# Symmetric Blind Information Reconciliation for Quantum Key Distribution with Dense Pool of Low Density Parity Check Matrices

Matan Ostrovsky

September 2019

**Abstract**

This work will examine the error reconciliation, or interchangeably information reconciliation, stage of a quantum key distribution (QKD) scheme. The focus will be on low density parity check (LDPC) matrices based reconciliation algorithms, mainly implementing and examining the Symmetric Blind error reconciliation scheme[1]. The Symmetric Blind scheme was chosen since it performs better on chosen metrics. We further introduce and analyse the use of dense pools of LDPC matrices, in which one holds a large number of LDPC matrices used for error correction. This method requires relatively large use of memory, and we will present methods that reduce memory usage and the complexity of calculations.

We first will present two classical encryption schemes, and view the key distribution problem and the difference between information theoretical security and computational security. Later we present QKD, a set of key distribution schemes combining classic and quantum communication. We will see the BB84 QKD scheme and the Coherent One Way QKD scheme. Following this, we will present the use of LDPC matrices in error correction, and specifically in QKD. We also show LDPC matrix generation using the progressive edge growth (PEG) algorithm. Afterward, we will examine rate adaptive error reconciliation, a family of algorithms optimizing the LDPC based reconciliation algorithm. We will focus on, implement and test the Symmetric Blind rate adaptive scheme. To conclude, we will introduce the use of a dense LDPC matrix pool and analyse the method's performance.

## 1 Introduction

Quantum Key Distribution (QKD) is a method of distributing secret keys between two communicating parties. It does so while solving existing security challenges, and addressing future security risks arising from quantum technology.

To operate, QKD requires the two parties to exchange information on both quantum and classical channels. The quantum channel, within current limitations of technology, is extremely prone to transmission errors. These errors need to be corrected by the parties to facilitate communication. This work focuses on the error correction, or error reconciliation, process in QKD.

To understand the initial problem solved by QKD, we will start by viewing two classical encryption methods, the One Time Pad, and RSA.

## 2 Classical Cryptography

Looking at different fields of mathematics, few of them can produce the same level of drama as cryptography. This is the science of information hiding, deceit, and espionage, dating back to early history. Over time, especially in the second half of the 20th century, cryptography has developed from an art form to an elaborate mathematical field. Computers increase the complexity of the encryption techniques, but also enable systematic decryption of ciphers. Bound by these technological improvements, modern cryptography analyzes encryption schemes while considering the strength of an adversary.

A mathematical discussion of encryption schemes will often consist of three parties - Alice (party A), Bob (party B) and Eve (an eavesdropper). Alice encrypts a message $m$, resulting in a cipher $c$. Alice then sends $c$ to Bob, which in turn will decrypt it back to the original message $m$. The communication between Alice and Bob is monitored by Eve, which may have various sets of abilities. The durability of a scheme to the resources available to Eve creates different definitions of security. An encryption scheme that cannot be realistically decrypted unless Eve possesses computational abilities exponential in the length of the message is called "Computationally Secure". If the scheme cannot be decrypted regardless of the computational abilities of Eve, it is called "Information Theoretic Secure".

To better understand the challenges involved in encoding schemes, we will briefly examine two methods: The One Time Pad, and the RSA public key encryption.

### 2.1 The One Time Pad

One Time Pad (OTP) is a symmetric key encryption scheme, meaning that both Alice and Bob hold an identical key known only to them, used to encrypt and decrypt the message. In the OTP scheme, we use a random key of length identical to the message. To maintain secrecy, the key must not be used to encrypt more than one message, hence we refer to it as a one time pad. We mark the key and message respectively as $k, m \in \{0,1\}^*$[i], when as said $|k| = |m|$. Alice will create a cipher $c \in \{0,1\}^*$ by performing $c = k \oplus m$, when the $\oplus$ sign stands for logical XOR. Bob will decrypt the message similarly by performing $m = c \oplus k$. Making no assumptions on the capabilities of Eve, one can prove that the cipher does not reveal information on the original message. We should pay attention to the high cost in key bits - for every message bit sent we require a single key bit. Additionally, the protocol does not provide an answer as to how to secretly distribute the key between Alice and Bob in the first place. This is the key distribution problem -

[i] $\{0,1\}^*$ is the set of all finite strings consisting of 0s and 1s.

to have secret communication using a symmetric key scheme, we must first exchange secret information.

### 2.2 RSA

RSA[2], which stands for its inventors' names (Rivest, Shamir, and Adleman), is a public key encryption protocol. Such protocols consist of a publicly exposed key used for encryption, and a private key known only to Bob. Any person, let alone Alice, can encrypt the message using the public key, and using the private key Bob alone will be able to decrypt it efficiently. Unlike OTP, these methods allow the two sides to encrypt information without ever sharing secret information. Specifically, in RSA, the security is based on the assumed hardness of integer numbers factorization. Decoding the cipher without the private key is as hard as factoring the multiplication of two unknown primes. The hardness of this problem is not proven, but there is no known classical algorithm that solves it efficiently. It is not realistic to decrypt the cipher with existing technology. This said, recent technological developments make this scheme extremely vulnerable. Shor's Algorithm[3], which allows efficient integer factorization using a quantum computer, renders RSA ineffective in the age of quantum computing.

## 3 Introduction to Quantum Key Distribution

Having seen the classical methods and the challenges they are facing, we will now introduce Quantum Key Distribution (QKD). QKD is a family of protocols allowing for key distribution, secured in different levels against an adversary equipped with quantum technology.

### 3.1 Qubits and Measurement

To define quantum schemes of encryption, we must first define a quantum information model. The basic unit of information will be the qubit, some quantum state encoding bit-like information. Formally, it will be a vector

$$|\phi\rangle \in \mathbb{C}^2 \quad , \quad |\langle\phi|\phi\rangle|^2 = 1 \tag{1}$$

To obtain the bit value of a qubit we need to measure it. The process of quantum measurement is not trivial and probabilistic. Defining two orthogonal vectors $|v_0\rangle, |v_1\rangle$ to represent the values 0 and 1 correspondingly, the outcome of measurement will be $i \in \{0, 1\}$ with probability $|\langle v_i|\phi\rangle|^2$. We should notice that the probability to measure some value is dependant on the initial choice of basis and that we have the freedom to choose a measurement basis in the first place.

Quantum computers can prepare quantum states in ancillas (qubit registers created in advance), and to manipulate sets of qubits using linear operations. Since it is required to preserve the qubits' normalization, linear operations on qubits will be unitary. A unitary transformation on qubits is called a quantum gate.

For later use, we shall define and use two orthogonal measurement bases. The first pair, which we will name as the up-down (UD) basis for convenience, will be defined as

$$|\uparrow\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |\downarrow\rangle = \begin{bmatrix}0\\1\end{bmatrix} \tag{2}$$

The second basis, which we will name plus-minus (PM) basis, will be defined as

$$|+\rangle = \begin{bmatrix}1/\sqrt{2}\\1/\sqrt{2}\end{bmatrix}, |-\rangle = \begin{bmatrix}1/\sqrt{2}\\-1/\sqrt{2}\end{bmatrix} \tag{3}$$

We will assign the value 0 to $\{|\uparrow\rangle, |+\rangle\}$, and 1 to $\{|\downarrow\rangle, |-\rangle\}$. We would note that the UD / PM is not a naming convention, but rather our arbitrary choice.

### 3.2 The QKD Setup

A QKD system consists of the two communicating parties, Alice and Bob, and the eavesdropper Eve (Fig. 1). Alice and Bob are communicating via two channels, a quantum channel carrying qubits, and a classical channel carrying classical bits. The goal of the communication is that Alice and Bob will hold an identical string of bits, the key, with a content unknown to Eve. When analyzing the protocol, we assume that Eve can view and alter the values on both channels. To authenticate the messages on the classical channel, Alice and Bob will have to share a secret key a priori. We also assume that the units of Alice and Bob are verified and there is no additional information leakage other than on the communication channels. Like the classical case, we may grant Eve various computational capabilities, including quantum, thus defining different types of attacks and different standards of security.

In contrast to classical cryptography, in QKD one assumes the only limitation on the power of Eve is that her apparatus and attacks must obey the rules of quantum mechanics. This allows a much wider set of attacks compared to classical cryptography. Nevertheless, Eve is ultimately limited by the so-called no-cloning theorem of quantum mechanics. According to the no-cloning theorem, Eve cannot deterministically intercept a qubit and accurately resend it to its destination[15].

Regarding the information leaked on the classical channel, the rule is that every key bit or bit calculated using the key, posted on the channel is equivalent to a single bit of information exposed to Eve[4,5]. This excludes communication that might be used to facilitate the connection itself. To maintain the secrecy of the key, by the end of the process we will discard the number of bits exposed, meaning that every bit transmission is equivalent to the consumption of a key bit.

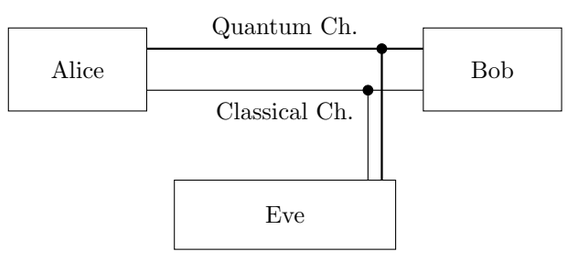



Figure 1: QKD system setup. Alice and Bob, communicating via exposed quantum and classical channel.

### 3.3 Possible Attacks

Various studies have been made on attack schemes targeting different QKD applications[6,7]. Broadly dividing the different schemes by the ability of the attacker, these are the main types:

1. **Individual Attacks:** Eve is interacting with each of the qubits independently. Such interaction includes preparation of ancillas, use of quantum gates and measurements, all done on each qubit separately. Intercept-resend[15] is an example of this type of attacks.
2. **Collective Attacks:** Eve prepares ancillas and operates on each qubit separately, but may measure several qubits together.
3. **Coherent Attacks:** Eve may interact and measure several qubits together.

It is clear that each attack type contains the previous. Individual attacks are weaker while coherent attacks are not stronger than collective[6,7].

## 4 BB84

The first QKD protocol suggested was BB84[16]. Suggested by Bennett and Brassard in 1984, it is an information theoretic secure protocol, particularly secured against a quantum adversary. We will use this protocol as a base for understanding QKD in general. The frame of this scheme, the two distinct phases and the inner stages of post-processing, are common to different QKD schemes. The differences are in specific variations and implementation details. Though the frame is similar, the implementation affects the security and feasibility of the system.
The protocol is divided into two distinct phases as follows, which are as said common to other QKD schemes.

### 4.1 The Quantum Phase

Alice sends Bob $N$ qubits over the quantum channel. For every qubit sent, independently and repeatedly, Alice makes two coin flips. The first determines if she should send a bit value of 0 or 1, and the second determines if the encoding basis will be the up-down basis or the plus-minus basis. Receiving each qubit, Bob randomly chooses one of the two bases for its measurement. If Bob chooses the same basis that Alice used to encode the bits, they will hold the same bit value. Otherwise, there is a probability of 0.5 that they will hold the same value. Over the classical channel, Bob tells Alice which basis was used for each qubit measurement. Alice in turn replies stating which of measurements Bob has done with the same basis used for encoding, and both discard every bit disagreed on. The result is what is called a sifted key. A quick calculation will show that on average $N/2$ bits will be discarded at the sifting stage.

### 4.2 Post-Processing (Classical) Phase

At the end of the quantum phase Alice and Bob are left with bits known to be encoded and measured in the same basis, but there is no guaranty that the bits are with equal values. A bit that was sent by Alice could be altered during transmission, as a result of malicious intervention, or noise over the quantum channel. That said, we would like to perform an error correction between the sifted keys. Additionally, we need to compensate for any information that may have been leaked to Eve during the protocol flow. The compensation is done by discarding bits from the generated key. The main subject of this work is the error reconciliation process mentioned below.

