HUMBOLDT-UNIVERSITÄT ZU BERLIN

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.) im Fach Physik

Implementierung Einer Steuerung Für Ein Quantum Key Distribution (QKD) Experiment Inklusive Postprocessing

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Institut für Physik
Humboldt-Universität zu Berlin

von
Robert Riemann
geboren am 06.06.1988 in Berlin

eingereicht am:
1. Februar 2013

Betreuung:
Prof. Dr. Oliver Benson
Dr. Alejandro Saenz

Master Thesis

Implementation Of A Control Unit For A Quantum Key Distribution Experiment Including Postprocessing

Humboldt University of Berlin
Faculty of Mathematics and Natural Sciences I
Department of Physics

Robert Riemann
Berlin, 1st of February 2013

Supervised by Prof. Dr. Oliver Benson,
Dr. Alejandro Saenz and
Matthias Leifgen

Contents

Chapter 1.

Introduction

The transition from an industrial to an information society, which began in the 1970s, has a deep influence on many aspects of our everyday life. It changed the way we learn, do our work and communicate with each other. Last but not least, there is a high impact on our economy as well, which is often referred to as knowledge economy. In times of international cooperation and global markets, the possibility of exchanging information safely and secretly is of particular importance.

The application, research and development of techniques to provide means of secure communication is called *cryptography*. The idea of hiding information from third parties, commonly called adversaries, dates back at least to the first century BC, when Julius Caesar used a substitution cipher to encrypt his private correspondence. During the last century cryptography evolved within information theory sciences into its own field of intense research.

1.1. Classical Cryptography

Cryptography belongs to the field of cryptology, which includes as well cryptanalysis, the art of code breaking. Cryptography occurs in two steps: during the *encryption*, a message is transformed using an algorithm called cryptosystem or cipher to form a *cryptogram*. The inverted process is called *decryption*. Often these algorithms depend on some additional input information, which is referred to as *key*. To implement unconditional secure cryptography, it is necessary to find a suited algorithm that renders the task to find the message without knowledge of the key impossible. As these algorithms are not convenient for practical use, most ciphers use algorithms providing only conditional security, i.e. it is very unlikely to extract the message from the cryptogram without any knowledge about the key, but it is not generally impossible [Gisin et al., 2001]. These algorithms are based on assumptions like the computational complexity to factor big integers. It is assumed that recovering the message would take much more time than the message is supposed to be useful. However it is unquestionable that further developments could allow to break cryptograms in much less time.

Cryptosystems can be divided into two subgroups, which are described in the following subsections.

1.1.1. Asymmetrical Cryptosystems

Asymmetrical Cryptosystems use keys consisting of two parts. A message is encrypted using a publicly known part of the key and can be decrypted with a different, private part of the key [Diffie and Hellman, 1976]. The first implementation was published by Rivest, Shamir, and Adleman [1978] and is commonly known as RSA. Generally, the receiver of a message, who is called Bob in the cryptography community, generates a private key and subsequently computes a public key depending on the private key. After openly communicating this public part to the sending party, called Alice, messages can be encrypted and decrypted without exchanging private keys beforehand. This is a fundamental advantage, which makes these so-called *public-key* cryptosystems very attractive to use.

Asymmetrical cryptosystems base on one-way functions $f(x)$, which can be computed from x in polynomial order with increasing size of x, whereas it takes exponential time to compute x for a given $f(x)$. To give an illustrative example, let us consider prime factorisation.

$$f(x) = 4591 \cdot x \tag{1.1}$$

It is reasonable that $f(2879) = 13\,217\,489$ is much easier to calculate than finding both prime factors given the result. Of course, it becomes much easier if one factor is already given.

Nowadays public-key cryptograms like RSA are often used in internet protocols (e.g. in ssh/https) to realise authentication, encryption or to exchange securely temporary keys for a consecutive symmetrical encryption.

1.1.2. Symmetrical Cryptosystems

Symmetrical cryptosystems provide comparable security to asymmetrical systems with much less computing power and smaller key sizes, thus providing a better overall performance.

Both parties use the same key, which needs to be exchanged in advance, either by a personal meeting or by a public-key encryption scheme.

The most simple symmetrical cryptosystem uses the XOR operation ($\oplus$) to apply the bits of a common key k to a binary message m_1 as shown in (1.2). It is obvious that this operation is symmetrical, as the second application with the same key (decryption)

reverses the transformation and delivers m_2, which is identical to the original message m_1 (1.3).

$$\begin{aligned} s &= k \oplus m_1 && (1.2)\\ m_2 &= k \oplus s && (1.3)\\ &= k \oplus (k \oplus m_1) = (k \oplus k) \oplus m_1 = m_1 \end{aligned}$$

If the key is as long as the message itself, this encryption scheme is in fact the *one-time pad* introduced by Vernam [1926]. Interestingly, this is the only known cryptosystem today which is proven to be secure. As long as the key is not reused, it is not possible to obtain any information about the message, neither by statistical analysis nor by infinite computing power.

Regardless of its good properties, it is only rarely used today. For the most applications it is not convenient to deploy a cryptographic key as long as the message. A constantly growing message would require ongoing meetings between both parties or continuous use of a trustworthy courier service. These conditions render the one-time pad inappropriate to replace classical cryptosystems with fixed key length.

1.2. Unconditional Security

Both classical cryptosystems have crucial drawbacks. The robustness of public-key systems depends on one-time functions. Today there is not any one-time function known which is provably computational expansive to invert. Currently there is no practical algorithm known to realise integer factorisation easily, but this does not mean, that further developments in information theory could not bring up more efficient solutions. The progress in the field of quantum information theory by Shor [1996] puts additional doubts on this. It could be shown that prime factorisation can be computed efficiently using a quantum computer. Of course, this discovery does not have any practical impact as long as today's quantum computer experiments can only handle prime number factorisation of 15, as it was demonstrated by Vandersypen et al. [2001], Lu et al. [2007] and recently by Lucero et al. [2012]. However, with on-going developments in the field, it is foreseeable that quantum computers are going to improve.

The security of symmetrical cryptosystems depends on the distribution of the key. As long as the key is negotiated using public-key systems, which have not yet been proven to be secure, symmetrical cryptosystems have to be considered to be open to attacks as well.

This motivates the ongoing efforts during the last decades to develop new methods to implement a cryptosystem that overcomes these issues. In fact, it is not expected in the foreseeable future to find either a new cryptosystem, which is more convenient than the

one-time pad, but equally secure, or to finally find a one-time function, which is provably expensive to invert.

That is why the problem of solving this algorithmic problem was shifted to the task of finding solutions to make the one-time pad more convenient. Basically, the cryptography community wants to resolve the current limitation of this cryptosystem by providing new ways of a secret key exchange, which does not require ongoing meetings. Interestingly, the fundamental laws of physics allow to implement a solution which also stands the requirements of unconditional security.

1.3. Quantum Key Distribution

The theory of quantum mechanics provide some properties, which makes it quite promising for applications in cryptography.

1. Every measurement perturbs the system.[1]

2. Quantum objects in unknown states cannot be cloned
   (*no-cloning theorem* by Wootters and Zurek [1982]).

Wiesner is acknowledged to be a pioneer in the investigation of new applications of quantum mechanics related to coding. He published ideas of *quantum banknotes* in 1983, but the initial idea dates even back to 1969, when it was rejected from the journal for being too futuristic [Galindo and Martin-Delgado, 2001].

Quantum banknotes were supposed to provide a possibility to check banknotes for counterfeiting. Such banknotes would be manufactured with a few indefinitely trapped photons next to a printed serial number. These photons would have a dedicated polarisation only known to the institute/factory responsible for distributing the notes. To verify the authenticity of a banknote, the institute would measure the polarisation of each individual photon. This can be achieved without perturbing the states as long as the supposed states are known in advance. A forger is not able to clone these quantum banknotes because it would require copying the quantum states of photons, which is provably not possible due to the no-cloning theorem.

Though the idea of quantum banknotes has not yet been realised due to the lack of techniques to preserve single quantum states efficiently, it motivated Bennett and Brassard to think of a scheme to realise quantum key distribution using polarised photons and that's why it has to be considered as an important milestone towards quantum cryptography [Brassard, 2006].

[1] In the exception of a system, which is compatible to the eigenstates of the measurement, the system is not perturbed.

1.3.1. BB84 Protocol

In 1984 Bennett and Brassard presented their suggestion on how to realise a quantum cryptosystem, which is nowadays referred to as BB84 protocol [Bennett and Brassard, 1984]. It describes in detail how to realise the key distribution between two parties, Alice and Bob, using a uni-directional quantum channel and a bi-directional classical communication channel.

A potential eavesdropper, commonly known as Eve, might use unlimited computing power and is only bounded by the laws of quantum mechanics when manipulating the transmissions on the quantum channel. The classical channel is supposed to be public and authenticated, hence it does not allow for any manipulations (think of a newspaper). The quantum channel is used to transmit information commonly called *qubit*, which is the standard unit of information in quantum information theory, analogous to the classical bit (sometimes called cbit) of classical information theory. A qubit may be physically represented by any two-state quantum-mechanical system. It can be represented by a spin system or a trapped ion with a superposition of its ground state $|g\rangle$ and its excitation state $|e\rangle$. Of course, these trapped ions can not be transferred easily between Alice and Bob, which is a key requirement in quantum key distribution. Instead the polarisation state of a single photon as proposed by the BB84 protocol is the natural choice to exchange quantum information.

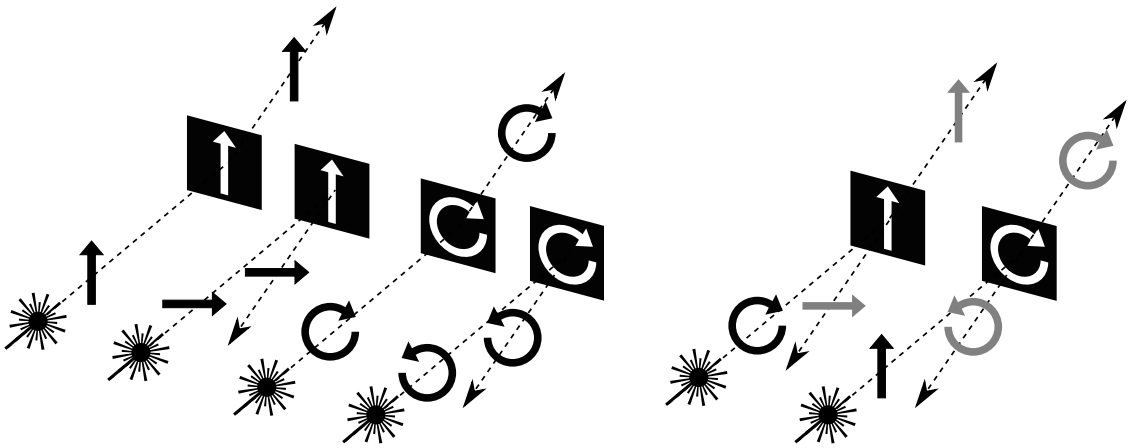
The polarisation of photons can be described within a base formed out of both states $\updownarrow$ and $\leftrightarrow$, which are usually assigned to the classical bits 0 and 1, respectively. Furthermore, a second base needs to be defined in a way that any pair of elements are maximal conjugate with respect to the first bases. The circular polarisation base $\circlearrowright$ (bit 0) and $\circlearrowleft$ (bit 1) fulfils this requirement.

$$|\langle\circlearrowright|\updownarrow\rangle|^2 = |\langle\circlearrowright|\updownarrow\rangle|^2 = \ldots = \frac{1}{2} \tag{1.4}$$

The basic protocol uses four states, but later on there were many proposals on possible extensions published, including variations with more states [Kern and Renes, 2008].

Before a single photon can be transmitted, both parties have to choose one base at random. Additionally, Alice has to select randomly one out of both states of her base. For an unconditionally secure implementation, it is crucial to use true random generators, which rely ideally on the inherent random nature of quantum mechanical effects. Alice prepares her photon to be in the state she selected in advance. At the same time, Bob is calibrating his devices to measure the photon in either the vertical-horizontal (VH) or the circular base with respect to his base choice. To do so, he is equipped with two detectors to be able to detect both states in his base with certainty.

Depending on the compatibility of the selected bases, there are two possible interpretations of the measurement result of Bob. In case of two conjugate bases, Bob will measure the correct photon with respect to Alice's state choice accordingly to (1.4) only with a probability of $p = \frac{1}{2}$ (fig. 1.1b). So this measurement does not give any information about



(a) matching bases (b) conjugate bases

Figure 1.1.: Schematic use of bases in BB84 protocol. The black plates symbolise arrangements of polarising filters and birefringent crystals. Photons with prepared polarisation are transmitted or reflected depending on the polarisation property of the polarising beam splitter. In (b), the outcome is probabilistic.

the state Alice has sent. When both states are compatible, as depicted in fig. 1.1a, the measurement of Bob allows him to deduce to the state Alice has sent.

Of course, both parties (and possible adversaries) do not have any knowledge about the base selection of the other person. So at the first glance, Bob cannot decide which measurements contains actual information and which are measured at random. Using the public, authenticated and bi-directional communicating channel, Bob sends Alice his list of base choices. Afterwards Alice responds Bob by announcing the measurements, which have been done in compatible bases without telling the actual base. This allows both to omit these sent/measured qubits which were processed in conjugate bases. This action is called *sifting*, after which the *raw key* is remaining.

Since Alice's states and Bob's base selection are each controlled by a random number generator (RNG), the key is going to be a random bit sequence as well, which is crucial to the security of the protocol. Therefore it is not possible to use this technique to transfer a message, but it is convenient to use this generated key in conjunction with the one-time pad, which relies fundamentally on such true random keys.

Ignoring the shortcomings of technical implementations, the BB84 protocol provides this way a provably secure solution to distribute keys [Shor and Preskill, 2000; Gottesman et al., 2004].

It has to be mentioned that since the publication of the BB84 protocol other protocols have been suggested as well. For instance, there is a quantum key distribution (QKD) scheme presented by Ekert [1991], which is based on quantum entanglement between photon pairs. This protocol, which is commonly called Ekert91, uses the Bell inequalities to detect

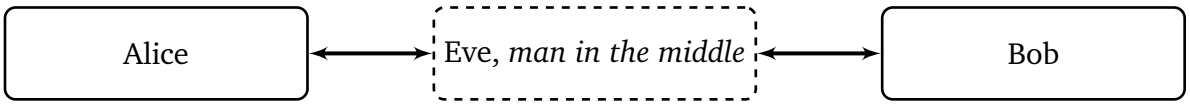


Figure 1.2.: man in the middle

potential forgery and was already experimentally realised [Marcikic et al., 2006; Poppe et al., 2004].

1.3.2. Attacks

To get a more detailed understanding of the BB84 protocol, it is helpful to consider the possibilities of the adversary Eve. The trivial idea for Eve would be an attack commonly called *intercept-resend*. The classically analogy would be the *man-in-the-middle* attack. In both attacks Eve pretends to be Bob when communicating with Alice, whereas she pretends to be Alice when talking to Bob (fig. 1.2).

In classical communication, the man-in-the-middle might decide to save a copy of the data stream along the transmission, which is impossible with quantum channels, to try different methods of code breaking afterwards. As the eavesdropper stays undetected and has unlimited time, there is a non-zero chance that the encrypted message might ultimately be revealed.

Considering single photon transmission, Eve has no choice other than to measure these single photons using an apparatus like Bob's. Afterwards, Eve sends a single photon in this measured state to Bob.

Alice and Bob proceed later on by sifting their data. Eve has full knowledge about this communication because the classical channel is assumed to be public. Other than that, Eve cannot secretly manipulate the authenticated sifting process. Of course, it is not possible for Eve to measure photons in the same base since Bob is declaring his random base selection only after the measurement. Statistically, Eve has a chance of 50 % to have the photon measured in the same base as Bob. In this case, Alice and Bob can not recognise a forgery. It is different when Eve measured states conjugate to both states of Alice and Bob, who would expect a deterministic result. Then Eve still has a chance of 50 % to stay undetected, because the photon is transformed to the conjugate base again during the measurement of Bob, which might result in the expected state of Bob. In 25 % of all cases, Eve is sending a verifiably wrong photon. An overview is presented in fig. 1.3.

Before proceeding with encoding their message, Alice and Bob will do some error estimations by comparing a part of the secret key, which will reveal an unexpected high error ratio. The key is going to be rejected and Alice and Bob are starting again from the beginning.

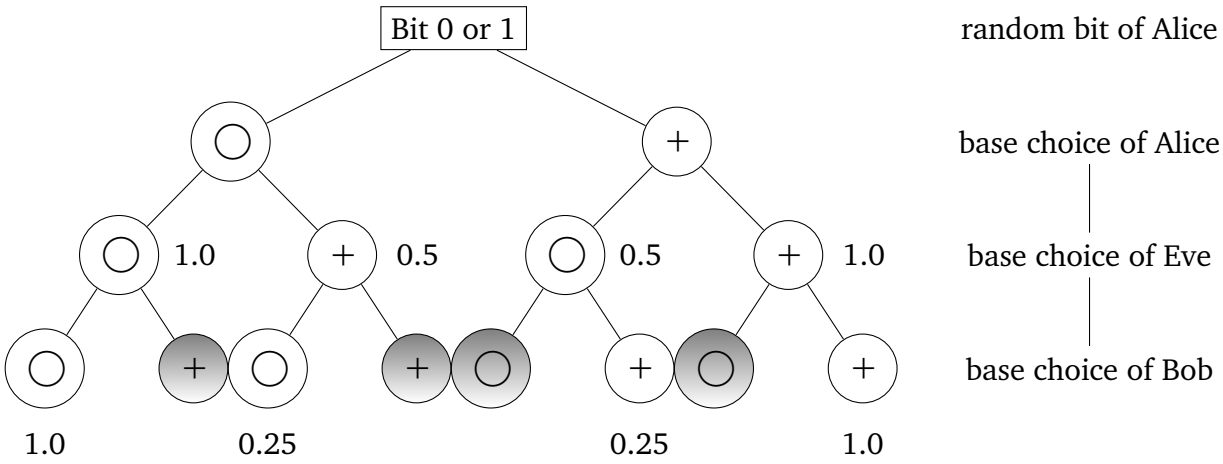


Figure 1.3.: Probability graph for a intercept-resend attack. The graph represents the absolute probabilities of Bob and Eve to measure the original bit despite of Eve's manipulations. Shaded configurations are subject of sifting and are not taken into account. The depicted states are meant to represent their base. ◯ represents the circular base and + stands for the vertical-horizontal base.

The key is only used to encode messages after both parties are convinced to have not been subject to eavesdropping. Eve might prevent Alice and Bob of establishing a secure key by interrupting the quantum channel, but theoretically it is not possible to run an undetected intercept-resend attack.

Beside the intercept-resend attack, there exist several other eavesdropping strategies, e.g. *photon number splitting* (PNS) attack [Bennett et al., 1992], which explicitly exploit the implementational shortcomings of using weak coherent pulses (WCPs) instead of single photons.

Vakhitov et al. [2001] presented a new kind of attack called *Trojan horse attack* or *light injection attack*, which enables Eve to get the exact secret bit sequence. Without going into detail, the core idea is to learn the base selection of Alice and Bob before the transmission of the single photon by emitting light to both apparatus and analysing the reflected photons. These kind of vulnerabilities heavily rely on the actual implementation and thus the fixes are very specific as well. A very simple idea to overcome this issue would be to implement a Faraday rotator at the light exit of the apparatus of Alice, which prevents photons of Eve from injection.

Chapter 2.

Experimental Setup

For demonstration and testing purposes, the working group has a small QKD experiment at its disposal. This experiment uses a single photon source [Schröder et al., 2011] which operates at room temperature.

2.1. Apparatus

The whole single photon source setup (see fig. 2.1) fits on a sheet of paper. It is based on nitrogen-vacancy (NV) colour centres [Kurtsiefer et al., 2000], but can be equipped alternatively with silicon-vacancy (SiV) colour centres provided by Neu et al. [2011]. Both colour centres occur in nanodiamonds of a typical size of $\mathcal{O}(10\,\mathrm{nm})$ in diameter. The nanodiamonds, in the early stage of the experiment nitrogen-vacancy colour centres, later silicon-vacancy colour centres, are embedded on a substrate, which can be adjusted using a 3-axes piezo platform. A confocal setup [Webb, 1996] is used to assure that only the fluorescence light of one diamond at a time is projected to the free space beam plane.

An excitation laser beam is guided to the front of this nanodiamonds substrate, where single fluorescence photons are emitted. The reflected laser beam is later on filtered by a dichroic beam splitter, which reflects the excitation laser beam due to its smaller wavelength.

