Lab 02 report: QKD IR

Symmetric QKD IR using LDPC codes & Cascade

Author: **David Polzoni**

Student ID: **2082157**

Course: **Quantum Cryptography & Security**

Prof. **G. Vallone**, Prof. **N. Laurenti**

Università degli Studi di Padova

A.Y. 2023-2024

Date: February 4, 2024

Contents

1 Introduction

Quantum Key Distribution (QKD) relies on a combination of quantum and classical procedures to facilitate the generation of a secret random string, referred to as the key, exclusive to the two parties executing the protocol. The inherent limitations of the protocol, imperfections in devices, and the threat of eavesdropping introduce errors and potential information leakage. Consequently, it becomes necessary to refine the set of measured signals, known as the raw key, through a process called key distillation. This classical process involves the application of basis reconciliation, error correction, and privacy amplification protocols to the raw key, resulting in the creation of a final, information-theoretically secure key. The efficiency of the information reconciliation process is crucial to prevent a bottleneck in the overall performance of the QKD system. Brassard and Salvail [1] introduced a protocol called Cascade, which has become the de-facto standard for practical QKD implementations. However, the Cascade protocol is highly interactive, requiring extensive communication between the legitimate parties and exhibiting suboptimal efficiency. This characteristic imposes an early limit on the maximum tolerable error rate. Winnow, proposed by Buttler et al. [2], is a well-known reconciliation protocol in the field of QKD that minimizes communication between parties. The advantage of Winnow is its reduced number of communications, but its efficiency is inferior to that of Cascade within the relevant error range. Until recently, modern coding techniques were not applied to discrete variable QKD. LDPC (Low-Density Parity-Check) codes were introduced, but their efficiency, although offering forward error correction, was still inferior to Cascade. LDPC codes optimized for the Binary Symmetric Channel (BSC) exhibited a saw-like efficiency curve due to a lack of information rate adaptability in the proposed procedure. Given the variability in error rates during transmissions, adaptability to such changes is crucial for an effective protocol. In this report, we will explore a software implementation of error correction for symmetric QKD information reconciliation using LDPC codes and an implementation of the Cascade protocol introduced for comparison with LDPC performances.

2 Background

In this section, we'll introduce the reader to the theoretical aspects of the above expressed concepts. Moreover, in our analysis, we'll only focus on prepare and measure protocols as the BB84, firstly introduced by Bennet and Brassard [3].

2.1 Information reconciliation and performance metrics

Information reconciliation is a crucial step in QKD protocols, ensuring that two parties, usually referred to as Alice and Bob, share a secret key despite the inevitable presence of errors and discrepancies in their transmitted qubits. The primary goal of information reconciliation is to synchronize the information held by Alice and Bob while minimizing the (classical) information leaked to a potential eavesdropper, commonly known as Eve. In the following, we report some key details about information reconciliation in QKD:

- **Error correction**: quantum communication channels are susceptible to various errors, including bit flips and phase errors. Information reconciliation corrects these errors to ensure that Alice and Bob share identical secret keys;
- **Privacy amplification (deterministic)**: the process of reducing the amount of information that Eve might have gained during the quantum communication. This involves transforming the initial shared information into a shorter, more secure key.

Regarding security metrics, when considering the ideal scenario with:

$$k_A^* = k_B^* = k^* \sim U(\mathcal{K}) \tag{1}$$

where the keys are equal and uniformly distributed in the key space, maximally unpredictable, our objective is to attain a k^* that remains independent of any observation made by potential eavesdroppers, including quantum side information. In other words, we aim for k^* to be independent from ρ_E, c_A, and c_B, where ρ_E denotes the quantum side information leaked to the eavesdropper, and c_A, c_B encode the classical side information from the two parties. Overall, what we want to achieve can be described as follows:

$$\rho^*_{k_A k_B E} = \omega_k \otimes \delta_{k_A k_B} \otimes \rho_E \tag{2}$$

In this ideal case, uniformity and complete independence from eavesdropper information are represented by the tensor product. However, it is important to acknowledge that achieving this ideal state in real-world scenarios is challenging, and our actual attainment may only approximate the ideal. This concept can be formalized using the trace distance as follows:

$$d_{\mathrm{v}}\left(\rho_{k_A k_B E},\, \rho^*_{k_A k_B E}\right) = \frac{1}{2}\left\|\rho_{k_A k_B E} - \omega_k \otimes \delta_{k_A k_B} \otimes \rho_E\right\| \tag{3}$$

2.2 LDPC codes

Low-Density Parity-Check (LDPC) codes are a class of error-correcting codes that have gained significant attention in both theoretical and practical aspects of information theory. They belong to the family of linear block codes and are known for their exceptional performance in terms of error correction at relatively low encoding and decoding complexity. Key features of LDPC codes include their sparse parity check matrices, hence the name "low-density". These matrices have a low ratio of ones to total elements, contributing to efficient encoding and decoding processes. LDPC codes exhibit remarkable performance near the Shannon limit, making them particularly attractive for applications in modern communication systems, including wireless and optical communications. The LDPC decoding process involves the use of iterative algorithms, with belief propagation being a common technique. These algorithms allow for the exchange of information among variable nodes and check nodes in the code graph, gradually improving the accuracy of decoded information. Let us now introduce some formalism about LDPC codes. A linear block code $\mathcal{C}$ of rate $R = k/n$ can be defined in terms of a $(n-k) \times n$ parity check matrix $\mathbf{H} = [h_1, h_2, \ldots, h_n]$, where each h_j is a column vector of length $n-k$. Each entry h_{ij} of $\mathbf{H}$ is an element of a finite field $\mathbb{Z}_p$. Since we will only consider binary codes, each entry is either a "0" or a "1" and all operations are modulo 2. The code $\mathcal{C}$ is the set of all vectors $\mathbf{x}$ that lie in the (right) nullspace of $\mathbf{H}$, i.e. $\mathbf{H}\,\mathbf{x} = \mathbf{0}$. Given a parity-check matrix $\mathbf{H}$, we can find a corresponding $k \times n$ generator matrix $\mathbf{G}$ such that $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. In its simplest guise, an LDPC code is a linear block code with a parity-check matrix that is "sparse" i.e. it has a small number of non-zero entries. Let us now define formally regular and irregular LDPC codes:

Definition 1. *A regular (n, k) binary LDPC code with column weight λ and row weight ρ is a (n, k) block code that has a $\ell \times n$ parity check matrix $\mathbf{H}$ in which the rows h_i and the columns $h_j^{'}$ satisfy:*

$$\|h_i\|_{\mathrm{H}} = \rho,\ i = 1, \ldots, \ell \tag{4}$$

$$\|h_j^{'}\|_{\mathrm{H}} = \lambda,\ j = 1, \ldots, n \tag{5}$$

with the constraints $\rho\ell = \|\mathbf{H}\|_{\mathrm{H}} = \lambda n$.

Definition 2. *An irregular (n, k) binary LDPC code with column weight distribution p_λ and row weight distribution p_ρ is a randomly generated (n, k) block code that has a $\ell \times n$ parity check matrix $\mathbf{H}$ in which the rows h_i and the columns $h_j^{'}$ satisfy:*

$$\mathbb{P}\{\|h_i\|_{\mathrm{H}} = m\} = p_\rho(m),\ i = 1, \ldots, \ell \tag{6}$$

$$\mathbb{P}\{\|h_j^{'}\|_{\mathrm{H}} = m\} = p_\lambda(m),\ j = 1, \ldots, n \tag{7}$$

with the constraints $\mathbb{E}\{\rho\}\ell = \|\mathbf{H}\|_{\mathrm{H}} = \mathbb{E}\{\lambda\}n$.

An important advance in the theory of LDPC codes is made possible by using bipartite graphs to provide a graphical representation of a parity-check matrix. A bipartite graph is a graph in which the nodes may be partitioned into two subsets such that there are no edges connecting nodes within a subset. In the context of LDPC codes, the two subsets of nodes are the *variable* nodes and the *check* nodes. There is one variable node for each of the n bits in the code and there is one check node for each of the m rows of $\mathbf{H}$. An edge exists between the i-th variable node and the j-th check node if and only if the entry $h_{ij} = 1$ in $\mathbf{H}$. In general, the components of those bipartite graphs can be listed as follows:

- The n variable nodes represent the codeword bits;
- The ℓ check nodes represent the syndrome bits;
- $\mathbf{H}$ is the biadjacency matrix of the graph, which is sparse.

The number of edges incident upon a node is called the degree of the node. Thus, the bipartite graph of a $(d_\mathrm{v}, d_\mathrm{c})$ LDPC code contains n variable nodes of degree d_v and m check nodes of degree d_c. It is clear that the parity-check matrix can be deduced from the bipartite graph and thus the bipartite graph can be used to define the code $\mathcal{C}$. We can therefore start talking about codes as defined by a set of variable nodes, a set of check nodes, and set of edges. In figure 1, we will examine a specific instance of this representation using a matrix $\mathbf{H}$ that we intentionally omit for the sake of conciseness in our explanation.

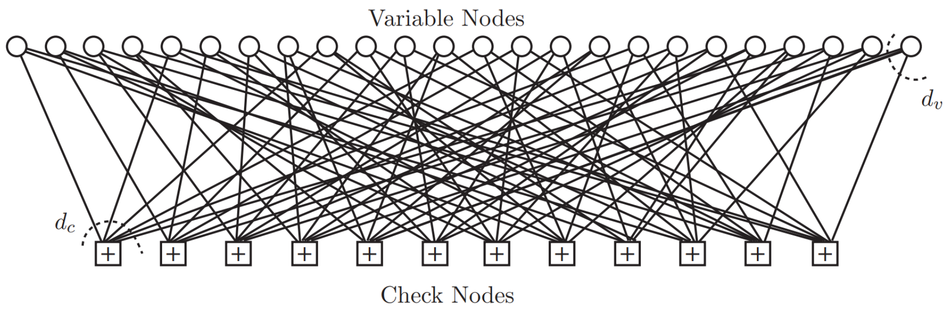


Figure 1: Example of a bipartite graph for (3, 6) regular LDPC code with parity-check matrix $\mathbf{H}$.

Let us now delve into the actual algorithmic framework for information reconciliation employing LDPC hashing. The procedure can be described as follows:

Algorithm 1 Information reconciliation via LDPC hashing

```
1: Input: $\underline{\mathbf{x}}$ ($k_A^s$: Alice sifted key), $\underline{\mathbf{y}}$ ($k_B^s$: Bob sifted key)
2: Output: $\underline{\mathbf{x}} = \underline{\mathbf{y}}$, minimizing the leaked information to eavesdropper
3: Split $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ in $L$ bits blocks
4: A computes the syndrome $\underline{\mathbf{s}}_i = \mathbf{H}\,\underline{\mathbf{x}}_i$ and sends it to B as $c_A = [\underline{\mathbf{s}}_1, \ldots, \underline{\mathbf{s}}_N]$
5: B performs syndrome reconciliation: finding $\hat{\underline{\mathbf{x}}}_i$ "closest" to $\underline{\mathbf{y}}_i$ s.t. $\mathbf{H}\,\hat{\underline{\mathbf{x}}}_i = \underline{\mathbf{s}}_i$
```

Modifications to code parameters are essential in the context of LDPC codes. Codes possessing robust algebraic properties or exhibiting optimal decoding performance are typically designed for specific values of n (length of codeword) and k (dimension of the code). However, it is imperative to acknowledge that these well-constructed codes are often optimized only for particular pairs of n and k. Consequently, there arises a need for code alterations to accommodate different values of n and k while still preserving favorable properties. These alterations are fundamental in deriving codes with similar desirable characteristics, thereby extending the applicability of LDPC codes to a broader range of parameters. Specifically, with reference to table 1, we will delineate the parameters of interest that yield optimal outcomes.

operation	n	k	$n-k$	rate (k/n)	minimum distance $(d_{\min})$
puncturing	$\downarrow$	$=$	$\downarrow$	$\uparrow$	$\downarrow$
shortening	$\downarrow$	$\downarrow$	$=$	$\downarrow$	$\uparrow$

Table 1: Table involving the parameters of interest for adapting LDPC codes. Puncturing involves selectively removing information bits from the codeword, resulting in a shorter but more resilient code (higher rate), while shortening reduces the length of the codeword by eliminating some bits, aiming for improved correction probability (higher $d_{\min}$) in specific applications.

Specifically, in terms of the bipartite graph representation, this corresponds to removing edges that connect variable nodes to certain check nodes. The resulting graph becomes sparser, reflecting the reduced redundancy in the code. On the other hand, shortening involves removing variable nodes and their associated edges. This results in a more compact graph structure, capturing the essence of the shortened codeword. The main idea is to maintain a consistent graph structure, ensuring that algorithms are constantly executed on the same underlying framework. That concept is encoded in figure 2, providing a visual representation of the aforementioned process.