**Error Estimation** The classical phase starts with an estimation of the discrepancies between the two sifted keys on both sides, both equal in length. Notating the sifted keys for Alice and Bob as $K^a_{sift}, K^b_{sift} \in \{0,1\}^*$ correspondingly, an error is defined as a case in which for some index $i$, the $K^a_{sift}[i] \neq K^b_{sift}[i]$. The error rate between the keys, or the Quantum Error Bit Rate ($q_{ber}$) will be defined as

$$q_{ber} = \frac{|\{i : K^a_{sift}[i] \neq K^b_{sift}[i]\}|}{|K^a_{sift}|} \tag{4}$$

The error estimation stage is crucial for monitoring the information leakage since errors hint that Eve is intercepting the information. An example of an attack that could be discovered via an increased error rate would be the mentioned intercept-resent attack[15]. In this scenario, Eve measures a qubit sent by Alice in either of the said measurement bases. Eve sends that bit to Bob, encoded with the same basis used for measurement. Bob will then measure the qubit sent by Eve in either basis, unaware of the interception.

Since each of the three acting parties could have chosen to encode or measure in the UD basis or the PM basis, there are eight possible combinations. Assuming that none of the parties can know of the others' choice, and assuming that the choice of basis is done with probability 0.5, the combinations are equally likely. Let us examine the possibilities. For a given qubit, if Alice and Bob chose different bases, the resulting bit would be removed in the preceding sifting stage. Thus, a qubit that was intercepted by Eve would be included in the error estimation stage if Alice and Bob chose the same basis, thus cutting half of the combinations. If the choice of bases is identical for all parties, Eve would accurately measure and resend the value sent by Alice, resulting in no error. If Eve chose a different basis from Alice and Bob, there is a 0.5 probability that the bit value measured by Bob will be erroneous. For example, when Alice sends $|\uparrow\rangle$, if Eve measures the qubit in PM basis she will equally likely receive 0 and 1. This translates to an equal probability of forwarding $|+\rangle$ and $|-\rangle$ to Bob. Measuring in the UD basis, regardless of which of the PM vectors Eve sent, Bob will measure $|\uparrow\rangle$ and $|\downarrow\rangle$ with equal probabilities, receiving values of 0 and 1 correspondingly. Since the original value of that bit is 0, there is a probability of 0.5 for that bit value to be erroneous. This analysis holds for the other cases where Alice sends a qubit which measured by Eve in the wrong basis, all cases being equally likely. Concluding, if Alice and Bob chose a different basis from Eve, there is a 0.5 probability of discrepancy. Out of the possibilities that survive the sifting, there is a possibility of 0.5 for all parties to use the same basis, resulting in no error, and 0.5 probability of Eve choosing the wrong basis, resulting in a 0.5 probability of error. In total, this is a 0.25 probability of error. Additionally, this analysis reveals that Eve has a probability of 0.75 to read accurately the value sent by Alice. We will regard the number of bits that Eve has discovered as $I_{EVE}$.

For this specific intercept-resend scheme, if Eve would intercept all of the qubits, we would see an

estimation of $q_{ber} = 0.25$ , and $I_{EVE} = 0.75|K^a_{sift}|$. Assuming the worst, that all of the errors are result of malicious activity, the calculation would be that $I_{EVE} = 3q_{ber}|K^a_{sift}|$. There are many types of possible attacks, and the intercept-resend is one of the simplest. The relationship between the error rate and the revealed information might be more complicated, and $I_{EVE}$ may be bigger than in this example. Nevertheless, there is a correlation between attacks and the error rate, and we must set a threshold of $q_{ber}$ over which the information loss is considered too high. In that case, we discard the sifted keys, and start over. Considering additional scenarios which we have not mentioned, this threshold is chosen to be $q_{ber} = 0.11$[5].

We now need to find a way to estimate the error as accurately as possible. Since comparing all the bits on the classical channel will break the secrecy, we will assume that the estimation will not be exact. One method would be for Alice and Bob to reveal an agreed section of the key, each comparing the other's sample to their own. This sample will serve as an estimation of the general $q_{ber}$, at the expense of the key bits produced. Instead of or together with this method, one could use previous runs of the algorithm to estimate the error rate. There is some work on this subject, suggesting and comparing methods[32], but no method currently holds a clear advantage over the rest.

**Error Reconciliation** Regardless of the origin of the error, we would like to correct the discrepancies between the sifted keys. To perform this task, we turn to the field of error correction. The problem at hand is slightly different from the common error correction case. One such difference is an inability to add parity bits before transmitting information. Usually, the sender adds information to the message, and the receiver uses the redundant information to identify and correct errors, extracting the original message. An assumption made here is that the length of the sent and received message is not changing. In the QKD case, we cannot add redundancy to the original sent qubits. This is because of two reasons: the first being that we cannot tell which and how many of the bits will end up in the sifted keys, and the second being that adding dependencies between qubits will likely reveal information.

An adjustment done for QKD is providing a way to create such redundancy. We will do so by revealing bits over the classical channel, using the revealed information to correct the errors. Mathematically, the requirement for redundant bits originates in the Noisy-Channel coding theorem. The theorem states a theoretical limit to the rate of information that can be successfully transmitted through a noisy channel. For QKD, the theorem sets that the minimal rate of redundant bits in the sifted keys as $\eta_b(q_{ber})$[1,33,34], the $\eta_b$ function being the binary entropy function

$$\eta_b(q) = -qlog_2(q) - (1-q)log_2(1-q) \qquad (5)$$

This number is the percent of bits that we need to consume. Calculating for the conjectured error rates, for $q_{ber} = 1\%$ we consume 0.08 of the bits, while for $q_{ber} = 10\%$ this number will rise to 0.47. Another key difference between the QKD error reconciliation case and the typical case is the error rate. In QKD, current technology produces an error rate on the scale of $1\% - 10\%$. Classical communication is much less noisy - for comparison, USB specification[8] require BER of less than $10^{-12}$.

A number of algorithms[1,20-31] and different variations of them has been suggested to solve this problem. The most prominent of these are Cascade[20], and LDPC based algorithms[1,24-31]. In this work we use the Symmetric Blind Information Reconciliation algorithm, an LDPC based algorithm that will be explained in Section 8.1.

**Error Verification** Existing error correction algorithms are capable of converging to a wrong correction result. This can happen when the algorithm corrects Bob's key in a way that answers the required condition, usually some parity check result, but this correction is not Alice's real key. Error reconciliation algorithms are designed to make this case improbable, but it is still possible.

To overcome this challenge, one can use a universal family of hash functions[1,35,36]. Each of Alice and Bob chooses a function and calculates the hashing result on its key. They will compare the result, and if it is identical they will consider the keys identical. Else, they will consider the reconciliation failed and discard the keys. Since the hash functions have a small probability of collision, this step reduces the chance of a false positive correction result, and it does it at the expense of additional bit sacrifice.

**Privacy Amplification** After the error correction, Alice and Bob hold identical keys, but Eve may have acquired information about it. They would transform the keys in a way that would lower Eve's information to bellow a negligible amount. This can be done by invoking a hash function[5], where the probability to have any knowledge of the output value without first knowing a significant portion of the input is negligible. The length of the result will take into account the bits shared over the classical channel, the information that is potentially available to Eve, and may include other parameters to set the security standard to a desired level.

The result of this stage is the final secret key.

**Message Authentication over the Classical Channel** During the post-processing phase, we use the classical channel, and to use it we must have a way to differentiate messages sent by Alice and Bob from messages sent by Eve. This is a problem existing in classical cryptography and is usually solved by adding some hard-to-guess tag to each message. Alice and Bob share some secret key needed to produce the tag as a function of the original message, while Eve has a negligible probability of guessing a tag.

This authentication problem must be addressed to ensure valid communication between Alice and Bob, requiring a secret key to be dedicated to that task. Work on that subject has been done in[5], where it is shown that the secret key used for authentication is consumed at a low rate.

# 5 Coherent One Way QKD Setup

Many of the demonstrations of BB84 systems are based on optical fibers. These systems are limited in range mainly due to noise[10]. A different implementation of QKD will be the Coherent One Way

(COW) setup[9], encoding the bits in the arrival times of weak coherent pulses. This method lowers the amount of noise in the system, but it is not proven secure against all attack types[12,13]. Using a less noisy method influence not only the error rate but also allows for transmission over longer distances[10].

In this scheme, illustrated in Figure 2, Alice holds a continuous wave laser, placing an intensity modulator at its end. This beam outputs either a pulse with a mean number of photons $\mu < 1$, or nothing at all. Dividing the timeline into periods of pulse length, sending a pulse in the $k$-th time unit would be a state $|\mu\rangle_k$, while no transmission is $|0\rangle_k$ - this is encoding using time basis.

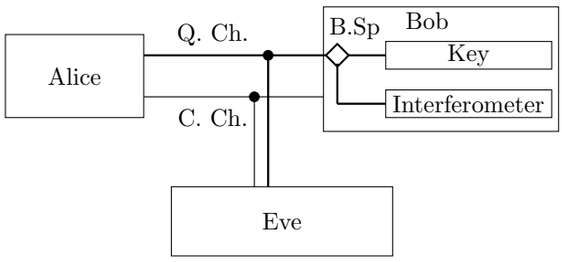



Figure 2: QKD COW system setup. Alice sends coherent pulses of laser encoded by different transmission times. At the side of Bob, the pulses are split between a photon detector (Key), and an interferometer detecting coherency errors between successive pulses.

Alice uses pairs of these states to encode a single qubit, sending one of these three states as the $k$-th logical unit

$$|\alpha_0\rangle_k := |\mu\rangle_{2k-1} |0\rangle_{2k} \text{ , a zero qubit} \quad (6)$$

$$|\alpha_1\rangle_k := |0\rangle_{2k-1} |\mu\rangle_{2k} \text{ , a one qubit} \quad (7)$$

$$|\alpha_d\rangle_k := |\mu\rangle_{2k-1} |\mu\rangle_{2k} \text{ , decoy state} \quad (8)$$

Bob holds a beamsplitter, directing $0 \leq t \leq 1$ of the qubits to a interferometer, and $1-t$ to a photon detector (key detector). The interferometer is checking phase coherency between two successive $|\mu\rangle$ pulses, that appears inside either of $|\alpha_1\rangle |\alpha_0\rangle$, $|\alpha_d\rangle$, $|\alpha_1\rangle |\alpha_d\rangle$, $|\alpha_d\rangle |\alpha_d\rangle$, $|\alpha_d\rangle |\alpha_0\rangle$. It has been shown that Eve cannot count the photons in any finite number of pulses without creating errors[11], that are identified by the interferometer.

The protocol is as follows:

1. Alice sends a long sequence of states to Bob, sending $|\alpha_d\rangle$ with probability $p > 0$, and $|\alpha_0\rangle$, $|\alpha_1\rangle$ with probability $\frac{1-p}{2}$ each.
2. Bob shares with Alice which qubit indices were measured in the key detector, and which were measured by the interferometer and identified with broken coherency.
3. Alice tells Bob which of the bits measured in the key detectors were decoy states, which are discarded. This is the COW sifting process.
4. Alice analyzes the coherency errors reported by Bob and evaluates Eve's information.
5. Now holding a sifted key and $q_{ber}$ estimation, Alice and Bob continue to the post-processing phase. This post-processing phase is identical to the one presented in Section 4.2.