The single photons are gathered by an objective lens with a high numerical aperture (NA) of 0.9, transmitted through the dichroic beam splitter, accumulated and guided as a free space beam into the experimental setup. Optionally, the beam can be fed into a fibre in-coupler.

In fig. 2.2 a scheme of the demonstration experiment is shown. At the cost of source light intensity, the photons are brought to an initial vertical polarisation state by a polarisation filter. In fact, the loss of photons can be reduced to $\mathcal{O}(10\,\%)$ as the single photon source already emits highly polarised photons. Hence, a $\lambda/2$-waveplate is placed before the polarisation filter to turn the bias polarisation of the single photon source to vertical polarisation.

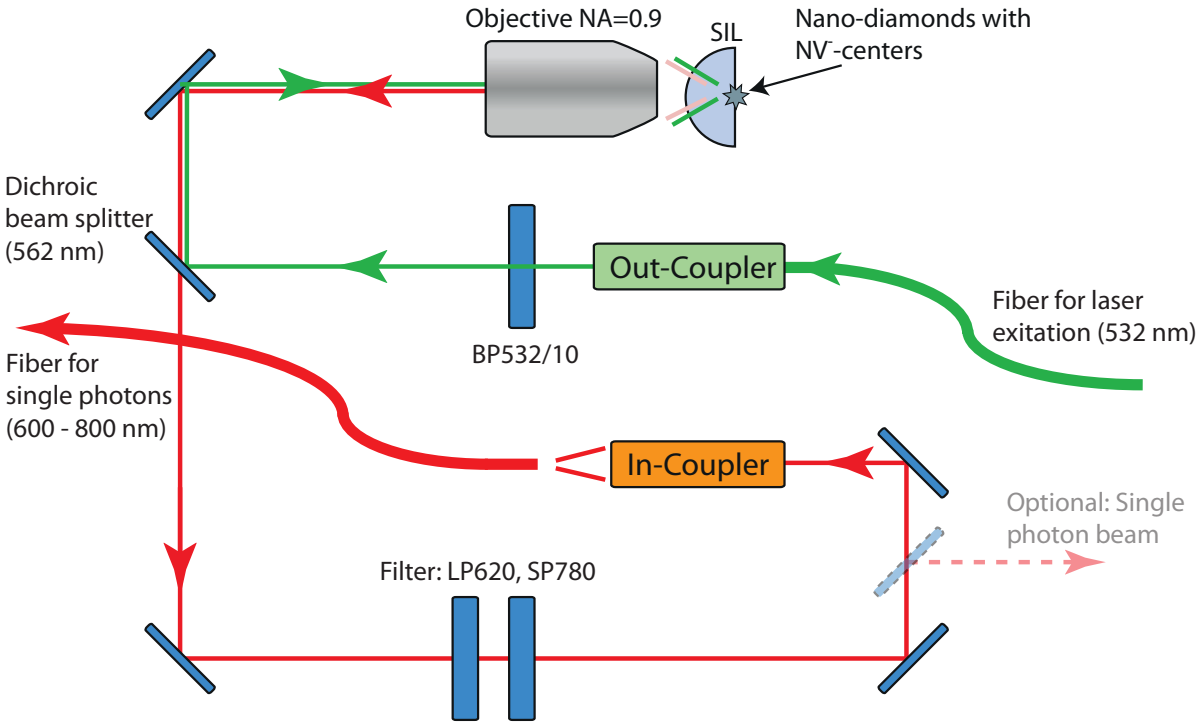


Figure 2.1.: Single photon source scheme. The scheme shows the setup of the used single photon source. The excitation laser (green) pumps the vacancy centres, which emit single photons (red). These are collected using a confocal setup [Webb, 1996] and in our experiment transmitted using a free space beam.

Source: Courtesy of Friedemann Gädeke.

To improve the efficiency and simplify the calibration of the beam, a set of lenses and a pinhole (PH) is put right after the single photon source.

Afterwards, the polarisation is modified using the first electro-optical modulator (EOM) controlled by Alice. The EOM uses the Pockels effect to transform the polarisation state depending on an input voltage. The built-in crystal has an effect comparable to a retarding waveplate, for which birefringent materials are used to modify the polarisation through two distinct light propagating speeds. The propagation speed depends on the orientation of the polarisation in the material, so that the effect can be influenced by changing the position and rotation, but also the extension. Of course, it is experimentally inconvenient to precisely change these parameters at high frequencies. Thus, one relies on crystals, whose birefringent property depend on an external electromagnetic field controlled by an input voltage.

Using a dedicated voltage, the EOM converts the polarisation like a $\lambda/4$-waveplate or a $\lambda/2$-waveplate. The necessary voltage ranges between 0 V and ± 250 V. Therefore a bridge circuit is used. The output voltage of the DA-converter is only in the range of ± 5 V, which makes it essential to use additionally a high voltage converter.

The input voltage is switched between different voltages corresponding to the polarisation states. The determination of these input voltages is part of the calibration process before the transmission can be started. The frequency of changing this voltage is synchronised with respect to the excitation frequency of the single photon source. After all, photons of state $|\updownarrow\rangle$ can be converted to states $|\leftrightarrow\rangle$ corresponding to the effect of a $\lambda/2$-waveplate, $|\circlearrowleft\rangle$ or $|\circlearrowright\rangle$ respectively corresponding to a $\lambda/4$-waveplate. The EOM is furthermore capable to not modify the polarisation states at all. In the end, the transformation to 4 different states on purpose can be realised. This is a key requirement to implement the BB84 protocol.

Between Alice and Bob there is a short free space distance of $\mathcal{O}(0.5\,\mathrm{m})$. Subsequently the EOM of Bob follows where the polarisation is transformed to be measured either in the vertical-horizontal (VH), which means actually to not transform the polarisation, or the circular base, which requires the EOM to act as a $\lambda/4$-waveplate. The latter one transforms circular polarised photons to vertical-horizontal polarisation, because the polarizing beam splitter (PBS) can only distinguish photons in this base.

After hitting a PBS, the photon is measured by one of two avalanche photodiodes (APDs). The insufficient efficiency of the PBS due to its manufacturing limits requires an additional polarisation filter to compensate the higher rate of mistakenly reflected photons and thus reducing the overall error.

Even if the usage of two APDs is not a strong requirement by the BB84 protocol, it notably improves the efficiency. The probability to emit a single photon after a laser excitation pulse ranges around $\mathcal{O}(0.5\,\%)$. Let us assume Bob is listening for photons in state $|\updownarrow\rangle$, but does not measure any photon and Alice reports afterwards to have used the VH base. In an ideal experiment with a source and detector efficiency of 100 %, Bob would conclude logically from not detecting a photon, that Alice has sent a photon in state $|\leftrightarrow\rangle$. In a real-world

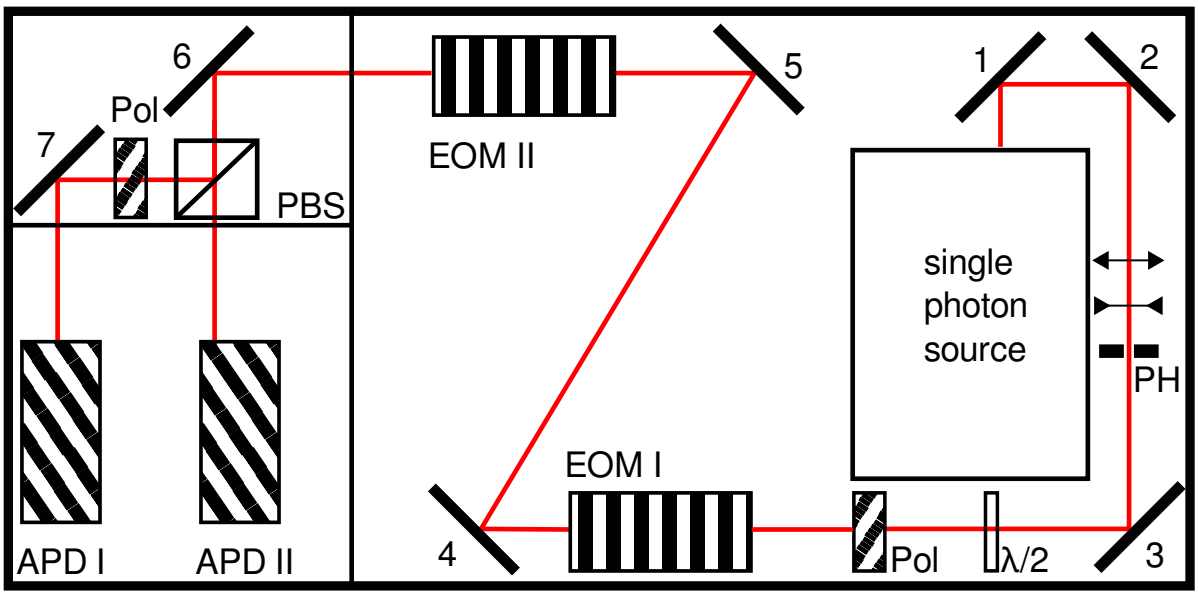


Figure 2.2.: Experimental setup scheme. Both parties, Alice and Bob, are placed on the same plate. The single photon source and the first EOM I belong to the sending unit of Alice. The second EOM II and both APDs setup the receiving unit of Bob. A photo of the experiment is given in fig. 3.12.

experiment, Bob does not know whether the state indeed has been $|\leftrightarrow\rangle$, or whether Alice has sent a photon at all. In fact, due to the low efficiency of the single photon source the latter case is significantly more likely. Using two APDs, Bob has the chance to measure one photon independently of its polarisation. Statistically this improves the efficiency by a factor of two, because Bob do not have to decide which state he would like to measure, but can measure both possible states without any other drawbacks.

2.2. Control and Data Acquisition

Quantum information experiments like quantum key distribution, but also quantum computing, are always evaluated by the benefits they provide in comparison with classical solutions. Properties like reliability, security, expenses and speed play an important role. While the theory of quantum information techniques, which includes the examination of security aspects, has matured, the practical implementation is lacking behind due to experimental shortcomings of today's technologies for storing, transmitting and manipulating of more than a few quantum states.

In contrast to quantum computing, the quantum key distribution is the first quantum technology to be deployed outside of physics laboratories since transmission and manipulation of single photons can be sufficiently realised. Commercial systems are already available.

However, there are some crucial factors to take care of when considering the implementation of a quantum key distribution experiment. Even with a provably secure theory, such an experiment is only as secure as the apparatus itself. A potential adversary would focus on exploiting the hardware. This was already demonstrated by Gerhardt et al. [2011]. To deal with such attacks would be outside the scope of this demonstration setup.

Instead, an effort was made to design a fast and reliable setup. Transmission rates in classical communication are counted in Gbit/s. Even if the current setup is limited to a secure key rate in the order of $\mathcal{O}(1\,\text{kbit/s})$, the presented software is capable of handling much higher transmission rates.

The interface between most components and the central controlling unit (a desktop computer in our case) is a field-programmable gate array (FPGA) controller. In addition to providing TTL-standardised I/O ports, the FPGA consists out of an integrated circuit, which can be configured after production. Vendor-specific tools allow to generate low level code[1] out of a subset of LabVIEW code, which can be used to program the information flow on the FPGA. It is very attractive to use such components, because they combine the advantages of integrates circuits (fast and massive parallel processing) with the benefits of computers (easy programming, fast developing and testing with multiple iterations at nearly no additional costs).

In our demo experiment, a NI PCI-7813R FPGA model was used. It is loaded onto a regular PC PCI card and provides means to transfer data directly between its own memory and the PC (the host system) using a FIFO system. It is important to realise that this chip has its own memory and clock and runs completely independently of the host system. The maximum clock frequency is a property of the hardware, but also of the code. In our case, the FPGA runs with a fixed real-time frequency of 40 MHz. An internal software counter f_{ratio} allows to stretch all actions like EOM setup, triggering etc. to certain time intervals. By increasing f_{ratio} the de-facto clock frequency $f_{[\text{Hz}]}$ can be lowered in real-time.

$$f_{[\text{Hz}]} = 40\,\text{MHz}/f_{\text{ratio}} \tag{2.1}$$

Even if the FPGA module is capable of executing many actions simultaneously during one single 40 MHz-cycle, it is an experimental requirement to do these actions in a fixed order, one after each other. To match the photon delay time, it is even necessary to wait a couple of cycles in between, meaning that there are cycles with no actions at all. The control graphical user interface (GUI) is depicted in fig. 2.3. A scheme of the underlying logic can be found in fig. 2.4, whereas a device data flow orientated scheme is presented in fig. 2.5.

In an initial approach, both Alice and Bob are controlled by the same PC/FPGA. This way, the synchronisation problems between the clock of Alice and Bob can be postponed in

[1] The FPGA is programmed using the specific-purpose language VHDL (Very-high-speed integrated circuit Hardware Description Language).

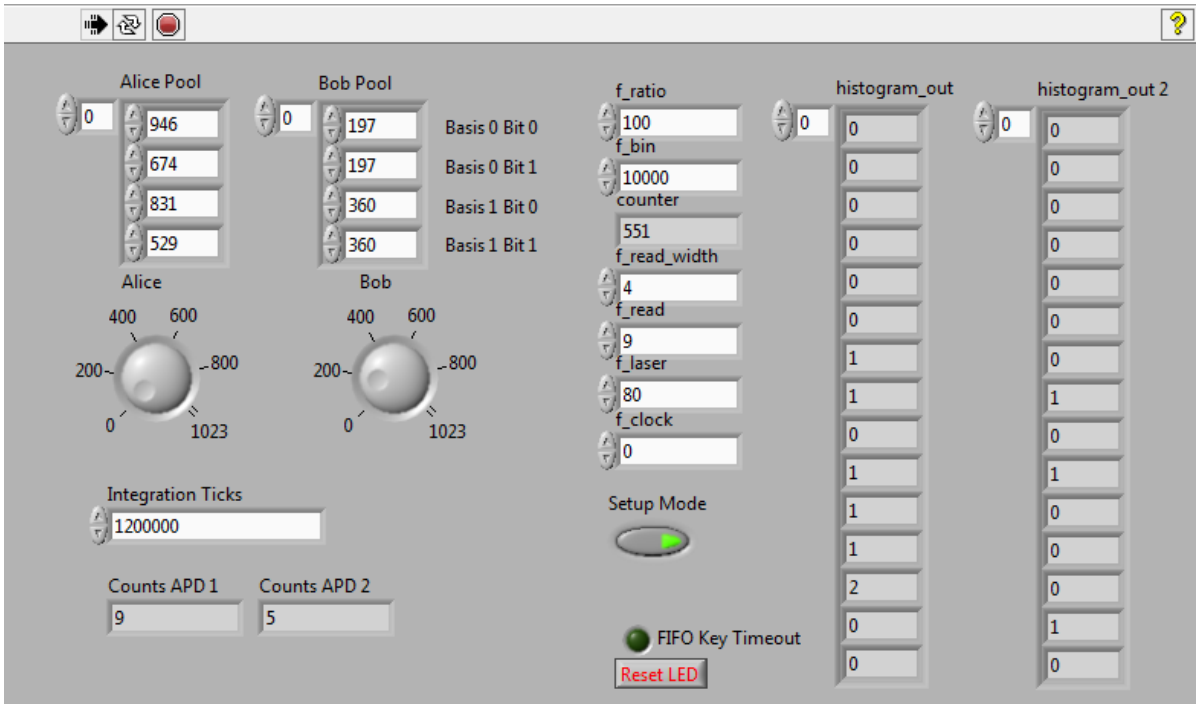


Figure 2.3.: Graphical user interface of LabVIEW FPGA module

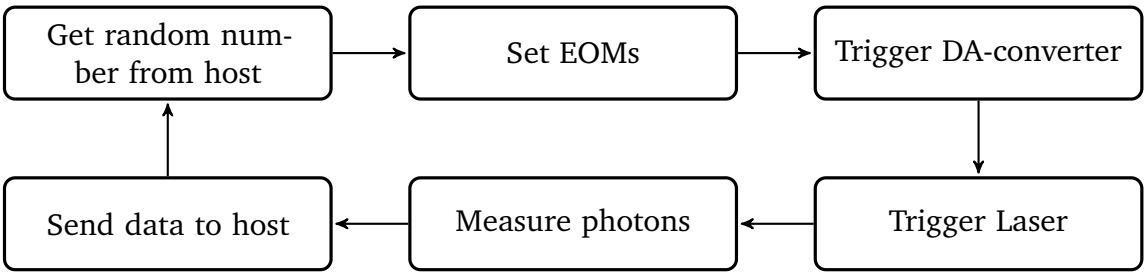


Figure 2.4.: Flow chart of internal FPGA process. This figure visualises the internal process of the FPGA module to transmit one qubit. The host denotes a system capable of delivering random numbers (i.e. from storage) and doing I/O-operations.

favour of focusing on the single photon transmission and data evaluation. The consequent separation between both control units is subject to future improvements. Until that point, the experiment is clocked using a single FPGA module.

To realise the required random bit/base choice of Alice and Bob, random numbers need to be accessible to the FPGA module. While it is often sufficient in traditional classic cryptography to use deterministic random number algorithms, it is important to use a true (non-deterministic) quantum random number generator (QRNG) to comply with the requirements of an unconditional secure implementation.

In cooperation with our group a publicly available online service was set up dedicated to provide true random numbers on demand, which are generated constantly by a QRNG [Wahl et al., 2011]. The FPGA module is set up to receive a stream of these numbers. One 4 bit-long number (0 - 15) is used to decide which base and bit should be sent and measured by Alice and Bob, respectively. The random bit choice of Bob is only necessary in case of a measurement setup of Bob with only one APD. As our measurement setup allows to register both bits, the bit choice of Bob is irrelevant in our case.

Depending on the selected states of Alice and Bob, the EOM voltage values are selected, which has to be entered in the settings (see left top corner in fig. 2.3) in advance before starting the experiment. The FPGA is submitting these 10 bit-values using its I/O-ports to the DA-converter during the cycle specified in f_{clock}.

In the context of the FPGA GUI, f values denote not frequencies, but each specific cycles from f_{ratio} total cycles, which is linked to the de-facto frequency by (2.1).

It takes a second FPGA cycle to activate the new settings finally, because the DA-converter is waiting for a raising edge on its own clock input, which has to be altered consecutively. So after all, the EOM setting is consuming 2 FPGA cycles.

Due to a comparatively small slew rate of the high voltage converters connected to the DA-converter, it takes several cycles to apply the new settings to both EOM devices. The quantity f_{laser} determines the cycle in which the excitation laser of the single photon source is triggered. The difference $f_{\text{laser}} - f_{\text{clock}}$ should therefore be maximised.

After that, a photon is expected in either APD 1 or APD 2. To reduce the influence of background photons, only photons measured in a small time frame of $f_{\text{read\_width}}$ cycle periods are recognised as signal photons. The quantity f_{read} denotes the number of cycles between triggering the laser and starting to expect the single photons to arrive.

For each APD, one bit is used to store if a photon was measured in the specified time frame or not. Additional two bits are required to store which state of four options was sent by Alice. Even if Bob actually only chooses one base, two bits are used as well to bring the software processing of both parties into line (1 bit unused). These bits are combined to an 8 bit integer value (cf. fig. 3.8) and transmitted in blocks of 10000 to the host PC. After f_{ratio} cycles, the loop in fig. 2.4 is starting again.

These 8 bit-integer values are internally sent to the host PC, received by another LabVIEW program and saved to binary files.

For calibration purposes, the FPGA module is providing some additional counters and a histogram functionality for each APD.

