
TECHNISCHE
UNIVERSITÄT
DARMSTADT

ULB

Software bundle for data post-processing in a quantum key distribution experiment

Notz, Pascal Markus; Nikiforov, Oleg; Walther, Thomas

(2020)

DOI (TUprints): https://doi.org/10.25534/tuprints-00014042

Lizenz: 

CC-BY 4.0 International - Creative Commons, Namensnennung

Publikationstyp: Report

Fachbereich: 05 Fachbereich Physik

Quelle des Originals: https://tuprints.ulb.tu-darmstadt.de/14042

Software bundle for data post-processing in a quantum key distribution experiment

Technical report

Pascal Notz, Oleg Nikiforov and Thomas Walther

September 28, 2020

TECHNISCHE UNIVERSITÄT DARMSTADT

Institut für Angewandte Physik

Abstract

Quantum key distribution (QKD) is an innovative technology which has reached its threshold for commercial use. A growing number of research groups and companies around the world implement QKD experiments. For educational purposes, we present a post-processing software bundle for a QKD experiment carried out in our research group. The software manages the readout from a time-tagger, sifting, error correction and privacy amplification of the exchanged key. For error correction, we employ LDPC codes with the sum-product decoding algorithm, provided by R. Neal. For privacy amplification, Toeplitz matrices are used. The software code can be found at: https://git.rwth-aachen.de/oleg.nikiforov/qkd-tools

Contents

1 Scope of Technical Report **4**

2 Preliminaries **4**
2.1 Quantum key distribution . . . 4
2.2 Post-processing in quantum key distribution . . . 4
2.2.1 Channel capacity and the achievable secret key rate . . . 5
2.2.2 Parameter estimation and key reconciliation . . . 6
2.2.3 Privacy amplification . . . 6

3 General description of the system **7**
3.1 Hardware for signal acquisition . . . 7
3.2 Software structure . . . 7
3.3 Filtering of the invalid key packets . . . 8
3.4 Parameter estimation and key reconciliation . . . 8
3.5 Privacy amplification . . . 9

4 Optimization of the matrix size **9**

5 Evaluation of the complete software system **10**

6 Implemented Software **11**
6.1 Data structure . . . 12
6.2 QKD manager . . . 12
6.3 Device manager . . . 13
6.4 FPGA control . . . 13
6.5 Network manager . . . 15
6.6 Key output . . . 15
6.7 Thread manager for post-processing . . . 15
6.7.1 LDPC Manager . . . 16
6.7.2 Privacy amplification . . . 16

7 Conclusion **17**

8 Acknowledgements **17**

1 Scope of Technical Report

The scientific exchange is a crucial element of successful research. Besides journal publications and conference contributions, a direct exchange of know-how and collaborations are important for accelerating progress in different fields.

Therefore, in this report we present the design and the usage of the software bundle that we implemented for control and evaluation of our QKD experiment, realizing a polarization based Bennet-Brassard-84 protocol. Within our software bundle, we also incorporated open source libraries already available.

2 Preliminaries

2.1 Quantum key distribution

Quantum key distribution (QKD) is an important method for secure communication based on the exchange of symmetric keys between the communicating parties based on the laws of quantum mechanics. Thus, its security employs the laws of physics with the additional feature that a potential eavesdropper will reveal itself by errors introduced into the exchanged key. The first protocol was proposed by Bennet and Brassard in 1984 [1]. Since then, a large number of different protocols have been proposed, implemented and analyzed. Several reviews and further details of various protocols can be found in the literature [2, 3, 4]. Here, we give solely a brief general description of a generic QKD protocol.

In general, QKD considers two parties, Alice and Bob, who want to communicate privately. Both parties are connected by a quantum channel for the transmission of quantum objects (the so-called *qubits*) and a classical authenticated channel, where Alice and Bob could exchange further information as well as the encrypted message. Thus, the privacy of the data transmission depends on the applied key and the security of the key transmission. Quantum theory guarantees an information theoretical secure key exchange over the quantum channel. This claim arises from the fact that the information is encoded within the state of single quantum objects (single photons or faint laser pulses). Those states can be described by functions, elements of a Hilbert space and could be prepared within several different given bases of that space. The states or rather the possible measurement outcomes within different bases represent classical bits 0 or 1.

The base and the state of every single qubit is chosen randomly by Alice. She prepares the qubit accordingly and sends it via the quantum channel to Bob. To obtain the correct classical information, a physical measurement within the matching base must be performed. The ignorance of the correct base leads to errors in information retrieving, if a wrong base is chosen. Bob chooses randomly a base for each qubit and notes the chosen base and the value obtained. Once the qubit exchange is finished, Alice and Bob announce publicly, which base was used for each qubit, so they can discard measurements within unmatching bases (the so-called *key sifting*). A potential eavesdropper, generally referred to as Eve, tries to remain undetected and must keep all her errors. Furthermore, according to quantum theory, she is not able to generate an exact copy of the state of any qubit and if she performs any measurement on it, in general, she changes its state, introducing additional errors.

After the key sifting, the remaining bits should be identical in a perfect experiment and with no eavesdropper present. However, the noisy environment or the presence of Eve affect the qubits during the exchange, leading to errors in the sifted key. By directly comparing a part of the exchange bits (and discarding those bits afterwards), Alice and Bob determine this quantum bit error rate (QBER). If this QBER is below a certain threshold, the key exchange is considered successful [5]. Otherwise, the parties have to start over. The key sifting finishes the stage of the qubit exchange and both communicating parties obtain a *raw key*. For generation of the secure key, classical *post-processing* procedures have to be carried out.

2.2 Post-processing in quantum key distribution

In an ideal QKD experiment without an eavesdropper and noise, the raw key is perfectly error free. However, in any real-world implementation, a large number of errors arises from the imperfections of the setup (there exist no perfect single photon sources and detectors), or the quantum channel (Brillouin scattering, polarization mode and chromatic dispersion etc.). During the post-processing procedures the errors introduced are identified (so-called *parameter estimation*) and corrected (*key reconciliation*). Subsequently, the *privacy amplification* step is carried out, minimizing the potential eavesdropper's information about the key.

2.2.1 Channel capacity and the achievable secret key rate

A perfectly secure key can be extracted from a sifted key pf the length n, if the mutual information between Alice and Bob $I(A,B) = H(A) - H(A|B)$ is smaller than the mutual information between Alice and Eve $I(A,E)$. Here $H(X)$ denotes the *Shannon entropy* of X and $H(X|Y)$ denotes the conditional entropy, i.e. the amount of uncertainty of the variable X, if the variable Y is known. Furthermore, it can be shown that due to noise, the mutual information of Alice and Bob decreases, that of Alice and Eve increases. $I(A,B) + I(A,E) \leq n$ holds for the sum. A perfectly secure key can then be extracted from the sifted key of the length n bits, if

$$I(A,B) \geq \frac{n}{2} \tag{1}$$

holds.

The quantum channel is assumed to be a *symmetrical binary channel*, i.e. a channel with possible bit flips and an equal probability p of each bit to flip from 1 to 0 and vice versa.

