`, the parameter `indices=True` is passed that returns for each element of the resulting array the indices in the two original arrays that got intersected. This is useful to create a sifted key containing only bit values 0 and 1 based on the array containing all repetition cycle numbers where Alice and Bob measured in the same bases.

When assigning bit values to the sifted key, it is important to respect the correlation coefficient C close to -1, leading to anti-correlation in the phase basis for Alice and Bob. Thus, whenever Alice and Bob detect entangled photons in the phase basis, Alice's left detector clicks when Bob's right detector clicks and vice versa. Alice and Bob therefore assign opposite bit values to their detectors for phase-basis events in the sifted key. For time-basis events, they assign the same bit values. As computers work with complete bytes instead of single bits, up to seven bits are discarded to truncate the sifted key to a multiple of $8\,\mathrm{bit}$. This makes processing the sifted key easier, but if there is interest in keeping those seven bits, padding could be used where the remaining bits are filled with 0 before sending the sifted key to the next processing step. Obviously, these padded bits must be removed before using the sifted key to, e.g., derive a secret key.

Assigning bit values like this for the sifted key, the information about the basis is lost, i.e., for a single bit in the sifted key it is not clear whether this bit was measured in the time or phase basis. For the latter calculation of the correlation coefficient C, however, it is necessary to remember the basis of each sifted key bit. Therefore, as part of this key sifting another bit string is created that is of the same length as the sifted key. In contrast to the sifted key, Alice and Bob don't put bit values based on the measured state into this string but 0 whenever they measured in the time basis and 1 when measuring in the phase basis. Like the sifted key, this bit string is also truncated to a multiple of $8\,\mathrm{bit}$. Both bit strings containing the sifted key and its corresponding bases are now ready to be passed to the post-processing.

Comparing Protocol Bandwidths

Two potential data protocols have been considered for this thesis for exchanging the list of bases after each measurement interval of $T_\mathrm{t} = 90\,\mathrm{s}$. Each protocol is discussed with the estimated data volume required for sending the list of bases. This means that the total required data volume is two times the volumes presented below, since each party sends and receives a list of bases.

Given the source produces entangled photons at a rate of $f_S = 215\,\text{MHz}$ [10], each cycle has a total length of $T_c = 4.65\,\text{ns}$. This leads to a total amount of $N = 19.35 \times 10^9$ cycles which can be encoded in $B = 35\,\text{bit}$. Encoding the cycle in B bits allows cycle lengths as low as $T_{\text{c, min}} = T_t/2^B = 2.62\,\text{ns}$ or repetition rates of up to $f_{\text{S, max}} = 381.77\,\text{MHz}$. Each party has two detectors which detect around 20×10^3 to $f_d = 30 \times 10^3$ photons per second and detector [10].

The first potential data protocol consists of sending two bits for each cycle, regardless of whether a photon got detected or not. The first bit indicates whether a valid photon got measured; the second bit indicates its corresponding base. Over a measurement period of $T_t = 90\,\text{s}$, this leads to a total data volume of

$$V_1 = 2\,\text{bit} \cdot N = 38.70\,\text{gigabit} = 4.84\,\text{gigabyte}\,. \tag{4.7}$$

The second protocol includes sending the full repetition cycle number encoded in B bits for every photon measured by the two detectors. The repetition cycle numbers are sent in two batches, where the first batch contains all cycles with photons measured in the phase basis and the second batch all photons measured in the time basis. This results in a data volume of

$$V_2 = 2 \cdot B \cdot f_d \cdot T_t = 0.19\,\text{gigabit} = 23.63\,\text{megabyte}\,. \tag{4.8}$$

The required photon detection rate $f_{\text{d, eq}}$ for equal data volumes $V_1 = V_2$ is

$$f_{\text{d, eq}} = \frac{N}{B \cdot T_t} = 6.14\,\text{MHz} \approx 205 \times f_d \approx 0.03 \times f_s\,. \tag{4.9}$$

Therefore, to have equal data volumes for both protocols, either the detection rate is increased by a factor of 205 or the losses are lowered to 15 dB so that in at least 3 % of the repetition cycles a photon is detected at the receivers. From these calculations, it becomes apparent that the second protocol is preferable over the first one in terms of required bandwidth. In this thesis, the second protocol has been implemented. However, instead of encoding the repetition cycle numbers in 35 bit, they are encoded in 64 bit since the underlying library used for calculations on the timestamps, numpy, does not support encoding integers in arbitrary bit lengths. This results in a data volume of

$$V_3 = 2 \cdot 64\,\text{bit} \cdot f_d \cdot T_t = 0.35\,\text{gigabit} = 43.20\,\text{megabyte}\,, \tag{4.10}$$

which is an increase of 83 % or 20 megabyte compared to V_2. If in the future there is interest to save this data volume, data compression algorithms like *gzip* can be used or numbers can be manually compressed to and decompressed from 35 bit when sending and receiving this data. Implementing the algorithms presented above with self-built integer types of 35 bit is not recommended since most likely this would make the algorithms much slower. In this project, performance is more important than saving bandwidth due to the 90-second analysis limit of each iteration.

4.6. Error Correction and Privacy Amplification

The sifted key derived from the measurement data in the previous section is not equal for Alice and Bob, but differs with a Quantum Bit Error Rate (QBER) of $2.5\,\%$ for the data analyzed in this thesis. To correct these errors, the sifted key is sent from Python — where the sifted key is generated — to Java using the socket connection presented in section 4.3.2. There, the sifted key is handed over to the post-processing code developed in ref. [11] that uses the Low-Density Parity-Check (LDPC) code algorithm to error-correct the key.

For the error-correction algorithm, an important size is the number of parity bits

$$M = f \cdot K \cdot H(\text{QBER})\,, \tag{4.11}$$

which is derived in ref. [11] using the desired LDPC efficiency f, the sifted key size K, and the Shannon entropy

$$H(\text{QBER}) = -\text{QBER} \cdot \log_2(\text{QBER}) - (1 - \text{QBER}) \cdot \log_2(1 - \text{QBER})\,, \tag{4.12}$$

which is also derived as function of the QBER in ref. [11]. Using the LDPC code algorithm, Alice sends M parity bits to Bob that allow him to correct his key so that it matches Alice's key. It is important for Alice to send enough parity bits, because otherwise Bob might not be able to error-correct his key. Therefore, it is important to have a precise value of the QBER. In this thesis, the value for the QBER is estimated by

$$\text{QBER} = \frac{e}{n}\,, \tag{4.13}$$

where e is the amount of flipped bits in the sifted key and n is the length of the sifted key. Alice and Bob do not know the exact value of the QBER for their sifted keys, since they only compare a subset of the sifted key. Thus, the QBER used for the error-correction is corrected by its uncertainty

$$\Delta\text{QBER} = \frac{\Delta e}{n} = \frac{2\sqrt{e}}{n}\,, \tag{4.14}$$

which was derived using Gaussian propagation of uncertainty with $\Delta e = 2\sqrt{e}$ as the uncertainty of the amount of flipped bits. This leads to a number of parity bits

$$M = f \cdot K \cdot H(\text{QBER} + \Delta\text{QBER})\,. \tag{4.15}$$

During the post-processing, Alice and Bob exchange three messages related to the LDPC code algorithm that are encoded as strings. In the first message, Alice asks Bob to send her a subset of his sifted key at randomly chosen indices. In the second message, Bob answers with revealing these sifted key bits. Alice uses this information to generate the LDPC code matrices and sends them to Bob, which allows him to correct his sifted key so that it matches Alice's one. After having exchanged this third post-processing message, both Alice and Bob use privacy amplification to obtain a secret key that is more secure than the sifted key before error correction. When Bob finishes his post-processing and successfully obtains a secret key, he informs Alice about this so that both save this secret key to their key database. In case Bob fails to obtain a secret key, both Alice and Bob discard this iteration.

There are two reasons why post-processing could fail: either the amount of parity bits sent from Alice to Bob was not enough for Bob to error-correct his key, or the amount of parity bits was too high and too much information was shared. The latter situation is equal to a QBER higher than $11.5\,\%$, which is the limit for generating a secure key [27]. To summarize, it is crucial to send the right amount of parity bits that is neither too high nor too low.