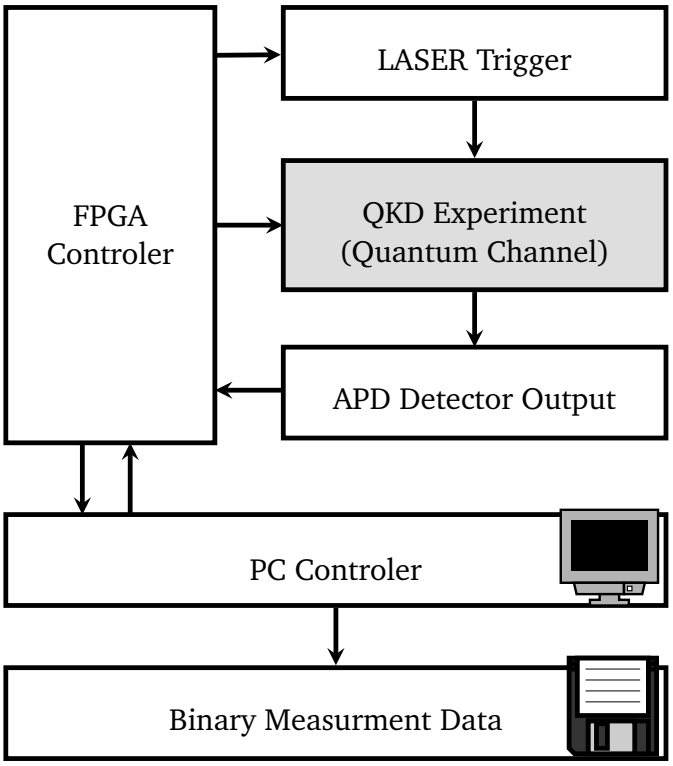


Figure 2.5.: Flow chart of device interaction with respect to data flow. The interaction between the major components is presented. The FPGA controller sets the states for both EOMs, which influences the quantum channel. Afterwards the FPGA controller triggers the laser and is looking for a photon to be measured by the APD detectors. The PC is used to switch the FPGA controller on and off, transmit random numbers and receive measurement data, which are saved to a binary file.

Chapter 3.

Postprocessing

After a general introduction to cryptography in chapter 1 and a brief a description of the experimental setup in chapter 2, this chapter presents the analysis of the measurement data, which is the central focus of this work. The first part from sec. 3.1 to sec. 3.5 is dedicated to the single steps of this analysis and the theory behind them. Finally, in the last sec. 3.6 details about our implementation are discussed and the achieved results are presented.

When Alice and Bob have finished their qubit transmission, both parties have to sort out qubits with non-matching bases (sifting) to get the raw key. Therefore they use a classical public communication channel.

Considering an ideal experiment, Alice and Bob could use this key directly to encode the message using the one-time pad, whereas a realistic experiment with imperfect detectors and a noisy quantum channel requires some additional steps, which are commonly called *postprocessing*. The postprocessing can be done offline, which means that the quantum channel is not needed at this time anymore and the further processing can be done independently anywhere at any later time as long as a classical communication channel is provided. There are no hard timing/syncing requirements anymore.

Due to measurement background (dark counts of APDs, stray light, etc.), the raw key of Alice and Bob differ from each other. A specific *error correction* algorithm has to be applied. Most of these possible routines require an initial guess of the estimated error rate in order to be efficient. Therefore it is necessary to run another algorithm for *error estimation* beforehand.

Finally, in a realistic environment, the public communication on a classical communication channel is open for different kinds of attacks, e.g. man-in-the-middle attacks. To prevent such offences effectively, it is common to add means of *authentication* to each transmitted message. An overview of all steps is presented in fig. 3.1.

Generally, the role of Alice and Bob can be exchanged in between during the postprocessing. Nevertheless, in a real experiment there might be some performance advantages for doing it the one or the other way.

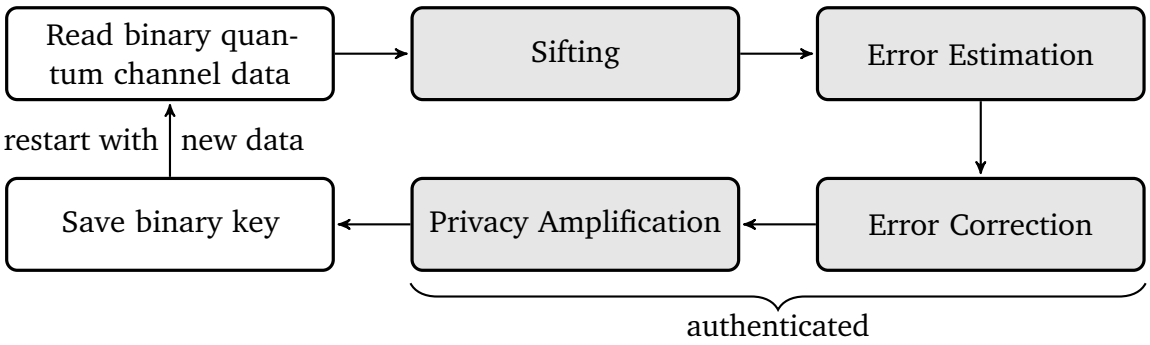


Figure 3.1.: Flow chart to visualise the components of postprocessing. The chart presents the necessary steps of the postprocessing and their order. Shaded steps are authenticated. The postprocessing has to be applied on chunks of measurement data. The depicted loop is repeatedly executed for each chunk.

3.1. Sifting

By definition, qubits that have been measured in maximal conjugate bases are projected randomly in one out of two measurement states (see fig. 1.1b). The information content of such a qubit equals zero. Thereby it is necessary to find and delete these random measurements in our sorted qubit string.

Up to now, Alice and Bob only know their own base selections. So Bob has to reveal his choices publicly. Afterwards Alice is able to distinguish between random and deterministic measurements and announces the positions of the former or latter ones publicly. The probability distribution for random and deterministic bits is expected to be uniform in most protocols [cf. Erven et al., 2009]. Hence, the choice of who announces the positions is arbitrary. In contrast, it is convenient to ask Bob to initially report its base choices because he only has to report position and measurement base for qubits that he actually measured. Consequently, Alice henceforth ignores all qubits that Bob has not mentioned. Considering the low efficiency of the quantum channel with its detectors, the performance advantage in matters of total data transmission by this role allocation can be significant.

Both Alice and Bob have to pay attention to verify that they are only accepting answers from each other and not from a third party Eve. The eavesdropper, who has already run an intercept-resend attack, could simply publish his measurement bases before Bob, hence pretending to be Bob. Eve would gain an exact copy of the key if Alice has no means to recognise Bob. Any quantum transmission between Eve and Bob would be unimportant. Therefore both parties are appending a short token, comparable to a signature, to their messages. A detailed explanation is following in sec. 3.5.

3.2. Error Estimation

The error correction algorithm, which is presented in the sec. 3.3, neither has a fixed efficiency nor really guarantees to deliver an error-free key afterwards. It is rather based on minimising the error probability while maximising the efficiency. A tuning parameter is derived by the estimated error probability, which has to be determined in advance.

The implementation, which forms the basis of this work, uses plain comparisons per bit [Gisin et al., 2001, pg. 7]. There is a customizable parameter, which is currently configured to expose 2 % of the key data publicly to allow the calculation of the average bit error probability. Of course, these comparison bits cannot be used as part of the final key anymore and must be discarded. The final number of bits for this purpose needs to be tuned to provide on the one hand a sufficient accuracy of the estimation, and on the other hand a largest possible secure key. As the performance of the following error correction algorithm can only be determined by sample tests, the appropriate bit ratio used for the error estimation and its uncertainty needs to be calibrated by benchmarks as well.

A possible eavesdropper could try to run a man-in-the-middle attack and thereby gain some influence on the parameter configuration of the error correction algorithm. To prevent these kind of attacks, either Alice or Bob must select the bits for comparison randomly and must announce the positions and their bit values authenticated to the other one.

While the sifting is done for single bits, the error estimation and the following postprocessing steps require operating on multiple bits (chunks).

3.3. Error Correction

The error finding and correction stage, also referred to as *reconciliation*, is an important part of every QKD experiment underlying realistic conditions. The fundamental idea is to provide ways and means to decrease the final bit error rate (BER) arbitrarily close to zero. While typical classical optical communication channels suffer from a BER of the magnitude of 10^{-9}, a typical quantum channel communication has to deal with a quantum bit error rate (QBER) in the order of a few percent [Gisin et al., 2001]. These bit errors might be caused by physical imperfections like optical misalignment or dark counts of the APD, as well as by Eve's eavesdropping attempts.

In the paper of Gisin et al. [2001, pg. 7] a very trivial correction scheme is presented to give an initial idea of such a reconciliation algorithm. Alice announces the XOR operation result of two randomly chosen bits and asks Bob for confirmation. If Bob accepts, only the first bit is going to be part of the final key. Of course, the XOR-bit will be the same in case

of two erroneous bits. Assuming a per bit error probability p with $p \leq 1$, the per bit error rate of the outcome is lowered to

$$p_{\texttt{XOR}} = p^2. \tag{3.1}$$

In return, the key size is dramatically shrunk to 50 %.

3.3.1. Preliminaries

A theoretical discussion on better performing reconciliation algorithms was published by Brassard and Salvail [1994]. The most important ideas and concepts are summarised below.

The information or uncertainty of a finite set X with an assigned probability distribution $\{p(x) \mid x \in X\}$ is measured by the *Shannon entropy* $H(X)$, which is phenomenologically introduced in the appendix A, and quantifies the minimum amount of necessary storage bits to fully represent the given information. The logarithms are to the base 2, if not otherwise expressed.

$$H(X) = -\sum_{x \in X} p(x) \log p(x) \tag{3.2}$$

In case of a binary choice (Bernoulli trial) with a probability distribution $\{p, 1-p\}$, the Shannon entropy is referred to as $h(p)$. The conditional entropy $H(X|Y)$ of X given Y with two finite sets X, Y and a joint probability distribution $\{p(x,y) \mid x \in X, y \in Y\}$ is defined by

$$H(X|Y) = -\sum_{x \in X} \sum_{y \in Y} p(y) p(x|y) \log p(x|y). \tag{3.3}$$

Now we want to classify the loss of information due to the noisy transmission channel. The binary symmetric channel (BSC) is defined to transmit bits with an individual error probability p per bit independently of the actual value. The binary messages of equal length n sent by Alice and received by Bob are denoted by A and B, respectively, where A is considered to be random. The information given by Alice is expressed using the Shannon entropy. It equals the number of bits needed to store A, which is exactly n, because A is already assumed to be represented in binary.

$$H(A) = n \tag{3.4}$$

The loss of information of Bob can be expressed using the conditional entropy.

$$H(A|B) = nh(p) \tag{3.5}$$

In fig. A.1, a plot of $h(p)$ is shown. In case of an error probabilty of $p = 0.5$, the loss of information matches the total information sent by Alice. Thus, Bob doesn't gain any information.

A reconciliation protocol R is supposed to be run on both messages A and B to produce a secret message $S \in \{0,1\}^n$ by exchanging the messages Q publicly. This can be summarised by stating $R(A,B) = [S,Q]$. Following the terminology introduced by Brassard and Salvail, a failed run, during which Alice and Bob do not share the same secret key afterwards, is indicated by $S = \perp$. The information on S given Q, which might be inferred by an eavesdropper Eve, is denoted by $I_E(S|Q)$.

It is stated, that a reconciliation protocol R is *ϵ-robust* with $\epsilon \in [0,1]$, if

$$(\exists N_0(\epsilon))(\forall n \geq N_0(\epsilon)) \sum_{\alpha,\beta \in \{0,1\}^n} \text{prob}(A=\alpha, B=\beta)\,\text{prob}(R(\alpha,\beta) = [\perp,\cdot]) \leq \epsilon. \tag{3.6}$$

Using this definition, Brassard and Salvail specify the theoretical limit of an optimal protocol R operating on a BSC with an error probability p. A protocol R is *optimal* if

$$(\forall \epsilon > 0)[R = [S,Q] \text{ is } \epsilon\text{-robust}] \tag{3.7}$$

and

$$\lim_{n\to\infty} \frac{I_E(S|Q)}{nh(p)} = 1 + \zeta \text{ with } \zeta = 0. \tag{3.8}$$

This result can be conducted from the noiseless coding theorem presented by Shannon [1948]. In the limit of long messages the information accessible to Eve should be equal to the information loss of Bob, which needs to be compensated publicly to complete the secret key S.

Below, the definition of an *ideal* protocol R by Brassard and Salvail is quoted. A reconciliation protocol R is:

1. *efficient* if there is a polynomial $t(n)$ such that $\overline{T}(n) \leq t(n)$ for n sufficiently large, where n is the length of the strings transmitted over the secret channel.

   $\overline{T}(n)$ is the expected running time of R operating on messages of a length n.

2. *ideal* if it is both optimal and efficient.

Unfortunately, there are not any known ideal protocols yet that might be suited for use in a real QKD application because these optimal protocols require a random choice from a large, exponential growing set of functions. This is a very costly action considering that it is obligatory to use a QRNG therefore, or Bob is required to decode specific messages from Alice, which cannot be done efficiently [cf. Bennett et al., 1988].

3.3.2. Cascade Protocol

A compromise between optimal reconciliation protocols and practical use is given by algorithms which are *almost-ideal*. It is accepted to reveal a relatively arbitrary small

amount $\zeta > 0$ of secret message S additionally to the theoretical minimum given in (3.8), if the protocol can be processed in only polynomial growing time $\overline{T}(n)$.

A possible implementation of such a practical protocol was presented by Brassard and Salvail [1994] as well. The postprocessing software implements this reconciliation algorithm called *Cascade*, hence it is explained in detail below.

Often, the concept of *parities* is used. A parity in terms of quantum key processing is a check bit c, which is the result of an XOR-chain of the single bits D_i part of a binary data chunk $D = \{D_1, \ldots, D_n\}$.

$$c(D) = \bigoplus_{i=1}^{n} D_i \tag{3.9}$$

The parity is used to compare the key bits $A = \{A_1, \ldots, A_n\}$ and $B = \{B_1, \ldots, B_n\}$ composing the message of Alice and Bob, respectively. In the case of matching messages $A = B$, the check bit matches as well with a probability of $p = 1$. Assuming long messages, the probability of having an even or odd number of errors per data chunk is nearly equal[1]. As a result, the check bit might match as well in the case of unequal messages with the probability of $p \approx 0.5$ since the parities of chunks with even number of errors match. Obviously, it is not sufficient to solely rely on the validation by only computing once the parity.

The Cascade protocol describes an algorithm to check the parity multiple times on different subsets of A and B. For this, a very important algorithm primitive is used. The error correction scheme *Binary*, which is capable of finding and correcting one error in a subset $D \leq A, B$ is explained below. An example is given in fig. 3.2. There, the message B of size $n = 16$ contains an odd number of errors. Hence, the initial parity check fails:

$$\begin{aligned} c(A) &= 0 \\ c(B) &= 1 \end{aligned}$$

Therefore the Binary algorithm gets triggered to correct one error. After Bob reported the parity mismatch, Alice is computing the parity again, but only for the first half (right column in fig. 3.2) of the data chunk. As Bob has an odd number of errors (in this example exactly one) in his corresponding chunk, the parities are unequal again (right column value in fig. 3.2 is shaded). Bob knows that there is at least one error in this part. Consequently, Bob asks again for the parity of the first half of this part (meaning the first quarter of the whole). He finds that this value actually matches and concludes that the error must be in the second quarter. This process continues up to the point, where Bob can determine the wrong bit. In the last row in fig. 3.2, Bob learns that Alice's check bit for the 1 bit-long

[1] The actual distribution of the total error count k is given by the Poisson distribution $P_\lambda(k)$. The expectation value is $\lambda = n \cdot p_{\text{per bit error}}$. For $\lambda \gg 1$, $|P_\lambda(k) - P_{\lambda+1}(k)| \approx 0$, which illustrates, that even and odd total error counts are approximately equally distributed.

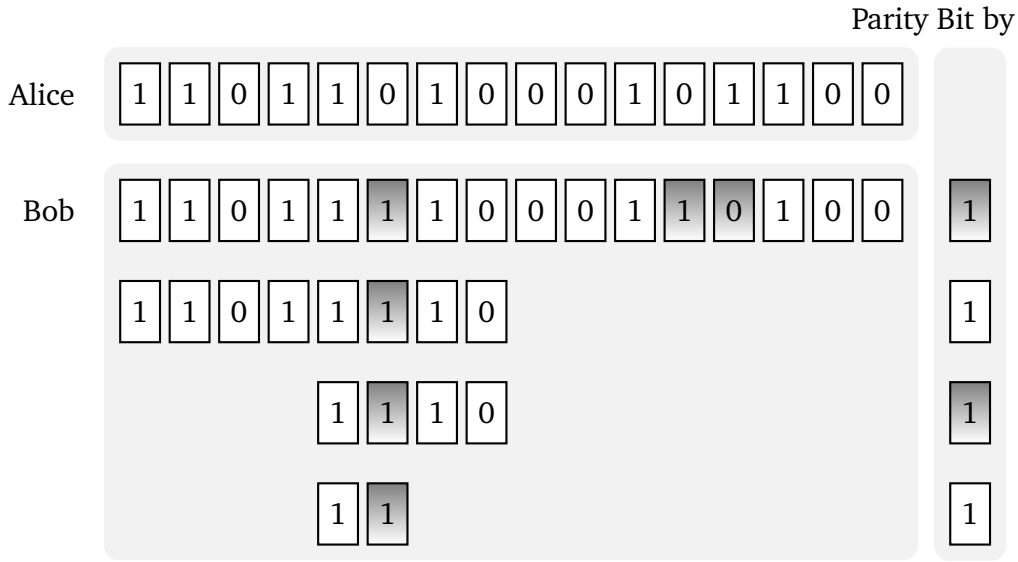


Figure 3.2.: Process scheme of the Binary error correction algorithm. The Binary error correction algorithm is used to find an odd number of errors in a block of size n and correct one of them in $\log_2 n$ steps. Thus, Alice announces the parity of the first half of the block. Bob evaluates the parity for the same block and answers. If the parity was wrong, the error is in the first half of the block; otherwise, the error is in the second half. In this figure, shaded fields mark non-matching bits or parities.

chunk $\{A_5\}$ is $c(A_5) = 1$, the bit A_5 itself. Bob's value is the same, so he can conclude that the erroneous bit must be in the second half, which only contains the bit B_6 at this point. The algorithm finishes with a bit flip of that bit.

With Binary, only one error per data chunk at a time can be corrected. Furthermore, an even count of errors per chunk stays undetected. To find and correct all errors, one has to embed the Binary algorithm into another algorithm, which pays attention to checking the parities on different block sizes and with different bit sortings to increase the probability to only have one error at maximum per data chunk. On the one hand, it is sufficient to run Binary once with an initial block size of $k_1 = 2$, but this would equal the naive `XOR` scheme presented above. Too much publicly exchanged key information Q renders this setting choice very inefficient. On the other hand, a large block size leads to a higher probability of having more than one error in such a block. Thus, the selection of a reasonable initial block size k_1 is crucial to the efficiency of the reconciliation protocol. Recommendable values can be derived by the evaluation of benchmark tests of the whole reconciliation process and are presented in fig. 3.3.

After the error estimation delivered the error probability p and the initial block size k_1 is derived, the Cascade algorithm can be initiated. The algorithm is repeated several times with different block sizes $k_i \geq k_1$, whereas each run with the same block size is applied on a unique bit order, which has to be chosen randomly by either Alice or Bob

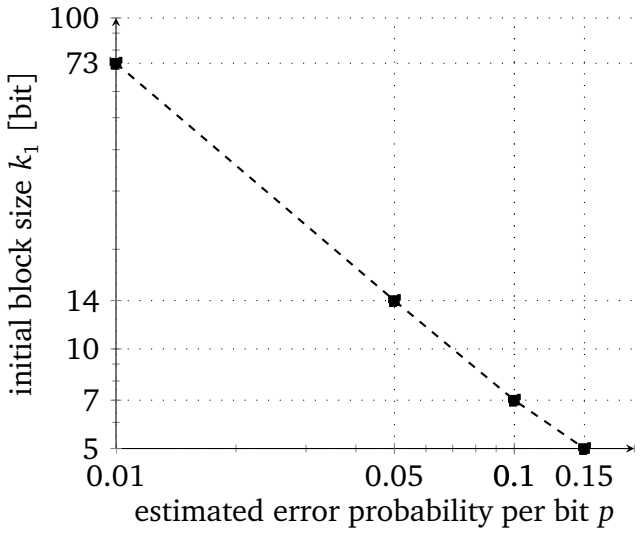


Figure 3.3.: Recommended Cascade block sizes. This logarithmic plot represents the Cascade block sizes, which were suggested in the work of Brassard and Salvail [1994]. The results were produced by running and evaluating benchmark tests.

and communicated to the other. The process is schematically depicted in the flow chart in fig. 3.4. First, the parity check with block size k_1 is processed. If blocks are detected to be corrupt, meaning that the parity comparison between corresponding blocks of Alice and Bob failed, errors get sorted out using the primitive Binary presented in fig. 3.2. Afterwards a new block size $k_2 > k_1$ is chosen. Usually the block sizes are simply doubled, so that $k_2 = 2 \cdot k_1$. Of course, the parity of two joint blocks each with matching parity will always match again, despite of possible errors. It is thereby necessary to choose another bit order at random with the intention of separating the even number of errors in the prior blocks.