# 6 Low Density Parity Check Codes

Low Density Parity Check (LDPC) codes are a family of parity check codes, unique by having sparse parity check matrices. Invented by Gallager in 1962[14], they were suggested for QKD error reconciliation[31] due to typically lower bit consumption and low number of communication rounds used in the decoding process.

$$\begin{bmatrix} 1 & 0 & \mathbf{1} & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

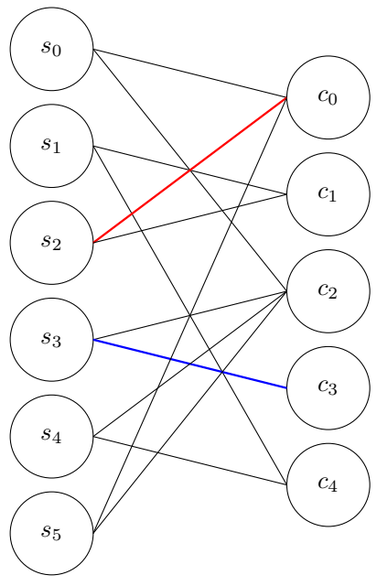



Figure 3: A parity check matrix and the equivalent Tanner graph. Nodes on the left are symbol nodes, and on the right are check nodes. The red and blue pairs of edge and cell are corresponding to each other.

**Introduction to Parity Check Codes** We will start with a description of the common approach to error correction codes. Alice holds a message, that she would like to transfer to Bob via a noisy channel. The original message is a string of 0s and 1s of length $l$, so we will refer to it as vector $x \in \{0,1\}^l$, while the message received by Bob will be $y \in \{0,1\}^l$. Addition is defined as logical XOR, with regular multiplication. The term for a vector space of messages is code, and the term for a vector in it is a code word. The noisy channel will be modeled as a binary symmetric channel with probability $p$ of a bit flipping. Assuming that errors on the channel are completely independent, the probability for $k$ bit flips is $p^k$. The difference between any two vectors will be defined using the Hamming distance, the number of indices with non-equal values.

Any vector of length $l$ might be a message sent to Bob, so when receiving a message containing an error he has no way of knowing that an error even occurred, let alone correct it. To overcome this, we would like to divide the messages to "good" and "bad" sets. Alice would only send good messages,

so if Bob receives a bad message, he can tell that there was an error. Given that we must be able to send at least $|\{0,1\}^l|$ messages, to have such division we would have to enlarge our messages space. For some decided $l < r \in \mathbb{N}$ we will inject the message space into a subspace $V \subsetneq \{0,1\}^r$, $dim(V) = l$. After finishing the correction phase we can return to the original $l$ length messages space.

As a linear space of dimension $l$, $V$ is the kernel of some $(r-l) \times r$ matrix $H$, with rank $(r-l)$. Therefore for each $v \in V$, $Hv = 0$, and $Hu \neq 0$ for $u \in \{0,1\}^r \setminus V$. Such a matrix is called a parity check matrix since it is equivalent to a set of parity checks. A parity check is the term for checking whether the number of 1s in some subset $J \subseteq [l]$ of indices in a message is odd or even, returning 0 for even and 1 for odd. This is equivalent to performing the calculation of $\bigoplus_{j \in J} v_j$ (XOR of all bits in indices $J$). When constrained to 0, this calculation is a linear equation. Altogether, we have $(r-l)$ parity check equations, when $H_{ji} = 1$ means that the i-th bit in a message is participating in the j-th parity check. The product of multiplying a vector with the parity check matrix is called a syndrome. The rate of a parity check matrix of size $(r-l) \times r$ is defined as the amount of a usable code in a transmission, $R = \frac{l}{r} = 1 - \frac{row\ number}{column\ number}$.

Assuming that Alice sends $x \in V$ to Bob, which received a possibly altered $y$, Bob can multiply $Hy$ to know if this message is in $V$. The hard work is to build $V$ in such a way that it is unlikely that a $y$ is closer to some $z \neq x$ in $V$ then it is close to $x$ itself. This will allow to uniquely identify the erroneous $y$ with $x$. According to the bit flip probability $p$, Alice and Bob will decide some threshold of $k$ bit flips that they would like to recover from. They need to make sure that there is only one word in $V$ within a distance of $k$ from any vector in $\{0,1\}^r$, so the minimal distance between two codewords in $V$ must be at least $2k$. This is called the minimal distance of $V$. After recovering the original $x \in V$, Bob can return the $r$ length message to the original $l$ length, concluding the transmission.

**LDPC Codes** These are a special case of the parity check codes, unique by having a sparse parity check matrix, meaning it has a significantly low amount of non-zero values comparing to the matrix dimensions. This is desirable since the decoding algorithm complexity is determined by the amount of such non-zero elements. A parity check matrix of an LDPC code will be called an LDPC matrix. An LDPC matrix in which all rows has the same number of 1s and all columns has the same number of 1s is called regular, and otherwise will be called irregular.

We can also represent such a matrix using a bipartite graph, called a Tanner graph. Given a parity check matrix $H$, with dimensions $m \times n$, the corresponding Tanner graph G will be defined as follows: For every bit of a message, which we will now refer as symbol, we will have a vertex in G - a symbol node. Similarly, we will have a vertex for each parity check, these will be called check nodes. We will have an edge between the i-th symbol node to the j-th check node iff $H_{ji} = 1$. It is not hard to see that this representation is equivalent and unique. In Figure 3 one can see a parity matrix and the equivalent Tanner graph.

Looking at the graph, we will define its girth as the length of its shortest non-trivial cycle. A cycle is a sequence of edges connecting vertices along the graph when the first and last vertex are the same and no other vertex is repeated twice. In Figure 3, we can identify the cycle of length 4, $s_0 \to c_0 \to s_5 \to c_2 \to s_0$. A nontrivial cycle must have at least two symbol nodes and two check nodes - so this cycle is minimal. Thus, the girth of the graph is 4. We will later see that bigger girth is improving the decoding process[17], making this property desirable. Finally, we will define the degree of a node as the number of edges connected to it. For a symbol node, the degree will be the number of parity checks it participates in, and for a check node, it will be the number of symbols participating in it.

## 6.1 Belief Propagation

LDPC codes are decoded using the Belief Propagation, or Sum-Product, algorithm[1,30]. This algorithm is based on message passing along the edges of a Tanner graph when the final result is the log-likelihood ratio (LLR) of each bit of the key. In this context, a message is a real number sent between the parties, used for the log-likelihood ratio calculation. The LLR of a given bit is defined as the log of ratios between the probability of being 0 to the probability of being 1. For a bit represented by a random variable X

$$LLR(X) = log\frac{P(X=0)}{P(X=1)} \tag{9}$$

One can see that higher probability of a bit being 0 (1) is associated with $LLR > 0$ ($LLR < 0$), while complete uncertainty is $LLR = 0$. The higher the absolute value of $LLR$ for a bit, the higher our certainty in the value of that bit. The specific calculation performed is explained in Section 8.2, but the general flow is as follows: The algorithm receives a vector $x$ that is to be corrected, and the goal is to change it to have a syndrome $s$ relative to some LDPC matrix $H$. Each symbol node is assigned an initial LLR value, not dependant on other nodes. These LLR values are calculated using the bit flip probability and the input key value for that symbol. Following the initialization, each check node receives a message from each symbol node connected to it, containing the LLR value of the symbol node. After all symbol-to-check nodes are transferred, each check node uses the received information to create a message of its own, sending it back to the symbol nodes. Ending the iteration, each symbol node uses the received messages to reevaluate its LLR.

According to specific needs and defined stopping conditions, we will repeat the messaging process until a defined condition is reached. For example, the condition may be reaching some number of messaging iterations, or stabilization of the LLR values. At the end of the process, a presumably correct bit value will be constructed using the $LLR$ values. A bit with $LLR \geq 0$ is determined to be 0 and with $LLR < 0$ is determined to be 1. Determining the value for all bits, we create a corrected key $y$. The algorithm is successful if $Hy = s$ and fails otherwise. Upon success, $y$ is returned, and upon failure, it is discarded. One should note that success is defined as converge to some key value with the desired syndrome, but is not necessarily the original sent key. This version of the algorithm does not guaranty success, but in Section 7.2 we will see an improvement to it which

does.

### 6.2 Progressive Edge Growth

The Progressive Edge Growth (PEG) algorithm[17,18], is an algorithm for creating LDPC matrices. It receives as input the number of symbols, the number of parity checks, and the desired degree of each symbol node, and outputs a Tanner graph applying these characteristics. The algorithm is greedy in the sense of starting from an edge-less Tanner graph and adding edges in multiple iterations, each creating an edge with maximal possible distance. A pseudo code for the algorithm can be seen in Algorithm 1.

**Algorithm 1:** Progressive Edge Growth

```
// argmax_{a∈A}{f(a)} := {b ∈ A | f(b) = max_{a∈A} f(a)}
// argmin_{a∈A}{f(a)} := {b ∈ A | f(b) = min_{a∈A} f(a)}
// The ← sign means assignment.
Function PEG (N, M, Deg[N]) is
    Input  : N - desired column number.
             M - desired row number.
             Deg[N] - Desired degree per
             symbol.
    Output: G =< V_s, V_c, E > - A Tanner
             graph.

    V_s ← (v_k^s)_{k=0}^{N}
    V_c ← (v_k^c)_{k=0}^{M}
    E ← ∅
    for i ← 0 to N − 1 do
        for j ← 0 to Deg[i] − 1 do
            E ← E ∪ {getFarthestCheck(v_i^s)}
        end
    end
    return G =< V_s, V_c, E >
endFunction

Function getFarthestCheck (v^s) is
    nrch ← {v^c ∈ V_c : v^c unreachable from v^s}
    if nrch ≠ ∅ then
        minDeg ← argmin_{v^c∈nrch}{deg(v^c)}
    else
        maxDist ← argmax_{v^c∈V_c}{dist(v^c, v^s)}
        minDeg ← argmin_{v^c∈maxDist}{deg(v^c)}
    end
    return arbitrary v^c ∈ minDeg
endFunction

Function dist (u_1, u_2) is
    return length of shortest path between
     u_1, u_2
endFunction

Function deg (u) is
    return degree of vertex u
endFunction
```

## 7 Error Reconciliation Schemes

### 7.1 QKD Error Correction Using LDPC

As said, the problem we encounter in QKD is different from the regular problem solved with error correction codes, and the flow is slightly different.

**QKD Error Reconciliation Flow** Let be LDPC matrix $H$ of size $m \times n$. Alice and Bob have the sifted keys $K^a_{sift}, K^b_{sift} \in \{0,1\}^n$. We will notate the syndromes of Alice and Bob as $s_a = HK^a_{sift}$ and $s_b = HK^b_{sift}$, both in $\{0,1\}^m$. At this point the QKD scheme diverges from the typical case, since there is no guaranty that either $s_a$ or $s_b$ is 0. The goal now is to correct the key held by Bob such that its syndrom will be equal to $s_a$.

The decoding will begin with Alice sending her syndrome to Bob, which will now hold both syndromes. To maintain secrecy, this communication requires sacrificing the number of sent bits later on. Bob would use belief propagation to correct his key such that $s_b = s_a$, or $HK^a_{sift} = HK^b_{sift}$.

### 7.2 Rate Adaptive Information Reconciliation

**Rate Adaptive Information Reconciliation** We will now see a variation on LDPC based information reconciliation seen at Section 7.1, the Rate Adaptive Information Reconciliation[24-28]. The rate adaptive scheme is a way of optimizing bit consumption when correcting a sifted key using LDPC, by replacing key bits with auxiliary bits and changing the error rate of the key.