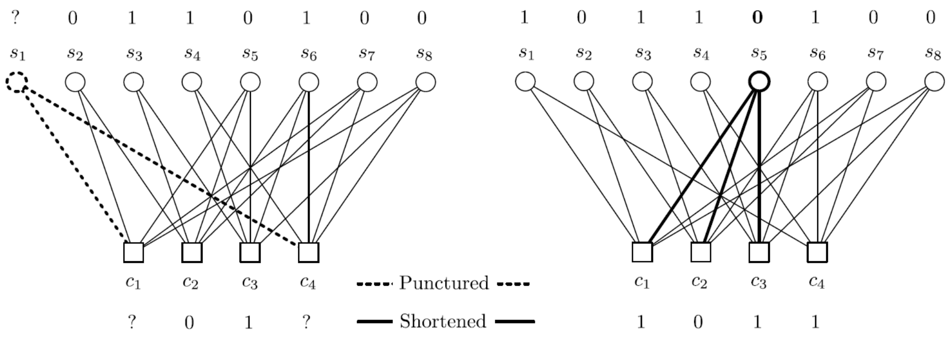


Figure 2: Examples of puncturing and shortening strategies applied to a linear code represented by its bipartite graph. In puncturing, one symbol is deleted from the word, transforming a (8, 4) code with a rate $R = 1/2$ into a (7, 4) code, increasing its rate. Instead, in shortening, one symbol is deleted from the encoding, converting the same (8, 4) code to a (7, 3) code, resulting in a decreased rate.

In terms of performance metrics in the information reconciliation procedure via LDPC hashing, we need first to consider the standard definition of efficiency which could be derived as follows:

$$f_1 = \frac{\lambda_{\text{EC}}}{n_Z \cdot h_2(Q_Z)} \tag{8}$$

where λ_{EC} represents number of disclosed bits, n_Z is the number of generated bits in the key basis, and $h_2(Q_Z)$ encodes the binary entropy of the QBER in the key basis. However, this metric in not so reliable if we consider low values of QBER; in fact, if we considered $Q_Z \approx 0$, also $h_2(Q_Z) \approx 0$. This would result in f_1 increasing and becoming much greater than 1, which is not the desired outcome. To address these issues, we can introduce another metric that solves these problems. The latter can be defined in the subsequent way:

$$f_2 = \frac{\lambda_{\text{EC}}}{n_Z} - h_2(Q_Z) \tag{9}$$

Therefore, f_2 can be interpreted as a composite metric that balances the efficiency of error correction (disclosed bits relative to generated bits) with a correction factor based on the binary entropy of the QBER. This provides a more refined assessment of the error correction process, considering both the quantity of successfully corrected bits and the information content associated with the randomness in the error distribution.

2.3 Cascade protocol

To draw a comparison with LDPC codes as discussed in section 2.2, we will implement the Cascade protocol. The subsequent parts will present a concise introduction and explanation of the Cascade protocol, delve into the information reconciliation for QKD algorithm, and provide some observations regarding the protocol. Cascade was firstly introduced by Brassard and Salvail [1] and it is known for its effectiveness in reconciling quantum key bits, especially in the presence of low QBER. It leverages iterative error correction and public discussion to progressively refine the raw key bits, enhancing the overall efficiency of the reconciliation process. The protocol is particularly suitable for QKD implementations where robust error correction is essential for establishing a secure and reliable cryptographic key between communicating parties. The Cascade protocol is designed to correct residual bit errors in Bob's key, which may deviate from the correct key initially sent by Alice. In a typical quantum key distribution scenario, like BB84, Alice possesses the correct key, while Bob's key has limited bit errors, indicating noise. To address this, they employ the Cascade protocol, where Bob, as the client, takes an active role in requesting information, and Alice, serving as the passive server, provides answers. Despite its association with quantum key distribution, Cascade is a fully classical protocol involving the exchange of classical messages, not quantum communications. Let's start by looking at the Cascade as a black box algorithm, and let's consider what the inputs and the outputs of the protocol are, as depicted in figure 3.

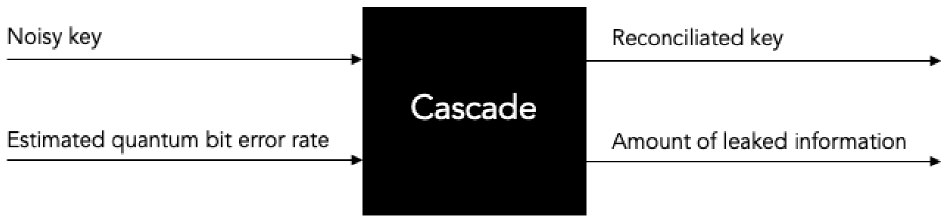

Noisy key
Estimated quantum bit error rate
Cascade
Reconciliated key
Amount of leaked information

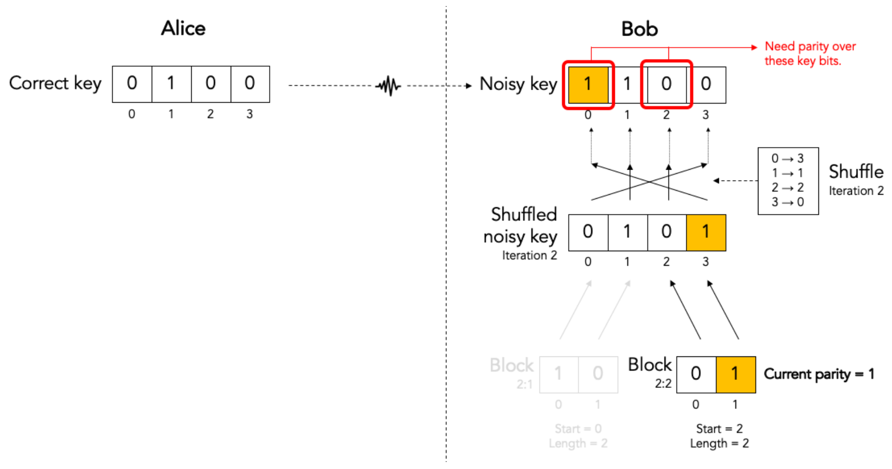
Figure 3: Scheme of inputs and outputs of the Cascade protocol.

In the Cascade protocol, Bob initiates the process after completing the quantum phase of quantum key distribution. The inputs to the protocol are Bob's noisy key, received from Alice, and an estimated Quantum Bit Error Rate (QBER) resulting from the quantum key distribution protocol. Bob's noisy key is essentially a classical string of bits with errors introduced during the quantum key distribution. The Cascade protocol aims to determine and correct these errors based on the QBER estimate. The output includes a reconciled key, but it's important to note that Cascade doesn't guarantee complete error correction. The reconciled key may still have a residual bit error rate, albeit significantly smaller than the original error rate. Cascade lacks a built-in mechanism to confirm the success of reconciliation or detect remaining errors and external validation is required. Additionally, Cascade tracks leaked information, especially regarding the parities computed during the process.

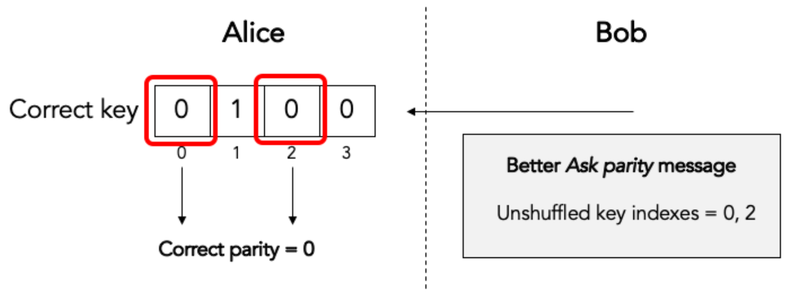
The amount of leaked information, measured in logical bits, is significant for the subsequent privacy amplification phase, helping determine the extent of amplification needed for enhanced security. Cascade involves multiple iterations, with each iteration representing an attempt by Alice and Bob to reconcile a single key. The original version of the protocol typically includes four iterations, and during each iteration, some bit errors in the key are corrected. While it is highly probable that all errors will be corrected by the end of the last iteration, it is not entirely certain. A distinctive feature of each iteration, except the first one, is the shuffling of the bits in Bob's noisy key at the beginning. Shuffling involves randomly reordering the bits in the key. It's essential to note that the shuffling is not meant to conceal the key from Eve, and the shuffling permutation may even be known to Eve. The key values, however, should remain undisclosed. Bob communicates the shuffle permutation to Alice for each iteration, and this information may be openly observed by Eve without compromising the security of the protocol. It's crucial to understand that in each iteration, every bit in the original unshuffled key ends up in a different position in the shuffled key. This process helps enhance the effectiveness of error correction over the course of multiple iterations in the Cascade protocol. During each iteration of the Cascade protocol, Bob shuffles the key and divides it into equally sized top-level blocks. The size of these blocks depends on the iteration number and the estimated QBER. The formula for the original Cascade protocol determines the block sizes for each iteration as follows:

$$\begin{aligned} k_1 &= \frac{0.73}{Q} \\ k_2 &= 2 \cdot k_1 \\ k_3 &= 2 \cdot k_2 \\ k_4 &= 2 \cdot k_3 \end{aligned} \tag{10}$$

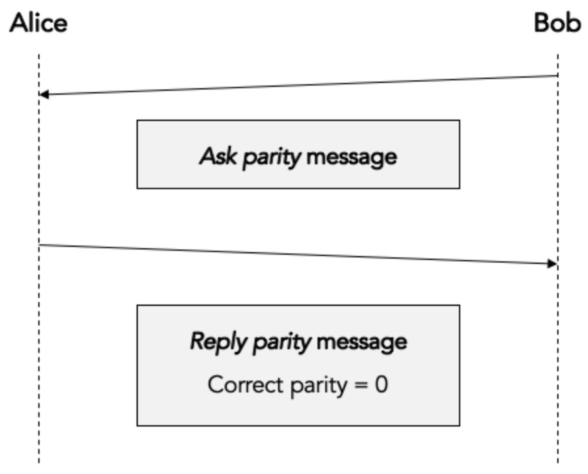
The block size is inversely proportional to the QBER, aiming to balance error correction efficiency and information leakage. Bob then initiates the task of detecting and correcting bit errors in each top-level block, ensuring minimal information leakage to eavesdropper Eve. This involves computing the current parity of each top-level block based on the shuffled noisy key. Bob also wants to know the correct parity from Alice's perspective. To achieve this, he sends an ask parity message to Alice, asking her to compute the correct parity. The whole procedure is depicted in figure 4. While a naive implementation of the ask parity message involves providing all information about the block, a more efficient approach involves Bob listing the unshuffled key indexes over which Alice must compute the parity (refer to figures 4a and 4b for additional details). Alice computes the correct parity and sends it back to Bob in a reply parity message (consult figure 4c for more details). Although this process does not reveal specific key bits, the disclosure of correct parity in the reply parity message leaks some information to Eve, which arises security concerns.



(a) Bob provides the specific unshuffled key indexes for which Alice needs to compute the parity.



(b) Alice computes the correct parity without reconstructing the shuffled key and block, streamlining the process.

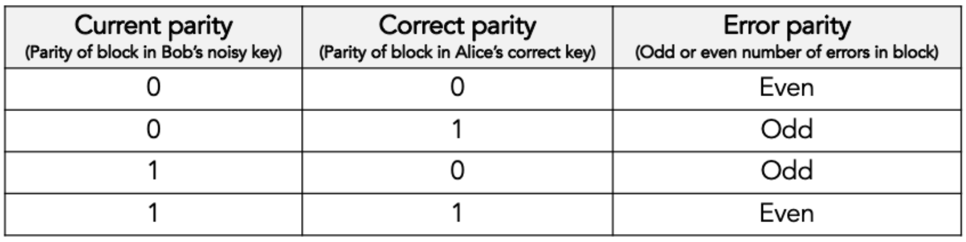


(c) Alice transmits the correct parity back to Bob through a reply parity message.

Figure 4: Full depiction of the parity bits communication between Alice and Bob.

Then, Bob proceeds with error correction using the Binary algorithm during each iteration. After shuffling the key and dividing it into top-level blocks, Bob aims to correct bit errors in each block.

He determines both the correct parity (from Alice's perspective) and the current parity of each block. Utilizing an error parity table, as illustrated in figure 5, Bob can identify whether the block contains an even or odd number of errors. Bob focuses on blocks with an odd error parity, indicating the presence of at least one error. For blocks with even error parity, he takes no action.



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

Figure 5: Error parity table used by Bob to identify odd/even number of errors in each block.

When correcting a single bit error in a block with an odd number of errors, Bob applies the Binary algorithm, which recursively identifies and corrects exactly one bit error. The Binary algorithm splits the block into sub-blocks, determining the error parity for each and recursively applying the algorithm to sub-blocks with odd error parity until reaching a sub-block of size one, as described in figure 6. The subsequent algorithm description offers a concise summary of this process.

Algorithm 2 Information reconciliation via Binary algorithm

```
1: Input: $\underline{\mathbf{x}}$ ($k_A^s$: Alice sifted key), $\underline{\mathbf{y}}$ ($k_B^s$: Bob sifted key)
2: Output: $\underline{\mathbf{x}} = \underline{\mathbf{y}}$, minimizing the leaked information to eavesdropper
3: Split $\underline{\mathbf{x}}$ and $\underline{\mathbf{y}}$ in $L$ bits blocks, such that:
       $$\mathbb{P}\left[d_{\mathrm{H}}(\underline{\mathbf{x}}_i, \underline{\mathbf{y}}_i) > 1\right] \approx 0, \ \forall \text{ block pair } (\underline{\mathbf{x}}_i, \underline{\mathbf{y}}_i) \quad (11)$$
4: A computes the parity of each block: $c_{A,i} = \bigoplus_{j=0}^{L-1} \underline{\mathbf{x}}_{ij}$ and sends the information
   to B over the public channel
5: B calculates the parity of each block: $c_{B,i} = \bigoplus_{j=0}^{L-1} \underline{\mathbf{y}}_{ij}$. Then, he verifies if:
   $c_{B,i} = c_{A,i} \ \forall i$, and a single parity check is sufficient due to the expectation
   of at most 1 bit error.
6: Upon detecting a mismatch ($c_{B,i} \neq c_{A,i}$), B transmits the indexes and corresponding parity values of affected blocks. The algorithm recursively divides
   each erroneous block in half, computes parity, and repeats until the error location is found. B then corrects the identified error in the block.
```

It's important to note that after correcting a single bit error in a block, the Cascade protocol may still leave an even number of errors in the block. These remaining errors are addressed through mechanisms like reshuffling in later iterations and the cascading effect.