During the post-processing, not only the QBER but also the correlation coefficient C is estimated based on the exchanged subset of the sifted key. Analogously to the calculation of the QBER, a value for C is calculated by comparing each bit of the subset of the sifted key. In contrast to the calculation of the QBER, bits originating from the time basis are ignored. At the end of the post-processing, the obtained value for C is sent back to the Python code where it can be used to adapt the interferometers' temperatures.

5. Results

With the software implemented in this thesis, it is possible to run a QKD session using the BBM92 protocol based on measurement data recorded by previous setups. Before discussing the obtained detector offsets and key rates in sections 5.2 and 5.3, respectively, the results of the inter-process communication are discussed in section 5.1. For these results, two different sets of measurement data are used that got captured in October 2021 by the setup from ref. [9]: one from the subnetwork Alice-Diana[1], featuring a total distance of $47.5\,\mathrm{km}$ between the parties, and one from the subnetwork Bob-Charlie[2], with a distance of $60.5\,\mathrm{km}$ between the parties.

5.1. Inter-Process Communication

Using QNCC as the main communication hub, Alice and Bob are able to exchange messages between the two parties, but also between the Java and the Python processes of one party. In this section, it is described how the memory leak in QNCC was fixed and some considerations on the security of the network communication are given.

5.1.1. Stopping the Memory Leak

As mentioned in section 4.1, QNCC had the problem of steadily using more and more memory with each iteration. This became a serious problem, since after roughly 25 iterations there was no more free memory space available of the development machine's $16\,\mathrm{GB}$ memory in total, resulting in a freeze of the whole machine. Only after some time the machine's operating system — Arch Linux in this case — killed the processes of QNCC so that the machine became usable again. This memory problem was investigated using an analysis tool specifically designed for Java programs[3]. With this tool, the memory usage can be monitored besides other quantities like CPU usage and total amount of classes.

In programming languages like Java, developers cannot *allocate* and *deallocate* memory space on their own, but instead have this handled by Java. Whenever creating objects, Java automatically allocates the right amount of memory. As soon as an object is no longer needed, deleting all references to it is all that is required for developers to do. For example, a Java variable could contain a list of objects that points to a certain amount of objects. As soon as the objects are no longer needed, a new empty list can be instantiated and assigned to this variable. The previous list of objects is then no longer accessible from the Java program, however, its data is still in the program's memory. At this point, it simply "forgot" that this data exists and where it lies in the memory.

[1]In the lab notebook, this measurement in the Alice-Diana subnetwork is called *attempt 3*.
[2]In the lab notebook, this measurement in the Bob-Charlie subnetwork is called *attempt 1*.
[3]*VisualVM* by Oracle Corporation

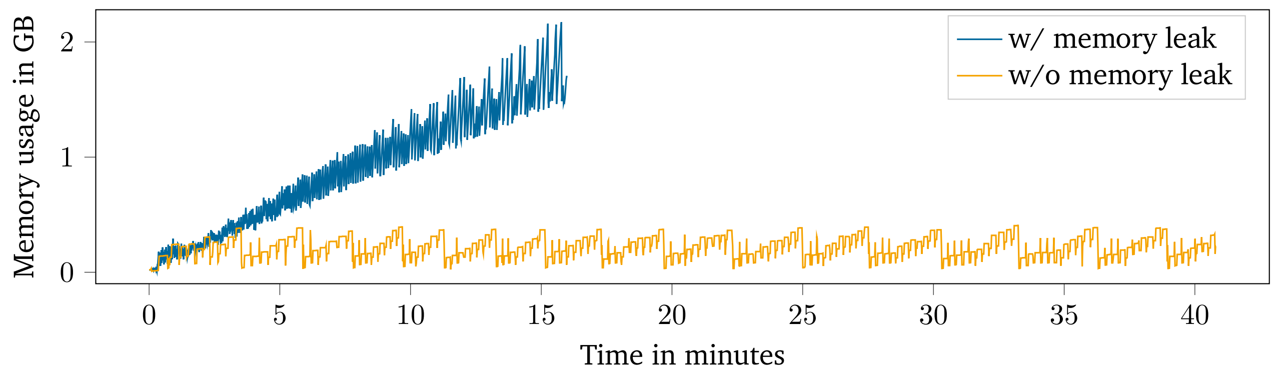

Figure 5.1.: The diagram shows the memory usage of QNCC with the bug that led to the memory leak in blue, and with the bug fixed in version 0.1.0 in orange. While the memory usage in orange represents a complete run of QNCC over 160 iterations, the program had to be stopped for the memory usage in blue after just 25 iterations because only little memory space was available anymore.

To prevent the memory size of Java programs to be ever increasing, programming languages like Java feature a *garbage collector*. The garbage collector is regularly run and removes inaccessible objects from a program's memory. It does not, however, remove objects that will not be used again in a program but are still referenced somewhere. The analysis tool used to analyze the memory problem allows to execute the garbage collector. Doing this does not change much about the memory size of QNCC. It stays at a high level and continues to increase with each iteration. To further analyze the memory problem, the analysis tool allows to do a *heap dump* which essentially is a capture of the program's memory at this point in time. This heap dump can be directly analyzed in the analysis tool. When doing this for QNCC after some iterations, it can be seen that the largest amount of classes is given by some class of the post-processing software. After running QNCC for some iterations, the number of instances of this class stays around a level of 100 000 instances without increasing in time. To find the part of the code using more and more memory space with each iteration, the *retained size* of the program's objects can be calculated. It gets apparent that it is the class `KeyGenerator` that is ever increasing in size. This class got implemented in ref. [12] to manage the key generation process in QNCC. The analysis tool allows to further inspect the object's attributes and their sizes. This leads to the final conclusion that it is the attributes of the type `ObjectInputStream` and `ObjectOutputStream` that are using the vast majority of memory space.

The documentation of these classes describes this behavior of storing and remembering all sent objects. When implementing the classical channel between Alice and Bob, the methods `readObject` of `ObjectInput-Stream` and `writeObject` of `ObjectOutputStream` got used. If storing the objects is not wanted, the methods `readUnshared` and `writeUnshared` should be used instead. The implementation in QNCC is therefore switched to latter methods, resulting in a nearly constant memory usage of around 350 MB, as shown in figure 5.1. The sawtooth pattern that becomes apparent in this figure is a result of the garbage collector. The garbage collector does not run continuously but only at specific points in time that depend on the concrete implementation of the Java programming language on the host machine. Therefore, the memory usage repeatedly increases until the garbage collector is run, which results in a drop of the used memory space. While the version with the memory leak must be stopped after just 25 iterations due to the memory running out of space, the version without the memory leak is able to analyze all 160 iterations without increasing memory usage in size. The current version in orange is much faster than the old one in blue, because it uses the optimized Java-Python communication channel that allows for faster analyses. With the memory usage now being limited over time, QNCC is capable of handling an arbitrary amount of measurement iterations.

5.1.2. Security Considerations on Serialization

When working with serialization, it is very important to deserialize only trusted data, as stated in the documentation of `ObjectInputStream`[4]: “Warning: Deserialization of untrusted data is inherently dangerous and should be avoided.”

Otherwise, attackers could abuse weaknesses in deserialization algorithms of any class present in the QNCC code tree for attacks such as *denial-of-service attacks* [32] and even *remote code execution* [33]. When being affected by such attacks, an attacker Eve sends a malicious message to its victim Bob that is deserialized by him. The message is specifically crafted by Eve to abuse a weakness in one of the classes present in Bob’s software. For denial-of-service attacks, deserializing this malicious message leads to a situation where Bob, e.g., needs a lot of time to make some complicated calculation. During this time, Bob might not be able to respond to other messages, thus the name denial of service. Attacks featuring remote code execution allow executing arbitrary code on the victim’s machine. By this, not only could the victim’s machine and thus potentially also its network be infected with some type of malware, but also all secret keys generated with QKD could be stolen. These types of attacks are possible because an attacker is able to freely specify which class a byte stream is deserialized to. Thus, any class present in the software that is `Serializable` offers an additional attack surface. This is especially critical for weaknesses in the Java Standard Library, since these weaknesses are present in any software written in Java.