The Shannon theorem [6] states a maximum transmission rate of reliable information over a noisy channel. This quantity is denoted as the capacity of the channel $C = 1 - h(p)$, with the Shannon entropy $h(p)$ for a symmetric binary channel [7]:

$$h(p) = -p\log_2(p) - (1-p)\log_2(1-p). \tag{2}$$

It quantifies the amount of information about the key that Eve can possess. With the condition for maximal rate $I(A,B) = n \cdot C = n \cdot (1 - h(p))$ and equation 1, the critical error rate can be found to be $p \leq 11\%$ [2]. This means that for QBER$\leq 11\%$ a secure key can be extracted.

The theoretically possible secure quantum key rate (also denoted as the secret capacity) is given by the amount of information, unknown to Eve, minus the amount of data needed by Bob to correct the transmission errors [7]:

$$K_{\text{th}} = H(A|E) - H(A|B) \tag{3}$$

For real error correcting algorithms the achievable rates are much lower and depend on the efficiency of the error-correcting protocol f, which is the ratio of the information, revealed by the implemented protocol, to an ideal protocol with $f = 1$. The achievable key rate is given by [7]:

$$K_{\text{real}} = H(A|E) - fH(A|B) = 1 - h(p) - fh(p) = C - fh(p), \tag{4}$$

The amount of the information announced during the error correction procedure is given by $h(p)$ for an ideal error correcting protocol with $f = 1$. If N_{key} denotes the key length, N_{parity} the length of the transmitted parity bits and f is given as:

$$f(p) = \frac{(N_{\text{parity}}/N_{\text{key}})_{\text{real}}}{(N_{\text{parity}}/N_{\text{key}})_{ideal}} = \left(\frac{N_{\text{parity}}}{N_{\text{key}}}\right)_{\text{real}} \frac{1}{h(p)}, \tag{5}$$

the achievable key rate yields

$$K_{\text{real}} = 1 - h(p) - \left(\frac{N_{\text{parity}}}{N_{\text{key}}}\right)_{\text{real}}. \tag{6}$$

It is obvious that the secure key is shortened due to two effects. On the one hand, $h(p)$ denotes the information that Eve could have obtained during the qubit exchange. And on the other hand, $\left(\frac{N_{\text{parity}}}{N_{\text{key}}}\right)_{\text{real}}$ describes the parity data announced publicly due to the error correcting procedure.

For experiments, the final key length is given by:

$$l = K_{\text{real}} \cdot N_{\text{key}} - 2\log_2\left(\frac{1}{\varepsilon_{\text{sec}}}\right), \tag{7}$$

where N_{key} is the length of the exchanged key and ε_{sec} is a security parameter. The expression $2\log_2\left(\frac{1}{\varepsilon_{\text{sec}}}\right)$ is due to the finite nature of the exchanged key [8]. This is required for a uniform distribution of the key and $\varepsilon_{\text{sec}} = 2^{-60}$ is a typical value.

2.2.2 Parameter estimation and key reconciliation

For the parameter estimation, Alice and Bob choose a subset of the exchanged qubit, announce and compare their values and estimate the error correction of the whole key from the error rate in the compared sample. The used bits are then discarded. Afterwards, the key reconciliation (error correction) is carried out.

There exist several different error correcting algorithms that widely vary in their en- and decoding costs, communication load and performance: Low Density Parity Check (LDPC) Codes [9], Cascade [10], Turbo codes [11], Winnow code [12], Polar codes [13]. From all those types of algorithms the LDPC code is the most promising, enabling encoding extremely close to the Shannon limit [14]. The run-time efficiency can be decreased by proper programming [15] or by the use of GPUs [16] or FPGAs [17]. Originally proposed by Gallager [9], it was rediscovered by Neal and McKay [14], who suggested the use of that algorithm for efficient error correction. Nowadays, LDPC codes are a part of the wireless network standard, are used in digital television, for optimization of solid state disks etc. Here, a short introduction to classical LDPC codes is given.

The LDPC codes are elements of the class of linear codes and can be expressed as linear functions with a sparse matrix. Let $m = (m_1, \ldots, m_n)$ be a message of n bits. It should be encoded into a code word $c = (c_1, \ldots, c_k)$ of length k by a *generator matrix* G of the size $n \times k$:

$$c = mG \tag{8}$$

After the encoded message is transmitted over a noisy channel, the presence of errors can be checked by the *parity-check matrix* H of the size $(k - n) \times k$. H is chosen such that:

$$GH^T = 0 \tag{9}$$

And the correctness is checked by:

$$cH^T = mGH^T = m0 = 0 \tag{10}$$

Alice sends the message c to Bob and he receives the message $c' \neq c$ containing errors. He then starts the decoding of the message, reconstructing c. Finding an efficient decode algorithm is non trivial and is still a research topic. The most widely used algorithm is the *belief propagation*, a subclass of the *sum-product* algorithms. A good description of the algorithm is given in [18].

Both parties agree on the generator and parity check matrices before the error correction procedure.

2.2.3 Privacy amplification

The privacy amplification creates from the error corrected key of bit length n partially known to a potential eavesdropper Eve, a key of bit length l entirely unknown to Eve. Typically, two-universal hash functions are used for that purpose. A two-universal hash function is a function $f : U \to V$ with the property for all $x, y \in U$:

$$P(f(x) = f(y)) \leq \frac{1}{|V|}, \tag{11}$$

i.e. the collision probability is kept as low as possible. In general, even tiny mismatches in the error corrected keys of the communicating parties, lead to a completely different key after the privacy amplification. R. Renner showed that a randomly picked two-hash function can create a secure key [3].

In our implementation, for privacy amplification we use the multiplication with a Toeplitz matrix [19], a $l \times n$ matrix of the form:

$$M = \begin{pmatrix} b_{l-1} & b_l & b_{l+1} & \cdots & b_{l+n-2} \\ b_{l-2} & b_{l-1} & b_l & \ddots & \vdots \\ \vdots & & \ddots & & \vdots \\ b_0 & & \cdots & & b_{n-1} \end{pmatrix} \tag{12}$$

The Toeplitz matrix is defined by a sequence b_k of the length of $(l + n - 1)$ bits. The secret key length l is given by equation 7.

3 General description of the system

3.1 Hardware for signal acquisition

We implemented a polarization-based BB84 protocol without decoy states and with a passive basis choice [20, 21]. The raw key rates obtained over roughly a meter free space transmission are around several kilobits per second and the error rate amounts to $8 \ldots 10\%$.

The Bob module has four single photon detectors connected to an Altera Cyclone FPGA board for data acquisition. The key sifting is carried out at the FPGA board and communicated to Alice by an additional classical channel implemented by a pulsed laser. The sifted key data is accumulated at the FPGA board, divided into blocks of 100 bits length and are transmitted over the USB 2.0 channel to a personal computer. Alice and Bob signal the beginning of each block over the sifting channel, in order to ensure s synchronous operation and error free transmission of packets from the FPGA to the PC. The Ez-USB driver for that interface is provided by the manufacturer. The data acquisition system on the FPGA is based on the open source solution of S. Polyakov [22] and is not the scope of the present report. For further details, please refer to [23] or contact us.