The algorithm starts again by doing a parity check to find corrupt blocks. If errors have been found, they are corrected using Binary. Now it is convenient to run the Cascade algorithm again with the original bit order and the original block size k_1, because flipped bits in the second run could lead to an odd (prior even) number of errors in the original blocks, which can be corrected by Binary only now. Changes in the bit string may reveal corrupt blocks in the bit sorting of k_2 as well. So this run with k_2 is also going to be repeated.

Cascade demands to run the parity check and binary correction alternating with the original bit sortings and their respective block until no further errors are detected anymore. At this point, a new bit sorting is introduced with a block size $k_3 = 2 \cdot k_2$. The Binary algorithm is applied accordingly. If corrupt blocks are found and corrected, Cascade is going to start again with the original sorting and block size. Otherwise, the block size is doubled again.

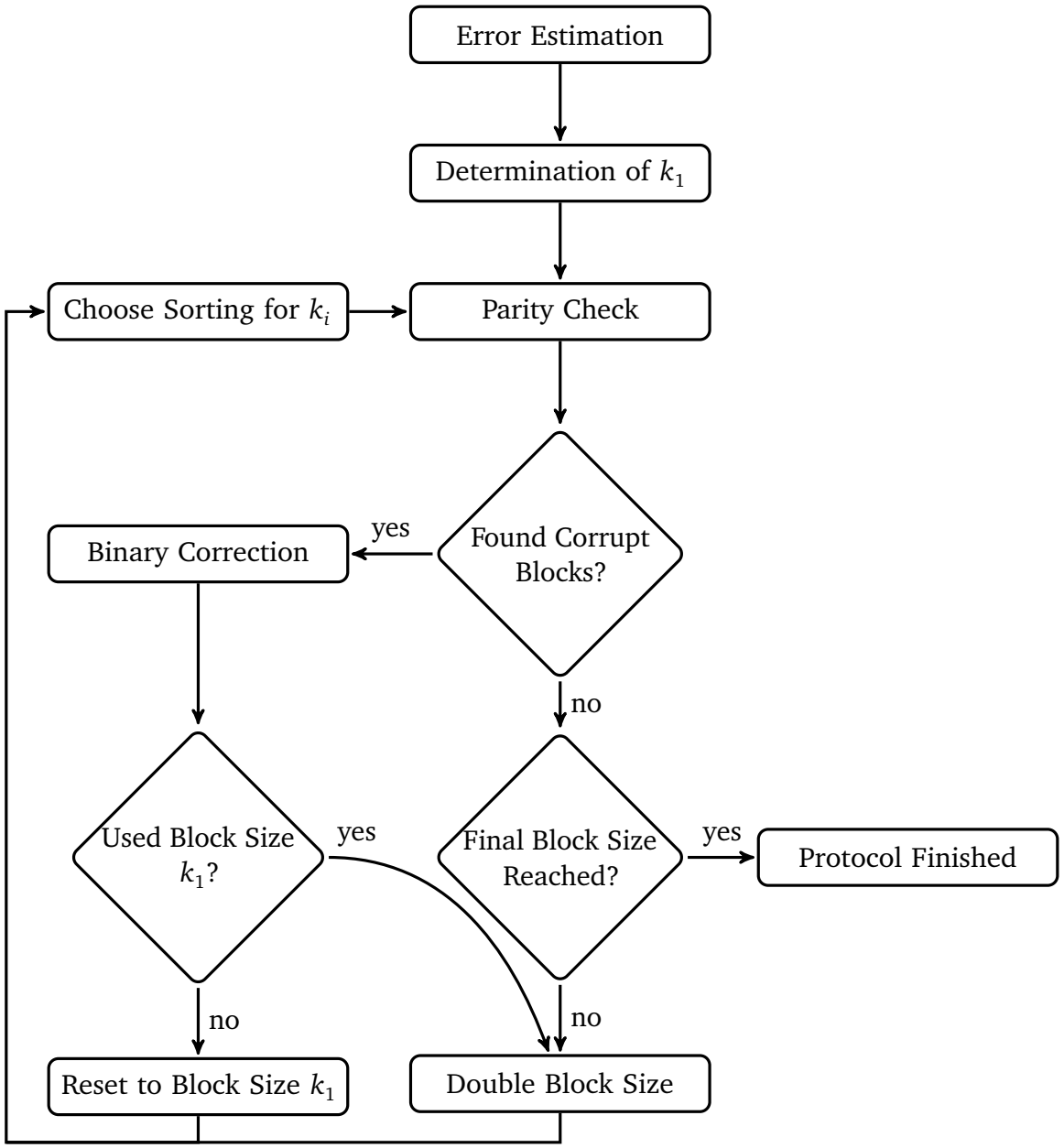


Figure 3.4.: Flow chart of Cascade protocol implementation. After an initial error estimation, the block size to start with k_1 is determined. Given this block size, the parities for all data chunks are computed. After Alice reported these values to Bob, he can make a comparison to find obviously corrupt blocks. On these blocks, the Binary correction algorithm presented in fig. 3.2 is applied. This process is repeated recursively with different sortings of the key bits each connected to a specific block size k_i until all parities match for all sortings.

This process is recursively executed up to a predefined maximum block size, where it is assumed that all errors have been found.

An analysis of the performance and security properties of Cascade is given in the paper of Brassard and Salvail [1994].

Due to public communication of parity bits, the secret shared key S is partly revealed. Hence, it is important to measure this exchanged key information Q and shorten the key size in a special way during a following postprocessing step named *privacy amplification*.

3.4. Privacy Amplification

Bennett, Brassard, Crépeau, and Maurer [1995] define privacy amplification to be the art of distilling highly secret shared information $\hat{S}$, e.g. for cryptographic keys, from a larger body of shared information S that is only partially secret.

The result of the reconciliation protocol S is considered to be equal with high probability for both parties, but it is only partially secret because a potential adversary could gain information by:

a) direct qubit measurements during the quantum channel transmission

b) the evaluation of parity bits during the reconciliation phase

c) the evaluation of authentication tokens (see sec. 3.5)

It is notable that unconditional security is only provided if the one-time pad is used in conjunction with a secret key K, which is totally unknown to any third parties. This was proven by Shannon [1949], who introduced in this context the idea of *perfect secrecy*, which means that the secret key K meant to be used with the one-time pad and the cipher text C of the private message need to be statistically independent. To achieve this, the definition of *mutual information* of two discrete random variables X and Y is used.

$$I(X;Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log_2\left(\frac{p(x,y)}{p(x)p(y)}\right) \tag{3.10}$$

$$= H(X) - H(X|Y) \tag{3.11}$$

$$= H(Y) - H(Y|X)$$

$p(x,y)$ is the joint probability distribution with respect to X and Y, and $p(x)$ and $p(y)$ are the marginal probability distributions.

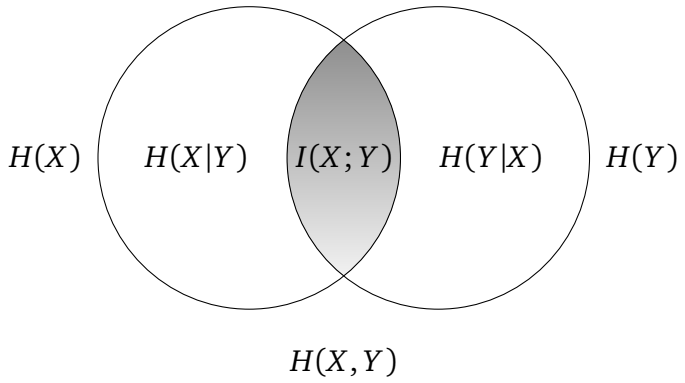


Figure 3.5.: Mutual information

It is therefore necessary that the mutual information $I(C;K) = 0$. This can be equally formulated by requiring the secret key K to be at least as long as the cipher text C.

$$H(K) \geq H(C) \tag{3.12}$$

The information $H(K)$ is obviously given by

$$H(K) = H(S) - I(S;I_E) = H(S|I_E) \tag{3.13}$$

with the information of Eve I_E, which was introduced in sec. 3.3.1 and is generally unknown to Alice and Bob. Hence, it is necessary to determine the upper boundary of I_E and extract the secret $H(K)$ by an privacy amplification algorithm.

A simple and illustrative example of such a privacy amplification algorithm is suggested by Gisin et al. [2001, pg. 7]. Alice announces random pairs of bit positions using the public channel. The XOR-result of those bits composes the new key. Considering that Eve only knows one bit of such a pair, it is not possible for her to deduce the resulting bit. To lower the chance that Eve finds a pair of bits of which she has complete knowledge, the algorithm might be applied repeatedly. Of course, this algorithm is not very efficient, because on each run the key is shortened by half.

The privacy amplification implementation that this work is based on was proposed by Bennett et al. [1995] and further examined by Kollmitzer and Pivk [2010]. It is assumed that Alice and Bob both share the same key S of size n. The information gathered by eavesdropping is denoted by I_E with an upper bound on its size of $t < n$. In this specific case, t is calculated using the estimation for the per bit error p and the number of parity checks n_{parity}, which is straightforwardly counted during the reconciliation phase.

$$t = 2 \cdot n \cdot p + n_{\text{parity}} \tag{3.14}$$

The factor of two is necessary due to the properties of the BB84 protocol. For the intercept-resend attack of Eve a measurement base has to be selected randomly. In case of a base mismatch between Eve and Bob, there is still a 50 % chance that Eve's state is projected

onto the state which represents the same bit as Alice has chosen originally, thus no error is introduced. Thereby, Alice and Bob are only noticing half of the eavesdropping attempts of Eve and the error rate p has to be doubled to take Eve's real knowledge into account.

The relation between S and I_E can be expressed using the Shannon entropy. The amount of usable information of the key S when taking Eve's knowledge I_E into consideration is n lowered at maximum by t.

$$H(S|I_E) \geq n - t \tag{3.15}$$

To reduce the knowledge of Eve, Alice and Bob choose a compression function

$$g : \{0,1\}^n \to \{0,1\}^r, \text{ with } r = n - t - s \tag{3.16}$$

from a finite set $\mathcal{G}$ at random. The information on this selection is denoted by G. The outcome may further be reduced by s bits, whereas $s \geq 0$ is commonly called *security parameter*.

The set of functions $\mathcal{G}$ is constructed to produce keys $\hat{S} = g(S)$, on which Eve's information is basically uniformly distributed and hence useless. This is achieved by a special class of functions, named *universal$_2$ hash functions*, which are discussed separately in appendix B. To prevent Eve from taking a dedicated compression function into consideration while eavesdropping to circumvent the privacy amplification, the function has to be chosen randomly only after the quantum channel transmission was completed.

To clarify the last threat, let us assume a very simple compression function already known beforehand by Eve. Alice and Bob simply drop every second bit to reduce the key to half its size. If this procedure is known before the measurement, Eve would not measure these bits going to be dropped definitively and thus the error estimation could underestimate the influence of the eavesdropper.

In the lectures of Kollmitzer and Pivk [2010, pp. 42] it is proven that the mutual information of Eve's knowledge $\{I_E, G\}$ and the privacy amplified key $\hat{S}$ is upper bounded by a function that decreases exponentially in the security parameter s.

$$I(\hat{S}; G\, I_E) \leq \frac{2^{-s}}{\ln 2} \tag{3.17}$$

Of course, it is not possible to achieve perfect secrecy as long as Eve's information is non-zero as shown in (3.17). The negligible share of information is due to the selection of the function set $\mathcal{G}$ and can be decreased arbitrarily. Other sets have been presented, which allow us to completely eliminate Eve's knowledge. Unfortunately, these sets contain so many functions to choose from that the necessary amount of random numbers and authenticated communication to share the selected function renders the protocol unusable. To give an example, consider the following function set, which fulfils the requirements of perfect privacy.

$$g : \{0,1\}^n \to \{0,1\}^r \tag{3.18}$$

This set holds 2^{r2^n} functions. The number of bits to specify a function from this set $\mathcal{O}(2^n)$ grows exponentially, which makes this set inefficient to use in a realistic field of application [Brassard and Salvail, 1994].

One practical approach to solve this issue was presented in Kollmitzer and Pivk [2010, p. 44]. Alice and Bob can derive random numbers out of their random base selection during the quantum channel transmission. These bases have to be communicated, but do not contribute to the raw quantum key, thus they can be used in the postprocessing as a source of true random numbers.

Statistically, there are roughly twice as many random bits available than bits in the sifted key. Although the key length is reduced during the sifting process, the random base bits referring to dropped measurements do not have to be omitted, but can be reused.

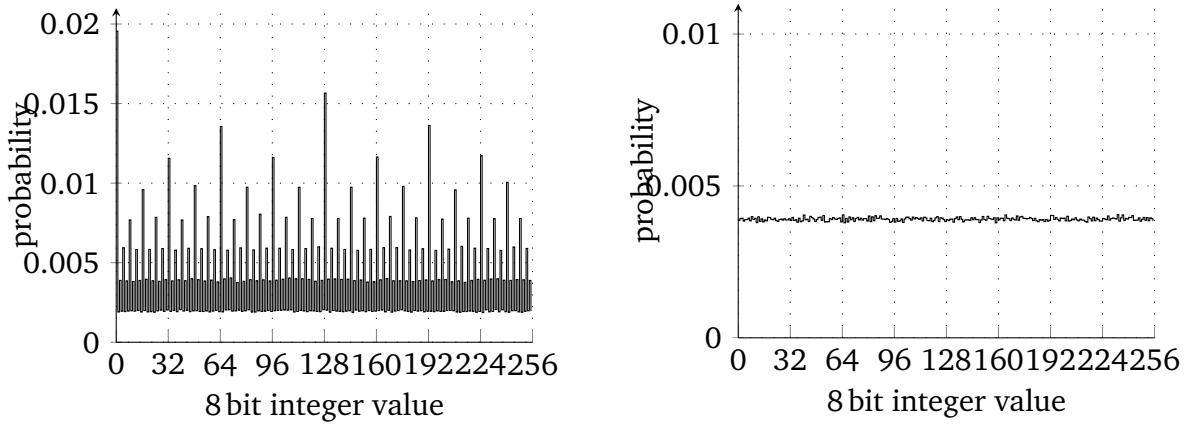
These random bits are used to form odd integer numbers c, which are necessary for a specific universal$_2$ class of hash functions, based on multiplication in finite fields initially suggested by Carter and Wegman [1977] and Dietzfelbinger et al. [1997]. While prime numbers are often crucial for these classes but are complicated to deal with, the proposed hash function given in (3.19) only requires very basic integer operations. Bit strings of size n are compressed to $r < n$ bits.

$$\mathcal{H} = \{g_c : x \to [(c \cdot x) \bmod 2^r] \in \{0,1\}^r \mid x, c \in \{0,1\}^n, c \text{ is odd}\} \tag{3.19}$$

It is remarkable that multiplication of integers and the modulus operation $m \bmod 2^r$ are both computational convenient operations that can be implemented very efficiently using today's computers. The latter just returns the r most lower-order bits of m, which can be done using the inherent integer overflow if certain conditions with respect to the size of integers are met.

The size of this specific set of hash functions is determined by the odd integer $c \in \{0,1\}^n$. Thus, there are 2^{n-1} odd elements in this class, yielding only a need of $\mathcal{O}(n)$ bits to select one function from this set.

Before proceeding with a simple example to demonstrate the usage of this hash functions, the importance of c as an odd number is explained a bit more in detail. For this, the function $g_c(x)$ with $n = r = 8$ and c and x, both chosen by a uniform RNG, was evaluated 10^6 times for each arbitrary c (fig. 3.6a) and odd c (fig. 3.6b). It is obvious that in case of arbitrary c, $g_c(x)$ does not have the property of universal$_2$ class hash functions because the result is not uniformly distributed. It is more improbable to find an odd result, because it can only be composed by the product of two odd numbers (c and r are odd), whereas the products of two even numbers, an odd and even number, or an even and odd number are always even. In the case of defining the first number c to be odd, there is only each time one case left to produce even and respectively odd results. This alone is still not sufficient for a uniform distribution because prime numbers, for instance, would be found less often, but due to the modulus operation there are 2^{r-1} numbers that are projected to the same result, giving ultimately a very smooth (theoretically uniform) distribution.



(a) c is even or odd

(b) c is odd

Figure 3.6.: Histograms to visualise the distribution of a universal class of hash functions g_c and a similar function. The universal$_2$ class of hash functions $g_c(x)$ as presented in sec. 3.4 was evaluated 10^6 times using uniformly distributed random numbers between 0 and 255 for both x and c. The condition that c must be odd, was ignored in (a), leading to a structured distribution that renders the function set useless in applications for QKD. In (b), this condition was met, thus the histogram shows the property of the uniform distribution of the hash functions.

To give a clear impression on how privacy amplification is actually implemented, an example key S of size $n = 27\,\text{bit}$ is processed. For this, the binary representation of numbers is denoted by the appended subscript base 2.

$$S = 111011100110010001001001011_2 = 124985931$$

This bit string needs to be compressed due to Eve's knowledge of $t = 8\,\text{bit}$. Additionally, the security parameter is set to $s = 2\,\text{bit}$. The reduced key size is $r = 17\,\text{bit}$. The base selection was used to derive a constant c of equal size of S.

$$c = 110010100000101011000000010_2 = 105928194$$

Accordingly, the compression function is given by:

$$g_c(x) = (105928194 \cdot x) \bmod 2^{17}$$

Finally the privacy amplified key $\hat{S} = g_c(S)$ can be evaluated.

$$\begin{aligned} g_c(S) &= 101111000010010100100100110101111011\underbrace{01111011010010110}_{\text{last 17 bit}}{}_2 \bmod 2^{17} \\ &= 10111011010010110_2 = 95894 \end{aligned}$$

3.5. Authentication

The authentication step is necessary to verify that Alice and Bob communicate with each other and not with a third party, commonly called *man-in-the-middle*. While historically, letters could be recognised to be authentic by the handwriting or a unique signature, for instance, it is necessary to introduce separate mechanisms to provide authentication for digital transmissions, as the messages do not provide any inherent possibilities anymore. Even worse, the advantages of easy duplication and manipulation of these messages are simultaneously a significant drawback with respect to authentication.

Of course, the problem of authentication is not unique to QKD alone. Thus one may find it convenient to use existing technologies for this purpose. For instance, instead of solely relying on a simple TCP/IP connection, a virtual private network (VPN)[2] connection could be used to establish authentication. However, VPN uses classical cryptography methods, which only provide computational security.

To provide provably unconditional security for authentication, it is necessary to stay within the realm of trusted tools. To achieve this, only QRNG numbers have to be used, private keys cannot be reused (cf. one-time pad) and compression functions have to be free of any kind of bias. The latter is crucial for authentication and can be fulfilled by using again universal$_2$ hash functions, which are clarified in appendix B.

In principle, the authentication is realised by calculating checksums out of the original message and pieces of a private key, which was already shared in advance, such as by a former QKD session or during a face-to-face meeting. This security token can be interpreted as a signature, which is then calculated for each message and transmitted right after the message itself. The receiver of this signed message has to repeat the checksum calculation to make a comparison. In case of not matching tokens, the transmission was either tampered or erroneous for any other reasons.

A major problem in the authentication process is the consumption of private key, which is used to choose compression functions within the family of universal$_2$ class hash functions. To authenticate a message in the set $\mathbb{M}$ of size $a = \log|\mathbb{M}|$, a private key of size in the same order $\mathcal{O}(a)$ is required. Hence, the QKD application would be very inefficient.

However, solutions of this problem were already presented. The amount of necessary bits to authenticate a message can be lowered to a logarithmic growth $\mathcal{O}(\log a)$ using a technique presented by Wegman and Carter [1981] and depicted in fig. 3.7. Equivalent to the definition of a, the bit size of the tokens in the set $\mathbb{T}$ is given by $b = \log|\mathbb{T}|$. The universal$_2$ class presented already above in sec. 3.4 is adjusted to compress input values of size $2s$ to tokens of size s with

$$s = b + \log\log a. \tag{3.20}$$

[2] A virtual private network is a common method within the realm of classical computer networks. It allows multiple local area networks (LANs) to be linked via an insecure connection, such as the Internet, and provides for these classical methods to establish authentication and encryption.