Nevertheless, using deserialization with untrusted data does not inevitably mean being vulnerable to attacks. A Java software could deserialize untrusted data without having any security vulnerability due to this. However, deserializing untrusted data makes attacks much easier, since a single class containing an otherwise non-critical weakness is enough to enable attacks such as the ones mentioned above.

Unfortunately, the classical communication channel of QNCC between Alice, Bob, and the source got implemented using Java’s serialization algorithms, meaning that QNCC deserializes untrusted data and thus is potentially vulnerable to these types of attacks. In appendix C, it is shown where QNCC deserializes untrusted data. In this section, a solution to this problem is sketched that, however, does not get implemented as part of this thesis, as the focus of this thesis is on making the BBM92 analysis software work on distinct computers and not on hardening QNCC.

To switch the communication of the classical channel of QNCC between Alice, Bob, and the source to some other protocol, first an analysis on which messages exactly are exchanged via this channel is needed. On the one hand, there are messages generated by the Python software during timestamp analysis and key sifting that are encoded to bytes in the Java-Python protocol presented in section 4.3.2. These messages are not interpreted by Java, but simply forwarded between Alice’s and Bob’s Python code. Therefore, the new protocol must be able to send and receive messages consisting of byte data. On the other hand, there are messages exchanged by QNCC for managing the QKD session, such as Alice asking Bob for a QKD session, the source informing Alice that photon-pair production has started, and so on. These types of messages can be handled by defining a set of message types and their corresponding integer values, starting from 0 for the first message type, 1 for the second, and so forth. Finally, the messages exchanged during post-processing are strings that can be converted to bytes, too. To summarize, the protocol implemented for the Java-Python communication is suitable for the Alice-Bob-source communication, as it allows sending messages indicating its type and containing byte data. Replacing the protocol of this public communication channel is definitively required, and until this is done, QNCC should not be publicly exposed to the Internet by, e.g., opening ports in a network’s firewall. Due to the firewall, QNCC is protected against attacks from the Internet, and for research and development, the computers running QNCC can be connected via a *virtual private network* (VPN).

[4] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ObjectInputStream.html`

Figure 5.2.: This table shows the relative offsets of the Alice-Diana subnetwork obtained via cross-correlation of the timestamp lists in the first iteration with Bob's left detector as the reference. To calculate the fiber length differences, $\Delta s = O_{\text{rel}} \cdot c_0/n$ was used with $n = 1.4682$ for the used wavelength of $\lambda = 1550\,\text{nm}$ [34].

Detector	O_{rel}	Δs
Alice left	33 930 831 ps	6.9 km
Alice right	33 950 337 ps	6.9 km
Bob right	18 867 ps	3.9 m

5.2. Timestamp Analysis

When analyzing the timestamp lists as the first step of each QKD session, information about the offsets of the detectors is obtained. This is discussed in section 5.2.1, before giving some security considerations on time-bin sorting in section 5.2.2.

5.2.1. Detector Offsets

To find coincidences between Alice's and Bob's detectors, their timestamp lists must get aligned to each other. This is done in the first iteration with a cross-correlation that uses Bob's left detector as the reference detector, as described in section 3.2.1. For the Alice-Diana subnetwork, this leads to the relative offsets from eq. (3.6) and fiber length differences shown in table 5.2. For Alice, the values Δs represent the length differences of the fibers connecting Alice and Bob with the source. Bob's offset corresponds to the additional distances introduced due to the additional fiber length via the optical circulator and the different lengths of the cables connecting the detectors with the time tagger.

In all subsequent iterations, the offsets of the detectors are recalibrated before the timestamps are sorted into time bins. The first step in this reconfiguration is the calculation of the central peak offset of the reference detector using the peak-finding tool. These values are used as reference offsets to pre-calibrate all other detectors. Only then the peak-finding tool is used to obtain the central peak offsets of the other detectors. The resulting values are shown in figure 5.3, where "Bob left" is the reference detector. Its values are 200 times larger than all other central peak offsets, which is a result of two effects: the restart of the time taggers at the end of each iteration and the cutting of the timestamp lists every $90\,\text{s}$. $90\,\text{s}$ is not a multiple of the repetition cycle length $9.1\,\text{ns}$, which leads to movements of the three-peak structure in every iteration. The larger influence, however, is given by the time tagger restarts that were required in the setup due to hardware problems of the time taggers [10]. These restarts take a different amount of time every time they are done. With the clock recovery from ref. [9], the restarts are no longer necessary, so smaller variations of the offsets are expected. The central peak offsets of the other detectors are much smaller with values around $\pm 100\,\text{ps}$. These mainly come from thermal expansion in the fibers.

The sum of these offsets leads to the total offsets O_{t} shown in figure 5.4. There, it becomes apparent that the total offsets are dominated by the reference offsets. Apart from the displacement due to the relative offsets from the initial iteration, plotting the total offsets of all four detectors on a scale of μs and ns results in the same curve being shown four times. The differences in these curves is on a scale of $\pm 100\,\text{ps}$ and therefore not visible.

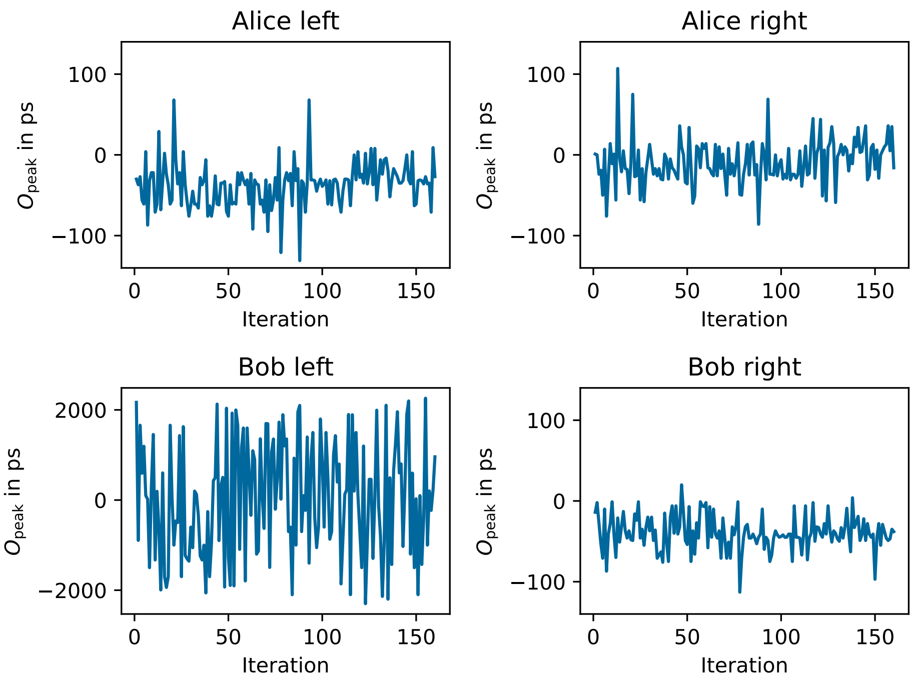

Figure 5.3.: The central peak offsets O_{peak} of Alice's and Bob's left and right detectors in the Alice-Diana subnetwork are shown. In this setup, Bob's left detector is chosen as the reference detector. Still, it has a central peak offset to center the peak structure and thus allow sorting events into time bins.