Given a sifted key of length $n$ with $q_{ber}$, the noisy channel theorem states that we must use a parity matrix with a row number of at least $m_{min} = \lceil \eta_b(q_{ber})n \rceil$. Creating the LDPC matrices is a relatively lengthy process, making it inadequate to create on-the-fly. But on the other hand, the $q_{ber}$ cannot be anticipated precisely. This understanding requires us to hold an already prepared set of matrices, with limited row number resolution. We would define a metric that would quantify our distance from the theoretic boundary: given a parity matrix $H$ of size $m \times n$ the efficiency of a information reconciliation process is defined as

$$f = \frac{\text{consumed bits}}{\eta_b(q_{ber}) \cdot \text{key length}} \tag{10}$$

or in the LDPC case.

$$f = \frac{m}{n\eta_b(q_{ber})} \tag{11}$$

The optimal case is $f = 1$, and key length is considered a constant for all practical use.

Taking the same sifted key, we would now search the matrix pool and pick the matrix with the smallest row number $m$ such that $m \geq m_{min}$. We would now consume $m - m_{min}$ bits more that we could theoretically, resulting in $f > 1$. The opposite scenario is when the available row number is too low for a given error rate. These two cases invites a way of adjusting the rate of an existing matrix on-the-fly. The main idea will be to add or remove discrepancies in the key, therefore adjusting the efficiency by need. Adjustment of efficiency is done with two complementing methods called puncturing and shortening. Puncturing is the process of replacing $p$ corresponding bits in the sifted keys with bits chosen randomly, without coordination between parties. This probability to create an error is 0.5. For an index set $P$ of size $p$, the output would be $K^a_{sift}{}', K^b_{sift}{}'$, when for either parties

$$K'_{sift}[i] = \begin{cases} K_{sift}[i] & i \in P \\ \text{random bit} & i \notin P \end{cases} \tag{12}$$

Shortening is done similarly by removing bits and replacing them with a coordinated value, thus reducing the discrepancies. This value can be conveniently chosen as 0 for all shortened bits, but theoretically the values can be chosen otherwise. No matter what the choice is, the shortened values are assumed to be known by Eve. For a similar set $S$ of $s$ indices

$$K'_{sift}[j] = \begin{cases} K_{sift}[j] & j \in S \\ 0 & j \notin S \end{cases} \tag{13}$$

In both of the puncturing and shortening cases, the original bit values replaced by the punctured or shortened ones are usable and kept by both parties for the next processing iteration. This seemingly reduces the output of the algorithm, since those bits are postponed to the next iteration, but the alternative is to consume a similar amount of bits due to inefficiency. Finally, at the end of the reconciliation process, the added bits are removed, leaving Alice and Bob only with bits from the original sifted keys. Notice that it is not prohibited to invoke both methods on the same key, but since they are addressing opposite scenarios this use is obscure.

When considering the consumption of bits due to information leakage, the punctured bits are not counted as leaked, while the shortened bits are counted[25]. Assuming the same matrix and dimensions and a sifted key where $p$ bits have been punctured and $s$ were shortened, the efficiency will be

$$f = \frac{m-p}{(n-p-s)\eta_b(q_{ber})} \tag{14}$$

The change in the denominator comes from reducing the length of the key being effectively transmitted.

**Blind Reconciliation** Inspired by the hybrid automatic repeat request method, the Blind Reconciliation method[26] is an enhancement of the rate adaptive scheme ensuring the decoding process will converge. The price is additional revealed bits.

With or without using the rate-adaptive scheme, so far if the decoding algorithm has not converged to a correct result, the sifted key is discarded. A possible way to avoid the loss of the key is that Alice would reveal additional key bits to Bob, making the task of recovering her key slightly easier. Bob will now perform the belief propagation decoding from the start with more information on the key, which just might help him to converge to a correct result. If the decoding still does not converge, Alice just reveals more information, until either the decoding succeeds or the entire key is revealed. An additional cost to consider is the increase in communication between Alice and Bob. As we see in Section 8.3, part of the initial choice of LDPC as a reconciliation scheme was that it is less interactive than the proposed Cascade method. The introduction of bit revealing rounds slightly changes the balance, but the Blind Reconciliation scheme is still less interactive than the Cascade[1].

For efficiency calculation, the additional revealing rounds are considered to expose key information, and it is added to the count of consumed bits. Assuming Alice and Bob have implemented the rate-adaptive scheme with $p_0$ and $s_0$ punctured and short bits and revealed additional $n_{add}$ bits in every iteration for $d$ iterations, the efficiency would be

$$f = \frac{m-p_0+n_{add}d}{(n-p_0-s_0)\eta_b(q_{ber})} \tag{15}$$

# 8 Symmetric Blind Information Reconciliation

Looking at the Blind Reconciliation scheme, a point that is not addressed is which bits should be revealed in each revealing round. A good heuristic would be that Alice should reveal the bits that Bob is less sure about their correct value. This information is contained within the $LLR$ values, and if Alice had held these values she could have revealed the bits more wisely.

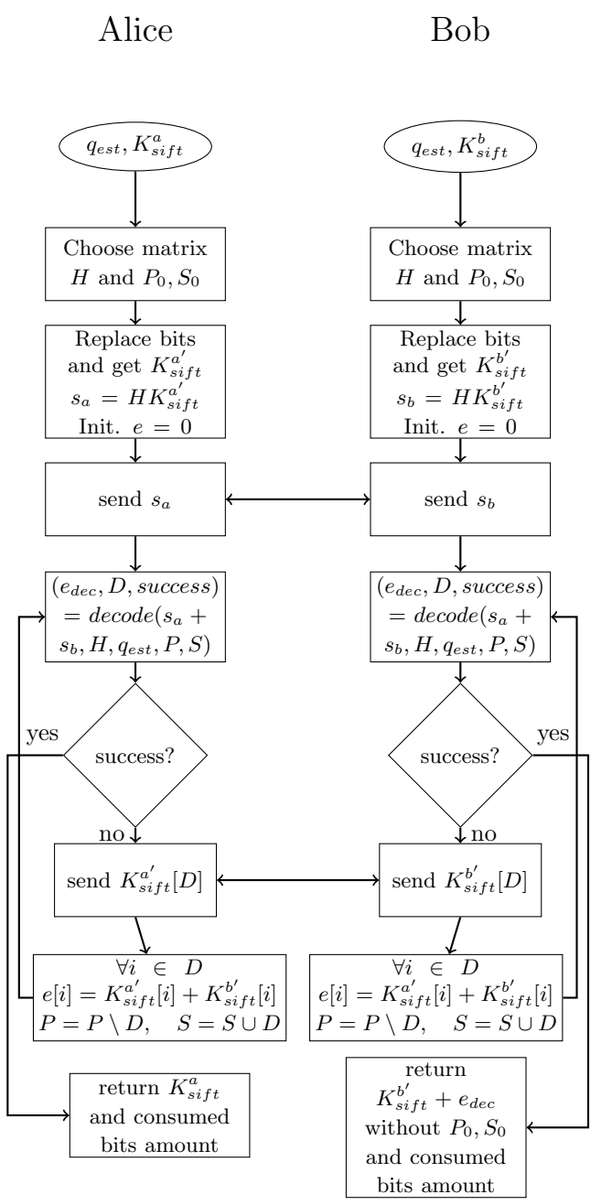



Figure 4: The Symmetric Blind Information Reconciliation flow[1]. Each of the parties holds its own sifted key, an independently calculated $q_{ber}$ estimation, and an LDPC matrices pool identical for both sides. By exchanging syndromes and additional bits, each side corrects the discrepancies between the keys independently. The output would be identical keys on both sides. Bob corrects his key according to the decoding result, and Alice simply removes the shortened and punctured bits from her original key.

The Symmetric Blind Information Reconciliation scheme provides that functionality by making

both Alice and Bob try to decode the sifted key together, arriving at the same results. Upon failure, both parties will reveal to each other the bits with least certain values. This symmetry allows Alice to know the LLR values, but it demands that both parties will act both the role of the decoder and the sender of bits. This doubled exchange of information is shown to reveal Eve the same amount of information[1], thus not harming the efficiency of the Blind scheme. The added value should be embodied in fewer rounds of bit revealing, resulting in improved efficiency. But, A price is paid by having both sides perform the same calculations. This price is arguably reasonable, considering that the priority is mostly to consume fewer key bits. The processing time is not expected to significantly increase, since Bob was already calculating the same calculation while Alice waits, meaning that this enhancement replaces the wait time with calculations. The algorithm flow is be described in Figure 4.

## 8.1 Symmetric Blind Scheme Flow

Each party holds a sifted key $K^a_{sift}$ or $K^b_{sift}$, an estimated error rate $q_{est}$, and an access to an array of LDPC matrices. They would now decode the keys simultaneously. The decoding flow, listed in Figure 4, is as follows. First, based on the error rate, each party will choose an $m \times n$ matrix $H$ from the array with the efficiency $f$ closest to 1. Afterwards they will puncture or shorten bits in their sifted keys according to need. Considering that the methods are complementary, the punctured and shortened sets $P, S$ will be of size:

$$|P| = \begin{cases} \lfloor \frac{m - n\eta_b(q_{est})}{1-\eta_b(q_{est})} \rfloor & f > 1 \\ 0 & f \leq 1 \end{cases} \tag{16}$$

$$|S| = \begin{cases} 0 & f \geq 1 \\ \lceil n - \frac{m}{\eta_b(q_{est})} \rceil & f < 1 \end{cases} \tag{17}$$

The $P, S$ bit locations are chosen randomly in our implementation, though there is a method claiming to perform better at choosing locations for puncturing[29]. To perform a coordinated random choice, Alice and Bob must own coordinated random number generators. By the end of this calculation, both parties have the same value of $P, S$.

Each of Alice and Bob replaces the chosen $P, S$ bits in the sifted key with punctured and shortened bits, creating $K^{a'}_{sift}$ at Alice's side and $K^{b'}_{sift}$ at Bob's. We will store $S_0 := S$, and $P_0 := P$ for later use.

Next, each side will calculate its syndrome and send to the other.

$$s_a = HK^{a'}_{sift} \quad , \quad s_b = HK^{b'}_{sift} \tag{18}$$

Now, both parties hold both $s_a, s_b$, and we can start decoding. Notating $K_{diff} := K^{a'}_{sift} + K^{b'}_{sift}$, we would like to find this $K_{diff}$. In previously mentioned schemes, when decoding only on Bob's side we would give the belief propagation algorithm Bob's key and Alice's syndrome. Then, it will try and find $K_{diff}$ such that $s_a = H(K^{b'}_{sift} + K_{diff})$. But in the symmetric scheme, both parties must perform the calculation, and they must do it the same to eventually reveal the same bits to each other. Therefore, we must only rely on syndromes on both sides. Moving Bob's syndrome on the last equation, we would like to find $K_{diff}$ such that

$$H(0 + K_{diff}) = s_a + s_b \tag{19}$$

Following this intuition, the input key handed to the belief propagation function would be a vector $e \in \{0,1\}^n$, the error vector, which will be initialized as $0^n$. We would like the algorithm to correct $e$ to a vector with syndrom $s_a + s_b$, the goal being to converge to the unknown $K_{diff}$.

Both sides will attempt to converge to a solution key with syndrome equal to $s_a + s_b$. **In case of decoding success**, Alice and Bob will hold $e_{dec}$ with the desired syndrome value. Bob will calculate

$$K^b_{corrected} := K^{b'}_{sift} + e_{dec} \tag{20}$$

when $HK^b_{corrected} = s_a$, and is perceived as equal to $K^{a'}_{sift}$. Alice and Bob will then remove the bits at $S_0, P_0$, arriving to the original $K^a_{sift}$. **In case of decoding failure**, both Alice and Bob will have identical set of indices with least LLR certainty, $D$. The calculation could be seen at length in Section 8.2. Both would send each other the bits $K^{a'}_{sift}[D], K^{b'}_{sift}[D]$, and will update the error vector

$$\forall i \in D \quad e[i] = K^{a'}_{sift}[i] + K^{b'}_{sift}[i] \tag{21}$$

The revealed bits are now updated as shortened, $S = S \cup D$, and removed from the punctured set - $P = P \setminus D$. The parties will now repeat the same decoding process with the updated $e$, until succeeding or until the entire key is revealed.