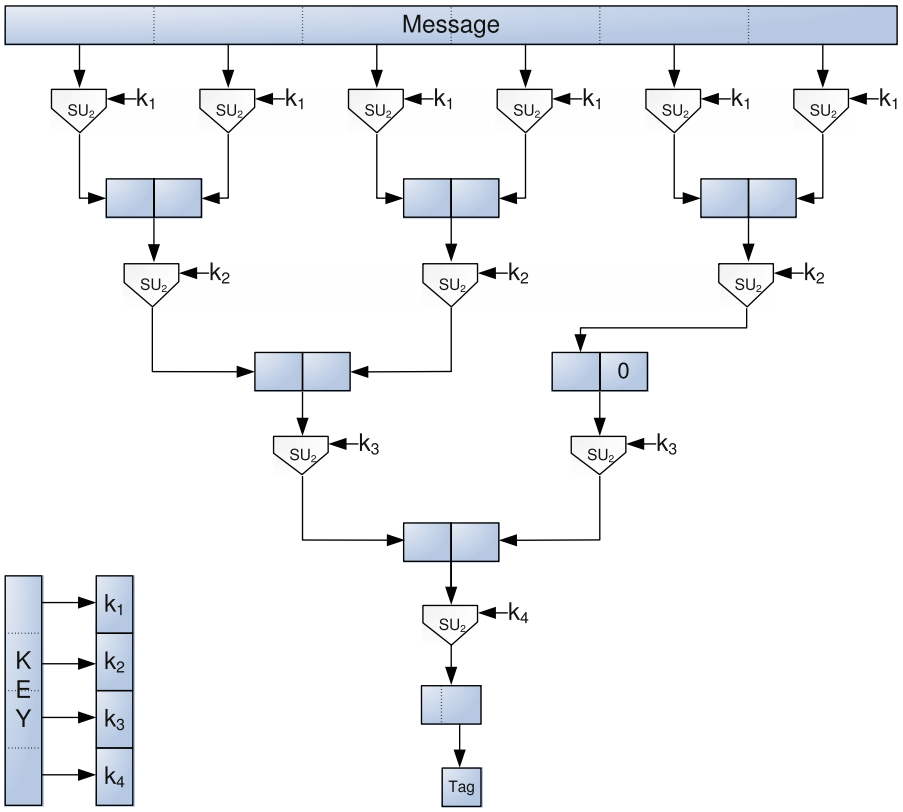


Figure 3.7.: Recursive authentication token generation. The authentication token (tag) is generated by recursively applying a compression function using a scheme suggested by Wegman and Carter [1981]. For this, the message is divided into parts of size $2s$. A hash function chosen by a private key k_i from a strongly universal$_2$ class (SU_2) is used to compress the single parts to even smaller parts of size s. The advantage of using such a scheme instead of using hash functions, which map entire messages directly to tokens of the specified length, is the ability to save private key bits.

Source: Kollmitzer and Pivk [2010, p. 17]

For this purpose, the parameters in (3.19) have to be set to $r = s$ and $n = 2s$. The original message is divided into parts of $2s$ and on each part the same compression function chosen by the private key k_1 is applied. When necessary, the message has to be extended by a zero padding to allow division without a remainder. In the next step, two consecutive outputs are chained together to be compressed again using the key k_2. This algorithm is repeated until only one part of size s is left. The lower-order b bits represent the final token. Wegman and Carter demonstrated that this routine fulfils the requirements to be set up an almost strongly universal$_2$ class (cf. appendix B).

There are $\log a - \log b$ iterations necessary that each consume a private key of roughly $2s = 2(b + \log\log a)$. Henceforth, the total private key size n_k of the joint key $k = \{k_1, k_2, \ldots\}$ can be calculated.

$$\begin{aligned} k &= 2(\log a - \log b)(b + \log\log a) && (3.21)\\ &= 2\log a(b + \log\log a) - \underbrace{2\log b(b + \log\log a)}_{\approx 0} \\ &= 2\log a(b + \log\log a) = \mathcal{O}(\log a) && (3.22) \end{aligned}$$

Hence, the cost of authentication with respect to private key consumption grows only logarithmic with the size of the message, assuming $a \gg b$.

The security provided by this authentication scheme only depends on the amount of possible tokens 2^b. In the original paper [Wegman and Carter, 1981], it is shown that the probability p for a potential adversary to find a valid authentication token for a manipulated message, given that one message-token pair is already known, is upper bounded by

$$p < \frac{2}{2^b} = \frac{1}{2^{b-1}}. \tag{3.23}$$

This scheme is used to authenticate all messages that are delivered using the classical public communication channel. This means that all messages concerning the sifting, error estimation and finally error correction are extended by a security token, which is checked by the receiver party. As mentioned above, the privacy amplification does not need any further classical communication because information gathered and authenticated during the sifting process can be used.

3.6. Implementation Details And Benchmarks

Above, mainly the underlying theory and ideas were presented. In this section, the own implementation of the postprocessing stack is shown. The most important software module is directly given in appendix C. Lastly, the results are presented with focus to either the gain/quantity of the secret key or the randomness/quality.

Software Implementation

The postprocessing software is not strictly integrated to the experimental setup, which means that the postprocessing software can be easily changed to supply the postprocessing chain to similar QKD experiments. Hence, the software only expects generic measurement data consisting of Boolean values (flags) describing the chosen settings of Alice to manipulate the qubit states or in Bob's case describing the configuration for his qubit measurement (base choice) as well as the detection result of his detectors.

In the current experiment only one FPGA module is used (cf. sec. 2.2), thus the Boolean flags representing the qubit transmission are not gathered separately, but instead in one place. Of course, this has to be changed in a further iteration of the experimental setup. This is acknowledged in the software by a very modular structure, which is already prepared to easily implement another file format only including either Alice's or Bob's information.

Up to this point, the data exchange between the FPGA module and the postprocessing software is realised using a simple binary file structure, which is described in fig. 3.8. Each qubit transmission is represented by an 8 bit integer value. The binary measurement file is consisting of many of these integers that are directly appended to each other. The single bits of one integer value represent the settings of Alice and Bob as well as the measurement details of the detector. The random selection of Alice's base (vertical-horizontal or circular) is memorised using a Boolean flag B_A.

The base represented by either $B_A = 0$ or $B_A = 1$ depends on the experimental setup and the FPGA processing. It can be defined arbitrarily and the software can be configured to meet this choice. The same applies to all other flags.

The flag b_A represents the random bit Alice intended to transmit. Analogous to B_A, there is a flag B_B, which denotes the base choice of Bob. The flag b_B is meant to save the bit Bob is looking for when only one detector is available and thus it is not possible to look for both states at the same time. In the current setup, two detectors are available. So this bit is not needed in our case and is consequently not processed. Finally there are two flags m_1 and m_2 in the integer representing the measurement result of the detectors, in which $m_{1/2} = 0$ indicates that no photon was measured in the given time frame. If the flag is set, meaning that the binary value is 1, a photon was measured.

Generally, the single flags are decoded in a defined position in the integer given by the file format specification.

To check the measurement data for consistency and allow debugging of the postprocessing software independently of the experiment, a separate *QKD Measurement Data Tool* has been developed to generate artificial measurement data files in the appropriate file format (fig. 3.9b) and run simple statistic tests (fig. 3.9a).

The QKD Measurement Data Tool as well as the postprocessing software and its GUI are developed using C++ and the cross-platform application framework Qt 4.7.4 provided by

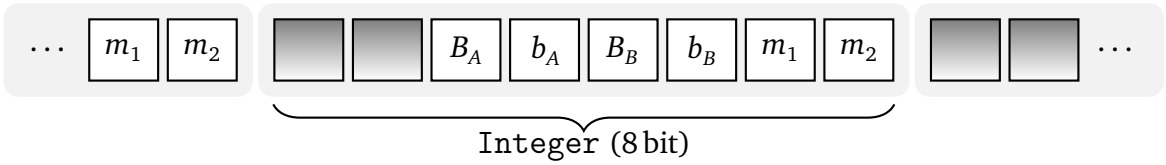


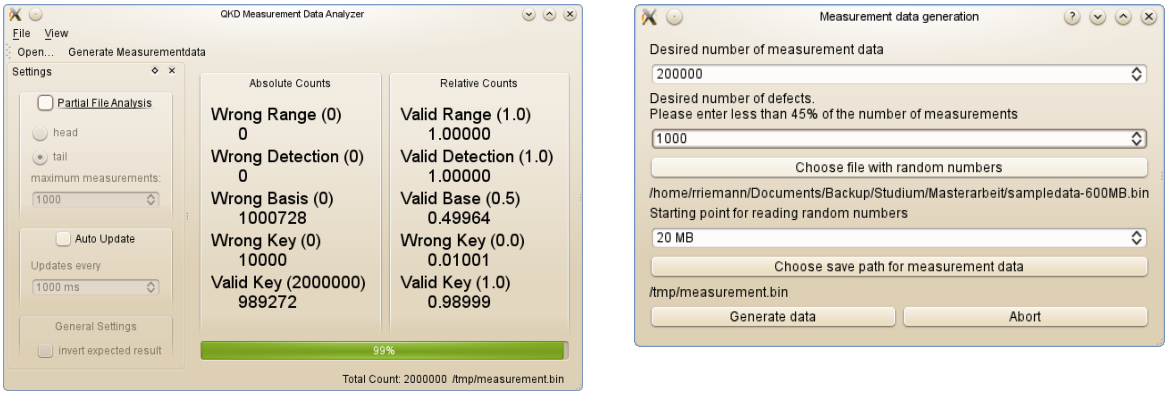
Figure 3.8.: Scheme of binary file format for measurement data. The measurement data are streamed from the FPGA module to a computer, which stores the information in files using a very simple binary scheme (cf. sec. 2.2). The bit B_A/B_B denotes the base choice of Alice and Bob, respectively. The bit intended to send/receive is represented by the flag b_A/b_B and the measurement result of the single APDs detectors (photon registered or not) is saved to the flags m_1 and m_2, respectively. Each measurement is represented by 8 bit (flags), of which the two higher-order bits (shaded) are not used. The flag b_B is also not used in our setup as well, but would be necessary in setups with only one detector. There is no separation between these 8 bit composing one integer value that represent one measurement. Hence, the single flags can only be distinguished by their position in the file.

the Qt Project [2012]. Hence, the software is supported in principle by all major desktop platforms, whereas it was so far only tested on openSUSE 12.1 64 bit (Linux version 3.1.10, GCC version 4.6.2), a GNU/Linux operating system published by the openSUSE Project.

The postprocessing software application, from now on referred to as *QKD Client*, has to be executed by Alice and Bob (fig. 3.10a). First of all, a measurement data file has to be opened. A second file with an initial private key for the authentication has to be provided as well, but for now it is compiled into the source code to ease the configuration in our proof-of-concept setup.

Afterwards both clients try to connect to each other using a TCP/IP connection, which provides the classical public transmission channel necessary for the postprocessing communication. For this, only very few configuration settings have to be set by the user. Either Alice's or Bob's client has to be provided with the IP address of the other client and the network port the client is listening to. Furthermore, the user has to select which client should behave as Alice (*master* client) and which as Bob (*slave* client). These settings only have to be given to the client who is establishing the connection. The other client is automatically configured to take the other role. To start the connection process, the connect button has to be clicked. The software is reporting constantly what happens in a log window. The state of Alice's client after a successful connection is depicted in fig. 3.10b.

At this point, the postprocessing can be initiated by pressing the execute button (gear icon). The key is processed as presented in fig. 3.1. The results of the error estimation are reported in the log window. The error correction is started afterwards and the Cascade algorithms run until the finishing criteria are fulfilled. In order for this to occur, the routine

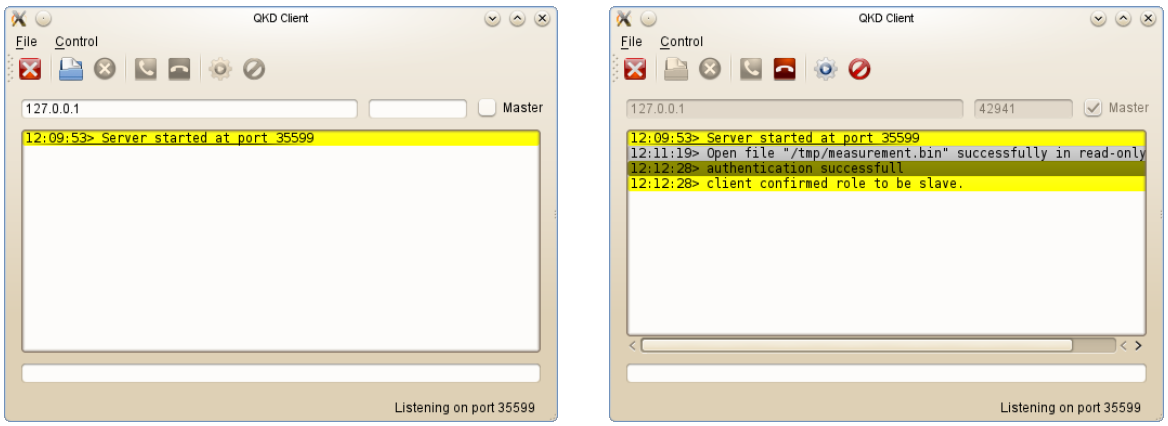


(a) QKD Measurement Data Analyzer

(b) QKD Measurement Data Generator Dialog; courtesy of Marcel Granetzny

Figure 3.9.: GUI of the QKD Measurement Data Analyzer. The QKD Measurement Data Tool allows to check binary measurement data files for consistency by running simple statistics checks (a). To detect potential file corruptions (i.e. bit misalignment), the 8 bit-integer values representing single qubit transmission are verified to have two leading-order zeros (cf. fig. 3.8). Due this restriction, the integer values are expected to be in a certain *Range*, which is checked and displayed in the GUI. Furthermore only one photon per measurement is allowed (valid detection). Finally the expectation of a uniform base distribution is analysed and the transmission error after sifting is evaluated. It has to be remarked that this kind of statistics is only possible as long as Alice's and Bob's information are both available as input data, which is the case in our demo experiment in contrast to any real world application.

Furthermore, this tool includes a measurement data generator (b) which takes random numbers from a binary file provided by an online service hosted by our work group [Wahl et al., 2011] to generate an artificial measurement data file convenient to test the postprocessing software. The amount of total transmissions and transmission errors is adjustable. The algorithm and the GUI for this generator was developed by Marcel Granetzny.



(a) QKD Client GUI before connection

(b) QKD Client GUI of Alice (master) after connection

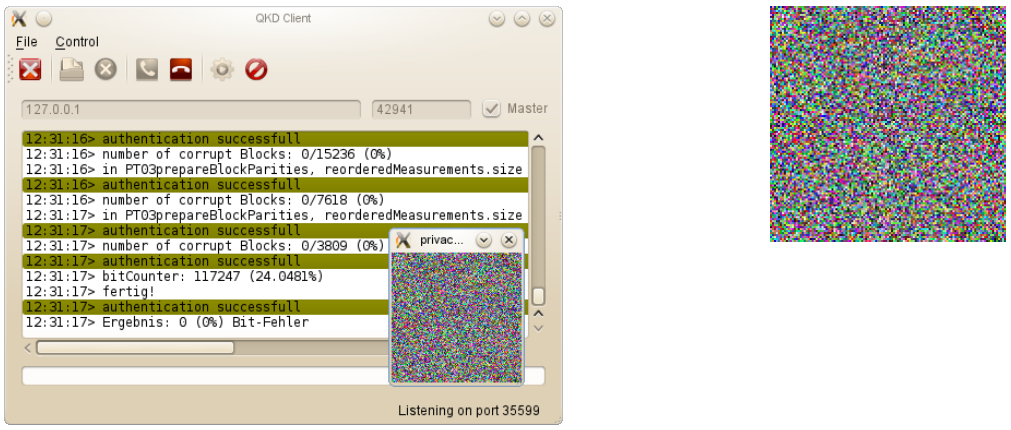
Figure 3.10.: GUI of the QKD Client. The application is very easy to use due to shortcuts, toolbar buttons with hints on mouse-over and an application menu. After the start-up depicted in (a), a measurement data file needs to be opened. After the setup of the network settings, the connection can be established. The log in (b) notes that the other slave client has recognised to act as Bob.

has to be passed through four different sortings with increasing block sizes without finding any corrupt blocks (cf. sec. 3.3.2). The number of parity bits that have to be interpreted as publicly known to a potential adversary is counted during the process and reported in the log. Finally, the privacy amplification is finished. The whole communication is authenticated using a private key file which has to be shared in advance. The successful authentication of all responses of Alice and Bob is reported in the log as well. In case of an authentication failure, the processing is halted immediately.

After the postprocessing is complete, Alice's and Bob's secret key are compared bit by bit. This improves the debugging capabilities and has to be removed, of course, for an application in a production environment. As a second aid to judge the randomness of the resulting key, an image from the key is generated which is automatically shown to the user immediately after the processing has finished (fig. 3.11).

The software consists of several separate modules (so called classes[3]), which helps to ease the further development. There is a dedicated class `QKDProcessor` for the postprocessing, which implements all necessary steps and can be easily modified or exchanged to realise the processing for different QKD protocols. There are different classes to handle the connection between both clients, which have been created in a way that the actual `QKDProcessor` class does not need to deal with the specific channel properties of the

[3]The QKD Client software was implemented using the object-oriented programming (OOP) paradigm. The whole software is divided by functionality into modules, which are called classes and feature properties and methods to use these.



(a) QKD Client GUI after postprocessing (b) graphical representation of the key

Figure 3.11.: GUI of the QKD Client after successful postprocessing. After the postprocessing has been finished, the generated secret key is saved to the working directory and additionally a graphical representation is presented to the user (a) which is shown in original size in (b). Every pixel in this pictures is composed of three colour channels RGB with each consuming 8 bit of the secret key. The picture might not represent the full secret key in cause of its quadratic constraints. It is expected to miss any structure, because of its input derived from quantum random numbers.

TCP/IP protocol stack. There is a separate `Authenticator` class which takes care of the authentication.

Quantitative Analysis

Measurement files have been recorded using the single photon source equipped with either nitrogen-vacancy (NV) or silicon-vacancy (SiV) colour centres. The property of the single photon source to emit mainly single photon pulses was investigated by measuring the intensity auto-correlation function $g^{(2)}(\tau)$ using the HBT setup by Brown and Twiss [1956].

$$g^{(2)}(\tau) = \frac{\langle I(t)I(t+\tau)\rangle}{\langle I(t)\rangle^2} \tag{3.24}$$

For this, the light beam is divided by an unbiased beam splitter, whose both outputs are equipped with an APD to measure the light intensity $I(t)$. For quantum single photon sources it is expected to observe $g^{(2)}(\tau = 0) \approx 0$, because the single photon can only be detected by either the one or the other APD. The results of this measurement are qualitatively presented in fig. 3.13 and meet the expectation of a $g^{(2)}(\tau = 0)$ value significantly below 0.5, thus implying single photon emission.

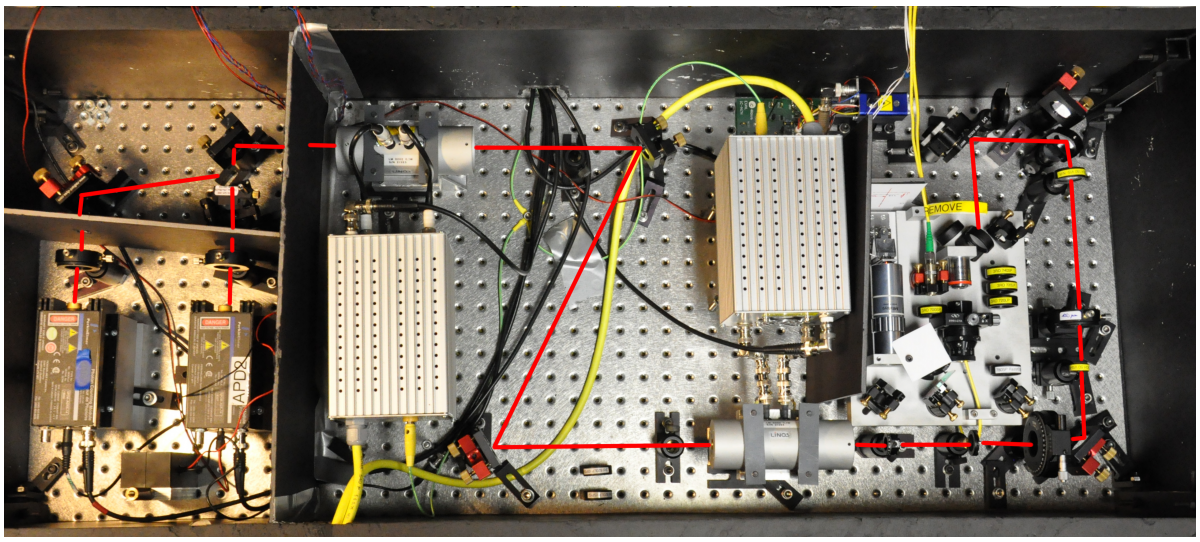


Figure 3.12.: Photo of the experimental setup. The setup of optical devices fits on a ground plate of 1.2 m × 0.5 m. On the right, the single photon source is situated. The beam is guided through two EOMs and afterwards pointed to two APDs on the left. The beam path is marked by a red line. A schematic view is presented in fig. 2.2.