3.2 Software structure

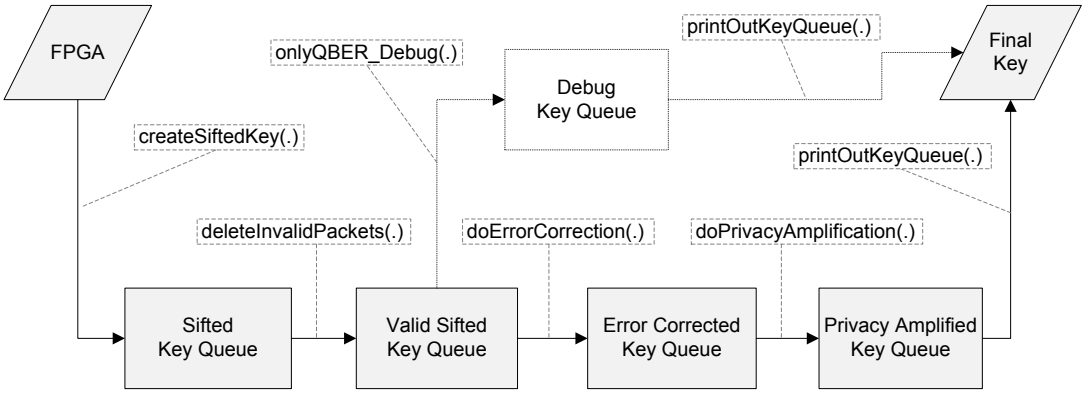

Figure 1: Scheme of the classical data management of the QKD experiment.

Our software system for post-processing is implemented for the Windows operating system, since it requires some proprietary libraries for the communication per USB-2.0 channel with our data acquisition system.

The key post-processing system operates parallel within several threads. There are threads for FPGA data readout, for post-processing as well as for output to a file. Thus, the run time of the algorithms are decreased for multi-core CPUs.

Fig. 1 shows an overview of the complete classical key management in our experiment: data acquisition, post-processing and secure key output. For each step the key data packets are taken from some queue, processed and added to the next queue. The regular processing line includes four queues:

1. *Sifted Key Queue*: Here, the sifted key packets from FPGA are lined up.
2. *Valid Sifted Key Queue*: After discarding of the invalid packets the remaining key data is stored.
3. *Error Corrected Key Queue*: The packets after the successful error correction are added.
4. *Privacy amplified Key Queue*: The packets after the successful privacy amplification are added.

The irregular processing line contains the *Debug Key Queue* for debugging purposes. Here the packets are put out without error correction and privacy amplification.

The exchanged sifted key can still contain some errors and invalid key blocks. During post-processing the following three steps are carried out in our post-processing routine:

1. Filtering of invalid sifted key packets

2. Parameter estimation and error correction

3. Privacy amplification

Each of those steps is executed in its own thread. In the following, we will discuss each step in more detail.

3.3 Filtering of the invalid key packets

In the first post-processing step, Alice and Bob check the validity of their sifted key blocks. Therefore, they control the number of the sifted key bits in each block received from the FPGA. A valid block contains exactly 100 bits. The communicating parties announce the results over the public network channel. The invalid blocks are then discarded by both of the parties. The synchronization of evaluation is ensured, by the simultaneous start of data acquisition with empty key queues. Additionally, the synchronization signals over the key sifting channel flag the begin of each block for both of the parties. Furthermore, the processing starts only after the network communication is finished. In this way, Alice and Bob start to process the matching key packets simultaneously, even if they possess unequal computing power.

3.4 Parameter estimation and key reconciliation

A bit error rate of the quantum bits (QBER) is an important measure for the security of exchanged data, as it enables the estimation of the information that a potential eavesdropper may have collected. Thus, QBER is a parameter for the privacy amplification.

During the parameter estimation step the QBER is determined. Therefore, Alice combines 110 key packets (each containing 100 bits) and chooses randomly from those 11,000 bits a subset of the length 1,000. She announces the coordinates and the value of the bits from that sample over the public channel and eliminates subsequently that subset from her key.

Bob also combines 110 key packets and compares the values received from Alice to the values that he had measured and estimates the QBER. Then, he also discards the publicly known fraction of data from his key and announces the QBER to Alice.

The QBER estimation is carried out as follows. For example, let $p = 9\%$ be the maximal allowed error probability. Then, the cumulative binomial distribution $P_{n,p}(X \leq k)$ describes the probability for up to k bits to be erroneous in a sample of n bits. For example, $P_{1000,0.09}(x \leq 74) \approx 0.04$. This means that the probability for the real QBER to be smaller than 9% amounts to 95%, if the maximal number of error bits is 74. All the blocks with a sample exceeding this threshold are thus discarded.

Because of samples for QBER estimation nine percent of the exchanged key length is used up. This number is a compromise in a trade-off between the precision of the QBER estimation and the amount of qubits used.

The non-discarded blocks, consisting now of 10,000 bits, are divided into 10 new blocks for the error correction procedure. In principle, it is possible to vary the block size. However, for smaller blocks the network latency leads to slower progress and the bigger blocks are detrimental for secure key length, if the QBER lies between 8.5 and 10% (see sec. 4)

Thus, after the parameter estimation Alice and Bob possess key packets of the size of 1,000 bits. For the error correction via LDPC codes we adapted the open source library *ldpclib* provided by Radford Neal [24]. Here, the *belief propagation* algorithm [14] is used in that implementation. The beginning of the algorithm, the error probability for each exchanged bit is set. Subsequently, the algorithm is carried out iteratively, until the correct code word is found or the maximal number of iterations is reached.

In our implementation, the error probability for parity bits is set to 0, since the parity bits are transmitted over the classical channel assumed to be noise-free. This is our only modification of the code by R. Neal.

The performance of the algorithm depends on the matrices used. However, the construction of appropriate parity-check matrices is non-trivial. In our case, generator and parity-check matrices are generated within *ldpclib* [24].

The success of the decoding algorithm is determined by the process described in 2.2.2: parity-check matrix checks the validity of the code word. If the code word is valid, no further verification of the key packet is necessary.

In the case of an invalid code word, the decoding failed and the key still contains errors. During the privacy amplification, the error rate would increase leading to a useless key. Thus, Bob informs Alice about the failed algorithm and both parties discard the present key packet.

3.5 Privacy amplification

According to equation 7, the security parameter during the privacy amplification is independent of the key packet size. For a 1,000 bit packet length, the security key length would amount to zero bits. Therefore, for the privacy amplification 10 key packets of length 1,000 are combined and processed simultaneously. For the estimated QBER$\leq 9\%$ and for the ratio of the parity data length to the key length of $N_{\text{parity}}/N_{\text{key}} = 0.522$ (see sec. 4) the final secret key length amounts to $l = 295$ bits.