Reshuffling in subsequent iterations creates opportunities to revisit blocks with an odd number of errors, allowing Bob to continue error correction. Correcting an error in one iteration has a ripple effect, impacting previous iterations and potentially causing a chain reaction of error corrections. This cascade effect enhances the overall efficiency of error correction in the Cascade protocol.

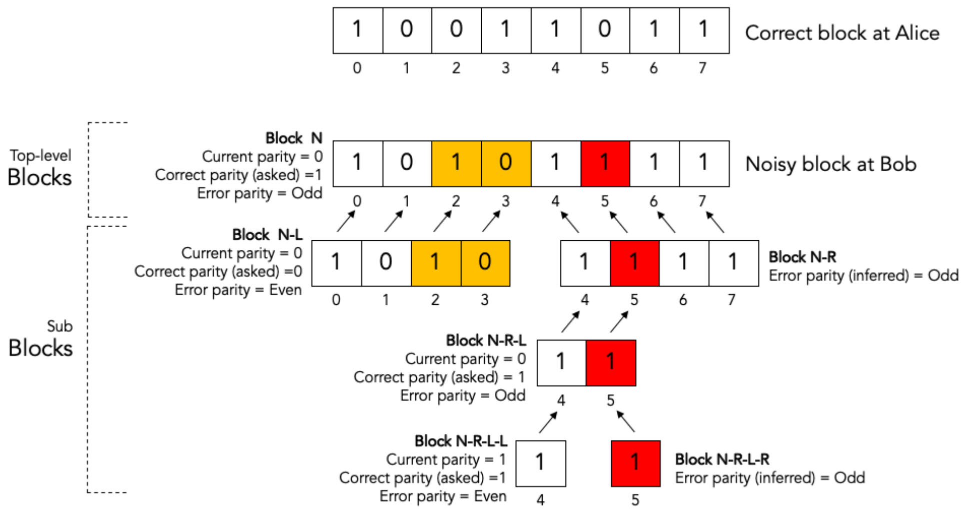


Figure 6: Simplified example of binary algorithm recursion procedure.

After examining the entire algorithm, let's analyze its limitations and how to quantitatively measure its performances. Considering the latter, we will assess the Cascade protocol using the metrics introduced in section 2.2. Specifically, equations 8 and 9 illustrate how to quantify its results. In terms of its limitations, the first one that comes to mind is the high message exchange overhead. The protocol involves frequent communication between Alice and Bob, particularly during the parity exchange phase. The need for numerous messages, including ask parity and reply parity messages, can result in increased latency and resource consumption. Moreover, Cascade relies only on classical communication between Alice and Bob. Unlike QKD protocols that leverage quantum communication, Cascade does not utilize quantum channels, limiting its ability to exploit quantum properties for enhanced security. Even within the algorithm, there are issues such as error correction over even blocks or dependence on the QBER estimate. To achieve improved results in terms of QKD information reconciliation, the Winnow protocol, proposed by Buttler et al. [2], partially addresses these concerns. For the sake of conciseness, we will not delve into Winnow as it is not part of our implementation.

3 Code implementation

Let us now proceed to the exploration of the code implementation encompassing all the aspects discussed in the preceding sections. Regarding LDPC codes, our reference will be the GitHub repository authored by Russian Quantum Center and Avesani [4]. The repository contains a Python 3 implementation of a symmetric reconciliation algorithm for QKD post-processing. The primary focus is on information reconciliation in the presence of QBER. The simulation involves generating random keys, introducing errors based on a specified QBER, and employing LDPC codes for error correction. Moreover, the repository provides tools to perform simulations, generate LDPC codes, and evaluate the reconciliation procedure. Let's now examine the repository's contents, as outlined below:

- `test_error_correction.py`: launches a simulation of the symmetric reconciliation and stores the results in `output.txt`;
- `error_correction_lib.py`: contains procedures for testing the reconciliation protocol, including key generation, error addition, LDPC code selection, syndrome encoding/decoding, and symmetric blind reconciliation. It also facilitates testing the entire reconciliation process;
- `codes_1944.txt`: pool of four standard LDPC codes with a block length of 1944 and specific rates;
- `codes_4000.txt`: pool of nine LDPC codes with a block length of 4000, constructed using an improved progressive edge-growing algorithm, with various code rates;
- `file_utils.py`: contains auxiliary procedures for reading files with LDPC codes.

Now, let's explore specific functions pivotal to our simulation, presented in `error_correction_lib.py`. Below, we will showcase details regarding two functions that play a significant role in our analysis.

```
def add_errors(a, error_prob):
    """
    Flip some values (1 -> 0, 0 -> 1) in 'a' with ↘
       probability 'error_prob'
    """
    error_mask = np.random.choice(2, size=a.shape, p↘
       =[1.0 - error_prob, error_prob])
    return np.where(error_mask, ~a+2, a)

def add_errors_prec(a, error_prob):
    """
    Add precisely 'error_prob' * length('a') errors in ↘
       key 'a'
    """
    len_a = len(a)
    n_er = int(round(len_a * error_prob))
    list_1 = list(range(0, len_a))
    list_2 = random.sample(list_1, n_er)
    K_cor = a.copy()
    for i in list_2:
        K_cor[i] = 1 - K_cor[i]
    return K_cor
```

Listing 1: BSC and deterministic errors simulation.

The role of `add_errors` is to simulate errors in a binary key by flipping bits based on a given error probability. This mimics the impact of quantum bit errors that can occur during the transmission of quantum states. Additionally, it models the Binary Symmetric Channel (BSC) scenario, where channel errors are simulated by a straightforward bit flip (XOR) based on the QBER. On the other hand, the function `add_errors_prec` introduces a precise number of errors into a binary key, determined by a specified error probability. This level of control is useful for fine-tuning the simulation and assessing the algorithm's performance under specific error conditions. In this context, it represents a deterministic scenario where errors are predefined based on a specified QBER. Now, let's explore some additional functions that will be of interest to us in the upcoming sections.

```
def choose_sp(qber, f, R_range, n):
    '''
    Choose appropriate rate and numbers of shortened and↘
       punctured bits
    '''
    def get_sigma_pi(qber, f, R):
        pi = (f * h_b(qber) - 1 + R) / (f * h_b(qber) - ↘
           1) # pi = ratio of punctured bits
        sigma = 1 - (1 - R) / f / h_b(qber) # sigma = ↘
           ratio of shortened bits
        return max(0, sigma), max(0, pi)
```

```python
    delta_min = 1
    R_min = None
    for R in R_range:
        sigma_c, pi_c = get_sigma_pi(qber, f, R)
        delta_c = max(sigma_c, pi_c)
        if delta_c < delta_min:
            delta_min = delta_c
            sigma_min = sigma_c
            pi_min = pi_c
            R_min = R
    if R_min is not None:
        return R_min, int(floor(n * sigma_min)), int(ceil(n * pi_min))

def generate_sp(s_n, p_n, k_n, p_list=None):
    '''
    Generates 's_n', 'p_n' and 'k_n' positions of shortened ('s_pos'), punctured ('p_pos') and key ('k_pos') symbols correspondingly.
    Punctured symbols are taken from 'p_list' if it is not None.
    If it is 'p_list' is None of 'p_n' is larger than number of elements in 'p_list', then they are token from the whole key.
    '''
    ## Refer to Github repository for more details ##

def extend_sp(x, s_pos, p_pos, k_pos):
    '''
    Construct extended key 'x' with shortened/punctured/key bits in positions 's_pos'/'p_pos'/'k_pos'
    '''
    ## Refer to Github repository for more details ##

def encode_syndrome(x, s_y_joins):
    """
    Encode vector 'x' with sparse matrix, characterized by 's_y_joins' (matrix multiplication mod 2)
    """
    m = len(s_y_joins)
    s = generate_key_zeros(m)
    for k in range(m):
        s[k] = (sum(x[s_y_joins[k]]) % 2)
    return np.array(s)
```

Listing 2: Shortening and puncturing bits selection, key extension, and syndrome encoding.

Let us provide a brief analysis to those functions, as follows:

- `choose_sp`: selects (automatically) an appropriate rate and the number of shortened and punctured bits based on the given parameters. It is crucial for determining the rate of key generation in a coding scheme, considering factors such as quantum bit error rate (`qber`), code efficiency (`f`), a range of rates (`R_range`), and the total number of bits (`n`);
- `generate_sp`; generates positions of shortened (`s_n`), punctured (`p_n`), and key (`k_n`) symbols in a key, either from a specified list (`p_list`) or from the entire key. It is useful for creating a key with specific configurations, such as shortened or punctured bits, based on the desired number of each type;
- `extend_sp`: constructs an extended key by placing shortened, punctured, and key bits at specified positions in the original key. It is essential for extending a key with specific configurations, as specified by the positions of shortened (`s_pos`), punctured (`p_pos`), and key (`k_pos`) bits.
- `encode_syndrome`: encodes a vector `x` using a sparse matrix defined by `s_y_joins` through matrix multiplication modulo 2. This function is integral to the error correction process, encoding the key vector with a sparse matrix to generate a syndrome.

In order to perform syndrome decoding, the `decode_syndrome_minLLR` function is designed for that task (refer to Russian Quantum Center and Avesani [4] for more details). Here's a summarized overview:

- **Objective**: decode a syndrome using the Min-Sum algorithm with Log-Likelihood Ratios (LLRs);
- Inputs:
  - `y`: decoding vector;
  - `s`: syndrome;
  - `s_y_joins`: parity-check matrix info (symbol nodes to check nodes);
  - `y_s_joins`: parity-check matrix info (check nodes to symbol nodes);
  - `qber_est`: estimated QBER;
  - `s_pos`, `p_pos`, `k_pos`: positions of shortened, punctured, and key bits;
  - `r_start`: vector of predefined LLRs;
  - `max_iter`: maximal number of iterations;
  - `x`: true vector for comparison and convergence check;
  - `show`: output parameter (1: decoding results, 2: iteration details);
  - `discl_n`: number of bits disclosed in each additional round;
  - `n_iter_avg_window`: iterations for mean LLR averaging for procedure termination.

- Ouputs:
  - `z`: decoded vector;
  - `minLLR_inds`: indices of symbols with minimal LLRs.
- Inner functions: `h_func`: approximation of $\log|e^x - 1|$, `core_func`: core function for LLR computation;
- Algorithm overview:
  1. Initialize parameters and LLRs;
  2. Iterate to update messages between check nodes and symbol nodes;
  3. Check for convergence based on correct syndrome;
  4. Output the decoded vector (`z`) and indices of symbols with minimal LLRs (`minLLR_inds`).
- Notes: it employs approximation methods for mathematical functions. Moreover, Min-Sum algorithm is used for message passing between check and symbol nodes. In addition, decoding continues until convergence or maximum iterations are reached.

To conclude, `perform_ec` and `test_ec` are designed to evaluate the efficiency and performance of error correction codes under the presence of QBER. Regarding the first function, we begin by creating an extended key that includes specified shortened and punctured bits. Subsequently, we encode the syndrome for both the original key and the key affected by errors. The procedure advances by computing the syndrome difference (`s_d`) and the sum of the extended keys (`key_sum`). We then employ the `decode_syndrome_minLLR` function to perform error correction. In cases where the correction is unsuccessful, we iteratively reveal additional information and repeat the correction process. Ultimately, we produce the corrected key along with the verification result. Regarding `test_ec`, we firstly choose a rate (`R`) and determine the number of shortened (`s_n`) and punctured (`p_n`) bits, then we generate an error correction code based on the selected rate and length. After that, we simulate error correction trials using the `perform_ec` function. Finally, we output and analyze the following statistics:

- `f_mean`: average efficiency of decoding. It represents the mean value computed across the number of trials (`n_tries`) for the previously introduced efficiency metric f_1 as outlined in equation 8;
- `com_iters_mean`: mean number of communication iterations;
- `R`: code rate;
- `s_n`, `p_n`: number of shortened and punctured bits;
- `p_n_max`: maximal number of punctured bits;
- `k_n = n - s_n - p_n`: effective bits of the key;
- `discl_n`: number of disclosed bits;
- `FER`: frame error rate.

4 Lab tasks and results

Let us now consider the assigned laboratory tasks. The initial one involves the manual adjustment of the count of shortened and punctured bits. This modification is executed on datasets characterized by varying QBER values, considering two distinct scenarios: a) a deterministic number of errors and b) a Binary Symmetric Channel (BSC). The primary aim is to determine the impact of these modifications on both the efficiency and the number of iterations inherent in the error correction process. Subsequently, the second task involves the implementation of an alternative error correction procedure, such as Cascade by Brassard and Salvail [1]. Following the code implementation, we need to conduct testing and evaluate the efficiency as a function of the QBER, drawing comparisons to the LDPC case. The analysis involves contrasting the efficiencies obtained using both the conventional definition of efficiency (refer to equation 8 for further details) and an alternative efficiency measure, denoted as f_2 (refer to equation 9 for more details).