Once finding an equal key on both sides (if we revealed the entire key it is done trivially), both sides will hold it and the number of bits consumed in the process. Those bits would be removed in the privacy amplification step.

## 8.2 Belief Propagation Algorithm Implementation in the Symmetric Blind Scheme

Explained in general lines in Section 6.1, we will now address the specific implementation of the Belief Propagation algorithm presented in[1,30]. This implementation takes into account the difference arising from introducing the punctured and shortened bits, and the possibility of bits revealed. The message sent from the i-th symbol node to the j-th is notated as $M_{i \to j}$, and the message sent from that same check to node to the symbol node is $M_{i \leftarrow j}$. Notice that messages exist only between nodes connected by an edge.

1. The algorithm receives a Tanner graph representing an $m \times n$ LDPC matrix $H$, an error vector $e \in \{0,1\}^n$, desired syndrom $s$, punctured and shortened indices sets $P, S$ ,and a $q_{ber}$ estimated value. Non punctured / shortened bits set will be marked as $K := [n] \setminus (P \cup S)$. The edge relation of the graph $E(i,j)$, equals 1 iff the i-th symbol node is connected to the j-th check node. The messages sent between the parties are real numbers used for LLR calculation.

2. The initial LLR value of the i-th bit is

$$r^{(0)}[i] := \begin{cases} (-1)^{e[i]} log\frac{1-q_{ber}}{q_{ber}} & i \in K \\ (-1)^{e[i]} r_s & i \in S \\ 0 & i \in P \end{cases} \tag{22}$$

Where $r_s \gg 1$. These values represents the current knowledge we have on the bits: We know the values of the shortened bits, and do

not know the values of the punctured. The original key bits are evaluated according to the possibility of a bit flip.

For all $i, j$ define $M^{(0)}_{i\leftarrow j} = 0$.
Initialize $k = 1$, the iteration number.

3. For every symbol node $i$, send messages to the connected check nodes.

$$M^{(k)}_{i\to j} = r^{(k-1)}[i] - M^{(k-1)}_{i\leftarrow j} \tag{23}$$

4. For every check node $j$, send messages to the connected symbol nodes.

$$M^{(k)}_{i\leftarrow j} = 2tanh^{-1}\Big[\prod_{i' \text{ s.t } E(i',j)} tanh\frac{M^{(k)}_{i'\to j}}{2}\Big](-1)^{s[j]} \tag{24}$$

5. Re evaluate the LLR for each symbol node $i$.

$$r^{(k)}[i] = r^{(0)}[i] + \sum_{j \text{ s.t } E(i,j)} M^{(k)}_{i\leftarrow j} \tag{25}$$

6. Decide the values of the key based on $r^{(k)}$

$$K^{(k)}_{est} := \begin{cases} 0 & r^{(k)}[i] \geq 0 \\ 1 & r^{(k)}[i] < 0 \end{cases} \tag{26}$$

7. Calculate the syndrome for the estimated key

$$s^{(k)} = HK^{(k)}_{est} \tag{27}$$

If $s^{(k)} = s$ return $K^{(k)}_{est}$ with success. Else, continue to the next stage.

8. The next step is to calculate the stopping condition. We will stop if we reached some configured maximal iteration $k_{max}$ or if the LLR certainty is stabilized to a degree. To check if the LLR values are stable we will calculate the average certainty on non-shortened symbols

$$\hat{r}^{(k)} := \frac{1}{n - |S|}\sum_{i\in K\cup P} |r^{(k)}[i]| \tag{28}$$

Shortened values are omitted since their values are known, and their LLR values are relatively high and might overshadow the rest of the values. The next stage is to check if the certainty level of the k-th iteration is smaller than the average last $l$ iterations, for some parameter $l$. If the following condition is met, we will halt the decoding algorithm.

$$\hat{r}^{(k)} \leq \frac{1}{l}\sum_{m=k-l}^{k-1} \hat{r}^{(m)} \tag{29}$$

If neither of the stopping conditions is met, we will increment k and start a new iteration at stage 3. Else we proceed to the next and last stage.

9. We will now halt the decoding and return with failure. Next, Alice and Bob will reveal bits to each other and will try to decode again. We will sort the non-shortened bits according to their LLR absolute value, $|r^{(k)}[i]|$, returning the lowest valued $d \geq 1$ bits.

## 8.3 LDPC Error Reconciliation vs. the Cascade Method

To understand the pros and cons of choosing LDPC for error reconciliation, we will compare it to the Cascade method. Here we followed the analysis presented in [1,19].

| Method | Efficiency | Com. Rounds | Guaranteed Conv. |
|---|---|---|---|
| Cascade | Depends on realization | $> 30$ | Yes |
| Straight forward LDPC | Typically better than Cascade ($q_{ber} > 2\%$) | 1 | No |
| Rate Adaptive | Better than S.F LDPC | 1 | No |
| Blind | Better than Rate Adaptive | Typically $< 10$ | Yes |
| Symmetric | Better than Blind | Typically $< 10$ | Yes |

Table 1: Comparison of QKD error reconciliation schemes[1].

**Metrics of Comparison** Examining different methods for QKD error reconciliation, we take into account a few factors:

1. *Does the decoding converge?* Are we promised that the algorithm will eventually halt with an output key?
2. *If converges, is it to a correct value?* This is generally a hard question because parity checks may return a wrong answer, but the answer would still satisfy the parity equations. This is not promised by any algorithm, but yet requires attention.
3. *What is the efficiency of the scheme? (Eq. 10)*
4. *How many communication rounds are needed?* We will notice that each communication round is not only consuming bits, but also consumes time and creates unwanted overhead to the complexity.

**Cascade** In this method[20], Bob divides his sifted key to several blocks of size $k$ and sends Alice a parity bit calculating the parity of each block. Alice performs the same calculation, announcing to Bob where differences exist. If an error found in any of the blocks, Alice and Bob exchange additional parity bits to perform a binary search inside the block to find the error and correct it. When this first pass over the blocks is done, Alice and Bob permute the keys' bits and go over the keys again with blocks of size $2k$. If an error is found in this iteration, it is fixed as before. Now, notice that parity checks miss errors when the check has an even number of them. Therefore, if an error was found in the second pass, it means that another error was hidden and masked in the first pass. The algorithm will permute the key to the locations of the first pass, run again over the same $k$ sized blocks, and fix the now revealed errors with binary search. Some errors must be revealed since the error that masked them was repaired when passing on the $2k$ size blocks.

This whole process is now repeated for $4k$ and so forth, repeating the previous runs once an error is found.

Notice that when performing each of the checks, many of the times there is a step going forward that is dependant on the previous result, making it obligatory to send it as a separate transmission to Alice. This makes the number of communication rounds in Cascade relatively large. On the other hand, we are promised that the algorithm will converge to some solution.

Kiktenko et al[1] have analyzed the different schemes mentioned, according to the metrics above, and the result can be seen in Table 1.

# 9 Symmetric Blind with Dense LDPC Pool

We have chosen to use the Symmetric Blind information reconciliation scheme, due to its benefits visible at Table 1. This algorithm performs with guarantied convergence, a critical advantage, and additionally, it typically performs with better efficiency than the other algorithms. We have additionally introduced the use of a large pool of LDPC matrices, created using the PEG algorithm. Each matrix row number corresponds to some $q_{ber}$, and the pool contains matrices distributed densely on a $q_{ber}$ interval of choice. We have found that the dense matrix pool improves the throughput of the decoding process by decreasing the number of consumed bits in every bit reveal round, excluding the initial syndrome round.

For some estimated rate, one would choose the closest available matrix and will fine-tune the choice using the Symmetric Blind scheme. The creation of matrices is done a priori according to chosen dimensions and chosen density.
We will notice that the amount of punctured and shortened bits is decreasing (Eq. 16,17) as the matrix row number gets closer to the theoretical bound. Thus as we increase the resolution of available matrices pool we will decrease the amount of puncture and shortened bits. Remembering that each bit replaced in these processes is effectively reducing the number of key bits generated over time, A dense pool increases the algorithm output.

The cost of a large number of matrices is in memory, both in the required space and time required for access. Considering the low cost of memory as a resource, this is arguably not a significant price. Additionally, the matrices can be organized efficiently to reduce that cost even more.

**Matrix Compression** Considering an LDPC matrix $h$ of dimension $m \times n$, with $c \ll mn$ cells containing 1s, the most common representation in memory would be by the non-zero cells. Each cell is represented by a row and column index, each a $\lceil log_2 n \rceil$ bits number, total of $2c\lceil log_2 n \rceil$ bits. For non zero cell indices $(i_k, j_k)$, the matrix will be represented as

$$i_1, j_1, i_2, j_2, ..., i_c, j_c \tag{30}$$

We will know the numbers apart by reserving an equal amount of bits to each number. If $\lceil log_2 n \rceil - log_2 n \geq 1$, we can use additional alphabet symbols to manage the information. Adding a 'line down' sign #, for each row from 1 to $m$ we will print the column indices appearing in it, followed by this sign. This will tell the reader that we are now moving to the next line. In the following example, the cells in the first row are in indices $i_1, i_2, i_3$, the second row has $i_4, i_5$ and so on and so forth. This requires $m$ # signs and $c$ additional numbers. at total $(m + c)\lceil log_2 n \rceil$.

$$i_1, i_2, i_3.\#i_4, i_5.\#i_6, i_7... \tag{31}$$

Last, in our implementation each column has exactly $l$ cells in it. Using this property, we will print the $l$ row indices of each column, starting from the first column to the last. We do not need the # delimiter, since we can count and tell the columns apart. We would put the number $l$ as the first number in the sequence, to let us know the amount in each column. This representation will cost us $(c + 1)\lceil log_2 m \rceil$ bits.

$$l, i_0^1, i_0^2 ... i_0^l, i_1^1 ... i_1^l i_2^1 ... i_N^l \tag{32}$$

Asymptotically these suggestions are not an improvement of the memory needed, since still requires $O(nlog_2 n)$ bits, but we have spared $(c - m)\lceil log_2 n \rceil$ bits in the first method and approximately half in the second.

**On-the-fly Linear Complexity Matrix Multiplication** The compression method in Equations 31 and 32 can be multiplied on-the-fly, as in with no need to remember past matrix bits or look at more than a single index at a time. The complexity is linear in cell number, which is some small integer times block length. The linearity is relying on an $O(1)$ implementation of random access to array elements. We will see the implementation for the Eq. 32 compression method in Algorithm 2. This can be easily modified for the Eq. 31 representation.

**Algorithm 2:** Linear Complexity on the Fly LDPC Multiplication

```
Function Mult (Key, H, M) is
    Input  : K - N bits length binary vector.
             H - LDPC as a vector of indices.
             M - matrix row number.
    Output: s - M length syndrom.

    s ← 0^M
    count ← 0
    col ← 1
    l ← H_1
    for each sign x in H, starting at 2nd do
        s[x] ← s[x] ⊕ K[col]
        count ← count + 1
        if count = l then
            col ← col + 1
            count ← 0
        end
    end
    return s
endFunction
```

**Memory Access** The matrices are uniquely identified by their row number, or equivalently by appropriate $q_{ber}$, creating an order that they can be stored in. Searching for some $m_{desired}$ rows matrix, one can search the matrix array with a binary search, fetching a matrix in $O(log_2 L)$ when $L$ is the size of the pool. Since the pool is not changed during run time, we can use a hash table or a lookup table, and achieve $O(1)$ average search time.