4 Optimization of the matrix size

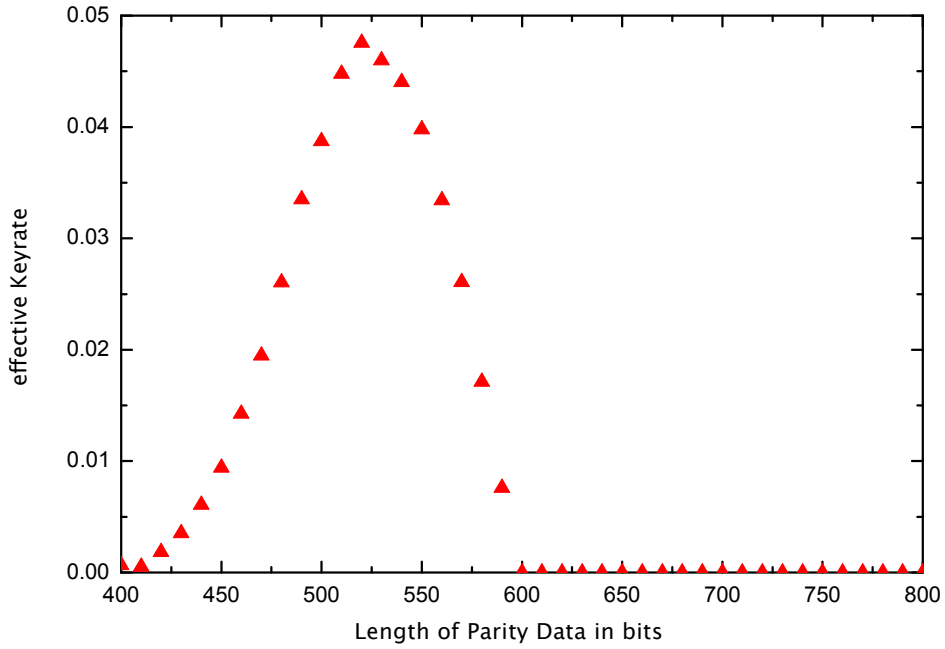

Figure 2: Simulated effective key rate K_{eff} as a function of the parity bit length.

For efficient error correction, a suitable LDPC matrix is necessary. Thus, the number of the parity bits to be transmitted is a crucial parameter. There exists a trade-off between the shortening of the key length due to public announcement of the parity bits and the increasing failure probability of the error correcting algorithm due to the lack of parity bits also resulting in discarded key packets. This leads to the expression for the effective key rate K_{eff}:

$$K_{\text{eff}} = (1 - BLER) \cdot K_{\text{real}} = (1 - BLER) \left(a - h(p) - \left(\frac{N_{\text{parity}}}{N_{\text{key}}} \right)_{\text{real}} \right). \tag{13}$$

Hereby, K_{real} is the real key rate given in 6 and $BLER$ denotes the block error rate, i.e. the number of blocks to be discarded due to a failed error correction. To maximize our K_{eff}, we test the implemented post-processing software for different parity bit lengths and evaluate the effective key rate K_{eff}. For this simulation, we created 5,000 of random 1,000 bit long key blocks with the error rate of 8%. For each parity bit length between 400 and 800 the error correction procedure was then carried out. Afterwards, the number of the unsuccessfully error corrected blocks was calculated and K_{eff} was determined. The result is shown in fig. 2. The effective key rate turned out to be non-vanishing for the parity bit length between 400 and 600 and reaches its maximum around 525 ± 5 bits.

The maximal possible key rate depends also on the QBER. In the second step, we repeated the previous simulation for the keys with the QBER between 5% and 11%. The result is shown in fig. 3. The effective key rate drops to zero for the QBER of $10 \ldots 11\%$ for all chosen parity bit lengths, i.e. the chosen algorithm is unsuitable for data with a high QBER. However, since roughly a QBER of 11% constitutes the limit for secure key exchange this is not a problem.

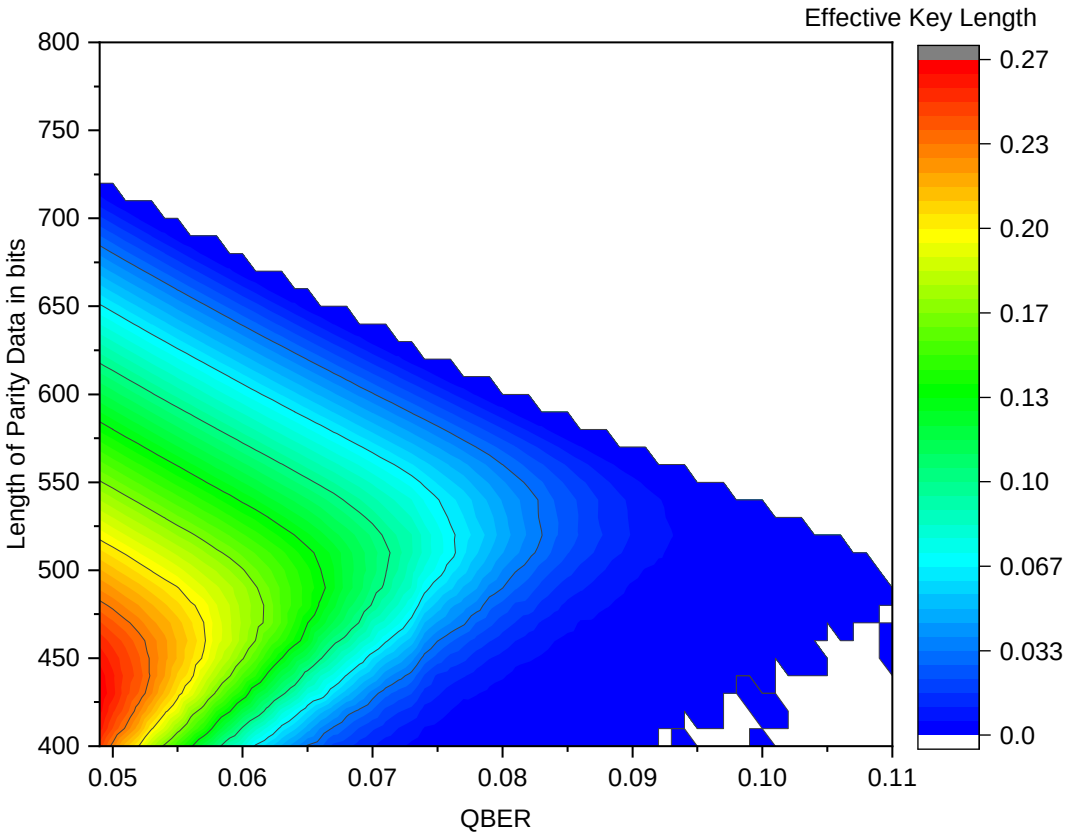

Figure 3: Simulated effective key rate K_{eff} as a function of the parity bit length and QBER of the key.

Fig. 3 suggests also the dependence of the optimal parity bit length on the present QBER. To specify it, we consider for our experiment the relevant QBER area between 7% and 10%. The results are shown in fig. 4). Here, for better clarity, the data points are connected by cubic hermite spline and the curve colors indicate different QBER. The dotted line denotes the optimal parity bit length. It turns out that the optimal parity bit length only slightly varies around $l = 521.6$ bits. Therefore, we determine the optimal parity bit length for our experiment to be 522 bits.

In the end, we consider the influence of the key block length on the performance of our error correction algorithm. Therefore, the key block length is set subsequently to 100, 1,000, 10,000 and 20,000 bit followed by a run the error correction procedure. To minimize the variance for the result, the number of the tested blocks are also different. For key block lengths of 100 and 1,000 an amount of 10,000 blocks was used, and for the lengths of 10,000 and 20,000 the amount of 1,000 blocks was sufficient. The parity bit length ratio is set to $N_{\text{parity}}/N_{\text{key}} = 0.52$ as determined above. In the fig. 5 the dependence of the number of unsuccessfully error corrected blocks versus the QBER in those blocks is shown. For clarity, the dots are connected by the cubic Hermite splines and the total number of the data points is reduced. From the data can be concluded that a higher key block length is only advantageous for QBER below 8.5%. Since the expected QBER in our experiment lies above that value, we chose a smaller key packet length. Additionally, for a QBER$> 8.5\%$ long blocks lead to a rapid increase of the BLER.