4.1 First task: implementation and results

To achieve the previously defined objective, it is imperative to make adjustments to the `test_ec` function, allowing for manual specification of the number of shortened and punctured bits. Additionally, a distinction must be introduced to differentiate between errors computed in a deterministic case and those computed on a BSC, which is the default error computation method in the given code. Furthermore, by fixing the code rate, we can assess how the mean efficiency varies concerning the QBER and the number of shortened and punctured bits. Finally, the introduction of a discrimination between the two efficiencies (refer to equations 8 and 9) will be crucial in the subsequent section. Below, we introduce the `custom_test_ec` function and outline the modifications made compared to the original code.

```python
def custom_test_ec(qber, R, codes, n, s_n, p_n, n_tries,↘
    f_start=1, show=1, discl_k=1, efficiency_metric='f_1↘
   ', error_model='bsc'):
    k_n = n - s_n - p_n
    m = (1 - R) * n
    code_params = codes[(R, n)]
    s_y_joins = code_params['s_y_joins']
    y_s_joins = code_params['y_s_joins']
    punct_list = code_params['punct_list']
    syndrome_len = code_params['syndrome_len']
    p_n_max = len(punct_list)
    discl_n = int(round(n * (0.0280 - 0.02*R) * discl_k)↘
       )
    qber_est = qber
    f_rslt = []
    com_iters_rslt = []
    n_incor = 0
    print("QBER = {}, R = {}, s_n = {}, p_n = {}, ↘
     p_n_max = {}, discl_n = {}".format(qber, R, s_n, ↘
     p_n, p_n_max, discl_n))
    for i in range(n_tries):
      print(i, end=': ')
      x = generate_key(n - s_n - p_n)
      if error_model == 'bsc':
       y = add_errors(x, qber)
      elif error_model == 'deterministic':
       y = add_errors_prec(x, qber)
      add_info, com_iters, x_dec, ver_check = ↘
        perform_ec(x, y, s_y_joins, y_s_joins, ↘
        qber_est, s_n, p_n, punct_list=punct_list, ↘
        discl_n=discl_n, show=show)
      if efficiency_metric == 'f_1':
       f_cur = float(m - p_n + add_info) / (n - p_n -↘
         s_n) / h_b(qber)
      elif efficiency_metric == 'f_2':
       f_cur = (float(m - p_n + add_info) / (n - p_n ↘
        - s_n)) - h_b(qber)
      f_rslt.append(f_cur)
      com_iters_rslt.append(com_iters)
      if not ver_check:
        n_incor += 1
    print('Mean efficiency:', np.mean(f_rslt),
       '\nMean additional communication rounds:', np.↘
        mean(com_iters_rslt),
       '\nEffective R:', (R - (s_n/n)) / (1 - s_n/n -↘
         p_n/n))
    return np.mean(f_rslt), np.mean(com_iters_rslt), R, ↘
     s_n, p_n, p_n_max, k_n, discl_n, float(n_incor)/↘
     n_tries
```

Listing 3: custom_test_ec function to properly test our simulation setup.

Now, let us focus our attention to the conducted tests and the various scenarios considered, as can be seen from the following tables. In each scenario under consideration, the QBER range remains the same, unless explicitly specified otherwise, and is defined as `range(0.02, 0.017, 0.01)`. Moreover, in each case considered, the number of keys processed for each QBER value (`n_tries`) will always be equal to 10. Another constant parameter throughout the first lab task is the efficiency f_1 (refer to equation 8 for more details). Additionally, the parameters remain constant across all cases and include the code size (`n`), fixed rate (`R`), shortened bits (`s_n`), punctured bits `p_n`, error model (`error_model`).

n	R	s_n	p_n	error_model
1944	0.5	range(0, 400, 50)	0	bsc
1944	0.5	0	range(0, 400, 50)	bsc

Table 2: Different parameter configurations for LDPC codes tests with code size (n) of 1944, fixed rate (R) of 0.5, varying numbers of shortened bits (s_n) and punctured bits (p_n), using BSC as error model.

n	R	s_n	p_n	error_model
1944	0.5	range(0, 400, 50)	0	deterministic
1944	0.5	0	range(0, 400, 50)	deterministic

Table 3: Different parameter configurations for LDPC codes with code size (n) of 1944, fixed rate (R) of 0.5, varying numbers of shortened bits (s_n) and punctured bits (p_n), using deterministic as error model.

n	R	s_n	p_n	error_model
4000	0.5	range(0, 800, 100)	0	bsc
4000	0.5	0	range(0, 800, 100)	bsc

Table 4: Different parameter configurations for LDPC codes with code size (n) of 4000, fixed rate (R) of 0.5, varying numbers of shortened bits (s_n) and punctured bits (p_n), using BSC as error model. In this particular case, the QBER range is slightly different, and is defined as `range(0.02, 0.14, 0.01)`.

n	R	s_n	p_n	error_model
1944	0.6667	range(0, 300, 50)	0	bsc
1944	0.6667	0	range(0, 300, 50)	bsc

Table 5: Different parameter configurations for LDPC codes with code size (n) of 1944, fixed rate (R) of 0.6667, varying numbers of shortened bits (s_n) and punctured bits (p_n), using BSC as error model.

n	R	s_n	p_n	error_model
1944	0.6667	range(0, 300, 50)	0	deterministic
1944	0.6667	0	range(0, 300, 50)	deterministic

Table 6: Different parameter configurations for LDPC codes with code size (n) of 1944, fixed rate (R) of 0.6667, varying numbers of shortened bits (s_n) and punctured bits (p_n), using deterministic as error model.

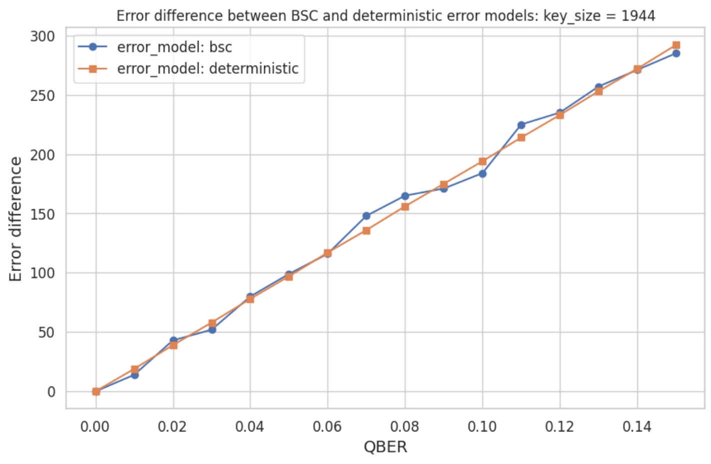
n	R	s_n	p_n	error_model
1944	0.75	`range(0, 200, 50)`	0	`deterministic`
1944	0.75	0	`range(0, 200, 50)`	`deterministic`

Table 7: Different parameter configurations for LDPC codes with code size (`n`) of 1944, fixed rate (`R`) of 0.75, varying numbers of shortened bits (`s_n`) and punctured bits (`p_n`), using deterministic as error model. In this particular case, the QBER range is slightly different, and is defined as `range(0.02, 0.14, 0.01)`.

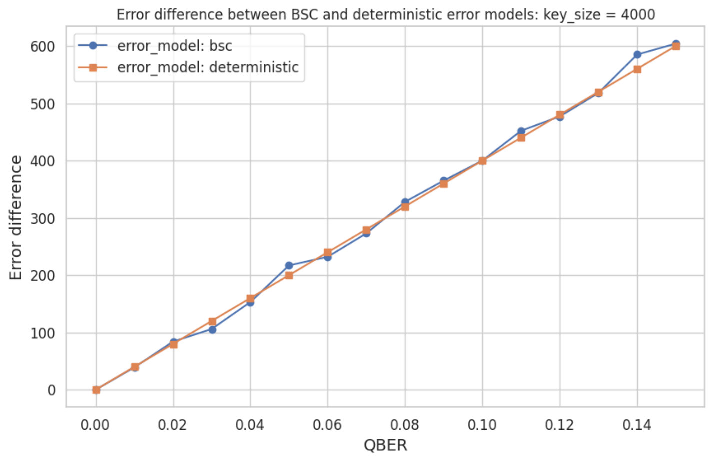
n	R	s_n	p_n	error_model
1944	0.8333	`range(0, 200, 50)`	0	`bsc`
1944	0.8333	0	`range(0, 200, 50)`	`bsc`

Table 8: Different parameter configurations for LDPC codes with code size (`n`) of 1944, fixed rate (`R`) of 0.8333, varying numbers of shortened bits (`s_n`) and punctured bits (`p_n`), using BSC as error model. In this particular case, the QBER range is slightly different, and is defined as `range(0.02, 0.12, 0.01)`.

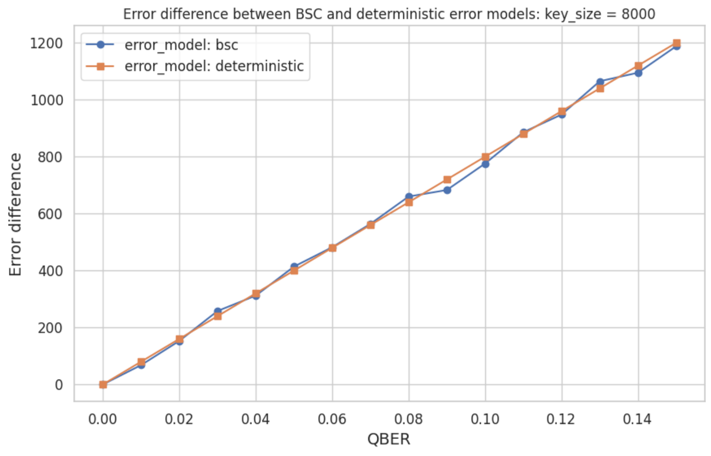
As evident from the preceding tables, the majority of tests were conducted under a BSC error model. The reason behind this choice, as opposed to performing identical calculations on both error models, is straightforward. The two error models align in the asymptotic limit ($n \to \infty$), and on average, they introduce the same errors in the cryptographic keys. This convergence is illustrated in figure 7, showcasing scenarios with key sizes of 1944, 4000, and double the size of the latter (8000) to emphasize the previously mentioned trend.



(a) Key size of 1944: still a noticeable difference in error models, although its magnitude is not so significant.



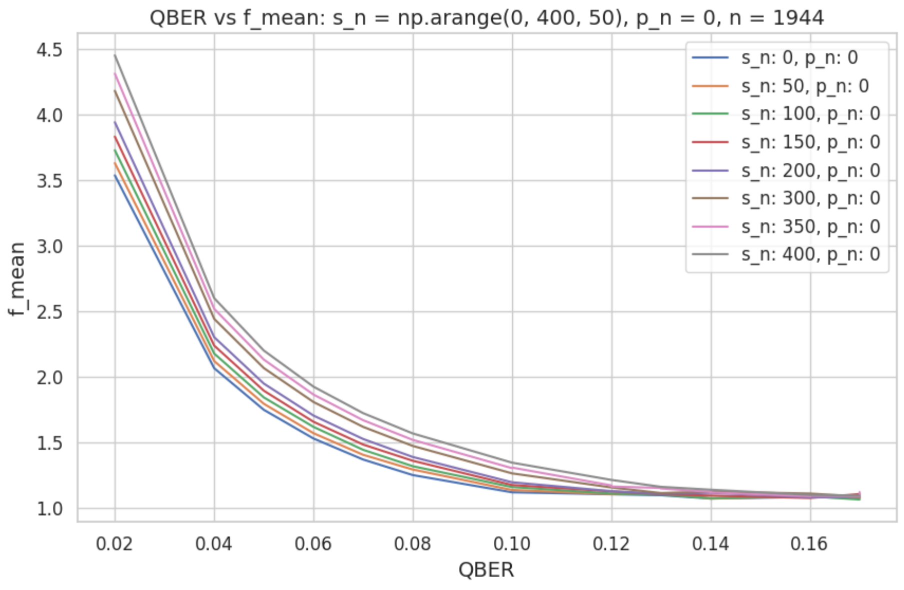
(b) Key size of 4000: error difference becomes increasingly minimal.



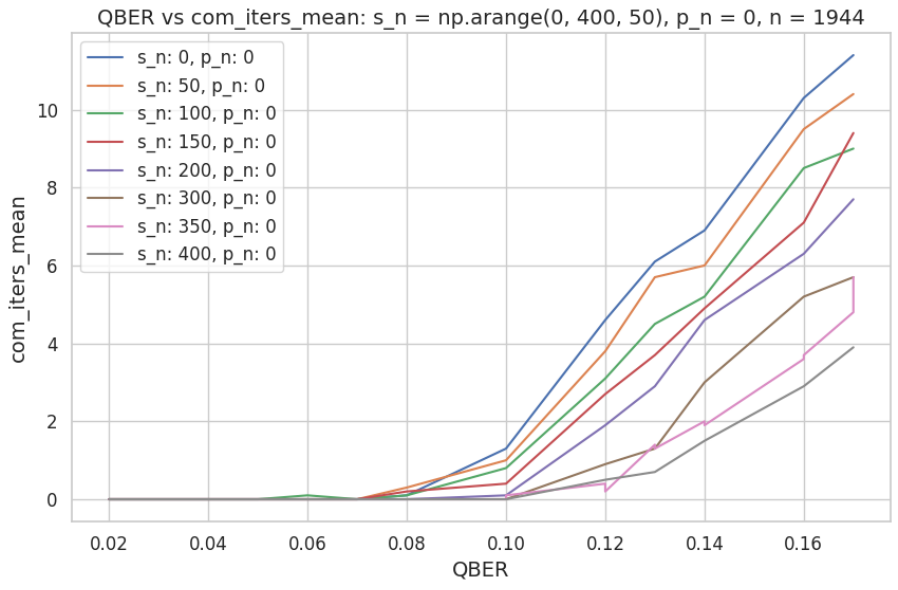
(c) Key size of 8000 bits: error difference is even more negligible.