**Matrix Parameters and Distribution** The matrix pool is created with the mentioned PEG algorithm. Regardless of the matrix dimensions, each column of each matrix will have exactly 3 cells in it. According to [14] this is the minimal number of cells in a column require to create a functioning LDPC code. This is an approximation on our side since there is no guaranty to the number of cells in each row. We will not use matrices with row number lower than 30, to prevent a situation where the matrix is not a sparse one. This is of course an abstract construct, and the threshold can be moved. Additionally, this decision is relevant for lower $q_{ber}$, in which arguably sparse matrices are not the most relevant.

While implementing the algorithm we found that the algorithm performance for $qber < 0.05$ is worse than on larger ranges, as seen in Figure 5c. In an attempt to slightly compensate for that weak spot, the distances between the matrices are smaller in this $q_{ber}$ scale.

### 9.1 Results

As part of a bigger attempt to create a working QKD COW system, we have made a software implementation of the Symmetric Blind reconciliation algorithm for the error reconciliation stage, using Matlab and C++. This software also used to test different parameters of the algorithm, paving the way for a possible, faster, hardware implementation of the scheme. To measure the performance of the algorithm, four metrics were used: efficiency (Eq. 15), decoding iterations, bits initially punctured or shortened (number of leftover bits kept for future iteration) and decoding failure rate. The decoding failure rate is defined as 0 if converged to the correct result, and 1 if converged to a wrong result yet with the desired syndrome. By looking at these matrices we get an insight into the algorithm's performance. The efficiency, failure rate, and the number of left bits together are showing the throughput of the algorithm, relative to the theoretical limit. The decoding iterations is directly connected to the time complexity of the algorithm. Additionally, these metrics were used to describe the performance of error reconciliation algorithm in past research, and using them allows us to compare our implementation to others'.

Each measurement was taken for 40 $q_{ber}$ values distributed uniformly on $(0, 0.11]$, each $q_{ber}$ value is measured 10 times and averaged. A pair of keys of length $n = 2000$ was produced with the given error rate and corrected for every such iteration. For each pool, the matrices are divided into two sections. 0.3 of the matrices are distributed uniformly on the $q_{ber}$ interval $(0, 0.1]$, and 0.7 are distributed uniformly on $(0.1, 0.11]$. in this context, the uniformity is for equal corresponding $q_{ber}$ values, meaning that the row numbers are distributed by $\eta_b(q_{ber})$. The code used for these measurements is shown in Appendix A.

Figure 5 demonstrates three pools with increasing densities, with 50, 100 and 150 matrices of row numbers in the range of $(0, 0.5n]$. The efficiency (a), decoding iterations (b) and failure rate (c) are not changing when density increases. On the other hand, we can see that the number of leftover bits (d) is significantly lower for the denser pools. The ratio between the peaks of the lowest density pool to the highest is 2.7. For the entire spectrum, the average difference in leftover bits between the 50 matrix pool and the 100 matrix pool is 6.7, while between the 100 matrix pool and the 150 matrix pool it is 5.7. Averaging on the entire spectrum the difference is spread out, but there is a great advantage to this much smaller bound on the high peak.

This is a direct increase in the throughput of the reconciliation algorithm for the denser pool. In that context, it is interesting to see that the efficiency as a metric is not capable to show this difference since is calculated relative to the key after being punctured and shortened.

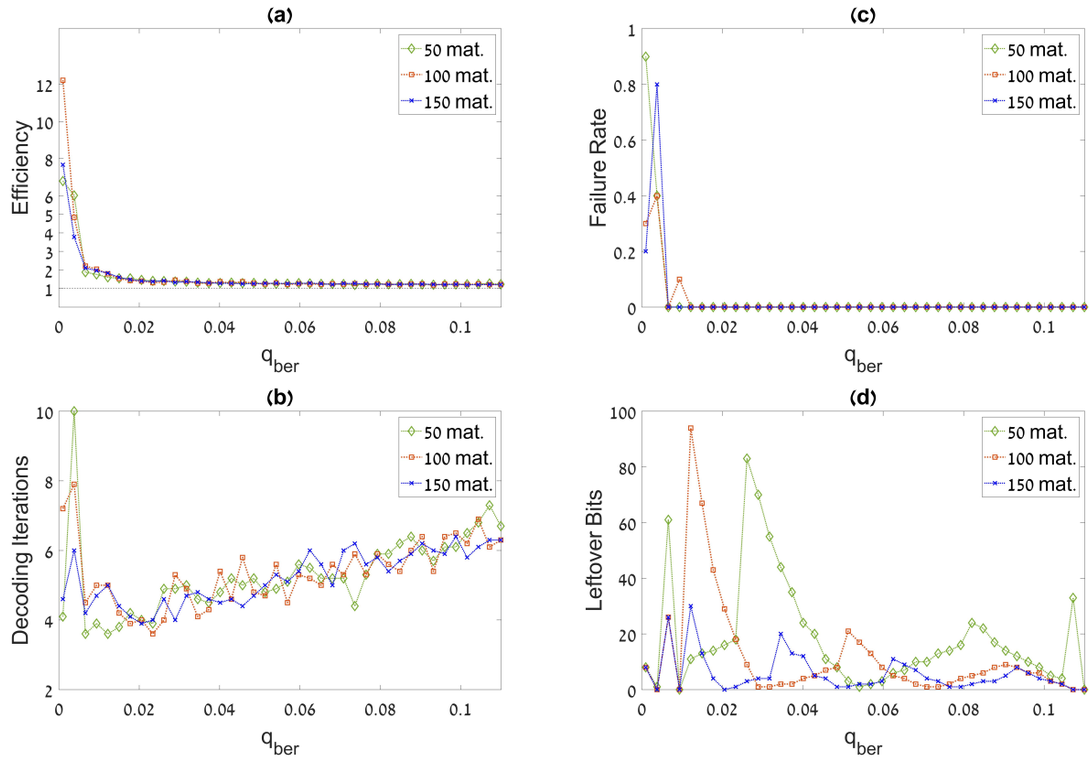
## 10 Discussion and Conclusion

In this is paper we presented QKD error reconciliation schemes, focusing and on the Symmetric Blind rate adaptive scheme. We implemented the said scheme while also implementing and presenting measurement results for dense matrix pools. It seems that the memory cost of such a large batch of matrices is tolerable, and is implementation dependent. We have shown that the operations on the matrices and their storage could be handled relatively efficiently. Additionally, we found that denser pools have the potential to increase the throughput of the reconciliation process by reducing the number of punctured and shortened bits in each iteration.

In a way, adding dense pools of matrices is "brute force", but arguably it is a price worth paying. An interesting direction for advancement would be to identify specific error-prone $q_{ber}$ rates in a system and optimize the pool selection for that system. Additionally, since it is expected that an hardware-based implementation would offer great improvement in time complexity, it is intriguing to see whether a large memory component storing the pool could be Incorporated practically without damaging performance.

An additional interesting question arising is the cause of the high failure rate for the low $q_{ber}$ matrices. The relevant matrices have a low number of rows, and arguably they are not sparse anymore. It is possible that in this range we should think of the matrices as non-sparse parity check matrices, and use different algorithms for matrix construction and possibly even for key correction.

Figure 5: Efficiency (a), decoding iterations (b), failure rate (c), and leftover bits (d) vs. qber for pool densities of 50, 100 and 150 matrices. The pools are denser at the range from 0 to 0.1 than on the rest of the spectrum. It is visible that the density effects only the leftover bits, while the other metrics stay the same. Also visible is the larger error rate when error rate is smaller, resulting in smaller matrices, possibly too small to function. The asymptotic rise of the efficiency when approaching $q_{ber} = 0$ is expected. This is because the denominator of the efficiency is proportional to $\eta_b(q_{ber})$, which goes to 0 on the limit of $q_{ber} = 0$.




# 11 References


1. Kiktenko, E. O., Trushechkin, A. S., Lim, C. C. W., Kurochkin, Y. V., & Fedorov, A. K. (2017). Symmetric blind information reconciliation for quantum key distribution. Physical Review Applied, 8(4), 044017.

2. Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.

3. Shor, P. W. (1994, November). Algorithms for quantum computation: Discrete logarithms and factoring. In Proceedings 35th annual symposium on foundations of computer science (pp. 124-134). Ieee.

4. Alléaume, R., Branciard, C., Bouda, J., Debuisschert, T., Dianati, M., Gisin, N., ... & Monyk, C. (2014). Using quantum key distribution for cryptographic purposes: a survey. Theoretical Computer Science, 560, 62-81.

5. Kiktenko, E., Trushechkin, A., Kurochkin, Y., & Fedorov, A. (2016, August). Post-processing procedure for industrial quantum key distribution systems. In Journal of Physics: Conference Series (Vol. 741, No. 1, p. 012081). IOP Publishing.

6. Renner, R. (2008). Security of quantum key distribution. International Journal of Quantum Information, 6(01), 1-127.

7. Bennett, C. H., Mor, T., & Smolin, J. A. (1996). Parity bit in quantum cryptography. Physical Review A, 54(4), 2675.

8. Universal Serial Bus Type-C Cable and Connector Specification, Release 1.4 March 29, 2019, Section 6.6.4.6.4.

9. Stucki, D., Brunner, N., Gisin, N., Scarani, V., & Zbinden, H. (2005). Fast and simple one-way quantum key distribution. Applied Physics Letters, 87(19), 194108.

10. Korzh, B., Lim, C. C. W., Houlmann, R., Gisin, N., Li, M. J., Nolan, D., ... & Zbinden, H. (2015). Provably secure and practical quantum key distribution over 307 km of optical fibre. Nature Photonics, 9(3), 163.

11. Gisin, N., Ribordy, G., Zbinden, H., Stucki, D., Brunner, N., & Scarani, V. (2004). Towards practical and fast quantum cryptography. arXiv preprint quant-ph/0411022

12. Mafu, M., Marais, A., & Petruccione, F. (2014). A Necessary Condition for the Security of Coherent-One-Way Quantum Key Distribution Protocol. Applied Mathematics & Information Sciences, 8(6), 2769.

13. Lim, C. C. W., Curty, M., Walenta, N., Xu, F., & Zbinden, H. (2014). Concise security bounds

for practical decoy-state quantum key distribution. Physical Review A, 89(2), 022307.

14. Gallager, R. (1962). Low-density parity-check codes. IRE Transactions on information theory, 8(1), 21-28.

15. Curty, M., & Lütkenhaus, N. (2005). Intercept-resend attacks in the Bennett-Brassard 1984 quantum-key-distribution protocol with weak coherent pulses. Physical Review A, 71(6), 062301.

16. Bennett, C. H., & Brassard, G. (2014). Quantum cryptography: public key distribution and coin tossing. Theor. Comput. Sci., 560(12), 7-11.

17. Hu, X. Y., Eleftheriou, E., & Arnold, D. M. (2001, November). Progressive edge-growth Tanner graphs. In GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270) (Vol. 2, pp. 995-1001). IEEE.

18. Jiang, X., Lee, M. H., & Qi, J. (2012). Improved progressive edge-growth algorithm for fast encodable LDPC codes. EURASIP Journal on Wireless Communications and Networking, 2012(1), 178.

19. Johnson, J. S. (2012). An analysis of error reconciliation protocols for use in quantum key distribution.

20. Brassard, G., & Salvail, L. (1993, May). Secret-key reconciliation by public discussion. In Workshop on the Theory and Application of of Cryptographic Techniques (pp. 410-423). Springer, Berlin, Heidelberg.