5 Evaluation of the complete software system

After implementing our software, we conducted a thorough test with following parameters. The error bit threshold for acceptance of the sifted key blocks with the size of 1,000 bits was set to 74 bits. That corresponds to a QBER$\leq 9\%$. The key block length for the error correction is set to 1,000 bits and the parity bit length was set to 522. In this way, after the privacy amplification with blocks of 10,000 bits length, we obtained the final key length of 295 bits corresponding to the expected real key length given in equation 7. For those parameters, the parity-check, the generator, and the Toeplitz matrices were created

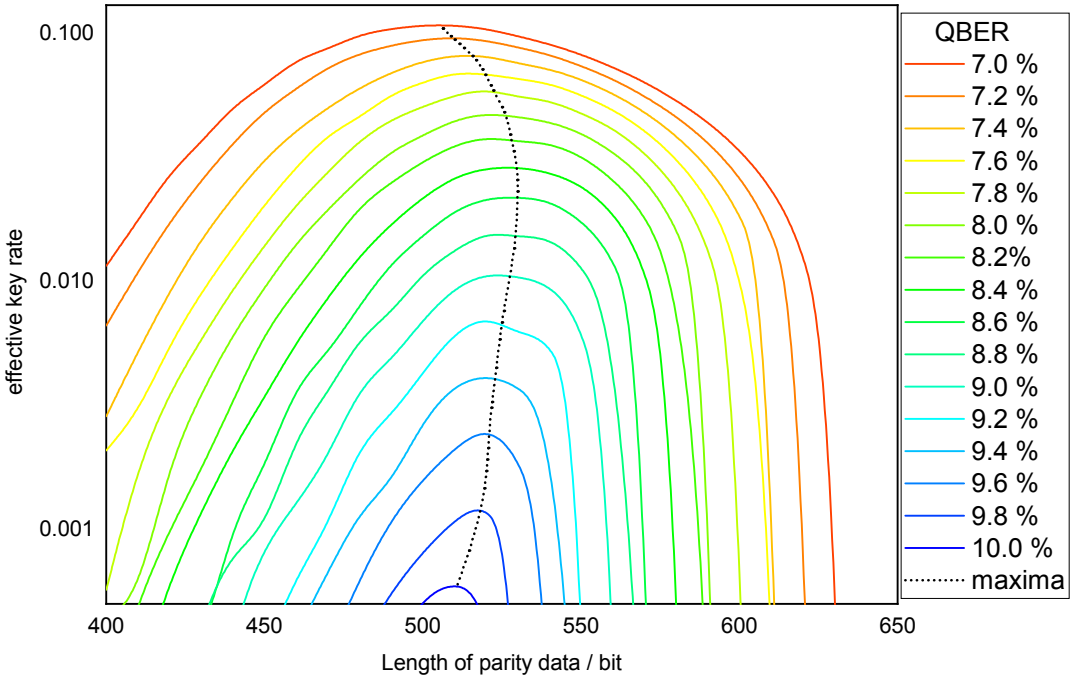

Figure 4: Simulated effective key rate K_{eff} as a function of the parity bit length. Color denotes different QBERs. The dotted line indicates the approximate maxima of the curves.

and the test conducted

For the test, we simulated several sets of the sifted keys with the QBER between 5% and 10% with the size of 10^7 bits per set. Subsequently, we applied our post-processing routine on those keys and obtained the actual secret key length, displayed in fig. 6. The data points are connected by cubic hermite splines. The data shows that the effective key rate vanishes for a QBER higher than 9%. For QBER below 6.5% the effective key rate approaches asymptotically a rate of 0.0268.

Here, the obtained actual rate K_{act} is smaller than K_{real} as well as K_{eff} due to several reasons. In each step of the error correction, the key is shortened:

- During the parameter estimation, 10% of the key length was consumed in order to estimate the QBER.
- During the key reconciliation a high number of packets was discarded, since the number of error bits exceeded 74 out of 1000. In fact, for a QBER$= 9\%$ the algorithm discards around 95% of all packets. And for a QBER$= 5\%$ only 0.04%.
- Due to a failed error correcting algorithm a lot of packets are discarded (compare fig. 5).
- Privacy amplification algorithms compress the key (see sec. 2.2.1).

According to the results of the test, the secure key rate drops rapidly for QBER exceeding 6.5%. For experiments with smaller QBER it would be recommended to increase the block length for error correction according to fig. 5 and to the number of discarded packets during the parameter estimation.

6 Implemented Software

In this section, technical details on the structure of our implementation are given. The source code of our software can be found at: https://git.rwth-aachen.de/oleg.nikiforov/qkd-tools.

The class-diagram of the implemented library is displayed in fig. 7. The main class is the quantum key distribution manager (`QkdMgr`) which takes care of all necessary steps for the successful secret key creation. The device manager (`DevMgr`) is responsible for the data acquisition with the FPGA board and the network manager (`NetworkMgr`) establishes a connection to the communicating partner. The thread manager (`ThreadMgr`) is responsible for the error correction within the `ldpcMgr`-class and the privacy

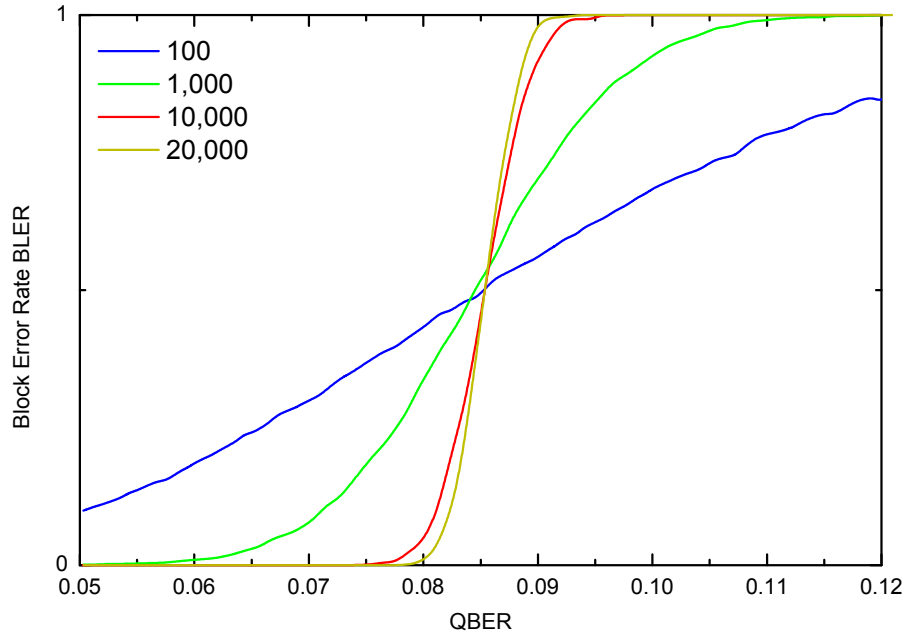

Figure 5: Block error rate BLER as a function of the QBER.

amplification within the `PrivAmp`-class. Finally, the class `printOutKey` enables the output of the key to a file.

6.1 Data structure

An efficient handling of the sifted keys requires a dedicated data structure. Therefore, a qubit stream detected and sifted at the FPGA is divided into packets (objects of class `KeyQueuePacket`) which are added to a queue (class `KeyQueue`) and subsequently processed (Fig. 8).