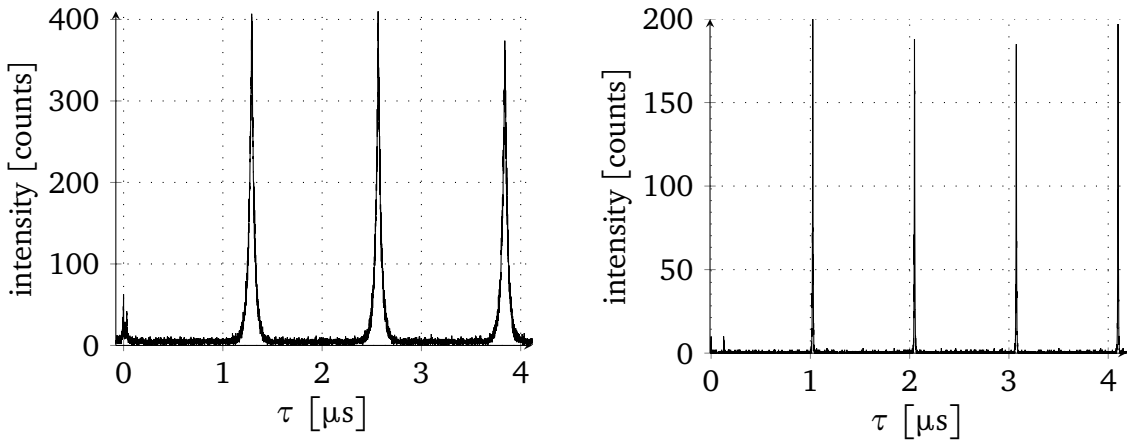
A photo of the experimental setup is depicted in fig. 3.12. Further discussion of other properties or components have not been subject of this work.

Then, the data files were processed using the QKD Client and different parameters were evaluated, which are listed in tab. 3.1.

The frequency of the excitation laser $f_{[\mathrm{Hz}]}$ was altered using the FPGA software at different values. Only very few excitation pulses can be converted to a single photon. Furthermore, the optical misalignment, imperfect optical devices and limits of the detection efficiency of the APDs influence the optical efficiency q_{dect}, defined as the quotient of detected photons and excitation laser pulses, which is typically in the order of $\mathcal{O}(5\,‰)$. The estimated error calculated by the software using a random sample from the measurement is denoted as p_{error}.

The ratio $q_{\mathrm{conversion}}$ is given by the bit size of the final secret key normalised by the number of measured photons after sifting. The value is defined to have a theoretical maximum at $q_{\mathrm{conversion}} = 1$, but depending on p_{error}, it decreases as parity bits have to be revealed during the error correction, leading to a smaller secret key in the end. During the classical communication, a secret key shared in advance is used for authentication purposes. The ratio between the already consumed secret key during the authentication phase (cf. sec. 3.5) and the produced secret key is given by q_{used}. It has to be remarked that $q_{\mathrm{conversion}}$ and the secret key size as given in these benchmarks are not corrected by this value.

Corresponding to our expectations, the conversion ratio decreases with growing error rates. To analyse this in detail, artificial measurement files are used, which were pro-



(a) NV colour centre

(b) SiV colour centre

Figure 3.13.: Intensity auto-correlation function for vacancy centre samples under pulsed excitation. The function $g^{(2)}(\tau)$ is a measure to distinguish between classic photon sources and quantum photon sources, e.g. single photon sources. The delay τ between two photons is measured and used to populate a histogram. Single photon sources are supposed to have $g^{(2)}(\tau = 0) = 0$, because otherwise two photons would have been measured at the same time and thus the source would not emitting single photons. The small peak in both histograms at $\tau = 0\,\mu s$ indicates that the photon source only emits very few multi-photon pulses using nitrogen-vacancy (NV) or silicon-vacancy (SiV) colour centres, of which the latter performs slightly better.

	$f_{[\mathrm{Hz}]}$ [kHz]	p_{error} [%]	$q_{\mathrm{conversion}}$ [%]	q_{used} [%]	sec. k. rate [kbit/s]	key size [kB]
NV	1000	3.0 ± 0.1	64.0	1.4 ± 0.4	2.77	104
NV	800	3.2 ± 0.1	66.0	1.5 ± 0.4	2.21	83
SiV	1000	3.3 ± 0.1	65.8	3.5 ± 1.0	1.03	39

Table 3.1.: Properties and results of used measurement data files. The data have been determined by executing the postprocessing chain three times. Significantly fluctuating parameters are given with an uncertainty based on Pythagorean error propagation. For the first two samples nitrogen-vacancy (NV) colour centres were used. The third sample was recorded using silicon-vacancy (SiV) colour centres instead. $f_{[\mathrm{Hz}]}$ is the frequency of the excitation laser as defined in (2.1). The estimated error of the measurement is denoted by p_{error}. The ratio between secret key size and measured photons after sifting is given by $q_{\mathrm{conversion}}$. The proportion of this used for authentication purposes is q_{used}. The *secure key rate* denotes the final size of the secret key file (*key size*) normalised by the measurement time.

duced by the measurement data generator belonging to the QKD Measurement Data Analyzer. To ease the processing of benchmarks, a special demonstration (demo) mode was implemented.

The very modular QKD Client also allows to use the `QKDProcessor` class in a demonstration mode. For this, the postprocessing of Alice and Bob, each represented by an instance of the processor class, is done within two separate threads in the same CPU process. Authentication and TCP/IP connection are not used. Both threads are communicating directly with each other. The demo mode is accessible by a command line flag followed by the measurement data file. A GUI is not started in this mode.

```
./privacy-amplification -demo measurementfile.bin
```

The demo mode is much faster and does not require to setup the network connection. Thus, it is very convenient for testing the `QKDProcessor` class.

To estimate the approximate performance of the QKD Client, several measurement data files of each $2 \cdot 10^6$ qubit transmission were generated using the generator tool. Thereby, the amount of erroneous transmissions is exactly known beforehand. The resulting secret key size represented by the conversion ratio was plotted in dependence of the error ratio in fig. 3.14.

The experimental maximum of a file without any transmission errors is given by approximately 94.8 %. This is due to some overhead consisting of the error estimation and a necessary minimum of parity checks.

A sample with 11 % of errors would already yield a negative net gain, because the secret key-wise authentication costs would probably overweight the rather small outcome corresponding to a conversion ratio of $q_{\text{conversion}} = 4.6\,\%$ (cf. tab. 3.1). A sample with 12 % errors leads already to an empty secret key because 100 % of information has to be removed in the privacy amplification step. This result is in agreement with the theoretical limit examined by Shor and Preskill [2000].

Between these extreme values, the conversion ratio is approximately linear. An increase of 1 % of the error ratio leads to a decrease of the conversion ratio of approximately 8 %. Divergences might be introduced by the non-continuous function, which determines the initial block size k_1 depending on the estimated error ratio for the Cascade algorithm.

An example of the decreasing key size during the postprocessing is depicted in fig. 3.15. The largest loss is given by the sifting phase, which is a fundamental property of the BB84 protocol. The decrease due to the error correction/privacy amplification highly depends on the error rate and is subject to appropriate adjustments in the experimental setup, i.e. qubit transmission and measurement process. The costs of authentication can be tuned to meet the necessary security standards and are generally relatively small.

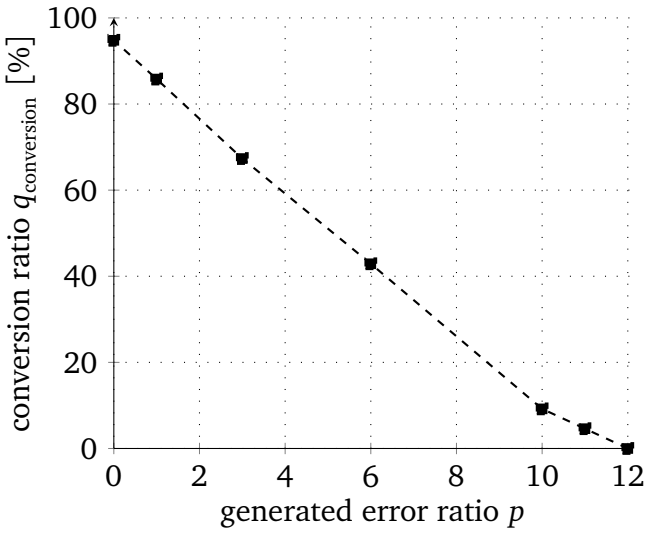


Figure 3.14.: Dependence between error ratio p and output conversion ratio $q_{\text{conversion}}$. Several measurement data files with each $2 \cdot 10^6$ qubit transmissions with known accurate error ratio p were generated to analyse its dependence on the relative secret key size, the conversion ratio. This conversion ratio $q_{\text{conversion}}$ shows how much information of each transmitted and sifted qubit was used on average in the final secret key. The results were produced with the demo mode of the QKD Client using single sample files and are subject of small fluctuations is cause of the non-deterministic property of the postprocessing. The uncertainty has not been further examined.

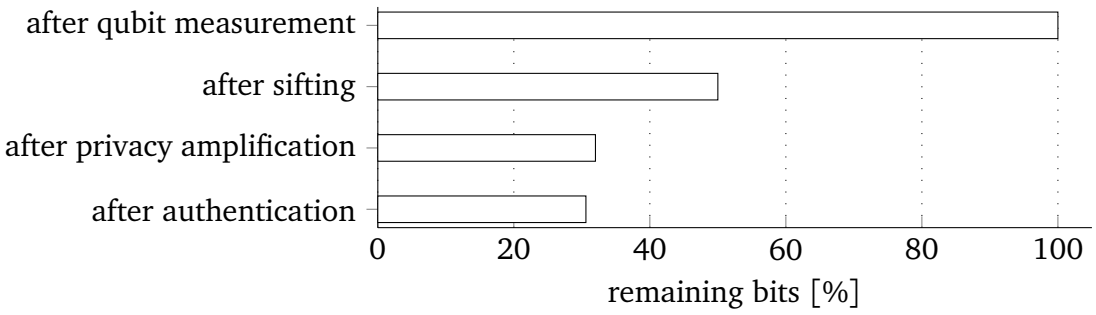


Figure 3.15.: Bar chart representing the development of the key size. This shown data is based on the measurement sample evaluation from the first row of tab. 3.1. The amount of qubits that were transmitted successfully are considered to equal 100 %. Statistically only half of the qubits are measured in the corresponding base leaving only roughly 50 % of initial key size. The error estimation and correction of 3.0 % erroneous bits enforce further key compression to approximately 32.0 %. The authentication costs of this process amount to 1.4 % leaving only approximately 30.6 % of bits to constitute the final secret key.

Qualitative Analysis

After the evaluation of the secret key's quantity, the results of the quality analysis of the final secret keys follows below.

The one-time pad requires a shared key, which ideally must be perfectly random. Throughout the whole measurement process, only QRNG numbers were used. The final secret key is only derived from this data, thus the key is expected to to share the same random properties.

Nevertheless, a potential bias in the measurement efficiency of the different photon states could introduce a less random final secret key, hence it is convenient to check our expectations.

For these tests the random number statistics suite `TestU01` published by L'Ecuyer and Simard [2007] was used. This library allows to analyse binary files with regard to the null hypothesis H_0 that the binary file is the result of a true RNG. Multiple kind of algorithms are used to find a structure in the provided sample and its results are expressed using the p-value, which describes the probability p to get such a data sample in case of a valid H_0 hypothesis. The default value to reject the H_0 hypothesis of the `TestU01` suite is configured to $p = 0.1\,\%$. This means that statistic tests are considered to be passed if the p-value is above this threshold. The results are given in tab. 3.2.

For all test samples, all algorithms to examine the randomness yielded a p-value above the threshold and thus confirm the H_0 hypothesis. The smaller values listed in tab. 3.2 do not contradict this conclusion and can be explained by a too small sample size.

Conclusion

The BB84 protocol was implemented successfully. For this purpose, a single photon source operating at room temperature was used. The intensity auto-correlation function was measured to verify its quantum nature.

Qubits were prepared by Alice using these single photons and were sent to Bob. For this, only true random numbers were used. The transmission results were saved to binary files, which were analysed afterwards using the postprocessing chain.

Taking tab. 3.1 into account, the theoretical maximum performance of the QKD postprocessing was achieved by approximately

$$q_{\text{conversion}} \cdot (1 - q_{\text{used}}) \approx 60\,\%.$$

The statistical tests to qualify the randomness of the resulting secret key (cf. tab. 3.2) do not give any evidence of a damaged key. Thus, the resulting files can be considered to be ready for use along with the one-time pad.

sample	size [kB]	CollisionOver	LempelZiv	AutoCor$_{d=2}$	passed tests
QRNG sample	897	0.93	0.25	0.86	38/38
QRNG sample	123	>0.99	0.33	0.23	33/33
NV sample $_{1\,\mathrm{MHz}}$	102	0.94	0.91	0.81	33/33
generated $_{p_{\mathrm{error}}=5\,\%}$	102	>0.99	0.56	0.43	33/33

Table 3.2.: RNG test suite results of used measurement data files. The suite TestU01 was used to evaluate statistically the randomness of the provided binary secret keys, which are are in turn the result of the postprocessing. Presented is an arbitrary selection of tests being part of the *Rabbit* benchmark [cf. L'Ecuyer and Simard, 2007] and its p-value. At first, the random number source file used for the measurements and the sample generation was analysed. To check the influence of the sample size, a subset with a size comparable to the real measurement files and a larger subset was examined. In both cases all tests were passed. Afterwards the same tests were undertaken for the secret key derived from the measurement using NV colour centres at 1 MHz (cf. tab. 3.1) and a generated sample with an error ratio $p_{\mathrm{error}} = 5\,\%$ of the same size. The results are in agreement with our expectations.

Chapter 4.

Outlook

Towards a more realistic quantum key distribution experiment, some drawbacks and simplifications should be addressed in future that were not in our scope.

4.1. Improvements Of The Experimental Setup

In our proof-of-concept setup there is no strict separation between Alice and Bob. Although the optical devices of both parties can be separated until the free space quantum channel transmission becomes to erroneous, the control unit is still shared to simplify the synchronisation of excitation laser, EOMs and APDs. To resolve this issue, Alice and Bob have to be controlled independently by an own FPGA module. The crucial synchronisation between both units can be achieved by additional classical laser pulses at a fixed frequency to detect and compensate possible internal clock shifts [Tang et al., 2011].

Furthermore, it would be desirable to integrate as much as possible related computations and postprocessing steps directly into the FPGA module. In this way, costs can be lowered and the experiment could be further miniaturised. Today's FPGA modules feature a comprehensive set of possibilities. Hence, it is imaginable to use these components not only as I/O interface to control the excitation laser, EOMs, APDs, but also to realise the synchronisation, to request true random numbers from remote (e.g. online services), handle the classical communicating via a classical Ethernet connection for the postprocessing, and finally save the privacy-amplified secret key to a pen-drive.

It is also planned to use the described control mechanisms and the postprocessing implementation with another QKD experiment based on fibre-optics. In this setup, the free-space beam quantum channel is replaced by a fibre cable link. Thereby, no line of sight is required. However, polarisation-encoded qubit states cannot be used anymore, because the states would be destroyed due to polarisation mode dispersion.

Instead, a phase-encoding scheme can be used [Bennett et al., 1992], in which the qubit states are generated and analysed by two unbalanced Mach-Zehnder interferometers. This scheme, referred to as time-bin encoding, was already successfully demonstrated by other groups, e.g. Townsend et al. [1993]. The fundamentals of the BB84 protocol and its

postprocessing remain unchanged. Hence, the software can be reused with hardly no modification.

4.2. Improvements Of The Postprocessing

Beside the planned modification of the experimental setup, there is room as well for software improvements regarding the postprocessing. Without introducing new concepts or changing the underlying mechanisms of the current processing chain, the application performance with respect to bandwidth usage of the classical communication channel can be further enhanced. In the current state, the TCP/IP packages contain much more meta information than necessary. Reducing the size of these packages would lower the use of bits for authentication as well. Hence, the overall QKD gain would improve.

A much larger benefit might result from improvements of the error correction algorithm. The block size within the Cascade algorithm (cf. sec. 3.3.2) is of particular interest for this. A modification called *Adaptive Cascade* was already presented by Kollmitzer and Pivk [2010].

Currently, the postprocessing software is still operating on files. To allow a continuous growing quantum key, it is advantageous to implement the handling of continuous input streams, which would require to buffer the stream first. When a specific buffer size would have been filled, the postprocessing would be run on it instead of files like it has been done before. Thus, the current secret key generation rate would be accessible already during the measurement.

As time evolves, the field of communication theory might present new research results, which also affect this software. On the one hand, new attacks on QKD might be developed that require modifications of the software to to detect or prevent these. On the other hand, further investigations might reveal new classes of hash classes, which provide an enhanced performance. In this spirit, the software can be assumed to be subject of constant changes.

Appendix A.

Shannon Entropy

In a pioneering paper by Shannon [1948], a quantity to measure the uncertainty or information content of a defined process or message is proposed, which is nowadays referred to as *Shannon Entropy*. While this "mathematical theory of communication" was developed with regard to a classical communication system, it is also meaningful for quantum communication aspects, as the post processing requires comparable statistical analysis in means of error correction and information condensation.

This chapter is dedicated to the introduction of the central ideas. As in Shannon's original paper, which provides a more detailed insight, the quantity is established constructively. First, the demands are formulated. Afterwards the quantity to meet these requirements is presented.

Consider a random variable X of a finite set $\mathcal{X}$ with elements n elements.

$$\mathcal{X} = \{x_1, \ldots, x_n\} \tag{A.1}$$

Additionally, there is a probability distribution $\{p(x)|x \in \mathcal{X}\}$ defined over all elements in $\mathcal{X}$. To give an example, consider a fair six-sided dice. The set $\mathcal{X}$ represents the sides of the dice, which are expected to show up equally distributed.

$$\mathcal{X}_{\text{dice}} = \{1, \ldots, 6\} \quad \text{with} \quad p(x_i) = p_{\text{dice}} = \frac{1}{6}$$

In terms of data transmission and related tasks, such as information compression, information redundancy and error correction, it is useful to quantify the amount of choice in $\mathcal{X}$. Obviously, $\mathcal{X}_{\text{dice}}$ offers $n = 6$ possibilities. Any monotonic increasing function in n might be suited to represent the uncertainty or, after the dice was thrown, the information.

In classical information theory, the information is stored and processed in binary digits (*bits* or *cbits* to emphasise the classical nature). There are

$$\log_2 6 \approx 2.585\,\text{bit}$$

necessary to represent a choice between 6 different states in a binary store. With respect to the close connection to digital (binary) processing of information and the properties, which are yet to be shown, the logarithmic measure of base 2 reflects the amount of information very clearly. If 1 bit is added, the choices increase quadraticly while the (binary) information only increases by 1.

To illustrate the influence of unequally distributed elements x_i, consider a communication protocol (inspired by the short message service), capable of transmitting messages consisting of 4 small characters a-z. There are altogether $26^4 = 456\,976$ possible messages, but taking a specific language into account, a lot of messages have a probability of $p = 0$, e.g. $p(\texttt{qrxtz}) = 0$, offering the possibility to compression. For instance, it is possible to define an indexed list of all n possible messages and rely on only transmitting the position instead of the characters. Hence, the transmission costs can be lowered from initially $\log_2 456976 \approx 18.8$ to $\log_2 n$ with $n < 26^4$. Another way to illustrate the amount of information can be given on a per-character level. Consider a message starting with `aq`. There is (nearly) no choice for selecting the next characters anymore, because the `u` has a very high probability to follow after the `q` to stay within the set of words given by our language. Thus, one expects a decreasing amount of choice in case of unequally distributed characters.

The quantity to represent the amount of information/choice is expected to reflect these properties. Hence, Shannon formulated these 3 properties, which should be fulfilled by such a quantity H of a probability distribution $\{p(x)|x \in \mathcal{X}\}$:

1. H should be continuous in the $p(x)$.
2. If all the $p(x)$ are equal, $p(x) = \frac{1}{n}$, then H should be a monotonic increasing function of n. With equally likely events there is more choice, or uncertainty, when there are more possible events.
3. If a choice is broken down into two successive choices, the original H should be the weighted sum of the individual values of H.