21. Pedersen, T. B., & Toyran, M. (2013). High performance information reconciliation for QKD with CASCADE. arXiv preprint arXiv:1307.7829.

22. Mink, A., & Nakassis, A. (2012). LDPC for QKD reconciliation. arXiv preprint arXiv:1205.4977.

23. Buttler, W. T., Lamoreaux, S. K., Torgerson, J. R., Nickel, G. H., Donahue, C. H., & Peterson, C. G. (2003). Fast, efficient error reconciliation for quantum cryptography. Physical Review A, 67(5), 052303.

24. Elkouss, D., Martinez-Mateo, J., & Martin, V. (2010). Information reconciliation for quantum key distribution. arXiv preprint arXiv:1007.1616.

25. Elkouss, D., Martinez-Mateo, J., & Martin, V. (2013). Analysis of a rate-adaptive reconciliation protocol and the effect of leakage on the secret key rate. Physical Review A, 87(4), 042334.

26. Martinez-Mateo, J., Elkouss, D., & Martin, V. (2012). Blind reconciliation. arXiv preprint arXiv:1205.5729.

27. Choi, E., Suh, S. B., & Kim, J. (2005, September). Rate-compatible puncturing for low-density parity-check codes with dual-diagonal parity structure. In 2005 IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications (Vol. 4, pp. 2642-2646). IEEE.

28. Martinez-Mateo, J., Elkouss, D., & Martin, V. (2013). Key reconciliation for high performance quantum key distribution. Scientific reports, 3, 1576.

29. Elkouss, D., Martinez-Mateo, J., & Martin, V. (2012). Untainted puncturing for irregular low-density parity-check codes. IEEE Wireless Communications Letters, 1(6), 585-588.

30. Hu, X. Y., Eleftheriou, E., Arnold, D. M., & Dholakia, A. (2001, November). Efficient implementations of the sum-product algorithm for decoding LDPC codes. In GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270) (Vol. 2, pp. 1036-1036E). IEEE.

31. Elliott, C., Colvin, A., Pearson, D., Pikalo, O., Schlafer, J., & Yeh, H. (2005, May). Current status of the DARPA quantum network. In Quantum Information and computation III (Vol. 5815, pp. 138-149). International Society for Optics and Photonics.

32. Kiktenko, E. O., Malyshev, A. O., Bozhedarov, A. A., Pozhar, N. O., Anufriev, M. N., & Fedorov, A. K. (2018). Error estimation at the information reconciliation stage of quantum key distribution. Journal of Russian Laser Research, 39(6), 558-567.

33. Slepian, D., & Wolf, J. (1973). Noiseless coding of correlated information sources. IEEE Transactions on information Theory, 19(4), 471-480.

34. Weaver, W., & Shannon, C. E. (1963). The mathematical theory of communication. Champaign, IL: University of Illinois Press.

35. Fung, C. H. F., Ma, X., & Chau, H. F. (2010). Practical issues in quantum-key-distribution postprocessing. Physical Review A, 81(1), 012318.

36. Walenta, N., Burg, A., Caselunghe, D., Constantin, J., Gisin, N., Guinnard, O., ... & Legré, M. (2014). A fast and versatile quantum key distribution system with hardware key distillation and wavelength multiplexing. New Journal of Physics, 16(1), 013047.

# Appendices

## A Measurement Code

This code is the Matlab code generating the measurements specified in Section 9.1. It uses the code documented in Appendix B. The code will be presented by files when the run is done by using the main.m file in Section A.3. The measure.m function is performing a decoding experiment for different values, and the print_figures.m function is setting these results in a figure. The main.m is generating matrices and run the measurements using the two said functions.

### A.1 measure.m

```
function [efficiency, dec_it, error_rate, left_bits, consumed] = measure(pool, qber)
tic;
[~,~] = movefile(pool,'matrices_bank');
KEY_LEN = 2000;
AVG_NUM = 10;

efficiency = zeros(size(qber));
dec_it = zeros(size(qber));
error_rate = zeros(size(qber));
consumed = zeros(size(qber));
left_bits = zeros(size(qber));
bin_ent_qber = bin_ent(qber);

for i=1:numel(qber)
    for j=1:AVG_NUM
        [sifted_key_a,sifted_key_b] = generate_keys(KEY_LEN,qber(i));
        [key_corrected_a, left_bits_a, consumed_bits_a, key_corrected_b, left_bits_b, ...
        consumed_bits_b, decoding_iterations, alg_success] ...
            = symmetric_flow(sifted_key_a, sifted_key_b, qber(i), rng);
        if ~alg_success
            error('No success');
        end
        if consumed_bits_a ~= consumed_bits_b
        error('consumed bits mismatch')
        end
        if numel(left_bits_a) ~= numel(left_bits_b)
        error('left bits mismatch')
        end
        if numel(key_corrected_a) ~= numel(key_corrected_b)
        error('key lengths mismatch')
        end
        if any(key_corrected_a ~= key_corrected_b) && alg_success
            %disp(qber(i));
            error_rate(i) = error_rate(i) + 1;
            %error('key correction mismatch')
        end
        left_bits(i) = left_bits(i) + numel(left_bits_a);
        consumed(i) = consumed(i) + consumed_bits_a;
        efficiency(i) = efficiency(i) + (consumed_bits_a / (numel(sifted_key_a) * bin_ent_qber(i)));
        dec_it(i) = dec_it(i) + decoding_iterations;
    end
    left_bits(i) = left_bits(i) / AVG_NUM;
    efficiency(i) = efficiency(i) / AVG_NUM;
    dec_it(i) = dec_it(i) / AVG_NUM;
    error_rate(i) = error_rate(i) / AVG_NUM;
    consumed(i) = consumed(i) / AVG_NUM;
end
movefile('matrices_bank',pool);
toc;
end
```

### A.2 print_figures.m

```
    function print_qkd(qber,eff,dec_it,frame_error,left_bits)

    function print_pools(x, y,label)
    lw = 1.5;
    ms = 10;
```

```
    y = y(1:3);
    figure('visible','on');
    set(gca,'FontSize',24)
    set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 30 20]);

    plot(x,y{1},':d','Color','#77AC30','LineWidth',lw,'MarkerSize',ms);
    hold on;
    plot(x,y{2},':s','LineWidth',lw+0.5,'MarkerSize',ms-2);
    hold on;
    plot(x,y{3},'b:x','LineWidth',lw,'MarkerSize',ms-2);
    xlim([0 0.1101]);
    set(get(gca,'ylabel'),'rotation',0)

    switch label
        case '(a)'
            label_name = 'Efficiency';
        case '(b)'
            label_name = 'DecIt';
        case '(c)'
            label_name = 'FailureRate';
        case '(d)'
            label_name = 'LeftoverBits';
    end
    title(label,'fontSize',35);

    if strcmp(label_name,'Efficiency')
        line([0 0.11],[1 1], 'Color','black','LineStyle','--');
        yticks([1:5 6:2:13]);
    end

    lgd = legend('50 mat.', '100 mat.', '150 mat.');
    lgd.FontSize = 30;
    get(gca, 'XTick');
    set(gca, 'FontSize', 20);
    get(gca, 'YTick');
    set(gca, 'FontSize', 20);
    savefig(label_name);
    print(gcf,label_name,'-djpeg','-r300');
    end

print_pools(qber, eff, '(a)');
print_pools(qber, dec_it, '(b)');
print_pools(qber, frame_error, '(c)');
print_pools(qber, left_bits, '(d)');

end
```

### A.3 main.m

```
KEY_LEN = 2000;
MAT_NUM = [50, 100, 150];
for j=1:3
    generate_matrices(KEY_LEN,linspace(0.0014,0.01,0.3 * MAT_NUM(j)));
    generate_matrices(KEY_LEN,linspace(0.01,0.11,0.7 * MAT_NUM(j)));
    movefile('matrices_bank',strcat('matrices_bank_density_',num2str(j)));
end

rng('shuffle');
QBER_NUM = 40;
qber = linspace(0.001,0.11,QBER_NUM);
eff = cell(1,3);
dec_it = cell(1,3);
frame_error = cell(1,3);
left_bits = cell(1,3);
consumed_bits = cell(1,3);

for j=1:3
    [eff{j}, dec_it{j}, frame_error{j}, left_bits{j}, consumed_bits{j}] = ...
        measure(strcat('matrices_bank_density_',num2str(j)), qber);
end
save('measure_per_density');
```

```
print_qkd(qber,eff,dec_it,frame_error,left_bits);

d13 = mean(left_bits{1} - left_bits{3});
d12 = mean(left_bits{1} - left_bits{2});
d23 = mean(left_bits{2} - left_bits{3});
max13 = max(left_bits{1}) ./ max(left_bits{3});
```

# B Code Documentation

## B.1 Overview

This package is performing error correction using the Symmetric Blind Error Reconciliation scheme explained in the paper, and specifically on Section 8.1. The package contains CPP and Matlab (.m) files and is meant to run on Matlab. The CPP files are implementing the core decoding and matrix creation processes, and are supposed to give an option for future compilation into FPGA. The Matlab files are running the CPP files using the MEX platform, compiling using the Matlab add-on 'MATLAB Support for MinGW-w64 C/C++ Compiler' version 19.1.0, on Windows OS.

## B.2 Package Files

These are the code files existing in the package, grouped by their containing directories. The division into directors is by the functionality of the files.

1. ***CPP*** CPP source code, used as core for LDPC matrix creation and decoding process.
   (a) ***qkd_objects.h***
   (b) ***qkd_functions.cpp***
   (c) ***qkd_functions.h***
   (d) ***Tanner_graph.cpp***
   (e) ***Tanner_graph.h***

2. ***MATLAB_LOGIC*** Matlab functions used for user operation of algorithm.
   (a) ***bin_ent.m***
   (b) ***build_mex.m***
   (c) ***decode.m***
   (d) ***find_closest_mat.m***
   (e) ***generate_keys.m***
   (f) ***generate_matrices.m***
   (g) ***generate_matrices_log_dist.m***
   (h) ***get_ldpc.m***
   (i) ***get_short_punc_ind.m***
   (j) ***insert_short_punc_bits.m***
   (k) ***is_bin_arr.m***
   (l) ***mod2mult.m***
   (m) ***symmetric_flow.m***

3. ***MEX*** CPP files, readable as Matlab functions, wrapping and managing the CPP logic.
   (a) ***mex_decode.cpp***
   (b) ***mex_print_ldpc_mat.cpp***

4. ***MEX_COMPILED*** Compiled MEX files. This is used by the Matlab console for code execution. Is not meant to be altered by user, nor contains any new logic. One can build exisiting MEX files using the build_mex.m script, resulting in these files being updated.
   (a) ***mex_decode.mexw64***
   (b) ***mex_print_ldpc_mat.mexw64***

5. ***Installation, Examples and Documentation*** Code examples and short documentation.
   (a) ***Install.m***
   (b) ***example_symmetric_flow.m***

6. ***Measurement code*** Code described in Appendix A used to create the measurements in Section 9.1.
   (a) ***measure.m***
   (b) ***print_figures.m***
   (c) ***main.m***

## B.3 Functional Documentation

### B.3.1 Install.m

*Install*

**Description:** Set file paths before each run.
**Input:**

1. None.

**Output:**

1. None.

---

### B.3.2 example_symmetric_flow.m

*example_symmetric_flow*

**Description:** Example on how to run the symmetric_flow function together with matrix generation.
**Input:**

1. None.

**Output:**

1. None.

---

### B.3.3 qkd_objects.h

**Defined Types:**

1. ***bit_t*** Abstraction of a single bit.
2. ***index_t*** Index of a matrix or key.
3. ***float_num_t*** A float number.
4. ***bit_vec_t*** Data structure for vectors of bits.
5. ***ind_set_t*** Data structure for indices sets of key positions.
6. ***indexed_float_num_t*** Floating point number connected to an index.
7. ***Decoding_result*** Struct for storing results of decoding process. Contains bit_vec_t decoded_key, ind_set_t d_vec, bool success.
8. ***Matrix_element*** Pair of two numbers representing an element of matrix. Element (i,j) of the matrix is the i-th check (check = i), and j-th symbol (symbol = j).
9. ***Ldpc_mat*** Data structure, abstraction of LDPC matrix. Set of Matrix_element instances. A matrix cell is 1 iff it exists in the set.
10. ***Sim_data*** Struct holding simulation configuration.