The attributes of the `KeyQueuePacket` are described in table 1. During the initialization of the `KeyQueuePacket`-Object, the validity-attribute should be set.

Attribute	Description
data	pointer to the key data bits
len	length of the key in the packet
qber	packet's error rate
sentParBits	number of the parity bits, sent for error correction
next	pointer to the next packet in the queue

Table 1: Attributes of the `KeyQueuePacket`-object.

6.2 QKD manager

First of all, the QKD manager has to be configured by the command `readConfigFromFile(.)`. The configuration file must match the rules in tab. 2.

Next, the key generating process can be controlled by `start()`, `stop()` and `reset()`. The command `close()` shuts the connection to the FPGA and closes the secret key output file.

For simple access `QkdMgr` provides an automatic mode which can be activated by a parameter in the configuration file (Tab.2). In this mode, Alice's `QkdMgr` opens a network connection to Bob's `QkdMgr` and controls it by suitable commands. This guarantees the synchronous operation of both communicating parties.

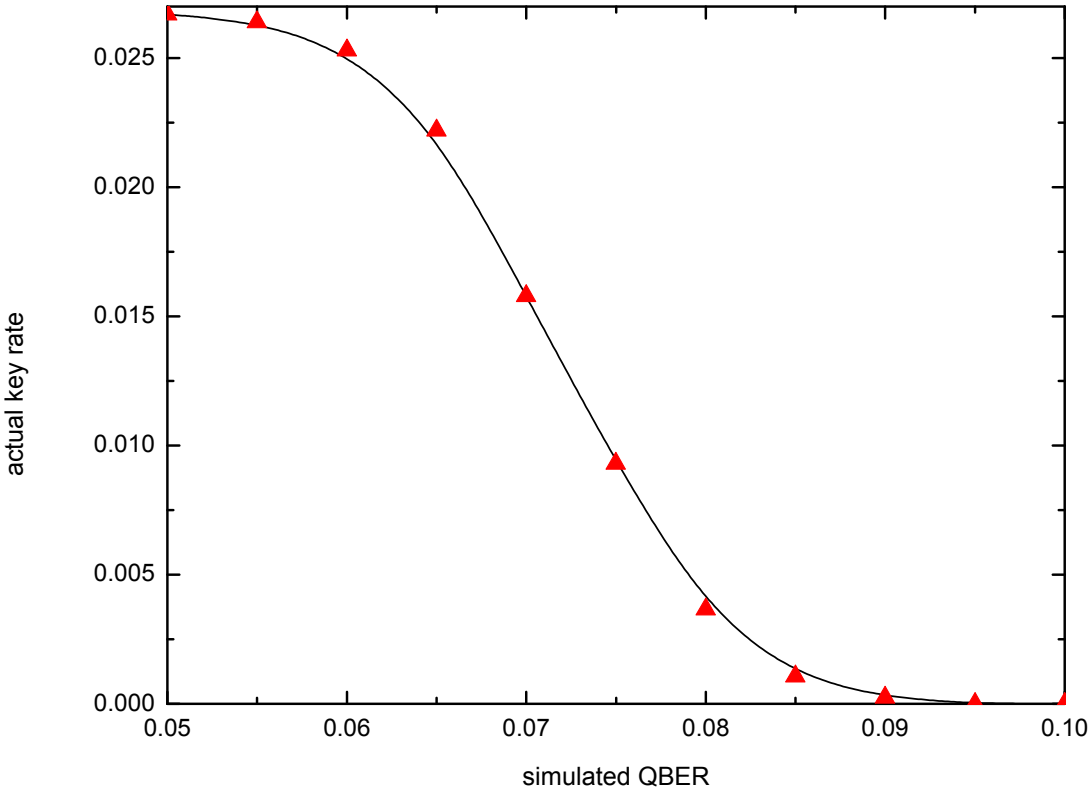

Figure 6: Simulated actual key rate K_{act} as a function of QBER.

The command `startDebug()` offers an alternative operation mode for debugging. Instead of the error correction and privacy amplification, here the packets are checked for validity, the QBER of the valid packets is determined and the raw key is written to a file.

6.3 Device manager

The communication with the FPGA board is carried out by the *Device Manager* (class `DevMgr`). It encapsulates the library *fpgaIO.dll* (see sec. 6.4), provides all its functions and offers some additional functionality, such as the data acquisition in a separated thread. First, the command `setQueue()` creates a queue for the sifted key packets. Subsequently, `openDev()` and `setDelayandStart()` opens a connection to the FPGA board, configures the delays and starts the data acquisition on the FPGA board. Then, the command `startDataCollectThread()` starts a new thread for data reception which collects new data with a repetitive call of `getData()`, divides them into packets and puts them into a queue.

The incoming data are divided into blocks of 100 bits at the FPGA. For detection of USB transmission errors, after receiving the number of the sifted key bits in each block is calculated. (A valid block consists of exactly 100 bits.) Subsequently, the bits are put into a new instance of `KeyQueuePacket` created with an according attribute `isValid` and then put into the queue. Both, valid and invalid packets are parts of the sifted key. By the command `stop()` the signal acquisition at the FPGA board is stopped, `stopDataCollectingThread()` finishes the thread for data transition from the FPGA board to the PC. The command `clearDeviceDataStorage()` wipes the FPGA cache and `closeDev()` shuts the USB connections to the FPGA board.

6.4 FPGA control

The library *fpgaIO.dll* enables the control of the FPGA board. It is based on the functions implemented in the open source project by S. Polyakov [22]. Therefore, the following commands are implemented:

- `FPGA_Open()` opens the connection to the FPGA and returns a handle.
- `FPGA_Close()` closes the connection to the FPGA board.