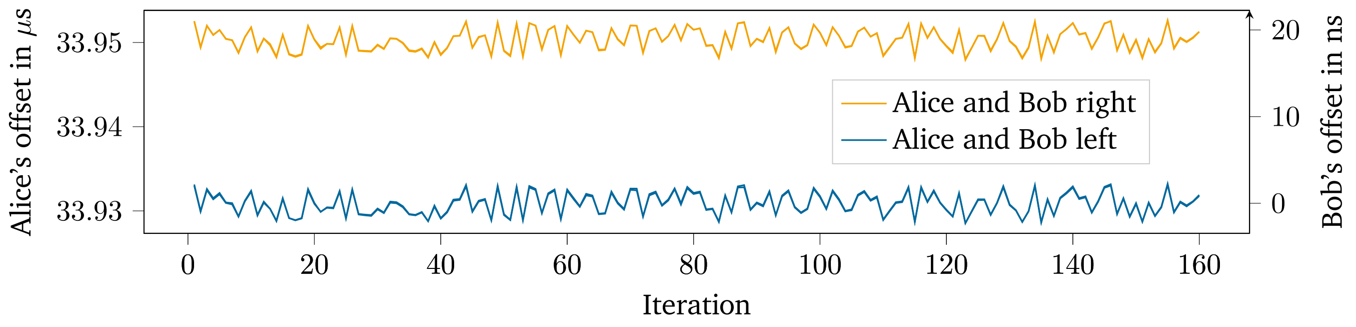

Figure 5.4.: The total offsets $O_t = O_{\text{rel}} + O_{\text{ref}} + O_{\text{peak}}$ of Alice's and Bob's left and right detectors are shown. In this setup, Bob's left detector is chosen as the reference detector that is used to obtain offsets for the other detectors. Therefore, the offsets of Bob's left detector are close to 0. It is not exactly 0 due to the central peak offset of the reference detector. All other offsets are dominated by these reference offsets. Therefore, all curves look identical on this scale and Alice's and Bob's detector offsets cannot be distinguished from each other.

5.2.2. Security Considerations on Time-Bin Sorting

In an ideal setup, the offset calibration and the time-bin sorting presented in the previous sections is all that is needed to start the key sifting process. In practice, however, the photon-pair production is not perfect, attackers might try to undermine the security of the key exchange, and the single-photon detectors are imperfect, too, because they, e.g, have nonzero dead times and dark count rates. Due to the nonzero dead time, whenever a photon is detected by the detector it is not able to detect another photon for a certain amount of time. Nonzero dark count rates lead to detections by the detector, even in the absence of photons. With an imperfect photon-pair production, it might happen that both entangled photons get directed to a single party instead of getting distributed among both parties, or that more than one photon pair gets created during SPDC. When this happens, a party could detect an event on each detector within the same repetition cycle. These events are filtered out by extending the algorithm presented in section 4.4. This extended algorithm is not presented here, but it can be found in the class `TimestampAnalysis`. Essentially, it is a combination of `merge`s and `intersect`ions of the library sortednp. For the data analyzed in this thesis, there are between 100 and 200 of such events that get filtered out of in total 6500 sifted key events.

However, two events with same repetition cycle number in two detectors' timestamp lists are only part of the problem. Philipp Kleinpaß elaborated in his Master thesis in ref. [35] on the vulnerability of the BBM92 protocol regarding using events from one detector while the other detector is in its dead time. In the setup of this thesis, the detectors are configured with a dead time of $10\,\mu$s while a repetition cycle has a period of 9.1 ns [10]. This means that after one detector clicked, there are about 1000 repetition cycles where only the other detector is capable of detecting photons. To mitigate this problem, events that got detected during the dead time of another detector should get filtered out before sorting timestamps into time bins. Obviously, this would lead to a lower sifted and thus also to a lower secure key rate, but would therefore eliminate such a loophole.

5.3. Sifted and Secure Keys

Having all steps required for a QKD session implemented in QNCC over the course of this thesis allows Alice and Bob obtain a shared secret key using historical measurement data obtained with the legacy measurement scripts. The measurement data of the Alice-Diana subnetwork primarily used in this thesis to test QNCC consists of 160 iterations, each with a period of 90 s and timestamp file sizes around 15 MB to 20 MB. Running QNCC with this measurement data takes around 35 minutes to analyze all iterations on a computer with Arch Linux as the operating system, 16 GB of memory, and a 4-core Intel Core i7-8550U. This means that analyzing a single iteration takes around 13 s which is well below the 90-second limit. In figure 5.5, the sifted and secure key rates are shown. The sifted key rates lie around 6500 bit in 90 s, or 72 bit/s. The secure key rate lies around 2500 bit in 90 s, or 28 bit/s. In contrast to the sifted key rate, the secure key rate shows large variations with rates as low as 1200 bit in 90 s (13 bit/s) and rates as high as 3700 bit in 90 s (41 bit/s). This variation is a result of the dependency of the number of parity bits M from eq. (4.11) on the QBER. It is empirically found that the uncertainty-corrected QBER together with a desired LDPC code efficiency of $f = 1.8$ allows Bob to error-correct the sifted key with the amount of parity bits sent by Alice. For other data sets than the Alice-Diana subnetwork's one, different LDPC code efficiencies might be required. In particular, for the measurement data of the Bob-Charlie subnetwork shown in figure 5.7 there are two iterations 5 and 26 where the secure key rate drops to 0. If this happens, Alice and Bob discard the iteration and store no secret key to their database. Looking at the QBER values in figure 5.8, it can be seen that these iterations had high QBER values in the $99\,\%$ sample and even higher values in the $12\,\%$ sample which led to too much parity bits

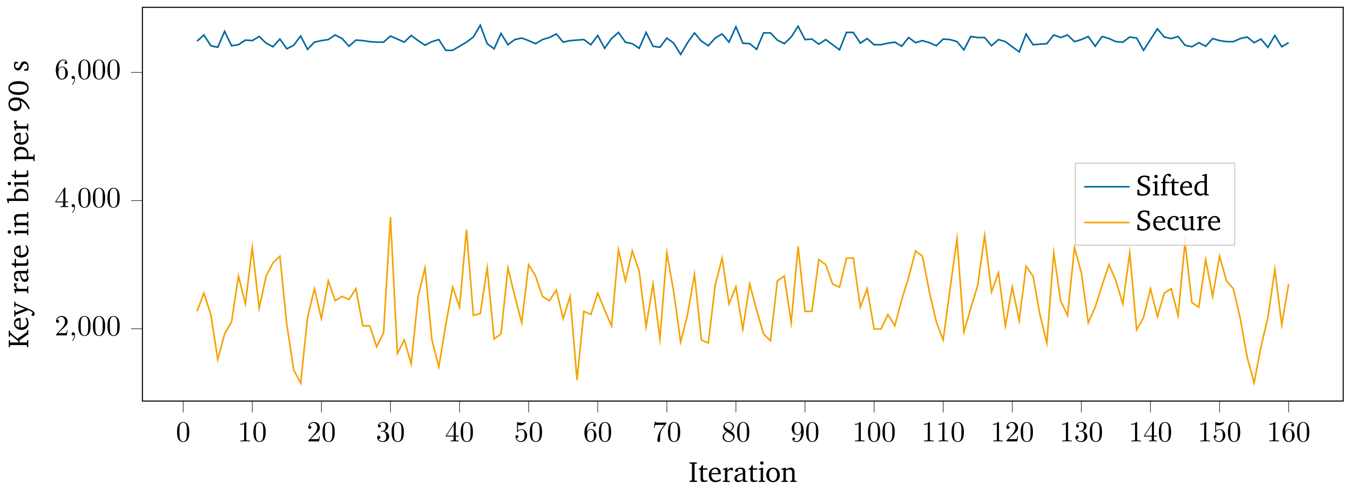

Figure 5.5.: The diagram shows the secure and sifted key lengths during an analysis period of 160 iterations of the subnetwork Alice-Diana. The sifted key rate shows a small variation around 6500 bit in 90 s, while the secure key rate shows large variations around 2500 bit in 90 s.

being shared. As stated in section 2.1.4, the limit for generating secure keys is a QBER of 11.5 %. Due to the LDPC code efficiency of $f = 1.8$, already lower QBER values lead to this limit getting crossed. Therefore, it is important to select the right amount of parity bits that is neither too high nor too low.

In figure 5.6, the obtained QBER value for different sizes of subsets is shown. In orange, QBER values are shown that are determined on a random 12 % subset of the sifted key with the uncertainty of eq. (4.14). These QBER values are used to determine the number of parity bits and thus error-correct the sifted key. In blue, the QBER values are shown as determined on a 99 % subset of the sifted key. It becomes apparent that the QBER values in blue still slightly vary around a level of 2.5 % with values as high as 3 % and as low as 2 %. Nevertheless, the QBER values obtained on the 12 % subset vary much more with values as low as 1.1 % and as high as 3.9 %. In average, both sets of QBER values lie around 2.5 %. To not send too few parity bits, the sum QBER + ΔQBER is used when determining the number of parity bits. In figures 5.6 and 5.8, it can be seen that by adding ΔQBER the QBER value obtained is above the QBER values in blue in most of the iterations, but there are still iterations where the value in orange is below the QBER value in blue. This means that most of the time enough parity bits would have been sent with a desired LDPC code efficiency closer to 1, but the other values required higher LDPC code efficiencies for error-correction to complete.