The last demand is explained in detail in the original work. It is derived that there is only one form of H to satisfy all 3 properties.

$$H(X) = -K \sum_{x \in \mathcal{X}} p(x) \log_2 p(x) \tag{A.2}$$

K is a positive constant and can be interpreted as the information unit of measure. In this work, $K = 1$ bit is assumed, whereas the unit may be omitted to present more generic formulas. In comparison to the H-theorem of Ludwig Boltzmann (1872), H is interpreted as the *Shannon entropy*.

In fig. A.1, the Shannon entropy H is presented for the probability distribution $\{p, q\}$ with $q = 1 - p$ of a single cbit, that means, for only two possible events. If both events (bit 0 or bit 1) have the same probability $p = q = 0.5$, the Shannon entropy reaches its maximum

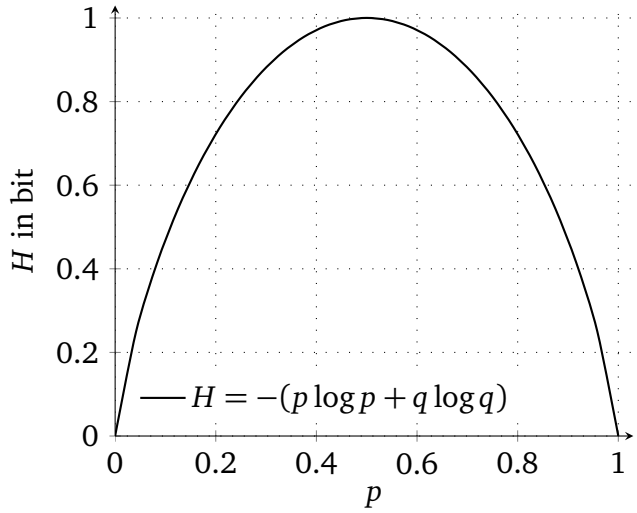


Figure A.1.: Shannon entropy H of a cbit. Shannon entropy H for the probability distribution of a cbit (set with only two events) in dependence of the bias. In agreement with our intuition, the amount of uncertainty is maximal in case of two equally likely events.

$H = 1\,\text{bit}$. If the outcome is certain, because either $p = 1$ or $q = 1$, the Shannon entropy gets minimal $H = 0$.

It is important to note that H is defined to be zero at minimum or positive. Next, the 2. demand on the Shannon entropy is investigated. Considering a probability distribution $\{p, p, \dots\}$ with $n \in \mathbb{N}^+$ elements of a probability $p = \frac{1}{n}$, the Shannon entropy H in (A.2) can be formulated in dependence of n only.

$$H(X) = K \log_2(n) \tag{A.3}$$

This result was already found and confirms the suitability of H.

Appendix B.

Universal Hash Functions

In modern communication application, it is often required to reduce the information using specific compression functions. For instance, it may be desirable to check the integrity of a transmitted data file. One possible, though inefficient, method is to transfer this file three times and run a bit per bit comparison. In case of errors, the majority vote for each bit would determine the final result. Of course, this procedure decreases the data bit rate dramatically. Furthermore, this method is very inflexible, because the security can only be influenced step-wise by increasing the repetition number.

It is more convenient to calculate a checksum from such a file, whereby the size of this checksum is a more accurately adjustable, but still step-wise parameter. This way, the probability to detect errors can be tuned to the expectation error rate of the specific case. Whereas the method presented above is not only detecting transmission errors, but is also able to correct errors without any further communication, this ability cannot be provided anymore if the checksum size is shorter than a characteristic minimum.

For the application within QKD, e.g. privacy amplification (sec. 3.4) and authentication (sec. 3.5), it is interesting to use compression functions with an output size much shorter than the input size. One reason is that today's quantum channel transmission bit rates are rather small and that's why it is preferable to save transfer volume by putting more effort in postprocessing, i.e. computing. Furthermore, the whole postprocessing communication needs to be authenticated. Authentication requires a secret key as well. Its size increases proportionally with the size of transferred data when equal security should be provided. In the worst case, the size of the secret key for the post processing is out-weighting the outcome of secret key. QKD is not possible anymore.

Wegman and Carter [1981] were the first presenting methods for data validation and authentication, which are not only computational secure, like e.g. RSA [Rivest et al., 1978], but also provide provable security, as it is often demanded in QKD applications.

Consider a domain $\mathbb{M}$ of messages and a second domain $\mathbb{T}$ of tokens. $\mathbb{T}$ could also be interpreted as a set of checksums, abbreviations or dictionary labels. The number of

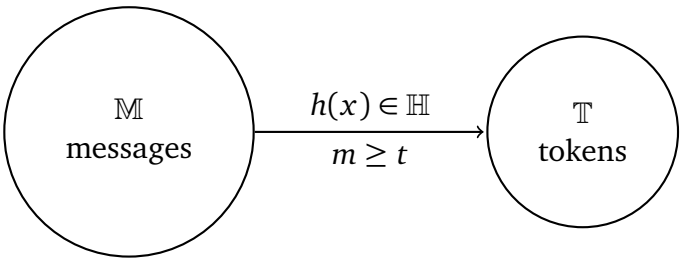


Figure B.1.: Hash function

elements in these domains is denoted by two bars.

$$m = |\mathbb{M}| \tag{B.1}$$

$$t = |\mathbb{T}| \tag{B.2}$$

Due to the need of compression, we assume furthermore, that $t \leq m$ is given. A *hash function* is defined as a function $h : \mathbb{M} \to \mathbb{T}$, which provides a mapping from a larger domain to a smaller one, as shown in fig. B.1. All hash functions h mapping between these domains $\mathbb{M}$ and $\mathbb{T}$ constitute the set $\mathbb{H}$.

If $t < m$, it is obvious that tokens are not unique. So there have to be messages $M_1, M_2, \ldots$ which share the same token T. If two messages $x \neq y \in \mathbb{M}$ have the same token $h(x) = h(y)$, it is called a *collision*. In the case of data validation, collisions are unwanted because it might be implied, that both messages are the same, because their token is equal, but in fact the messages are different. In matters of authentication, collisions are considered to lower the security because a potential man-in-the-middle could manipulate the message in a certain manner so that a collision is caused. The manipulation could remain undetected as the authentication token is still the same.

The technique of *universal hashing* allows to reduce the probability of a collision between two randomly selected messages M_1 and M_2 by uniformly distributing the chance of collision. This is accomplished independently of the input distribution, not by relying only on one hash function, but instead choosing a new hash function each time from a so-called universal$_2$ class.

The following formulas and definitions were proposed by Kollmitzer and Pivk [2010]. For a better mathematical description of the probability of collisions a function $\delta(x_1, x_2)$ with $x_1, x_2 \in \mathbb{M}$ is introduced.

$$\delta_h(x_1, x_2) = \begin{cases} 1, & \text{if } h(x_1) = h(x_2) \\ 0, & \text{if } h(x_1) \neq h(x_2) \end{cases} \tag{B.3}$$

Furthermore $\delta_{\mathbb{H}}(x_1, x_2)$ is representing the total number of collisons over all hash functions in $\mathbb{H}$.

$$\delta_{\mathbb{H}}(x_1, x_2) = \sum_{h \in \mathbb{H}} \delta_h(x_1, x_2) \tag{B.4}$$

Using (B.4), the collisions probability for two distinct messages x_1 and x_2 can be expressed by $\delta_{\mathbb{H}}(x_1, x_2)/|\mathbb{H}|$. Following the idea of a uniform distribution of collisions, this expression has to be upper bounded.

$\mathbb{H}$ is called *ϵ-almost universal$_2$* if

$$\forall(\epsilon \geq 1/t) \in \mathbb{R}, \forall(x_1 \neq x_2) \in \mathbb{M} : \delta_{\mathbb{H}}(x, y) \ \leq \ \epsilon|\mathbb{H}|. \tag{B.5}$$

The minimum of ϵ is given by $1/t$. In the specific case of $\epsilon = 1/t$, $\mathbb{H}$ is called a *universal$_2$* class.

$\mathbb{H}$ is *ϵ-almost strongly universal$_2$* if

a) $\forall x_1 \in \mathbb{M}, \forall y_1 \in \mathbb{T} : |\{h \in \mathbb{H} : h(x_1) = y_1\}| = |\mathbb{H}|/|\mathbb{T}|$ and

b) $\forall(\epsilon \geq 1/t), \forall(x_1 \neq x_2) \in \mathbb{M}, \forall(y_1, y_2) \in \mathbb{T} :$
$|\{h \in \mathbb{H} : h(x_1) = y_1, h(x_2) = y_2\} \leq \epsilon|\mathbb{H}|/|\mathbb{T}|.$

Like above, $\mathbb{H}$ is called *strongly universal$_2$* in case of $\epsilon = 1/t$.

The first condition requires an equal distribution of all tokens in the class $\mathbb{H}$. This means that each token has a probability of $1/t$. For a potential forger, it is not possible to find the token to a known message by an educated guess without knowledge of the random hash function choice from $\mathbb{H}$. The chance of finding a suitable token by accident can be lowered by increasing the token size and thus decreasing $1/t$.

The condition b) formulates a statement concerning the probability of finding a suitable token t_2 for a message m_2, given that a message-token pair m_1 and t_1 is already known. This probability is the interesting property for a potential forger, who might want to run a man-in-the-middle attack. For this, the original message and authentication token is known to him. The ϵ in the second condition describes the chance of finding a valid token for a manipulated message.

The lower index in the attribute *universal$_2$* emphasises that these functions provide provable security even for the second token assigning. This is not granted anymore when the forger already knows two message-token pairs and tries to figure out a third one. Hence, the hash function needs to randomly selected again right after one pair is exposed to the public communication channel.

The practical application of hash functions is explained in sec. 3.5.

Appendix C.

Source Code

Listing C.1: qkdprocessor.h

```
#ifndef QKDPROCESSOR_H
#define QKDPROCESSOR_H

#include <QObject>
#include <QVariant>
#include <QImage>

#include "measurement.h"

typedef quint32 Index;
typedef qint64 SIndex;
typedef QList<Index> IndexList;
typedef QPair<Index,bool> IndexBoolPair;
typedef QList<IndexBoolPair> IndexBoolPairList;
typedef QList<bool> BoolList;

class QKDProcessor : public QObject
{
    Q_OBJECT

public:
    explicit QKDProcessor(QObject *parent = 0);
    ~QKDProcessor();

    enum PackageType {
        PT01sendReceivedList = 50,
        PT01sendRemainingList,
        PT02errorEstimationSendSample,
        PT02errorEstimationReport,
        PT03prepareBlockParities,
        PT04reportBlockParities,
        PT05startBinary,
        PT06finished,
        PT07evaluation
    };

private:
    quint16 calculateInitialBlockSize(qreal errorProbability);
    bool calculateParity(const Measurements measurements, const Index index, const quint16 &size) const;
    inline IndexList getOrderedList(Index range);
    IndexList getRandomList(Index range);
    static Measurements reorderMeasurements(const Measurements measurements, const IndexList order);
    void clearMeasurements();
    static QByteArray privacyAmplification(const Measurements measurements, const qreal ratio);

    Measurements* measurements;
    bool isMaster;
```

```
    const static quint8 runCount = 4;
    const static qreal errorEstimationSampleRatio = 0.02;
    IndexBoolPairList list;
    IndexList remainingList;
    Index errorEstimationSampleSize;
    BoolList boolList;
    Index errorCounter;
    qreal error;
    quint16 k1; // initial block size
    BoolList parities;
    IndexList corruptBlocks;
    QList<Measurements> reorderedMeasurements;
    quint16 blockSize;
    quint16 binaryBlockSize;
    Index blockCount;
    quint8 runIndex;
    Index transferedBitsCounter; // number of bits known to eve
    static const int idIndexList;

signals:
    void sendData(quint8 type, QVariant data = QVariant());
    void logMessage(QString entry, Qt::GlobalColor backgroundColor = Qt::white);
    void finished();
    void imageGenerated(QImage image);

public slots:
    void start();
    void incomingData(quint8 type, QVariant data = QVariant());

    void setMeasurements(Measurements* measurements);
    void setMaster(bool isMaster);
};

Q_DECLARE_METATYPE(IndexList)
Q_DECLARE_METATYPE(IndexBoolPair)
Q_DECLARE_METATYPE(IndexBoolPairList)
Q_DECLARE_METATYPE(BoolList)
Q_DECLARE_METATYPE(Qt::GlobalColor)

#endif // QKDPROCESSOR_H
```

Listing C.2: qkdprocessor.cpp

```
#include "qkdprocessor.h"

#include <qmath.h>
#include <qdebug.h>
#include <QTime>
#include <QRgb>
#include <QFile>
#include <limits>
#include <algorithm>
using std::max;
using std::random_shuffle;

const int QKDProcessor::idIndexList = qRegisterMetaType<IndexList>();

QKDProcessor::QKDProcessor(QObject *parent) :
    QObject(parent), measurements(0), isMaster(false)
{
    // make sure tu initialize the pseudo RNG for std::random_shuffle
#ifdef QT_NO_DEBUG
    qsrand(QTime::currentTime().msec()); srand(QTime::currentTime().msec()+1);
#else
    qsrand(0); srand(0); // make the class deterministic for debugging purposes
#endif
```

```
    qRegisterMetaTypeStreamOperators<IndexList>();

    qRegisterMetaType<IndexBoolPair>();
    qRegisterMetaTypeStreamOperators<IndexBoolPair>();

    qRegisterMetaType<IndexBoolPairList>();
    qRegisterMetaTypeStreamOperators<IndexBoolPairList>();

    qRegisterMetaType<BoolList>();
    qRegisterMetaTypeStreamOperators<BoolList>();
}

void QKDProcessor::clearMeasurements()
{
    if(measurements) {
        qDeleteAll(*measurements);
        delete measurements;
    }
}

QByteArray QKDProcessor::privacyAmplification(const Measurements measurements, const qreal ratio)
{
    Q_ASSERT(ratio > 0);
    /* - count of bits to represent integer: N=ceil(log2(int))
     * - count of bits to represent squared integer: 2N
     * - must have double size of bufferTypeSmall to prevent integer overflow
     * - in any case only half of the bits of squared integer is used, so we can
     *   accept integer overflow and use the same bit size for bufferType, too.
     * - bigger types are better, because precision increasing in:
     *   floor(bitLimitSmall*ratio). step size: 100/64 => 1.56%
     */
    typedef quint64 bufferType;
    typedef quint64 bufferTypeSmall;

    static const quint8 bitLimitSmall = sizeof(bufferTypeSmall)*8;

    bufferTypeSmall bufferBits = 0;
    bufferTypeSmall bufferBase = 1; // bufferBase must be odd (-> hash functions)

    QByteArray finalKey;
    // only cstrings and char can be added to QByteArrays: we choose char
    typedef char finalKeyBufferType;

    finalKeyBufferType buffer = 0;
    const quint8 bufferSize = sizeof(finalKeyBufferType)*8;
    const quint8 bitCount = qFloor(bitLimitSmall*ratio);

    quint8 pos = 0;
    quint8 bufferPos = 0;
    const bufferType bufferType_1 = Q_UINT64_C(1);
    foreach(Measurement *measurement, measurements) {
        bufferBase |= (bufferTypeSmall)measurement->base << pos;
        bufferBits |= (bufferTypeSmall)measurement->bit  << pos;
        pos++;
        if(pos == bitLimitSmall) {
            pos = 0;
            bufferType temp = (bufferType)bufferBase*(bufferType)bufferBits;
            for(quint8 bitPos = 0; bitPos < bitCount; bitPos++) {
                // http://stackoverflow.com/a/2249738/1407622
                buffer |= ((temp & ( bufferType_1 << bitPos )) >> bitPos) << bufferPos;
                bufferPos++;
                if(bufferPos == bufferSize) {
                    bufferPos = 0;
                    finalKey.append(buffer);
                    buffer = 0;
```

```
                }
            }
            bufferBase = 1;
            bufferBits = 0;
        }
    }
    // there might be fewer bits in finalKey than ratio*measurements->size()
    // this is due remaining (unused) bits in buffers
    return finalKey;
}

void QKDProcessor::setMeasurements(Measurements *measurements)
{
    clearMeasurements();
    this->measurements = measurements;
}

void QKDProcessor::setMaster(bool isMaster)
{
    this->isMaster = isMaster;
}

QKDProcessor::~QKDProcessor()
{
    clearMeasurements();
}

quint16 QKDProcessor::calculateInitialBlockSize(qreal errorProbability)
{
    // values follow the proposal of Brassard and Savail(1994), but are
    // aligned to the power of 2 which is necessary for the binary scheme
    if(errorProbability < 0.02) {
        return 64;
    } else if(errorProbability < 0.05) {
        return 16;
    } else if(errorProbability < 0.1) {
        return 8;
    } else if(errorProbability < 0.15) {
        return 4;
    } else {
        emit logMessage(QString("EE: errorEstimation exceeded secure limit!"));
        return 4;
    }
}

bool QKDProcessor::calculateParity(const Measurements measurements,
                                   const Index index, const quint16 &size) const
{
    Q_ASSERT(size > 0);
    const Index end = index + size;
    Q_ASSERT((SIndex)end <= measurements.size());
    bool parity = false;
    for(Index i = index; i < end; i++) {
        parity = parity ^ measurements.at(i)->bit;
    }
    return parity;
}

IndexList QKDProcessor::getOrderedList(Index range)
{
    IndexList orderedList;
    for(Index i = 0; i < range; i++)
        orderedList.append(i);
    return orderedList;
}