Figure 7: Comparative analysis of error differences between BSC and deterministic error models for different key sizes.

Now, let's visualize all the preliminary findings from the conducted tests through plotting. Subsequently, we will derive conclusions based on the observed results.

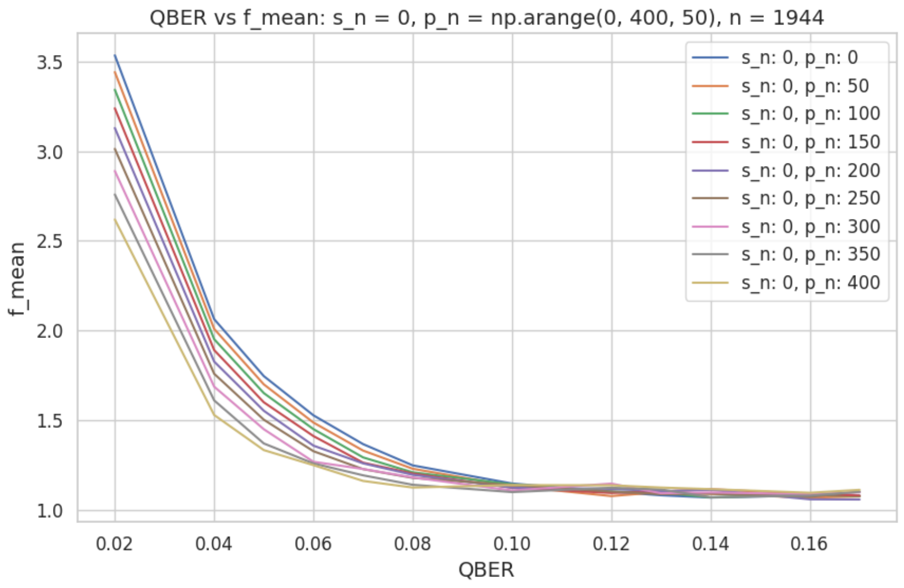


(a) QBER vs `f_mean`

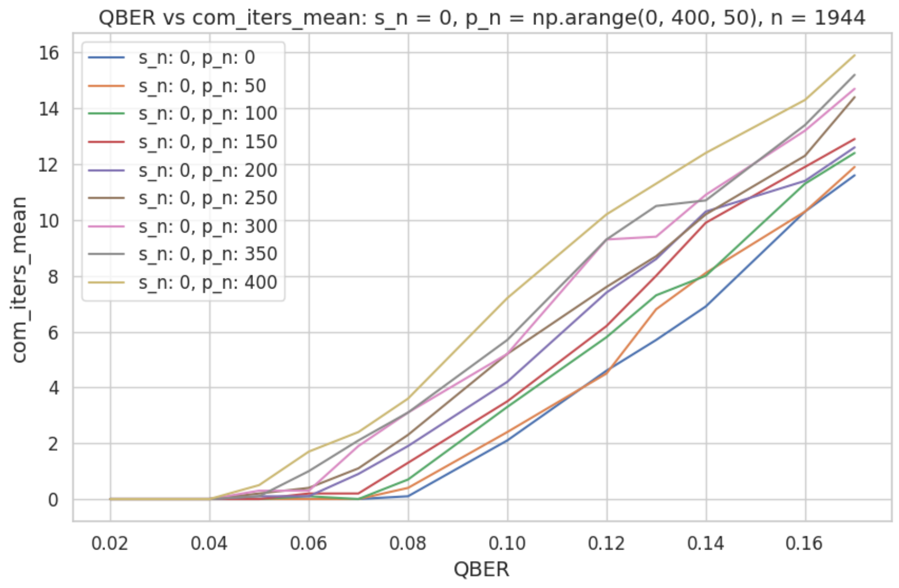


(b) QBER vs `com_iters_mean`

Figure 8: Preliminary results from the test set corresponding to table 2.

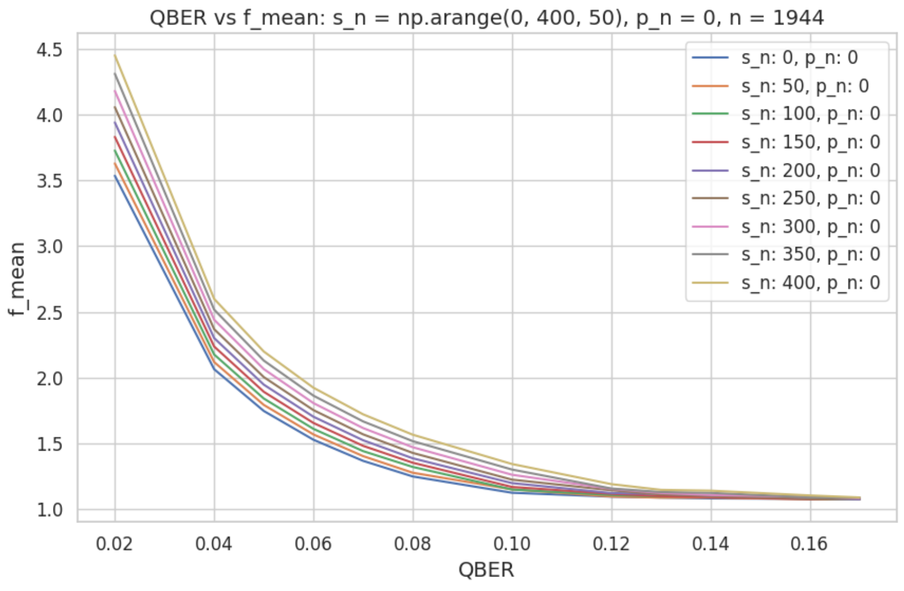


(a) QBER vs `f_mean`

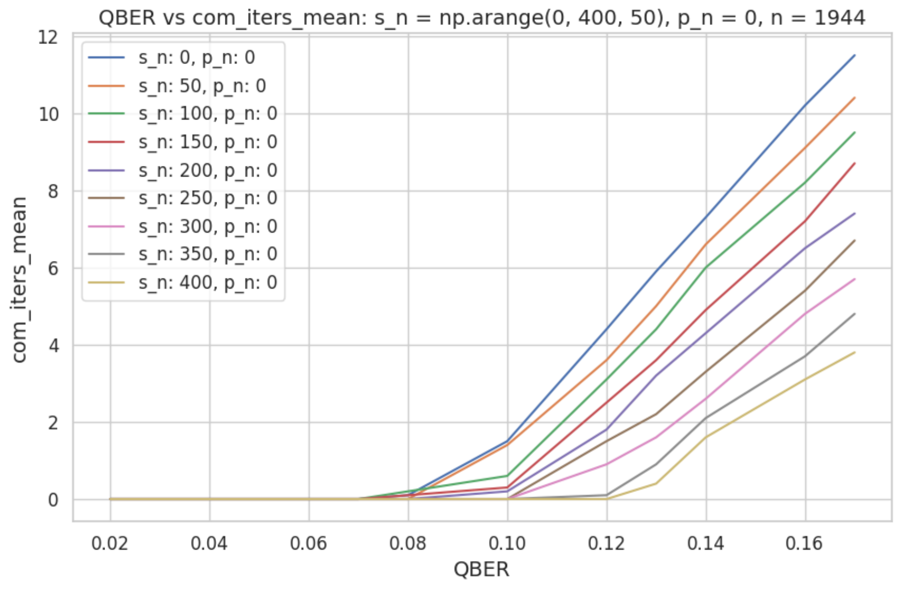


(b) QBER vs `com_iters_mean`

Figure 9: Preliminary results from the test set corresponding to table 2.

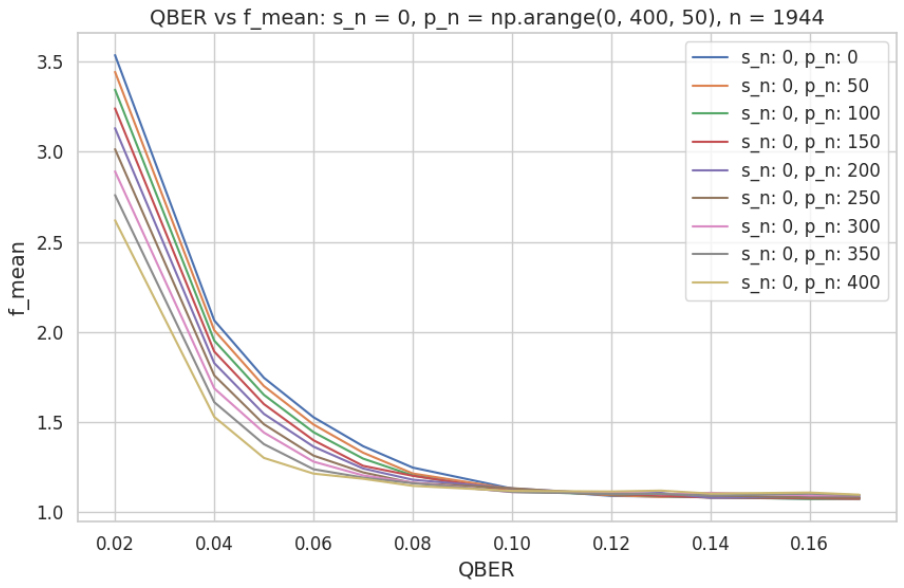


(a) QBER vs `f_mean`

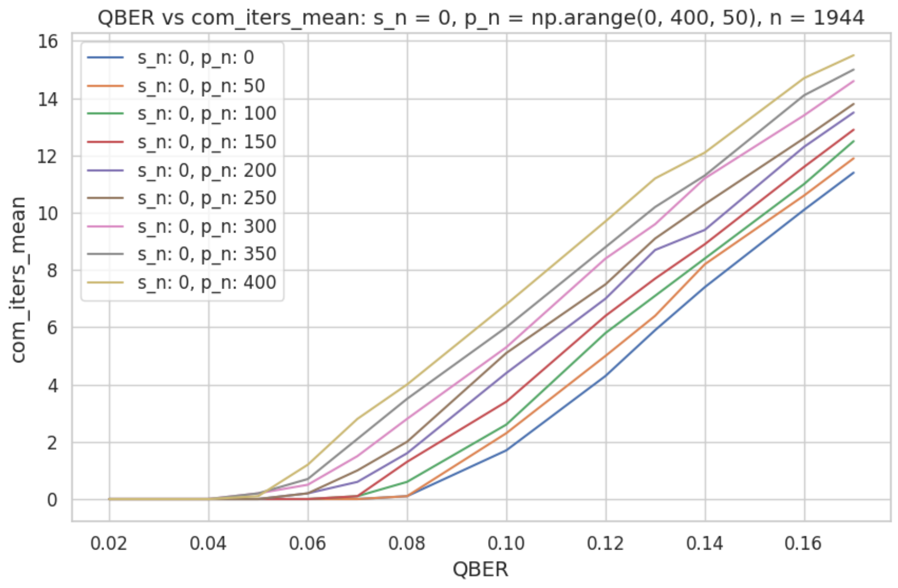


(b) QBER vs `com_iters_mean`

Figure 10: Preliminary results from the test set corresponding to table 3.

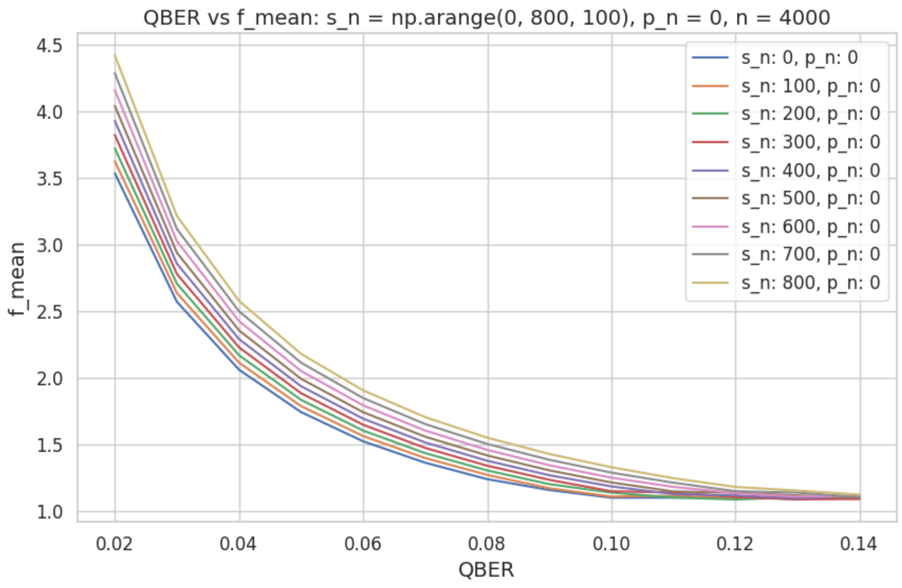


(a) QBER vs `f_mean`

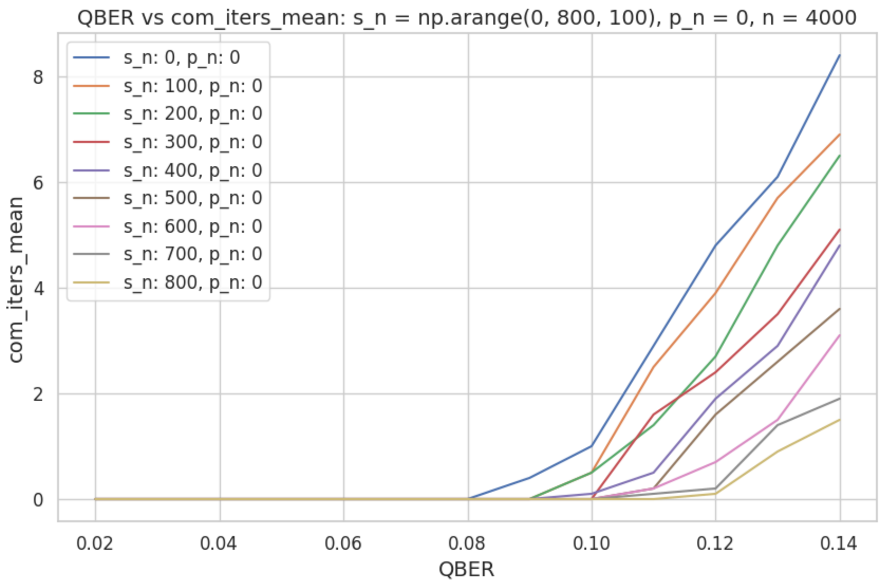


(b) QBER vs `com_iters_mean`

Figure 11: Preliminary results from the test set corresponding to table 3.