In figure 5.9, the correlation coefficient is shown as obtained from random 12 % and 99 % subsets in the Alice-Diana subnetwork. The correlation coefficient is used to adapt the temperatures of the interferometers. Like the QBER, the correlation coefficient on the 12 % subset shows larger variations than on the 99 % subset. In the legacy scripts, the temperatures are only adapted every second iteration to wait for the temperature change to distribute in the interferometer. When no longer obtaining the correlation coefficient from the whole sifted key, as in the legacy scripts, but from a subset, it might be necessary to wait even longer than two iterations when adapting the temperatures. Otherwise, the temperature might be adjusted due to statistical fluctuations even though there is no real phase drift in the interferometers. One possible option is to only adapt the temperatures if two correlation coefficients in subsequent iterations indicated such a necessity. By this, overcorrecting can be avoided.

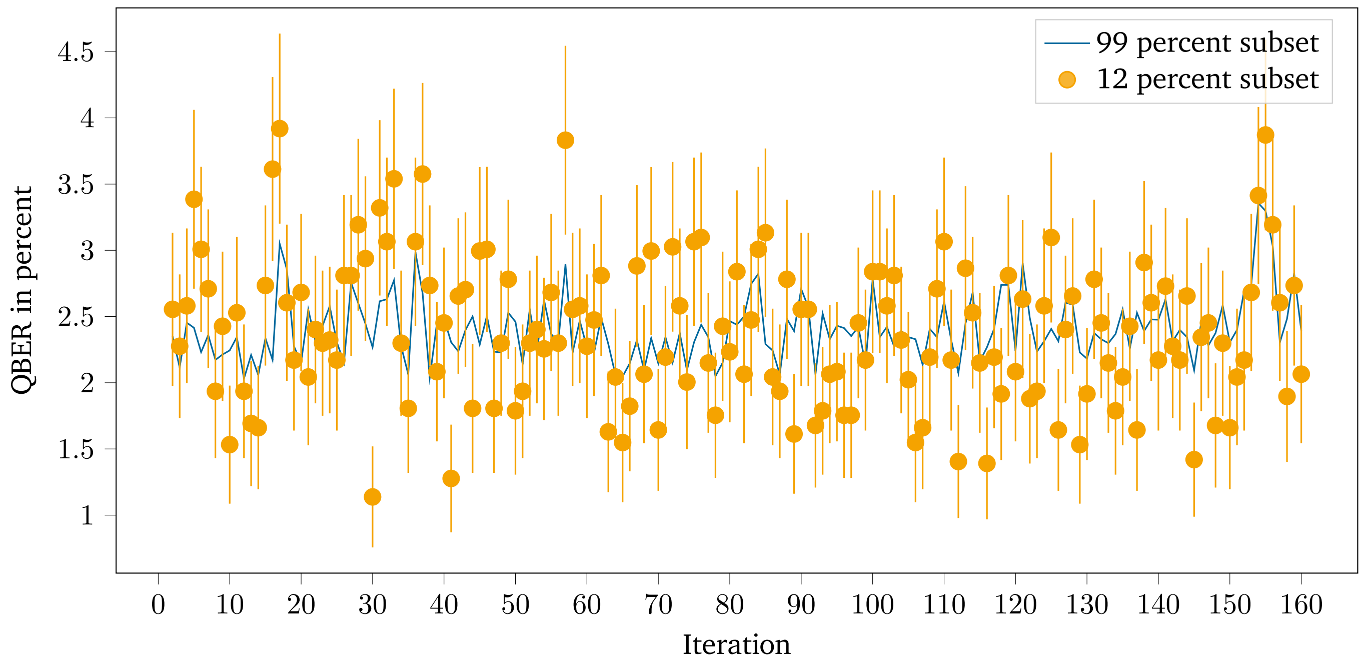

Figure 5.6.: This diagram shows the QBER of the Alice-Diana subnetwork as obtained on a subset of $12\,\%$ of the sifted key bits with a ΔQBER uncertainty and on a subset of $99\,\%$. For the error-correction algorithm to work, it is important to use a value larger than the blue QBER value when determining the number of parity bits.

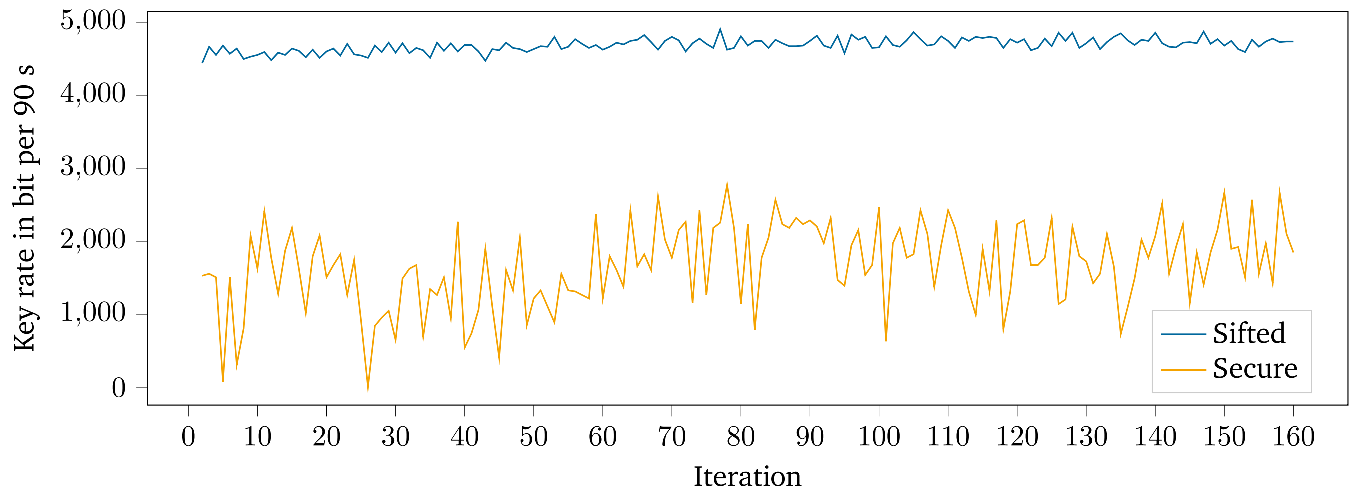

Figure 5.7.: The diagram shows the secure and sifted key lengths during an analysis period of 160 iterations of the subnetwork Bob-Charlie. The sifted key rate shows a small variation around $4500\,\mathrm{bit}$ in $90\,\mathrm{s}$, while the secure key rate shows large variations around $1500\,\mathrm{bit}$ in $90\,\mathrm{s}$. In iterations 5 and 26, too much parity bits were exchanged and thus no secure key could be generated.