### B.3.4 qkd_functions.h

*bool contains(const ind_set_t& set, index_t pos);*

**Description:** Returns true if a set of indices contains a given index, else false.
**Input:**

1. ***set*** A set of indices.
2. ***pos*** An index.

**Output:**

1. True if set contains pos.

---

*Decoding_result decode(const bit_vec_t& s, const bit_vec_t& e, const Ldpc_mat& ldpc_mat, const ind_set_t& short_pos, const ind_set_t& punc_pos, const Sim_data&);*

**Description:** Belief Propegation decoding algorithm.
**Input:**

1. ***s*** XOR of Alice and Bob's syndrom.
2. ***e*** The error vector.
3. ***ldpc_mat*** LDPC matrix used for decoding.
4. ***short_pos*** Shortened indices in Alice and Bob's keys.
5. ***punc_pos*** Punctured indices in Alice and Bob's keys.
6. ***sim_data*** Parameters for the simulation.

**Output:**

1. Decoding result object, containing decoded key, revealed indices, and decoding success flag. Decoded key is valid in case of success, and the revealed indices are valid in case of failure.

---

### B.3.5 Tanner_graph.h

**Defined Types:**

1. ***NodeType*** Enumeration defining types of nodes. Options are SYMBOL and CHECK.
2. ***Node_tanner*** A node in a tanner graph.
3. ***Node_comperator*** A nobject used to create an order between nodes.
4. ***Tanner_data*** An object storing meta-data on a matrix, to be printed on the top of the file.
5. ***Tanner_graph*** An object representing a Tanner graph.

### B.3.6 bin_ent.m

*bin_ent_result = bin_ent(q)*

**Description:** Calculate the binary entropy of q.
**Input:**

1. ***q*** A real number between 0 and 1.

**Output:**

1. ***bin_ent_result*** The binary entropy of q,

$$\eta_b(q) = -qlog_2q - (1-q)log_21 - q$$

---

### B.3.7 build_mex.m

*build_mex*

**Description:** Compile the MEX files.
**Input:**

1. None.

**Output:**

1. None.

---

### B.3.8 decode.m

*[error_dec, reveal_ind, success] = decode(error_syndrom, ldpc_mat, error_vec, punc_ind, short_ind, qber_est)*

**Description:** Belief propegation decoding function described in Section 8.2.
**Input:**

1. ***error_syndrom*** The XOR of Alice and Bob's syndroms with ldpc_mat.
2. ***ldpc_mat*** LDPC matrix used for the decoding.
3. ***error_vec*** The error syndrom, $K_{sift}^{a'} \oplus K_{sift}^{b'}$.
4. ***punc_ind*** Indices punctured.
5. ***short_ind*** Indices shortened.
6. ***qber_est*** $q_{ber}$ estimation.

**Output:**

1. ***error_dec*** The decoded error vector. Valid if success is true.
2. ***reveal_ind*** In case of failure, this contains the bits Alice and Bob should reveal. If success, this output is not valid.
3. ***success*** True if the algorithm convereged to an error_dec giving error_syndrom when multiplied by ldpc_mat. Else, false.

---

### B.3.9 find_closest_mat.m

*[ldpc_mat_path, chosen_m] = find_closest_mat(desired_m, key_len)*

**Description:** Find an LDPC matrix file with a given column number and row number to closest to a desired.
**Input:**

1. ***desired_m*** Desired row number of an LDPC matrix, a real positive number little or equal to key_len.
2. ***key_len*** Desired column number for an LDPC matrix.

**Output:**

1. ***chosen_m*** Row number of the found LDPC matrix.
2. ***ldpc_mat_path*** Path of the file containing the found matrix.

---

### B.3.10 generate_keys.m

*[varargout] = generate_keys(varargin)*

**Description:** Function generating keys, or batched of keys with errors, able to according to the following options.

1. Generate single Alice key: *[Alice_key] = generate_keys(key_len);*
2. Generate Alice key and single Bob key : *[Alice_key, Bob_key] = generate_keys(key_len,qber);*
3. Generate Alice key, Bob keys batch : *[Alice_key, Bob_key_0, ... ,Bob_key_n] = generate_keys(key_len,qber,num_keys);*
4. Generate single Bob key using Alice key from binary vector: *[Alice_key, Bob_key] = generate_keys(Alice_key_vec,qber);*
5. Generate multiple Bob keys using Alice key from binary vector: *[Alice_key, Bob_key_0, ... ,Bob_key_n] = generate_keys(Alice_key_vec,qber,num_keys);*

**Input:**

1. ***key_len*** Length of the required key.
2. ***qber*** Desired error rate between Alice and Bob's key.

**Output:**

1. ***Alice_key*** A key considered to be Alice. Used as reference for 'correct' key when creating Bob keys. When a batch is created, all Bob keys refer to this single Alice key.
2. ***Bob_key_**** Keys with $q_{ber}$ errors in relation to Alice's key. May be more than one of these.

---

### B.3.11 generate_matrices.m

*generate_matrices(symbol_num, check_num)*

**Description:** Generate the matrices with symbol_num columns and check_num(i) rows for each i in 1:length(check_num).
**Input:**

1. ***symbol_num*** Column number of the printed matrices.
2. ***check_num*** An array of row numbers to print matrices with.

**Output:**

1. The function does not return functions, but do print LDPC matrices into file matrices_bank.

---

#### B.3.12 generate_matrices_log_dist.m

*generate_matrices_log_dist(symbol_num, mat_num)*

**Description:** Generate the matrices with symbol_num columns rate distributed in a log scale.
**Input:**

1. ***symbol_num*** Column number of the printed matrices.
2. ***mat_num*** Number of matrices to print, distributed equaly along a logarithmic scale from 1 to symbol_num.

**Output:**

1. The function does not return functions, but do print LDPC matrices into file matrices_bank.

---

#### B.3.13 get_ldpc.m

*[ldpc_mat] = get_ldpc(symbol_num, qber_est)*

**Description:** Read an LDPC matrix from an LDPC file with closest appropriate $q_{ber}$.
**Input:**

1. ***symbol_num*** Column number for the desired matrix.
2. ***qber_est*** Error rate corresponding to the desired matrix.

**Output:**

1. ***ldpc_mat*** Read matrix.

---

#### B.3.14 get_short_punc_ind.m

*[punc_ind, short_ind] = get_punc_short_ind(check_num, symbol_num, qber_est, seed)*

**Description:** Caclulate the sizes of shortend and punctured bit sets (Section 8.1, and return randomly selected bit indices in that amount.
**Input:**

1. ***check_num*** Matrix row number, or parity check number.
2. ***symbol_num*** Matrix column number, or key length.
3. ***qber_est*** Estimated $q_{ber}$

**Output:**

1. ***punc_ind*** Indices to be punctured.
2. ***short_ind*** Indices to be shortened.

---

#### B.3.15 insert_short_punc_bits.m

*[sifted_key_altered, left_bits] = insert_punc_short_bits(sifted_key, punc_ind, short_ind)*

**Description:** Replace key bits in punctured and shortened bits as described in Section 7.2.
**Input:**

1. ***sifted_key*** The sifted key to emplace the bits inside.
2. ***punc_ind*** Indices to puncture.
3. ***short_ind*** Indices to short.

**Output:**

1. ***sifted_key_altered*** The sifted key after puncturing and shortening.
2. ***left_bits*** The bits that were replaced. The last —punc_ind— + —short_ind— bits in the input key.

---

#### B.3.16 is_bin_arr.m

*is_bin_arr_result = is_bin_arr(arr)*

**Description:** Checks if all the elements of an array are 0 or 1.
**Input:**

1. ***arr*** A Matlab array.

**Output:**

1. True if all the elements of the array are 0 or 1.

---

#### B.3.17 mod2mult.m

*[syndrom_row] = mod2mult(matrix,row_vec)*

**Description:** Multiply a matrix in the row vector, addition being mod 2.
**Input:**

1. ***matrix*** Sparse binary valued matrix to multiply in.
2. ***row_vector*** Row vector to multilpy in the matrix.

**Output:**

1. ***syndrom_row*** Transpose of the multiplcation result.

---

#### B.3.18 symmetric_flow.m

*[key_corrected_a, left_bits_a, consumed_bits_a, key_corrected_b, left_bits_b, consumed_bits_b, decoding_iterations, alg_success] = symmetric_flow(sifted_key_a, sifted_key_b, qber_est, seed)*

**Description:** Perform a complete error reconcilation process, as described in Figure 4.
**Input:**

1. ***sifted_key_a*** Alice's sifted key.
2. ***sifted_key_b*** Bob's sifted key.
3. ***qber_est*** Estimated $q_{ber}$
4. ***seed*** Shared seed between the two sides, used to generate random punctured / shortened indices.

**Output:**

1. ***key_corrected_a*** Alice's sifted key, truncated to the final key size.

2. ***left_bits_a*** Alice's bits replaced by punctured and shortened bits, for later use.
3. ***consumed_bits_a*** Number of consumed bits according to Alice's count.
4. ***key_corrected_b*** Bob's corrected sifted key.
5. ***left_bits_b*** Bob's bits replaced by punctured and shortened bits, for later use.
6. ***consumed_bits_b*** Number of consumed bits according to Bob's count.
7. ***decoding_iterations*** Number of decoding iteration performed.
8. ***alg_success*** True iff the algorithm convereged into a result with correct syndrom.

---

#### B.3.19 mex_decode.cpp

*[error_dec, reveal_ind, success] = mexDecode(ldpc_rows, ldpc_cols, error_vec, error_syndrom , punc_ind, short_ind, qber_est)*

**Description:** Belief propegation decoding function described in Section 8.2.
**Input:**

1. ***ldpc_rows*** The row indices of the cells in the LDPC matrix. Corresponding to the order in ldpc_cols.
2. ***ldpc_cols*** The column indices of the cells in the LDPC matrix. Corresponding to the order in ldpc_rows.
3. ***error_vec*** The error vector.
4. ***error_syndrom*** The error syndrom.
5. ***punc_ind*** Indices to be punctured.
6. ***short_ind*** Indices to be shortened.
7. ***qber_est*** Estimated $q_{ber}$.

**Output:**

1. ***error_dec*** In case of success, decoded error vector.
2. ***reveal_ind*** In case of failure, ndices to reveal in the next decoding iteration.
3. ***success*** True if the decoding was successfull, else false.

---

#### B.3.20 mex_print_ldpc_mat.cpp

*mex_print_ldpc_mat(check_num, symbol_num, char(file_path))*

**Description:** Create and print in the designated path an LDPC matrix of the needed proportions using PEG described in Algorithm 1 and Section 9.
**Input:**

1. ***check_num*** Row number of the matrix.
2. ***symbol_num*** Column number of the matrix.
3. ***file_path*** A path to print the matrix in, as a Matlab char array.

**Output:**

1. Does not return output, but prints the matrix in the given path.

---