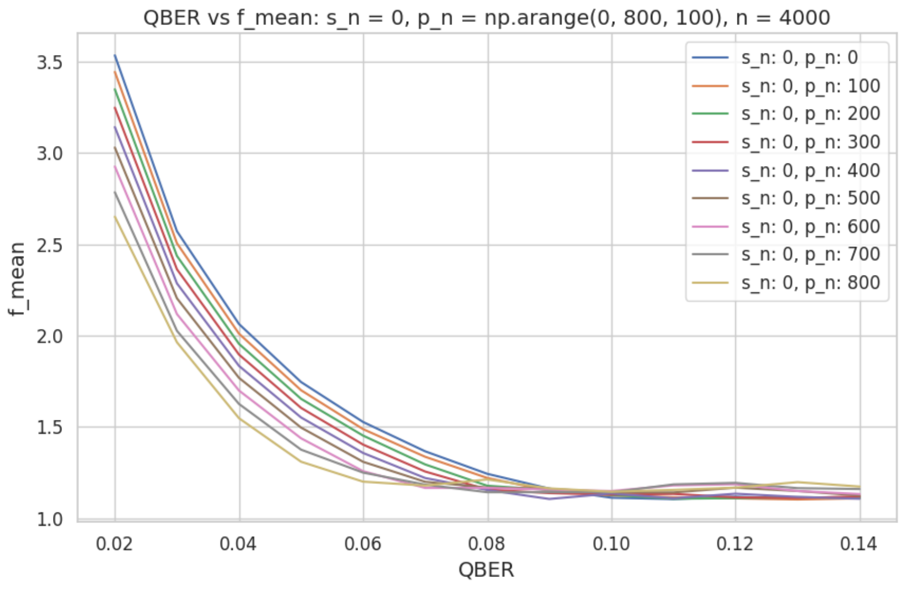


(a) QBER vs `f_mean`

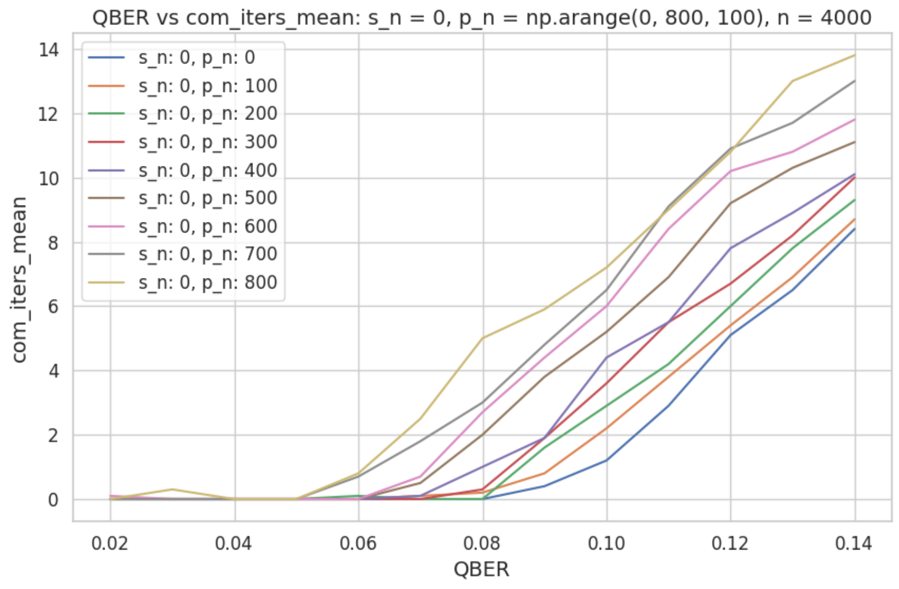


(b) QBER vs `com_iters_mean`

Figure 12: Preliminary results from the test set corresponding to table 4.

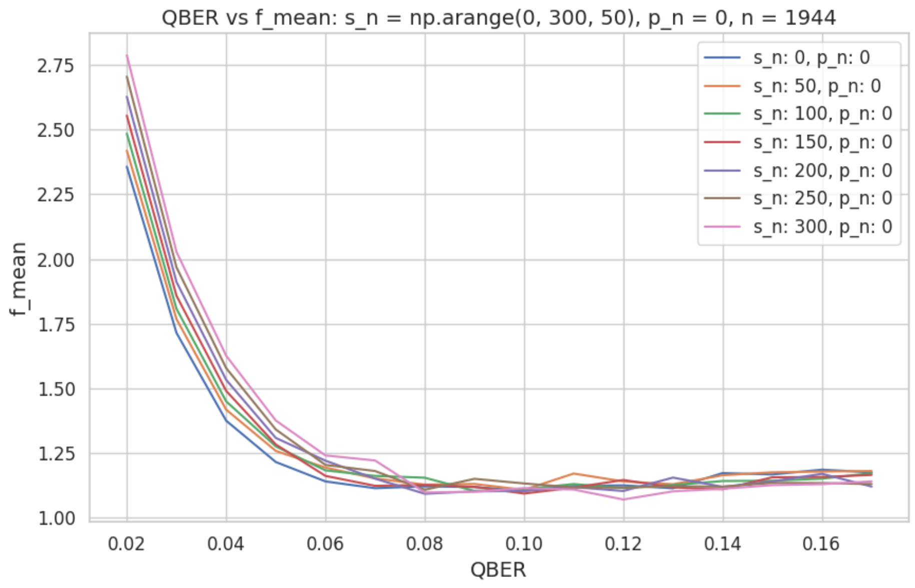


(a) QBER vs `f_mean`

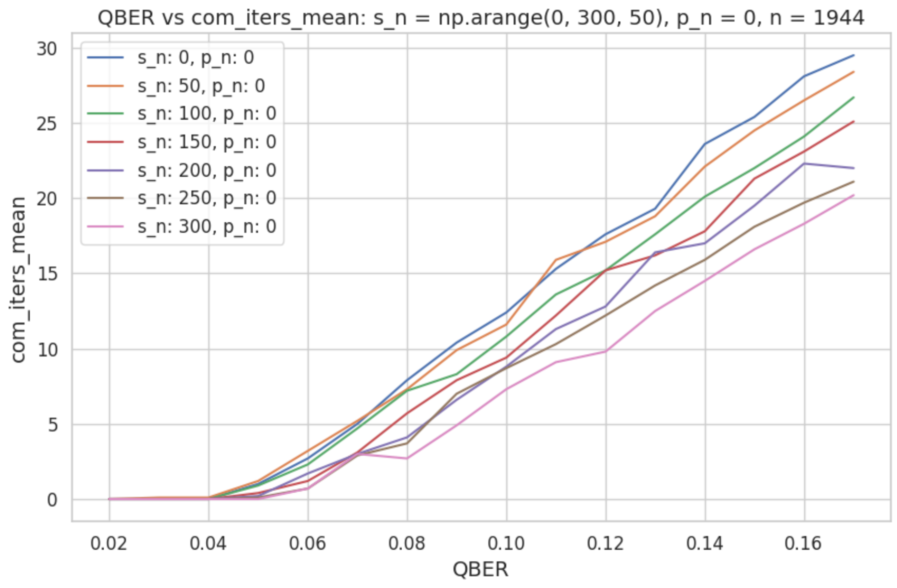


(b) QBER vs `com_iters_mean`

Figure 13: Preliminary results from the test set corresponding to table 4.

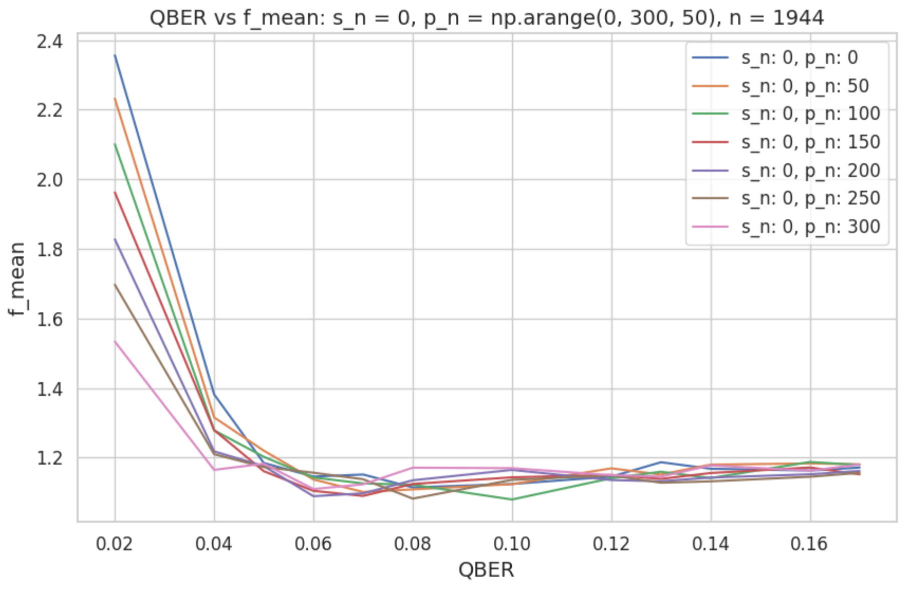


(a) QBER vs `f_mean`

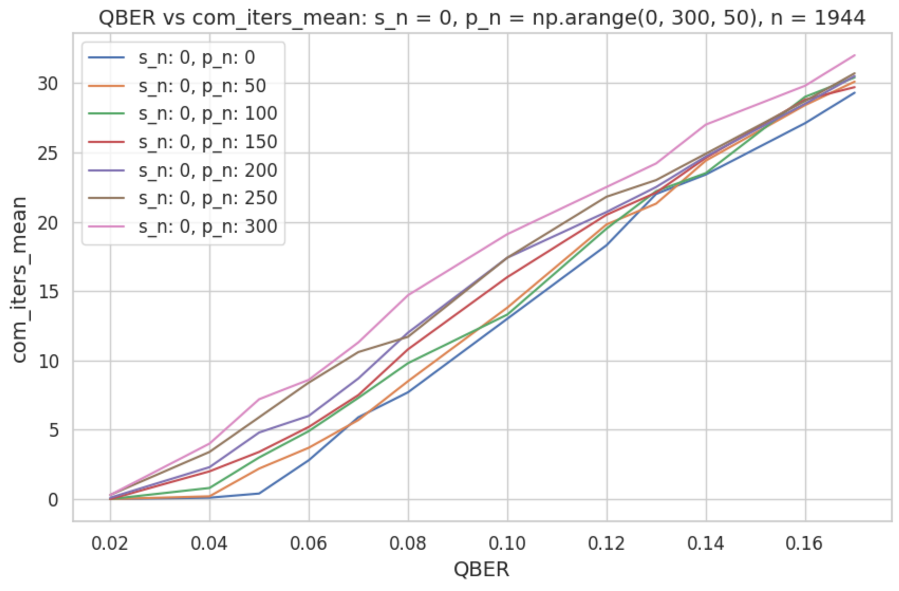


(b) QBER vs `com_iters_mean`

Figure 14: Preliminary results from the test set corresponding to table 5.