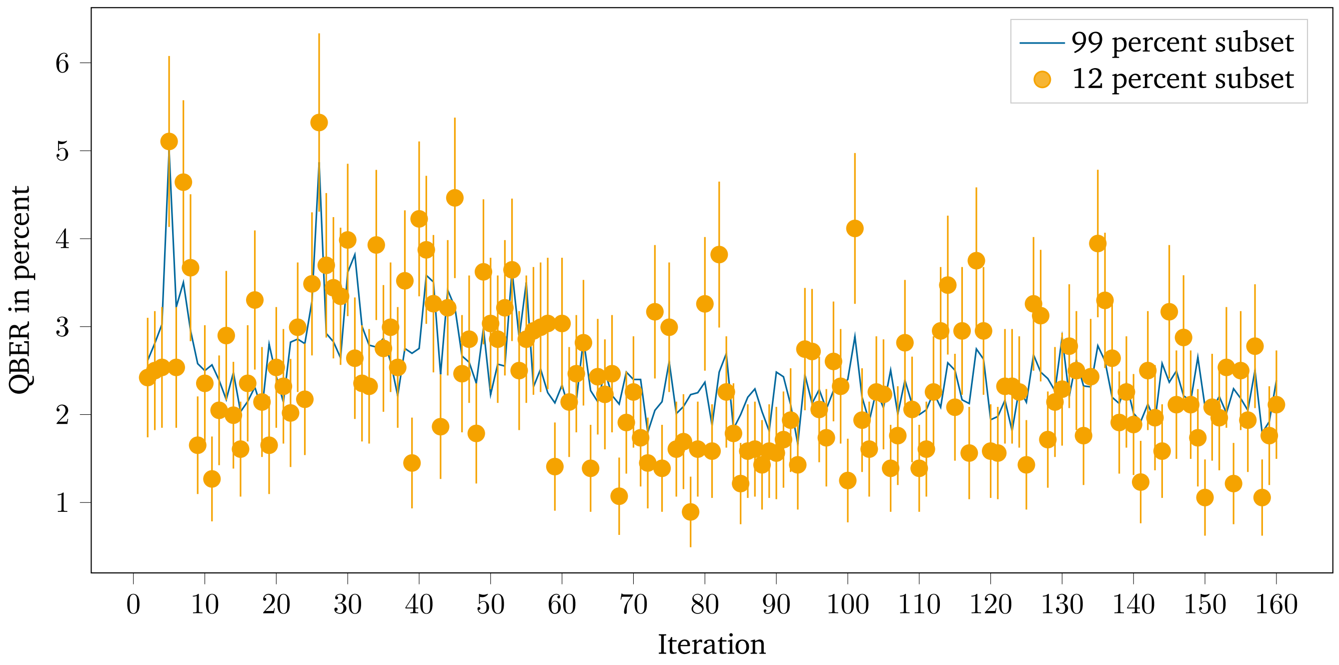

Figure 5.8.: This diagram shows the QBER of the Bob-Charlie subnetwork as obtained on a subset of $12\,\%$ with a ΔQBER uncertainty and on a subset of $99\,\%$. For the error-correction algorithm to work, it is important to use a value equal to or larger than the blue QBER value when determining the number of parity bits.

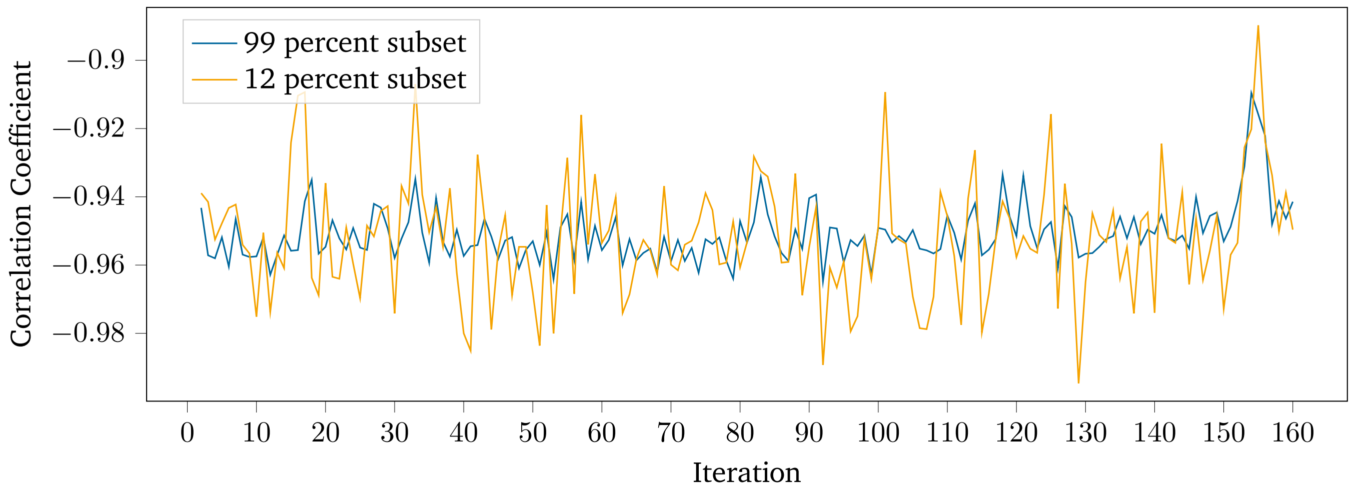

Figure 5.9.: This diagram shows the correlation coefficient C as obtained in the Alice-Diana subnetwork on a subset of $12\,\%$ and on a subset of $99\,\%$.

QNCC got tested on more historical data sets than shown here and was able to produce secure keys with comparable rates in all cases. While the development of QNCC happens on Linux, the computers running the measurement scripts must run Windows since the time taggers' software is only available on Windows. Running QNCC on Windows works and produces the same results as on Linux. Since both Python and Java are not pre-compiled to native machine code, the same binary files of QNCC can be used on Windows and Linux. Using the key storage functionalities of QNCC, it is seen that Alice and Bob obtain the same private keys. In the graphical user interface, a chat between Alice and Bob can be started that transfers end-to-end-encrypted messages using the secure keys obtained from the QKD session.

6. Summary and Outlook

Over the course of this thesis, a software got developed that implements the BBM92 protocol based on previously recorded measurement data which allows two communicating parties to establish a shared secret key using Quantum Key Distribution (QKD) on Windows- and Linux-based operating systems. Combining several previous works, the software uses network connections implemented in ref. [12] as means of communication between Alice, Bob, and the source, which allows all of them to be in distinct places. The post-processing software developed in ref. [11] got integrated into this thesis' software to obtain secret keys. Taking around $13\,\mathrm{s}$ to generate a secret key from measurement data collected over a period of $90\,\mathrm{s}$, it produces secret keys at a rate of $28\,\mathrm{bit/s}$ out of a sifted key rate of $72\,\mathrm{bit/s}$ with QBER values around $2.5\,\%$. The data used for this QKD session got recorded in October 2021 using a fiber deployed between Darmstadt and Griesheim with a length of $26.8\,\mathrm{km}$, resulting in a total distance of $47.5\,\mathrm{km}$ between Alice and Bob (subnetwork Alice-Diana in ref. [9]).

The software got implemented using the Quantum Network Control Center (QNCC) developed in ref. [12] as a basis. QNCC is written in Java, requiring a communication channel to the Python software developed during this thesis that generates the sifted key and eventually will control the measurement devices. This communication channel got implemented in this thesis using same-device network connections, removing the slow, insecure, and unsuitable file-system communication originally implemented in ref. [12]. Reusing parts of code from the previous setup, the software automatically calibrates detector offsets using cross-correlation and peak-finding tools. The subsequent time-bin sorting and key sifting of the BBM92 protocol got implemented from scratch using suitable Python libraries[1] designed to work with large data sets. During the key sifting, invalid events are filtered out that could otherwise lead to attacks against the QKD session. Finally, the obtained sifted keys are error-corrected and privacy-amplified using the post-processing code implemented in ref. [11]. This results in identical secure keys being obtained by both parties that can be used by other programs for symmetric encryption like, e.g., AES. QNCC allows to directly chat with the other party from within the graphical user interface using the secret keys obtained during the QKD session.

Having this implementation of the data analysis steps of BBM92 in place, the next step is to decentralize the management of the measurement devices. The scripts used for this in previous setups only work in a centralized mode where all measurement devices are connected to a single computer. Once the measurement devices can be controlled from separate computers in a decentralized setup, QNCC can be used to run QKD sessions among parties in distinct places. Before this thesis, the QKD setup was not able at all to be used in distinct places because the analysis steps required the full knowledge of all parties' measurement results. As shown in ref. [9], the setup is able to connect up to 34 parties with distances of more than $100\,\mathrm{km}$ which can finally also be demonstrated in a decentralized setup once the measurement-device control is in place.

Another remaining question is the estimation of values for QBER and correlation coefficient C based on the knowledge of only a subset of the sifted key. Previous centralized setups used the knowledge of the complete sifted key to obtain a value for C and adapt interferometers' temperatures based on this value. Comparing the

[1]numpy and sortednp

whole sifted key is not possible in decentralized setups, since the communication channel used to compare the key is assumed to be public and non-confidential. Therefore, techniques must be found that allow strong estimations for QBER and C that are not too much influenced by statistical fluctuations. Once these techniques are developed, secure key rates are expected to increase significantly.