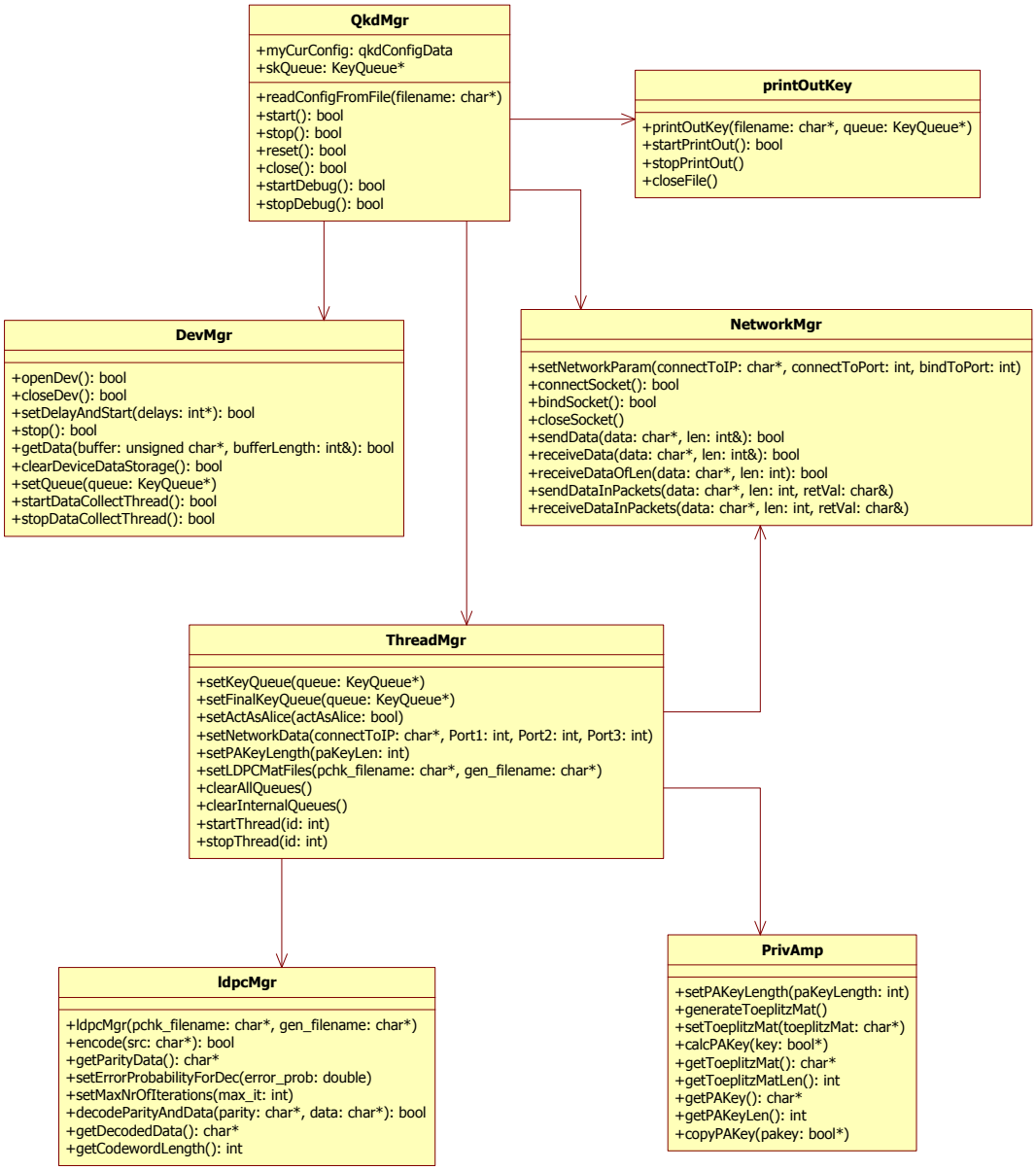

Figure 7: Class diagram of the qkdtools.dll library

- `FPGA_Interface(.)` communicates with the FPGA board.

Each call transmits a command to FPGA of length one byte. The commands in Table 3 define the delay times and trigger the begin or the end of data acquisition at the FPGA.

For setting up the delay time, all six bytes are required. The first four bytes define the relative delay of SPADs. And the last two bytes define the global delay: the penultimate byte sets seven lower bits and the ultimate byte the 5 higher bits of the global delay, that is 12 bits long.

The global delay should be carefully adjusted. Therefore, the class *qkdtools* (not shown in the class diagram) provides the command `optimizeGlobalDelay(.)`. This function requires all relative delays and an estimated global delay as parameter. The function then varies the global delay within ± 5 clock cycles around the received estimated value and returns the exact value of the global delay which maximizes the absolute number of detected events.

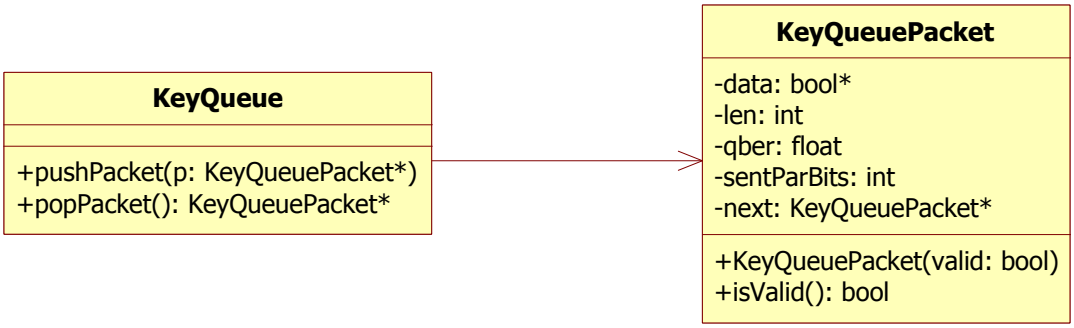

Figure 8: Class diagram of the data structure

Line number	Content
1	comment/header
2	auto mode (on:1, out:0)
3	identity (Alice:1, Bob:0)
4-7	relative delay of the four detectors in terms of the clock periods (a number between 0 and 127)
8	global delay in terms of the clock periods (a number between 0 and 4095)
9	IP-address of the opposite communication partner (XXX.XXX.XXX.XXX)
10-13	four open communication ports (e.g. from 12396 to 12399)
14	final length of the secure key, -1 for a dynamic calculation
15	maximal number of the error bits allowed from 1000 investigated bits.
16	filename of the LDPC parity-check matrix (*.pchk)
17	filename of the LDPC generator matrix (*.gen)
18	key output filename

Table 2: Configuration file rules for the QKD manager object.

6.5 Network manager

The class `NetworkMgr` is capable of communicating over the Transmission Control Protocol (TCP). Prior to data transmission and after the configuration via `setNetworkParam(.)` a network connection between the communicating partners should be established. Therefore, one of the partners, working as a *server*, opens a port and accepts incoming connections via `bindSocket()`. The opposite partner, the *client*, connects to the server's open port by `connectSocket()`. After the connection has been established, the communication becomes symmetric: both, Alice and Bob are able to send (`sendData(.)`) and receive data (`receiveData(.)`). Hereby, a large amount of data is divided into smaller packets by the protocol. The command `receiveDataOfLen(.)` waits until a certain amount of data is received.

In the case the data exceeds 1,500 bytes, it must be divided into packets, transmitted and subsequently restored from all smaller packets by `sendDataInPackets(.)` and `receiveDataInPackets(.)`. The command `closeSocket()` shuts the connection.

6.6 Key output

The class `printOutKey` consumes the key packets from some queue and puts it out in a text file. Therefore, a new thread is started and stopped by `startPrintOut()` and `stopPrintOut()` respectively. The command `closeFile()` closes the key output file.

6.7 Thread manager for post-processing

The exchanged sifted key still contains some errors and invalid key blocks. During post-processing the errors are discarded and the secret key is established. Three steps are included in our post-processing routine:

1. Filtering of invalid sifted key packets

Table 3: Commands for the FPGA control.

FPGA-command	Description
00000000	invalid command
00000010	stop acquisition
00000100	start acquisition
10000110	set a delay (The 7th bit defines the command, other bits define the value of the delay. Here: 6)

2. Parameter estimation and key reconciliation

3. Privacy amplification

Each of those steps is carried out in its own thread. For each step the key packets are taken out of a queue, processed and lined up into the next queue, according to fig. 1. Therefore, the threads are organized by the *Thread Manager* (class `ThreadMgr`). The initialization of the thread manager requires the following steps. Its IP address and open ports are configured by `setNetworkData(.)`. By `setActAsAlice(.)` the role of the communicating party is defined. The command `setLDPCMatFiles(.)` defines the path to the matrices for error correction algorithm. And the command `setPAKeyLength(.)` the final key length is set (value -1 denotes the automatic calculation of the size). The command `setMaxBitErrors(.)` defines the number of error bits for a sample of 1000 bits, the threshold for discarding of the key block. Finally, the *Sifted Key Queue* (by `setKeyQueue(.)`) and the final *Privacy Amplified Key Queue* (by `setFinalKeyQueue(.)`) are created.