```

```
IndexList QKDProcessor::getRandomList(Index range)
{
    IndexList list = getOrderedList(range);
    // http://www.cplusplus.com/reference/algorithm/random_shuffle/
    random_shuffle(list.begin(), list.end());
    return list;
}

Measurements QKDProcessor::reorderMeasurements(const Measurements measurements, const IndexList order)
{
    Measurements list;
    foreach(Index index, order) {
        list.append(measurements.at(index));
    }

    return list;
}

void QKDProcessor::incomingData(quint8 type, QVariant data)
{

    switch((PackageType)type) {
    // Sifting Procedure
    case PT01sendReceivedList: {
        emit logMessage(QString("Sifting Procedure started"), Qt::green);
        emit logMessage(QString("#01: File contains %1 measurements").
                        arg(measurements->size()));
        if(isMaster) {
            list = data.value<IndexBoolPairList>();
        } else {
            list.clear();
            Q_ASSERT((SIndex)std::numeric_limits<Index>::max() >= measurements->size());
            for(int index = 0; index < measurements->size(); index++) {
                if(measurements->at(index)->valid)
                    list.append(IndexBoolPair(index,measurements->at(index)->base));
            }
            emit sendData(PT01sendReceivedList, QVariant::fromValue(list));
        }
        emit logMessage(QString("#01: Valid measurements (containing 1 received photons): %1 (%2%)").
                        arg(list.size()).arg((double)list.size()*100/measurements->size()));

        if(isMaster) {
            IndexBoolPair pair;
            remainingList.clear();
            BoolList basesList;
            foreach(pair, list) {
                const bool &ownBase = measurements->at(pair.first)->base;
                basesList.append(ownBase);
                if(ownBase == pair.second)
                    remainingList.append(pair.first);
            }
            emit sendData(PT01sendRemainingList,
                          QVariant::fromValue(basesList));
        }

        reorderedMeasurements.clear();

        return;
    }
    case PT01sendRemainingList: {
        if(isMaster) {
        } else {
            BoolList basesList = data.value<BoolList>();
            Q_ASSERT(list.size() == basesList.size());
            remainingList.clear();
```

```
            for(SIndex index = 0; index < list.size(); index++) {
                const IndexBoolPair &pair = list.at(index);
                if(basesList.at(index) == pair.second)
                    remainingList.append(pair.first);
            }
            emit sendData(PT01sendRemainingList);
        }
        emit logMessage(QString("#02: Remaining measurements after sifting (same base): %1 (%2%)").
                        arg(remainingList.size()).
                        arg((double)remainingList.size()*100/list.size()));
        {
            Measurements siftedMeasurements;
            foreach(Index index, remainingList) {
                siftedMeasurements.append(measurements->at(index));
            }
            reorderedMeasurements.append(siftedMeasurements);

        }

        {
            int bit = 0;
            int base = 0;
            Measurements measurements = reorderedMeasurements.first();
            foreach(Measurement *measurement, measurements) {
                bit += measurement->bit;
                base += measurement->base;
            }

            emit logMessage(QString("Stats ratio: bit = %1%, base = %2%").
                            arg(100.0*bit/measurements.size()).
                            arg(100.0*base/measurements.size()));
        }

        list.clear();
        remainingList.clear();
        emit logMessage(QString("Sifting Procedure finished"), Qt::green);

        if(isMaster) {
            IndexList order = getRandomList(reorderedMeasurements.first().size());
            reorderedMeasurements.replace(0,reorderMeasurements(reorderedMeasurements.last(),
                                                                order));
            emit sendData(PT02errorEstimationSendSample,
                          QVariant::fromValue<IndexList>(order));
        }
        return;
    }
    // Error Estimation
    case PT02errorEstimationSendSample: {
        emit logMessage(QString("Error Estimation started"), Qt::green);
        if(!isMaster) {
            IndexList order = data.value<IndexList>();
            reorderedMeasurements.replace(0,reorderMeasurements(reorderedMeasurements.last(),
                                                                order));
        }
        Q_ASSERT(reorderedMeasurements.size() == 1);
        Q_ASSERT(reorderedMeasurements.first().size() > 1);
        errorEstimationSampleSize = qCeil(reorderedMeasurements.first().size()*
                                          errorEstimationSampleRatio);

        boolList.clear();
        for(Index i = 0; i < errorEstimationSampleSize; i++) {
            /*
             * takeLast() removes elements from the list
             * new orderings will take the size of the last ordering into account
             */
            boolList.append(reorderedMeasurements.first().takeLast()->bit);
```

```
        }

        if(!isMaster) {
            emit sendData(PT02errorEstimationSendSample,
                          QVariant::fromValue(boolList));
        } else {
            BoolList compareBoolList = data.value<BoolList>();
            int size = boolList.size();
            Q_ASSERT(size == compareBoolList.size());
            errorCounter = 0;
            for(int i = 0; i < size; i++) {
                if(boolList.at(i) != compareBoolList.at(i))
                    errorCounter++;
            }
            emit sendData(PT02errorEstimationReport,
                          QVariant::fromValue<Index>(errorCounter));
        }
        return;
    }
    case PT02errorEstimationReport: {
        if(!isMaster) {
            errorCounter = data.value<Index>();
            emit sendData(PT02errorEstimationReport);
        } else {
        }
        error = (double)errorCounter/boolList.size();
        emit logMessage(QString("#01: Estimated error rate: %1 / %2 (%3%)").
                        arg(errorCounter).arg(boolList.size()).arg(error*100));
        boolList.clear();
        k1 = calculateInitialBlockSize(error);
        emit logMessage(QString("#02: resulting block size (k1): %1").arg(k1));
        emit logMessage(QString("Error Estimation finished"), Qt::green);

        emit logMessage("Initial Parity Check started", Qt::green);
        Index maximumBlockSize = k1*pow(2,runCount-1);
        Index removeBits = reorderedMeasurements.first().size() % maximumBlockSize;
        Measurements::iterator end = reorderedMeasurements.first().end();
        reorderedMeasurements.first().erase(end-removeBits,end);
        Q_ASSERT(reorderedMeasurements.first().size()/(SIndex)maximumBlockSize > 0);

        runIndex = 0;
        transferedBitsCounter = 0;
        this->incomingData(PT03prepareBlockParities, (uint)runIndex);

        return;
    }

    case PT03prepareBlockParities: {
        if(data.userType() == idIndexList) {
            reorderedMeasurements.append(reorderMeasurements(
                reorderedMeasurements.last(), data.value<IndexList>()));
            runIndex = reorderedMeasurements.size() - 1;

        } else if(data.type() == (QVariant::Type)QMetaType::UInt) {
            runIndex = data.toUInt();
        }
        Q_ASSERT(runIndex < reorderedMeasurements.size());

        blockSize = k1*pow(2,runIndex);
        binaryBlockSize = blockSize;
        blockCount = reorderedMeasurements.at(runIndex).size()/blockSize;
        emit logMessage(QString("in PT03prepareBlockParities, reorderedMeasurements.size = %1,"
                                "runIndex = %2, blockSize = %3").
                        arg(reorderedMeasurements.size()).arg(runIndex).arg(blockSize));
        parities.clear();
        Index lastIndex = reorderedMeasurements.at(runIndex).size() - blockSize;
```

```
        for(Index index = 0; index <= lastIndex; index += blockSize) {
            parities.append(calculateParity(reorderedMeasurements.at(runIndex),
                                            index, blockSize));
        }

        if(isMaster) {
            emit sendData(PT04reportBlockParities,
                          QVariant::fromValue<BoolList>(parities));
        }

        return;

    }
    case PT04reportBlockParities: {
        if(!isMaster) {
            qint64 size = parities.size();
            Q_ASSERT(size > 0);
            BoolList compareParities = data.value<BoolList>();
            Q_ASSERT(compareParities.size() == size);
            corruptBlocks.clear();
            for(Index index = 0; index < size; index++) {
                if(compareParities.at(index) != parities.at(index))
                    corruptBlocks.append(index*blockSize);
            }
            emit sendData(PT04reportBlockParities,
                          QVariant::fromValue<IndexList>(corruptBlocks));
        } else {
            corruptBlocks = data.value<IndexList>();
        }
        transferedBitsCounter += corruptBlocks.size();
        if(blockSize == binaryBlockSize) {
            emit logMessage(QString("number of corrupt Blocks: %1/%2 (%3%)").
                            arg(corruptBlocks.size()).
                            arg(blockCount).
                            arg((double)corruptBlocks.size()/blockCount*100));
        }

        if(isMaster) {
            if(corruptBlocks.empty()) {
                runIndex++;
                // startBinary finished and there is another run
                // to return to (orders.size > 0)
                if((binaryBlockSize == 1) && (reorderedMeasurements.size() > 1) &&
                   (runIndex == reorderedMeasurements.size())) {
                    runIndex = 0;
                }
                if(runIndex < reorderedMeasurements.size()) {
                    emit sendData(PT03prepareBlockParities, (uint)runIndex);
                    this->incomingData(PT03prepareBlockParities);
                } else if(reorderedMeasurements.size() < runCount) {
                    QVariant data = QVariant::fromValue<IndexList>(
                        getRandomList(reorderedMeasurements.last().size()));
                    emit sendData(PT03prepareBlockParities, data);
                    this->incomingData(PT03prepareBlockParities, data);
                } else {
                    emit sendData(PT06finished);
                }
            } else { // start Binary
                this->incomingData(PT05startBinary);
            }
        }
        return;
    }

    case PT05startBinary: {
        binaryBlockSize = binaryBlockSize/2;
```

```
        parities.clear();
        if(isMaster || (!isMaster && binaryBlockSize > 1)) {
            foreach(Index index, corruptBlocks) {
                parities.append(calculateParity(reorderedMeasurements.at(runIndex),
                                                index, binaryBlockSize));
            }
        }

        if(isMaster) {
            emit sendData(PT05startBinary,
                          QVariant::fromValue<BoolList>(parities));
        } else {
            BoolList compareParities = data.value<BoolList>();
            Index size = compareParities.size();
            Q_ASSERT(corruptBlocks.size() == (SIndex)size);
            if(binaryBlockSize > 1)  {
                Q_ASSERT(parities.size() == (SIndex)size);
                for(Index i = 0; i < size; i++) {
                    if(compareParities.at(i) == parities.at(i)) {
                        // the 2nd half of the block must me corrupt
                        corruptBlocks.replace(i,corruptBlocks.at(i)+binaryBlockSize);
                    }
                }
            } else { // binaryBlockSize == 1: just toggle wrong bits
                for(Index i = 0; i < size; i++) {
                    // position in reorderedMeasurements: bit1 | bit2
                    // compareParities contains bit1 from Alice (master)
                    bool &bit1 = reorderedMeasurements.at(runIndex).
                            at(corruptBlocks.at(i))->bit;
                    bool &bit2 = reorderedMeasurements.at(runIndex).
                            at(corruptBlocks.at(i)+1)->bit;
                    if(compareParities.at(i) == bit1) {
                        bit2 = !bit2;
                        // corruptBlocks.replace(i, corruptBlocks.at(i)+1);
                    }
                    bit1 = compareParities.at(i);
                }
                corruptBlocks.clear();

            }
        }
        if(!isMaster) {
            emit sendData(PT04reportBlockParities,
                          QVariant::fromValue<IndexList>(corruptBlocks));
            transferedBitsCounter += corruptBlocks.size();
        }
        return;
    }
    case PT06finished: {
        if(!isMaster) {
            emit sendData(PT06finished);
        } else {
            this->sendData(PT07evaluation);
        }
        transferedBitsCounter += reorderedMeasurements.last().size()/
                k1*(2-pow(2,1-runCount));
        emit logMessage(QString("bitCounter: %1 (%2%)").arg(transferedBitsCounter).
                        arg(100.0*transferedBitsCounter/reorderedMeasurements.last().size()));

        // An increasing security parameter s lowers the upper bound of Eve's
        // information on the key I <= 2^(-s)/ln(2); 2^(-7)/ln(2) = 0.01 bits
        const quint8 securityParameter = 7;
        qreal removeRatio = error*2
                        + (qreal)(transferedBitsCounter+securityParameter)/
                                  reorderedMeasurements.last().size();
        if(!(removeRatio < 1.0)) {
```

```
            emit logMessage(QString("too much information revealed for privacy-amplification: %1%").
                            arg(removeRatio*100), Qt::red);
            return;
        }

        QByteArray finalKey = privacyAmplification(reorderedMeasurements.last(), 1-removeRatio);
        emit logMessage(QString("finished! (%1 byte)").arg(finalKey.size()));
        {
            QFile file(QString("outfile_%1.dat").arg(isMaster ? "alice" : "bob"));
            file.open(QIODevice::WriteOnly);
            QDataStream out(&file);
            out << finalKey;
            file.close();
        }
        int imageSize = qFloor(qSqrt(finalKey.size()/3));
        QImage image(imageSize, imageSize, QImage::Format_RGB888);
        int pos = 0;
        for(int x = 0; x < imageSize; x++) {
            for(int y = 0; y < imageSize; y++) {
                image.setPixel(x,y, qRgb(finalKey.at(pos),
                                         finalKey.at(pos+1),
                                         finalKey.at(pos+2)));
                pos+=3;
            }
        }
        image.save(QString("outfile_%1.png").arg(isMaster ? "alice" : "bob"));
        emit imageGenerated(image);
        return;
    }

    case PT07evaluation: {
        parities.clear();
        Index size = reorderedMeasurements.last().size();
        for(Index index = 0; index < size; index++)
            parities.append(calculateParity(reorderedMeasurements.last(), index, 1));
        if(!isMaster) {
            emit sendData(PT07evaluation, QVariant::fromValue<BoolList>(parities));
        } else {
            BoolList compareParities = data.value<BoolList>();
            Q_ASSERT(compareParities.size() ==(SIndex)size);
            Index errors = 0;
            for(Index index = 0; index < size; index++) {
                if(parities.at(index) != compareParities.at(index))
                    errors++;
            }
            emit logMessage(QString("Ergebnis: %1 (%2%) Bit-Fehler").arg(errors).
                            arg(100.0*errors/size));
        }

        emit finished();

        return;
    }

    }
}

void QKDProcessor::start()
{
    if(isMaster) { // this is Alice
        // we kindly ask Bob for a list a received bits in first range from 0-quint32
        emit sendData(PT01sendReceivedList);
    } // Bob doesn't do anything here
}
```

Acronyms

APD	avalanche photodiode
BER	bit error rate
BSC	binary symmetric channel
EOM	electro-optical modulator
FPGA	field-programmable gate array
GUI	graphical user interface
LAN	local area network
OOP	object-oriented programming
PBS	polarizing beam splitter
PH	pinhole
QBER	quantum bit error rate
QKD	quantum key distribution
QRNG	quantum random number generator
RNG	random number generator
VPN	virtual private network
WCP	weak coherent puls

List of Figures

Bibliography

C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, page 175, India, 1984. doi:10.1016/j.tcs.2011.08.039.

C. H. Bennett, G. Brassard, and J.M. Robert. Privacy amplification by public discussion. *SIAM journal on Computing*, 17:210, 1988.

C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. Smolin. Experimental quantum cryptography. *Journal of cryptology*, 5(1):3–28, 1992.

C. H. Bennett, G. Brassard, C. Crépeau, and U.M. Maurer. Generalized privacy amplification. *Information Theory, IEEE Transactions on*, 41(6):1915–1923, 1995.

G. Brassard and L. Salvail. Secret-key reconciliation by public discussion. In *advances in Cryptology EUROCRYPT '93*, pages 410–423. Springer, 1994.

Gilles Brassard. Brief History of Quantum Cryptography: A Personal Perspective. *Proceedings of IEEE Information Theory Workshop on Theory and Practice in Information Theoretic Security, Awaji Island, Japan, pp. 19-23, October 2005*, April 2006. URL `http://arxiv.org/abs/quant-ph/0604072v1`.

Robert Hanbury Brown and Richard Q. Twiss. Correlation between Photons in two Coherent Beams of Light. *Nature*, 177(4497):27–29, 1956. doi:10.1038/177027a0.

J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (Extended Abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM. doi:10.1145/800105.803400.

Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19–51, 1997. ISSN 0196-6774. doi:10.1006/jagm.1997.0873.

Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.

A. K. Ekert. Quantum cryptography based on Bell's theorem. *Physical review letters*, 67(6): 661–663, 1991.

Chris Erven, Xiongfeng Ma, Raymond Laflamme, and Gregor Weihs. Entangled quantum key distribution with a biased basis choice. *New Journal of Physics*, 11(4):045025, 2009. doi:10.1088/1367-2630.

A. Galindo and M. A. Martin-Delgado. Information and Computation: Classical and Quantum Aspects. *Rev.Mod.Phys.74:347-423,2002*, December 2001. doi:10.1103/RevModPhys.74.347.

Ilja Gerhardt, Qin Liu, Antia Lamas-Linares, Johannes Skaar, Christian Kurtsiefer, and Vadim Makarov. Full-field implementation of a perfect eavesdropper on a quantum cryptography system. *Nat Commun*, 2:349, jun 2011. doi:10.1038/ncomms1348. URL `http://arxiv.org/abs/1011.0105`.

Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. Quantum Cryptography. *Rev.Mod.Phys.74:145-195,2002*, September 2001. doi:10.1103/RevModPhys.74.145.

Daniel Gottesman, Hoi-Kwong Lo, Norbert Lütkenhaus, and John Preskill. Security of Quantum Key Distribution with Imperfect Devices. *Quantum Info. Comput.*, 4(5):325–360, September 2004. ISSN 1533-7146.

Oliver Kern and Joseph M. Renes. Improved one-way rates for BB84 and 6-state protocols. *Quantum Information & Computation*, 8(8):756–772, September 2008. ISSN 1533-7146. URL `http://arxiv.org/abs/0712.1494`.

Christian Kollmitzer and Mario Pivk. *Applied Quantum Cryptography*, volume 797 of *Lecture Notes in Physics*. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-04831-9.

Christian Kurtsiefer, Sonja Mayer, Patrick Zarda, and Harald Weinfurter. Stable Solid-State Source of Single Photons. *Phys. Rev. Lett.*, 85:290–293, July 2000. doi:10.1103/PhysRevLett.85.290.

Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007. ISSN 0098-3500. doi:10.1145/1268776.1268777.

Chao-Yang Lu, Daniel E. Browne, Tao Yang, and Jian-Wei Pan. Demonstration of Shor's quantum factoring algorithm using photonic qubits. *Phys. Rev. Lett. 99, 250504 (2007)*, December 2007. doi:10.1103/PhysRevLett.99.250504.

Erik Lucero, Rami Barends, Yu Chen, Julian Kelly, Matteo Mariantoni, Anthony Megrant, Peter O'Malley, Daniel Sank, Amit Vainsencher, James Wenner, Ted White, Yi Yin, Andrew N. Cleland, and John M. Martinis. Computing prime factors with a Josephson phase qubit quantum processor, February 2012. URL `http://arxiv.org/abs/1202.5707v1`.

Ivan Marcikic, Antia Lamas-Linares, and Christian Kurtsiefer. Free-space quantum key distribution with entangled photons. *Appl. Phys. Lett. 89, 101122 (2006)*, August 2006. doi:10.1063/1.2348775.

Elke Neu, David Steinmetz, Janine Riedrich-Möller, Stefan Gsell, Martin Fischer, Matthias Schreck, and Christoph Becher. Single photon emission from silicon-vacancy colour centres in chemical vapour deposition nano-diamonds on iridium. *New Journal of Physics*, 13(2):025012, 2011. doi:10.1088/1367-2630/13/2/025012.

openSUSE Project. General purpose GNU/Linux operating system, December 2012. URL `http://www.opensuse.org`.

A. Poppe, A. Fedrizzi, T. Loruenser, O. Maurhardt, R. Ursin, H. R. Boehm, M. Peev, M. Suda, C. Kurtsiefer, H. Weinfurter, T. Jennewein, and A. Zeilinger. Practical Quantum Key Distribution with Polarization-Entangled Photons. *Opt. Express 12, 3865-3871 (2004)*, April 2004. doi:10.1364/OPEX.12.003865.

Qt Project. Qt 4 cross-platform application and UI framework for developers using C++, December 2012. URL `http://qt-project.org`.

Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. doi:10.1145/357980.358017.

Tim Schröder, Friedemann Gädeke, Moritz Julian Banholzer, and Oliver Benson. Ultrabright and efficient single-photon generation based on nitrogen-vacancy centres in nanodiamonds on a solid immersion lens. *New Journal of Physics*, 13(5):055017, 2011. doi:10.1088/1367-2630/13/5/055017.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, October 1948. URL `http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`.

Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28:656–715, October 1949.

Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J.Sci.Statist.Comput. 26 (1997) 1484*, January 1996. URL `http://arxiv.org/abs/quant-ph/9508027v2`.

Peter W. Shor and John Preskill. Simple Proof of Security of the BB84 Quantum Key Distribution Protocol. *Phys. Rev. Lett.*, 85:441–444, July 2000. doi:10.1103/PhysRevLett.85.441.

Feng Tang, Shuang Gao, Xiaofei Wang, and Bing Zhu. A novel synchronization scheme for free-space quantum key distribution system. In *Communications and Photonics Conference and Exhibition*, pages 1–7, nov. 2011. doi:10.1117/12.917952.

P. D. Townsend, J. G. Rarity, and P. R. Tapster. Single photon interference in 10 km long optical fibre interferometer. *Electronics Letters*, 29(7):634–635, April 1993. ISSN 0013-5194. doi:10.1049/el:19930424.

Artem Vakhitov, Vadim Makarov, and Dag R. Hjelme. Large pulse attack as a method of conventional optical eavesdropping in quantum cryptography. *Journal of Modern Optics*, 48(13):2023–2038, 2001. doi:10.1080/09500340108240904.

Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature 414, 883-887 (20/27 Dec 2001)*,

December 2001. doi:10.1038/414883a. URL `http://arxiv.org/abs/quant-ph/0112176v1`.

G. Vernam. Cipher printing telegraph system for secret wire and radio telegraphic communications. *Journal of American Institute of Electrical Engineers*, 45:109–115, 1926.

Michael Wahl, Matthias Leifgen, Michael Berlin, Tino Rohlicke, Hans-Jurgen Rahn, and Oliver Benson. An ultrafast quantum random number generator with provably bounded output bias based on photon arrival time measurements. *Applied Physics Letters*, 98(17): 171105, 2011. doi:10.1063/1.3578456.

Robert H. Webb. Confocal optical microscopy. *Reports on Progress in Physics*, 59(3):427, 1996. doi:10.1088/0034-4885.

Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. ISSN 0022-0000. doi:10.1016/0022-0000(81)90033-7.

Stephen Wiesner. Conjugate coding. *SIGACT News*, 15(1):78–88, January 1983. ISSN 0163-5700. doi:10.1145/1008908.1008920. (Manuscript *circa* 1970.).

W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886): 802–803, October 1982. doi:10.1038/299802a0.

Acknowledgements

I would like to thank Prof. Dr. Oliver Benson for giving me the opportunity to work on such an interesting project. During the whole process of preparing and writing this thesis, I was supported by Matthias Leifgen, whom I owe a debt of gratitude.

Furthermore, the work on our project was supported by Friedemann Gädeke, Tim Schröder, Valentin Métillon and Marcel Granetzny. Without them, I would still waiting in the lab for finding some single photons.

At last, I want to thank Anne Mackinney for her valuable help to improve style and orthography.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Robert Riemann	Berlin, den 1. Februar 2013
	Ort, Datum