If QNCC should ever get used in real-world applications, some more considerations and modifications have to be made. In particular, a *threat model* is needed that clearly states what an attacker is capable of doing and which type of attacks are considered to be withstood by QNCC. Based on this threat model, the severeness of potential vulnerabilities like the one presented in section 5.1.2 can be assessed and addressed, if necessary. Additionally, QNCC should not use RSA but some quantum-resistant algorithm for authentication, since RSA is broken by quantum computers. Besides these security concerns, QNCC should be usable without a graphical user interface to allow it to be included in other programs as a library offering services for obtaining secret keys using QKD.

A. Setting up the Quantum Network Control Center

The Quantum Network Control Center (QNCC) is a software written in Java and Python, with additional libraries as dependencies in both languages. The Java dependencies are recorded in the file *pom.xml*, which the building tool *Maven* uses to automatically download and include these dependencies when building the software. Similarly, the Python dependencies are recorded in the file *src/main/python/requirements.txt* which Python's dependency manager *pip* uses to automatically install all dependencies. For the Python dependencies on Windows to work, the *Visual Studio Build Tools* are required to be installed on the machine. In the *README.md* file of QNCC it is recorded how to install them.

Once the Python dependencies are installed, there are two ways to start QNCC: one is to execute the *run configuration* "Run all" or "Build .jar" in the *integrated development environment*, Intellij IDEA, and the other is to build QNCC on the command-line using the building tool Maven by calling `mvn package`. The generated *.jar* file can be found in the directory *target* and can be executed on any Windows or Linux machine. QNCC is started with that .jar file by calling

```
java -jar QuantumnetworkControlcenter-0.1-jar-with-dependencies.jar
```

The code of QNCC is stored in the project's GitLab group on the GitLab instance of RWTH Aachen. By installing Git on the developer's machine, the code can be downloaded onto the machine by calling `git clone [url]`. It is strongly recommended to make use of Git's and GitLab's functionalities in order to facilitate cooperative work on the code of QNCC.

B. Configuring the Quantum Network Control Center

The Python software developed in this thesis allows to configure various different parameters via a configuration file. The file is found at *src/main/python/configuration.py* and can be modified with a text editor to match the experiment's needs. All configuration parameters are described in this appendix.

`KEY_GENERATION_MODE`
Possible values: 0 or 1
Explanation: 0 starts a key generation that controls the experimental setup and obtains data from measurement devices, while 1 works without any experimental setup by using historical data.
Notes: Mode 0 is not usable in this thesis, but to be implemented by future students. Mode 1 requires historical data to be present in the folder *historical* of the user's QNCC directory. QNCC can be adapted to the structure of the historical timestamp data in the file *src/main/python/measuring/historical_measuring.py*.

`DEBUG`
Possible values: `True` or `False`
Explanation: Determines whether debugging information is logged to the terminal.

`REPETITION_TIME`
Possible values: 1 positive integer
Explanation: This integer is used to analyze the timestamps for time-bin sorting. With the setup of this thesis, it is set to 9100 ps.

`BIN_BOUNDS`
Possible values: A Python tuple with three tuples, each containing two positive integers with values between 0 and `REPETITION_TIME`.
Explanation: These integers are used as boundaries for the early, central, and late time bins during time-bin sorting. In this thesis, its values are 1020 and 2020, 4050 and 5050, and 7080 and 8080.
Notes: The integers must have the same unit as `REPETITION_TIME`.

`INITIAL_NESTING_OFFSETS`
Possible values: Tuple containing an arbitrary amount of integers, all with values between 0 and `REPETITION_TIME`.
Explanation: These integers are used to shift the timestamps when analyzing data with dense packing of pulse sequences. The length of the tuple implicitly defines the nesting level. In this thesis, its values are 0 and 4050.
Notes: The integers must have the same unit as `REPETITION_TIME`.

`ID900_RESOLUTION`
Possible values: 1 positive integer
Explanation: Determines the resolution of the time tagger. In this thesis, it is set to 13.
Notes: The integer must have the same unit as `REPETITION_TIME`.

`RELATIVE_PEAK_PROMINENCE`
Possible values: 1 float between 0 and 1
Explanation: Used when finding the central peak offsets. In this thesis, it is set to 0.8.
Notes: Value should be larger than 0.5, since otherwise satellite peaks would be identified that are roughly half the size of the central peak.

`CC_FIRST_BLOCK_FRACTION_US`
Possible values: 1 float between 0 and 1
Explanation: Determines the fraction of our timestamp data analyzed when cross-correlating in the initial calibration. In this thesis, it is set to 0.05.

`CC_FIRST_BLOCK_FRACTION_THEY`
Possible values: 1 float between 0 and 1
Explanation: Determines the fraction of the other party's timestamp data analyzed when cross-correlating in the initial calibration. In this thesis, it is set to 0.05.

C. Code Sections where Untrusted Data is Deserialized

In line 142 of the class `ConnectionManager` in the package `networkConnection`[1], incoming connections from the network on the server socket of QNCC get accepted and transferred into a new network socket:

```
clientSocket = masterServerSocket.accept();
```

In line 153 of the same class, this newly created socket gets transferred to an object of the class `Connection-EndpointServerHandler`:

```
cesh = new ConnectionEndpointServerHandler(clientSocket, this);
```

In line 100, it gets configured to interpret data flowing into this socket with `ObjectInputStream`:

```
serverIn = new ObjectInputStream(clientSocket.getInputStream());
```

Finally, in line 107 of class `ConnectionEndpointServerHandler`, any data coming into QNCC from the network via this socket gets deserialized using the method `readUnshared`:

```
receivedMessage = (NetworkPackage) serverIn.readUnshared();
```

This is the place where attackers could start potential attacks. While the code tries to interpret the object deserialized by `readUnshared` as an object of the type `NetworkPackage`, attackers can choose which class an object is deserialized to and thus abuse weaknesses of any other class, not just `NetworkPackage`. If the class with that weakness is not a subtype of `NetworkPackage`, QNCC would possibly terminate at this point, but this would already be too late in case of an attack.

[1]All line numbers, code samples, and class and package names refer to version 0.1.0 of QNCC.

List of Figures

2.1. Experimental setup 6
2.2. Simplified arrival histogram 6

3.1. Setup scheme of receivers 12
3.2. Arrival histogram with data from Alice-Diana subnetwork 12
3.3. State diagram of Quantum Network Control Center (QNCC) 13

4.1. Class diagrams of `AbstractKeyGeneration` and its subclasses 19
4.2. Sequence diagram of QNCC in first iteration 21
4.3. Sequence diagram of QNCC in first iteration 22

5.1. Memory usage before and after bug with memory leak 32
5.2. Relative offsets in the Alice-Diana subnetwork 34
5.3. Peak offsets in the Alice-Diana subnetwork 35
5.4. Total offsets in the Alice-Diana subnetwork 35
5.5. Secure and sifted key rates in the Alice-Diana subnetwork 37
5.6. QBER on $12\,\%$ and $99\,\%$ subsets in the Alice-Diana subnetwork 38
5.7. Secure and sifted key rates in the Bob-Charlie subnetwork 38
5.8. QBER on $12\,\%$ and $99\,\%$ subsets in the Bob-Charlie subnetwork 39
5.9. Correlation Coefficient C on $12\,\%$ and $99\,\%$ subsets in the Alice-Diana subnetwork 39

Bibliography

[1] Suetonius. “Exstant et Ad Ciceronem, Item Ad Familiares Domesticis de Rebus, in Quibus, Si qua Occultius Perferenda Erant, per Notas Scripsit, Id Est Sic Structo Litterarum Ordine, Ut Nullum Verbum Effici Posset; Quae Si Qui Investigare et Persequi Velit, Quartam Elementorum Litteram, Id Est D pro A et Perinde Reliquas Commutet.” In: *Vita Divi Julii*. 121, p. 56.6 (cit. on pp. 1, 3).