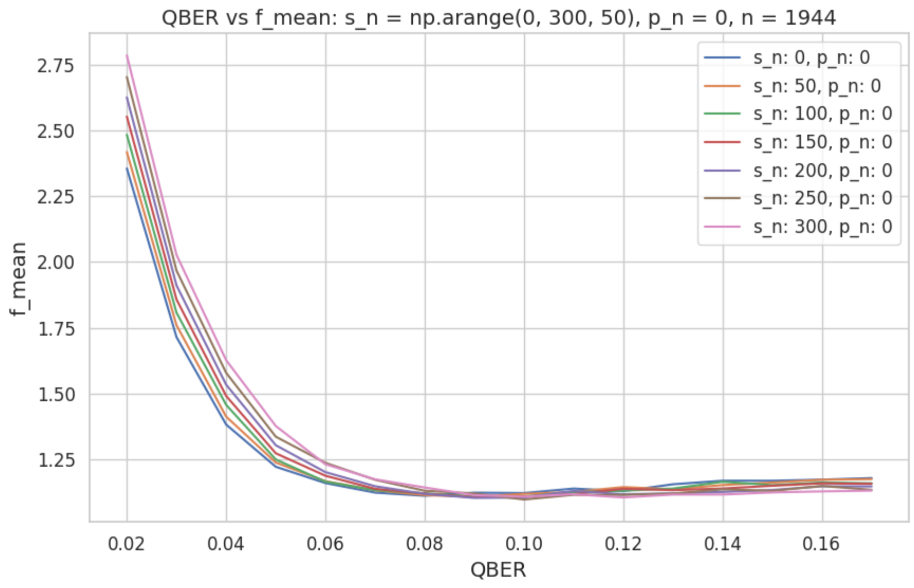


(a) QBER vs `f_mean`

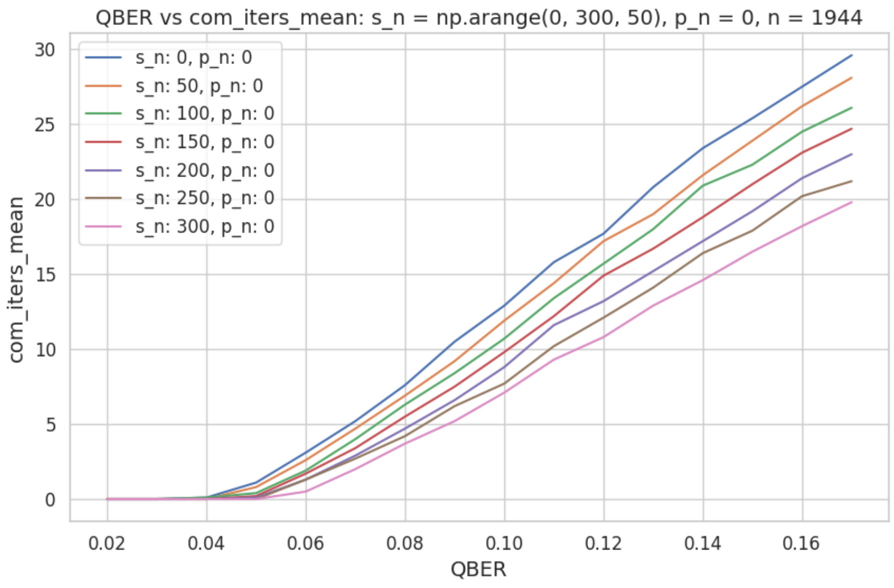


(b) QBER vs `com_iters_mean`

Figure 15: Preliminary results from the test set corresponding to table 5.

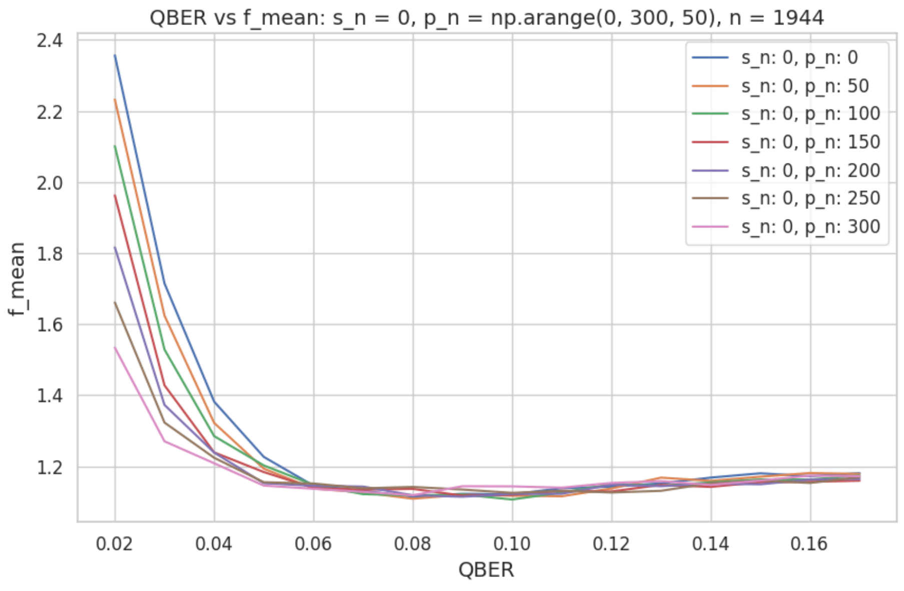


(a) QBER vs `f_mean`

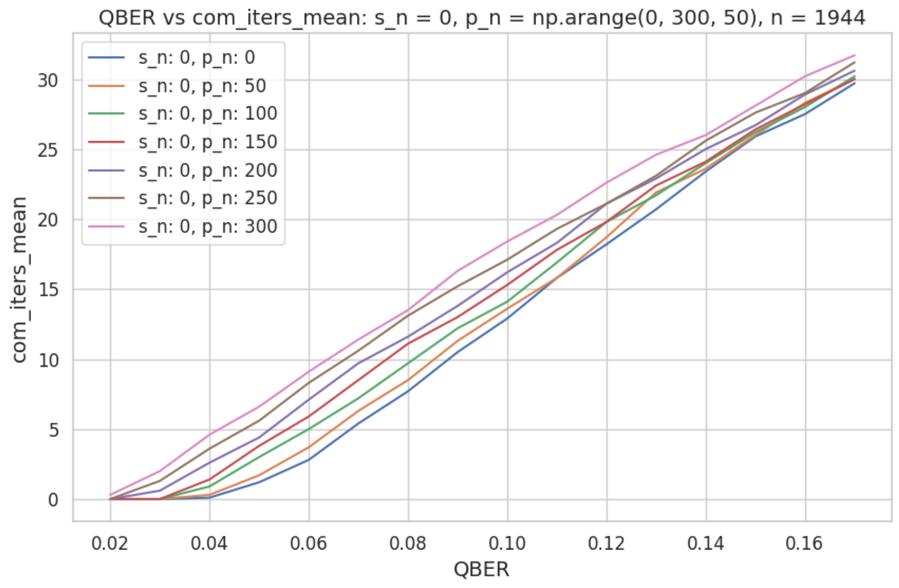


(b) QBER vs `com_iters_mean`

Figure 16: Preliminary results from the test set corresponding to table 6.

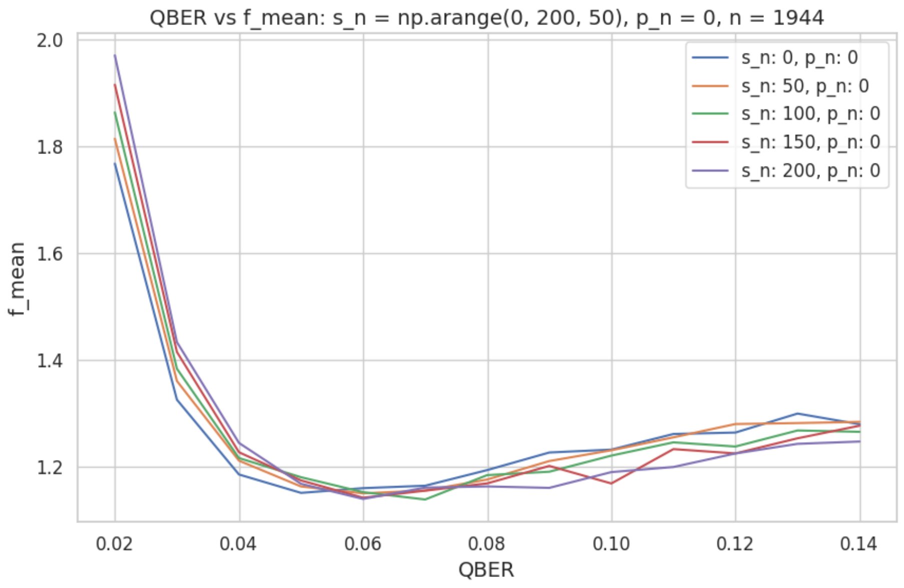


(a) QBER vs `f_mean`

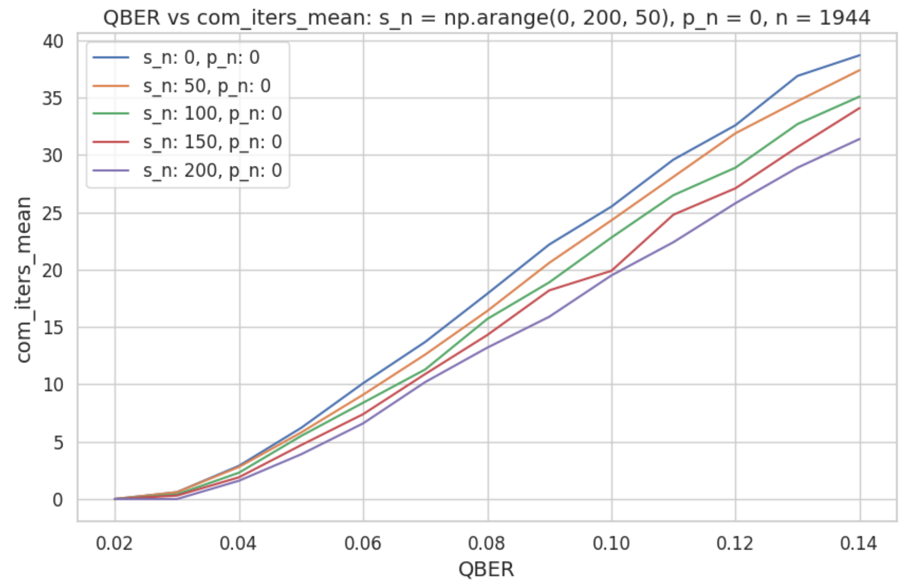


(b) QBER vs `com_iters_mean`

Figure 17: Preliminary results from the test set corresponding to table 6.

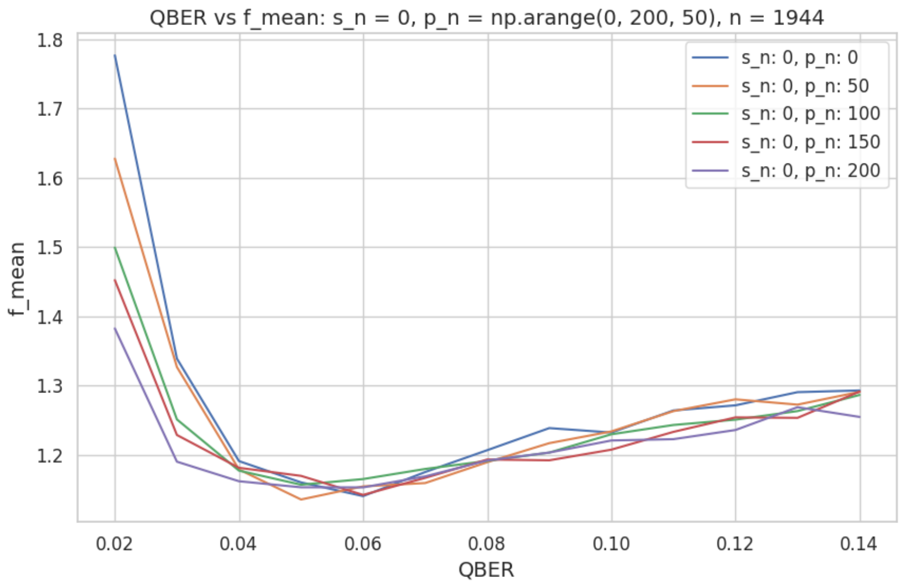


(a) QBER vs `f_mean`

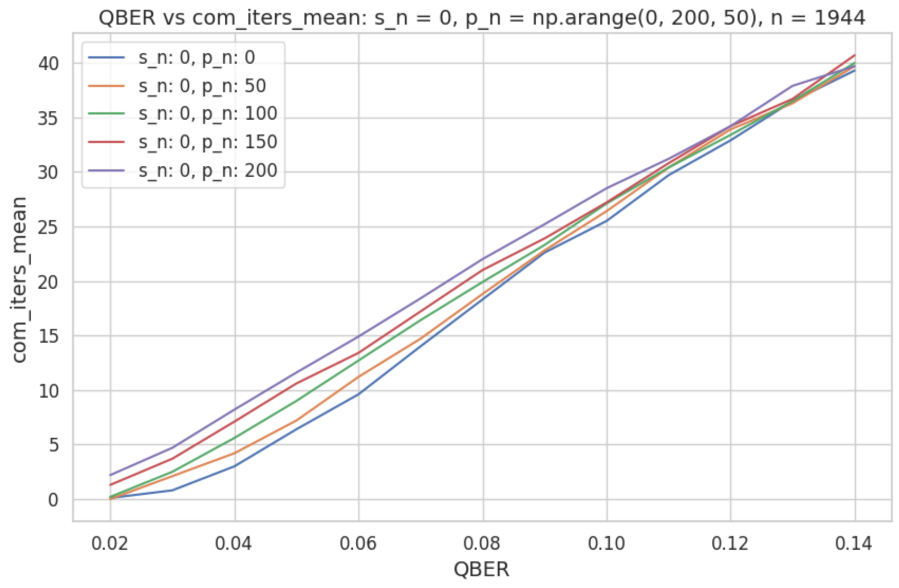


(b) QBER vs `com_iters_mean`

Figure 18: Preliminary results from the test set corresponding to table 7.