After the configuration the three threads can be managed by `startThread(.)` and `stopThread(.)`. Those functions require the thread ID as a parameter: 0, 1 or 2 for the regular threads and 3 for the debugging thread.

The queues are cleared by `clearAllQueues()` or `clearInternalQueues()`, after the threads are stopped. The first command clears all of the queues, but the latter one keeps the sifted key queue and the privacy amplified key queue untouched.

6.7.1 LDPC Manager

The class *ldpcMgr* provides the interface for the library *ldpclib*. Its instantiation requires a filename with the corresponding control ($*.pchk$) and generator ($*.gen$) matrices. The size of those matrices determines the size of the key to be corrected and the amount of the required parity bits.

Error correction is carried out within the following steps: Alice encodes a data block by `encode(.)` and receives a pointer to the parity data by `getParityData()`. She transmits the parity data over the public channel to Bob. For the decoding of the data, Bob sets an estimated error probability for the key data by `setErrorProbabilityForDec(.)` and the maximal number of algorithm iterations by `setMaxNrOfIterations(.)`. Subsequently, he combines his measured key data with the received parity data and starts decoding by `decodeParityAndData(.)`. A successful decoding returns a *true*. Thus, the command `getDecodedData()` returns a pointer to the now error-free key bits. Subsequently, Bob informs Alice about his success and both parties put the key packet into the next queue.

6.7.2 Privacy amplification

The class `PrivAmp` enables the privacy amplification procedure. First of all, the final key length must be set by `setPAKeyLength(.)`. This parameter is either constant or can be varied for each packet as a function of the QBER and the number of the transmitted parity bits (see sec. 2.2).

Thus, Alice takes 10 error corrected key packets from the queue, calculates the final key length and creates a random Toeplitz matrix by `generateToeplitzMat()` and receives access to its pointer by `getToeplitzMat()`. Then she transmits this matrix to Bob over the public channel.

Bob receives the matrix and passes it by `setToeplitzMat(.)` to his `PrivAmp`-Instance. Analogously to Alice, he picks 10 key packets out of the queue and both parties calculate the final key by `calcPAKey(.)`. The command `copyPAKey(.)` allows for the transmission of the final key to the key packet that is then put into the output key queue.

7 Conclusion

In this technical report, we present the implementation of the post-processing software for our BB84 quantum key distribution experiment. We discuss the optimal parameters to be used for different observed quantum key error rates in order to maximize the final secret key rate.

8 Acknowledgements

We thank Radford Neal for development of the open source LDPC library [24] and Sergey Polyakov for his open source data acquisition system [22]. We thank all students who participated in the development of this experiment: Daniel Rieländer, Tobias Diehl, Sabine Wenzel, Stefanie Lehmann, Micha Ober and Sabine Euler as well as many others.

References

[1] 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*. 1984, pp. 175–179.

[2] N. Gisin et al. “Quantum cryptography”. In: *Rev. Mod. Phys.* 74 (1 2002), pp. 145–195.

[3] R. Renner. “Security of quantum key distribution”. In: *International Journal of Quantum Information* 6.01 (2008), pp. 1–127.

[4] E. Diamanti and A. Leverrier. “Distributing secret keys with quantum continuous variables: principle, security and implementations”. In: *Entropy* 17.9 (2015), pp. 6072–6092.

[5] P. W. Shor and J. Preskill. “Simple Proof of Security of the BB84 Quantum Key Distribution Protocol”. In: *Phys. Rev. Lett.* 85 (2 July 2000), pp. 441–444.

[6] C. E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.623-656 (1948), pp. 379–423.

[7] D. Elkouss et al. “Efficient reconciliation protocol for discrete-variable quantum key distribution”. In: *Proceedings of the 2009 IEEE international conference on Symposium on Information Theory - Volume 3*. ISIT’09. Coex, Seoul, Korea: IEEE Press, 2009, pp. 1879–1883. ISBN: 978-1-4244-4312-3.

[8] M. Tomamichel et al. “Tight finite-key analysis for quantum cryptography”. In: *Nature Communications* 3 (2012), p. 634.

[9] R. Gallager. “Low-density parity-check codes”. In: *IRE Transactions on information theory* 8.1 (1962), pp. 21–28.

[10] G. Brassard and L: Salvail. “Secret-Key Reconciliation by Public Discussion”. In: *Advances in Cryptology — EUROCRYPT ’93*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 410–423.

[11] F. Vatta, R. Romano, and M. T. D. Alizo. “Turbo Codes for Quantum Key Distribution (QKD) Applications”. In: *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*. New York, NY, USA: Association for Computing Machinery, 2011.

[12] W. T. Buttler et al. “Fast, efficient error reconciliation for quantum cryptography”. In: *Phys. Rev. A* 67 (5 May 2003), p. 052303. DOI: 10.1103/PhysRevA.67.052303.

[13] S. Lee and J. Heo. “Efficient Reconciliation Protocol with Polar Codes for Quantum Key Distribution”. In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2018, pp. 40–43.

[14] R.M. Neal. “Near Shannon limit performance of low density parity check codes”. English. In: *Electronics Letters* 33 (6 Mar. 1997), 457–458(1).

[15] M. Benhayoun et al. “New Memory Load Optimization Approach for Software Implementation of Irregular LDPC Encoder/Decoder”. In: *2019 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS)*. 2019, pp. 1–6.

[16] S. Kang and J. Moon. “Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing”. In: *2012 IEEE International Conference on Communications (ICC)*. 2012, pp. 3692–3697.

[17] V. A. Chandrasetty and S. M. Aziz. “FPGA Implementation of High Performance LDPC Decoder Using Modified 2-Bit Min-Sum Algorithm”. In: *2010 Second International Conference on Computer Research and Development*. 2010, pp. 881–885.

[18] A. Shokrollahi. “LDPC codes: An introduction”. In: *Digital Fountain, Inc., Tech. Rep* 2 (2003), p. 17.

[19] C.-H- F- Fung, X Ma, and H. F. Chau. “Practical issues in quantum-key-distribution postprocessing”. In: *Phys. Rev. A* 81 (1 Jan. 2010), p. 012318. DOI: 10.1103/PhysRevA.81.012318.

[20] S. Euler. “Preparation and Characterization of Single Photons from PDC in PPKTP for Applications in Quantum Information”. PhD thesis. Technische Universität, 2017.

[21] S. Lehmann. “Testing a Transmission Line for Quantum Key Distribution”. MA thesis. Germany: Technische Universität Darmstadt, 2014.

[22] Sergey Polyakov. *Simple and Inexpensive FPGA-based Fast Time Resolving Acquisition Board*. Technical Description, Installation and Operation Manual. Jan. 2010. URL: http://www.nist.gov/pml/div684/grp03/multicoincidence.cfm.

[23] T. Diehl. *Quantum Cryptography – Building a quantum cryptographic transmission line*. Master-Thesis. Nov. 2011.

[24] R. M. Neal. *Software for Low Density Parity Check Codes*. Feb. 2012. URL: http://www.cs.utoronto.ca/~radford/ftp/LDPC-2012-02-11.