[2] Han-Sen Zhong et al. “Quantum Computational Advantage Using Photons”. In: *Science* 370.6523 (Dec. 2020), pp. 1460–1463. DOI: `10.1126/science.abe8770` (cit. on pp. 1, 3).

[3] Peter W. Shor. “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Nov. 1994, pp. 124–134. DOI: `10.1109/SFCS.1994.365700` (cit. on pp. 1, 5).

[4] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmén, eds. *Post-Quantum Cryptography*. Berlin: Springer, 2009. ISBN: 978-3-540-88701-0 (cit. on p. 1).

[5] Johannes Buchmann, Kristin Lauter, and Michele Mosca. “Postquantum Cryptography—State of the Art”. In: *IEEE Security Privacy* 15.4 (2017), pp. 12–13. ISSN: 1558-4046. DOI: `10.1109/MSP.2017.3151326` (cit. on p. 1).

[6] Tommaso Gagliardoni, Juliane Krämer, and Patrick Struck. “Quantum Indistinguishability for Public Key Encryption”. In: *IACR Cryptol. ePrint Arch.* 2020. DOI: `10.1007/978-3-030-81293-5_24` (cit. on p. 1).

[7] Johannes Buchmann. *Einführung in Die Kryptographie*. Sixth. Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. ISBN: 978-3-642-39774-5. DOI: `10.1007/978-3-642-39775-2` (cit. on pp. 1, 4, 18).

[8] Nicolas Gisin et al. “Quantum Cryptography”. In: *Reviews of Modern Physics* 74.1 (Mar. 2002), pp. 145–195. ISSN: 0034-6861, 1539-0756. DOI: `10.1103/RevModPhys.74.145` (cit. on p. 1).

[9] Erik Fitzke et al. “Scalable Network for Simultaneous Pairwise Quantum Key Distribution via Entanglement-Based Time-Bin Coding”. In: *PRX Quantum* 3.2 (May 2022), p. 020341. DOI: `10.1103/PRXQuantum.3.020341` (cit. on pp. 1, 6, 9, 11, 12, 15, 31, 34, 41).

[10] Lucas Bialowons. “Completion of a 4-Party Time-bin Entanglement QKD System”. Master thesis. TU Darmstadt, Aug. 2021 (cit. on pp. 1, 7, 11, 13, 15, 17, 25, 27, 34, 36).

[11] Jendrik Seip. “Fehlerkorrektur und Schlüsselaufbereitung für den Quantenschlüsselaustausch”. Master thesis. TU Darmstadt, Feb. 2021 (cit. on pp. 2, 9, 17, 23, 28, 41).

[12] Jonas Hühne et al. *Quantum Network Control Center - User Guide*. Tech. rep. TU Darmstadt, Mar. 2022 (cit. on pp. 2, 16, 17, 20, 23, 32, 41).

[13] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. 2015. ISBN: 978-1-4665-7026-9 (cit. on p. 3).

[14] C. E. Shannon. “Communication Theory of Secrecy Systems”. In: *The Bell System Technical Journal* 28.4 (Oct. 1949), pp. 656–715. ISSN: 0005-8580. DOI: `10.1002/j.1538-7305.1949.tb00928.x` (cit. on p. 3).

[15] Matt Curtin. *Brute Force: Cracking the Data Encryption Standard*. New York: Copernicus Books, 2005. ISBN: 978-0-387-27160-6 (cit. on p. 4).

[16] Joan Daemen and Vincent Rijmen. “The Block Cipher Rijndael”. In: *Smart Card Research and Applications*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 277–284. ISBN: 978-3-540-44534-0. DOI: `10.1007/10721064_26` (cit. on p. 4).

[17] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: `10.1145/359340.359342` (cit. on p. 4).

[18] Charles H. Bennett, Gilles Brassard, and N. David Mermin. “Quantum Cryptography without Bell’s Theorem”. In: *Physical Review Letters* 68.5 (Feb. 1992), pp. 557–559. DOI: `10.1103/PhysRevLett.68.557` (cit. on pp. 5, 8).

[19] Artur K. Ekert. “Quantum Cryptography Based on Bell’s Theorem”. In: *Physical Review Letters* 67.6 (Aug. 1991), pp. 661–663. DOI: `10.1103/PhysRevLett.67.661` (cit. on p. 5).

[20] Jürgen Brendel et al. “Pulsed Energy-Time Entangled Twin-Photon Source for Quantum Communication”. In: *Physical Review Letters* 82.12 (Mar. 1999), pp. 2594–2597. DOI: `10.1103/PhysRevLett.82.2594` (cit. on p. 5).

[21] Wolfgang Tittel et al. “Quantum Cryptography Using Entangled Photons in Energy-Time Bell States”. In: *Physical Review Letters* 84.20 (May 2000), pp. 4737–4740. DOI: `10.1103/PhysRevLett.84.4737` (cit. on p. 5).

[22] J. D. Franson. “Bell Inequality for Position and Time”. In: *Physical Review Letters* 62.19 (May 1989), pp. 2205–2208. DOI: `10.1103/PhysRevLett.62.2205` (cit. on p. 7).

[23] I. Marcikic et al. “Time-Bin Entangled Qubits for Quantum Communication Created by Femtosecond Pulses”. In: *Physical Review A* 66.6 (Dec. 2002), p. 062308. DOI: `10.1103/PhysRevA.66.062308` (cit. on p. 7).

[24] James L. Park. “The Concept of Transition in Quantum Mechanics”. In: *Foundations of Physics* 1.1 (Mar. 1970), pp. 23–33. ISSN: 1572-9516. DOI: `10.1007/BF00708652` (cit. on p. 8).

[25] W. K. Wootters and W. H. Zurek. “A Single Quantum Cannot Be Cloned”. In: *Nature* 299.5886 (Oct. 1982), pp. 802–803. ISSN: 1476-4687. DOI: `10.1038/299802a0` (cit. on p. 8).

[26] Dennis Dieks. “Communication by EPR Devices”. In: *Physics Letters A* 92.6 (Nov. 1982), pp. 271–272. ISSN: 0375-9601. DOI: `10.1016/0375-9601(82)90084-6` (cit. on p. 8).

[27] Norbert Lütkenhaus. “Estimates for Practical Quantum Cryptography”. In: *Physical Review A* 59.5 (May 1999), pp. 3301–3319. DOI: 10.1103/PhysRevA.59.3301 (cit. on pp. 9, 29).

[28] Tim Berners-Lee. *Information Management: A Proposal*. Mar. 1989. URL: https://www.w3.org/History/1989/proposal.html (visited on 06/16/2022) (cit. on p. 9).

[29] Vinton Cerf and R. Kahn. “A Protocol for Packet Network Intercommunication”. In: *IEEE Transactions on Communications* 22.5 (May 1974), pp. 637–648. ISSN: 1558-0857. DOI: 10.1109/TCOM.1974.1092259 (cit. on p. 9).

[30] Andrew S. Tanenbaum, Nick Feamster, and David Wetherall. *Computer Networks*. Sixth edition, global edition. Harlow, United Kingdom: Pearson, 2021. ISBN: 978-1-292-37401-7 (cit. on p. 9).

[31] Lucas Bialowons et al. *A Scalable Multi-User QKD Hub for Entanglement-Based Phase-Time Coding*. OPTICA Quantum Information and Measurement VI, Jan. 2021 (cit. on p. 12).

[32] Marc Schönefeld. *Pentesting J2EE*. Black Hat USA 2006, Jan. 2006. URL: https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf (visited on 07/21/2022) (cit. on p. 33).

[33] Chris Frohoff and Gabriel Lawrence. *Marshalling Pickles*. AppSec California 2015, Jan. 2015. URL: https://frohoff.github.io/appseccali-marshalling-pickles/ (visited on 07/21/2022) (cit. on p. 33).

[34] *Corning® SMF-28® Ultra Optical Fiber*. Nov. 2021. URL: https://www.corning.com/media/worldwide/coc/documents/Fiber/product-information-sheets/PI-1424-AEN.pdf (visited on 07/28/2022) (cit. on p. 34).

[35] Philipp Kleinpaß. “Description and Simulation of Entanglement-Based Phase-Time Quantum Key Distribution”. Master thesis. TU Darmstadt, June 2022 (cit. on p. 36).