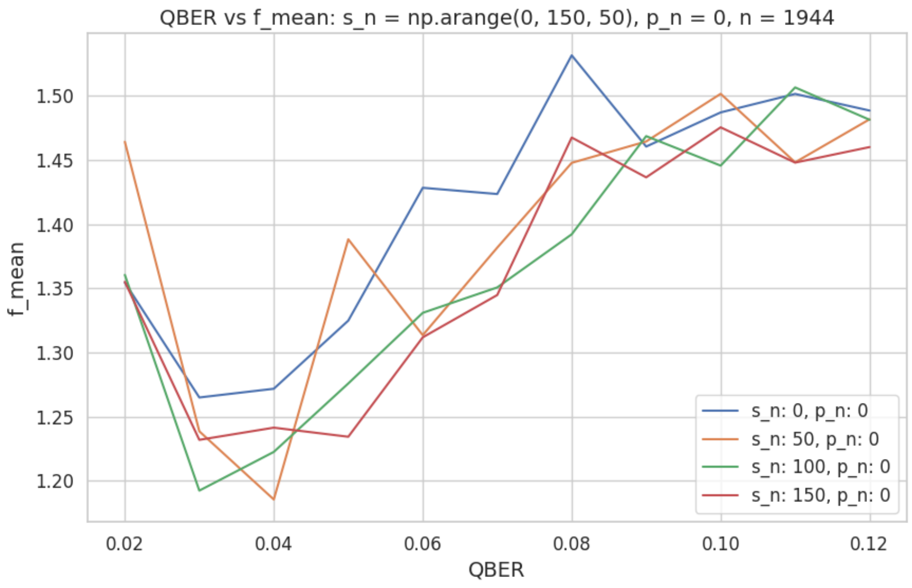


(a) QBER vs `f_mean`

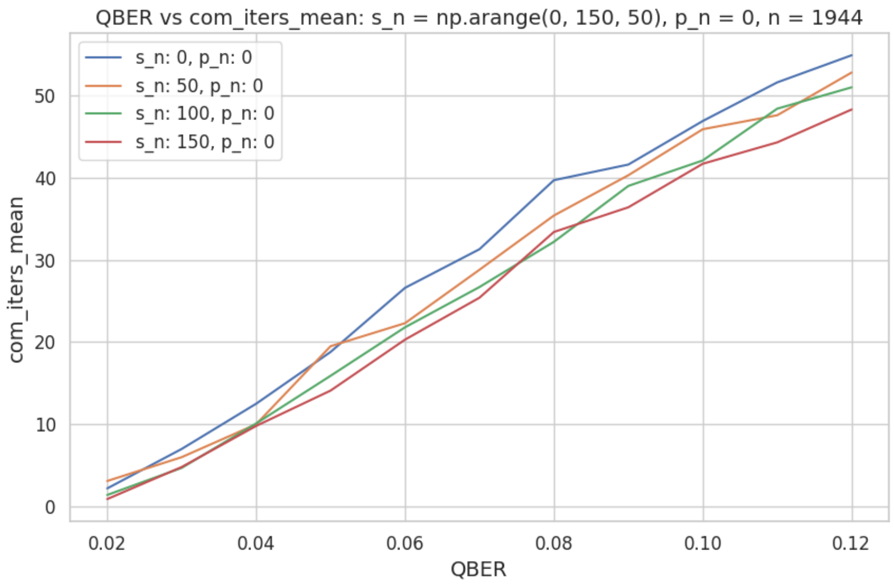


(b) QBER vs `com_iters_mean`

Figure 19: Preliminary results from the test set corresponding to table 7.

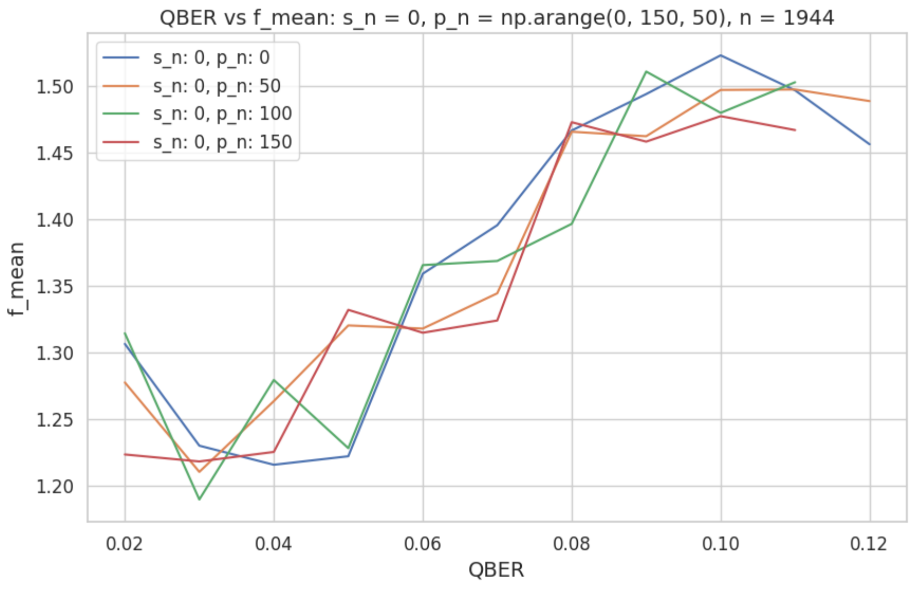


(a) QBER vs `f_mean`

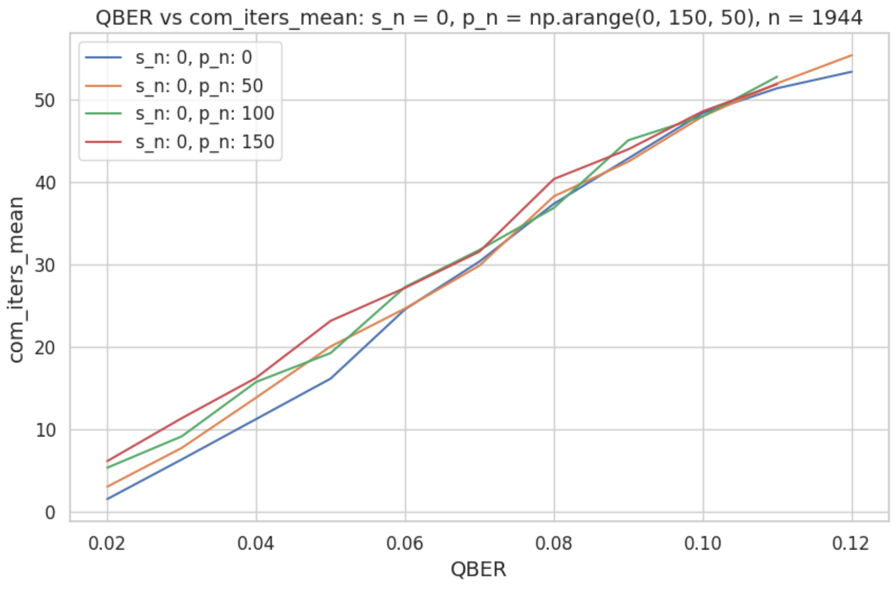


(b) QBER vs `com_iters_mean`

Figure 20: Preliminary results from the test set corresponding to table 8.



(a) QBER vs `f_mean`



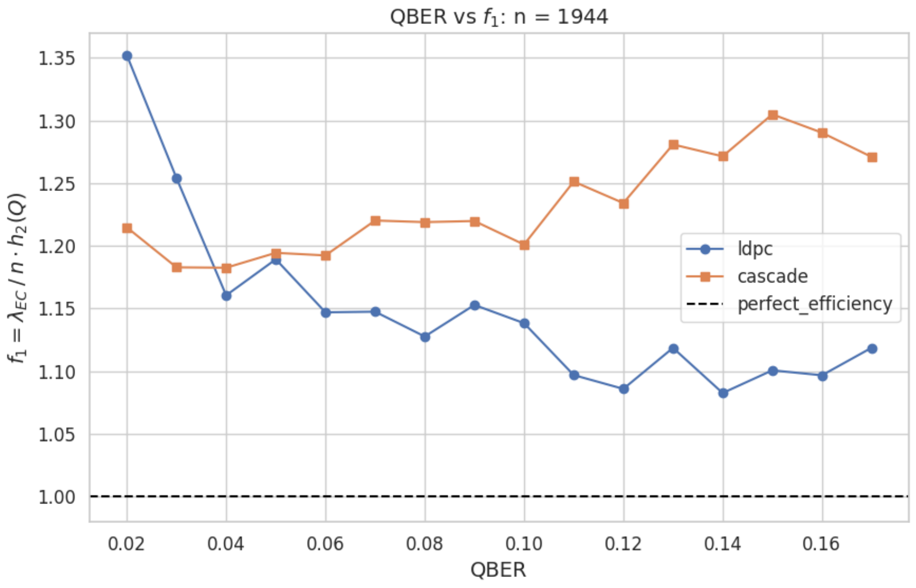
(b) QBER vs `com_iters_mean`

Figure 21: Preliminary results from the test set corresponding to table 8.

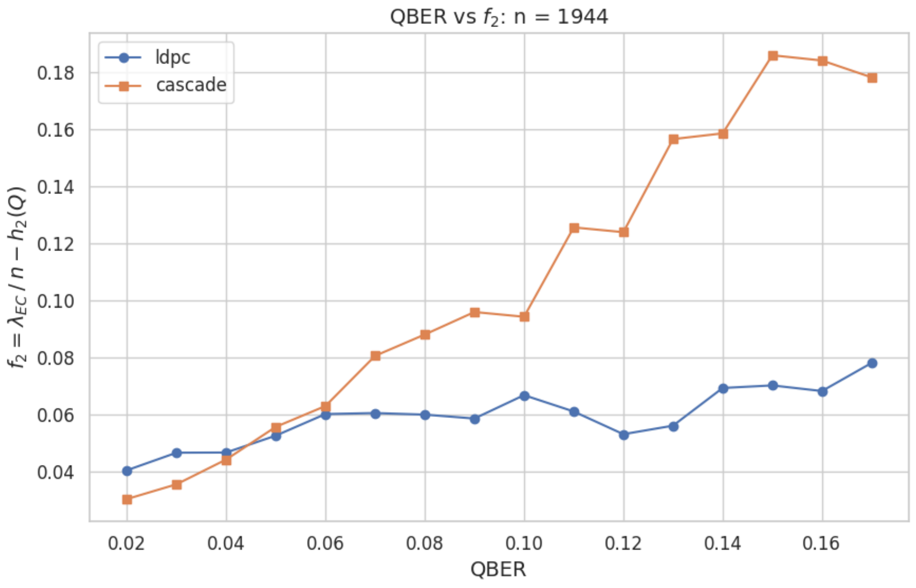
Firstly, let's specify that the conducted tests were considered based on `p_n_max` (maximum number of punctured bits), ranging from zero bits of shortening and puncturing up to `p_n_max` with intervals of 50 or 100 units. Additionally, most tests were performed on LDPC codes with 1944 bits, as those with 4000 bits required approximately 15 hours to execute at a rate higher than 0.5. However, trend variations due to different cases are also observable in the 1944 bits scenario, which can thus be extended to the 4000 bits case. As can be seen from figures 8, 9, 10 and 11, for the code with the lowest rate of 0.5, there is not much difference, except that when using puncturing, the `f_mean` starts at a more favorable (lower) value. This is attributed to the fact that a rate of 0.5 is too low for low QBERs, but for high QBER, it can attain good efficiency irrespective of puncturing/shortening. Regarding the `com_iters_mean` (average number of iterations per communication), the trend is generally the same, and they clearly increase with higher QBER: in fact, for low QBER values, there are approximately 0 `com_iters_mean`. Also, in the case of puncturing, the number of iterations is higher compared to the shortening case. This phenomenon can be explained by the impact of puncturing on enhancing the effective rate of the code, particularly when the initial rate is low. The higher effective rate, in turn, leads to an increased number of communications per iteration during the decoding process. All these observations can be extended to the case of 4000 bits, as illustrated in the figures 12 and 13. In the scenario involving the code with a rate of 0.6667, the general pattern remains consistent with previous plots. Initial efficiencies show improvement, especially when employing puncturing. As the QBER rises, however, the efficiency eventually decreases. Notably, the `com_iters_mean` exhibits a similar trend, increasing with higher QBER values. Those results are depicted in figures 14, 15, 16, 17. Concerning the code with a rate of 0.75, as shown in figures 18 and 19, puncturing is notably effective in enhancing the effective rate of the code. This results in an efficiency boost, reaching 1.38 compared to the baseline of 1.8 without puncturing. However, it's noteworthy that higher rate codes experience a decline in performance under conditions of elevated QBER. However, it's important to note that higher code rates lead to an increase in `com_iters_mean`, indicating a more challenging decoding process, particularly evident in cases of high QBER. Finally, we will conclude our preliminary analysis with the observations on the code with a rate of 0.8333. This specific code exhibits optimal performance for low QBERs, particularly when accompanied by a high number of puncturing bits. However, it experiences a rapid decline in efficiency at higher QBER values, as can be seen from figures 20 and 21. Note that executing simulations for this rate requires a significant amount of time, particularly when subjected to high QBER conditions, leading to a reduced number of data points in the generated plots.

4.2 Second task: implementation and results

So far, we have exclusively addressed LDPC codes for information reconciliation and all the tests have been conducted solely on the f_1 efficiency metric. Moving forward, we will implement the Cascade protocol, precisely as described in section 2.3, to compare its performance with LDPC codes in terms of the two efficiencies, f_1 and f_2 (refer to equations 8 and 9). Furthermore, we will consider in LDPC codes the `choose_sp` function to automatically adapt the number of shortening/puncturing bits to the QBER. For instance, if the QBER is low, a high rate will be chosen, and consequently, a certain number of shortening and puncturing bits will be associated with it. Clearly, this procedure optimizes and speeds up the use of LDPC codes in error correction. Let us now plot what presented so far and draw some conclusions from our analysis.

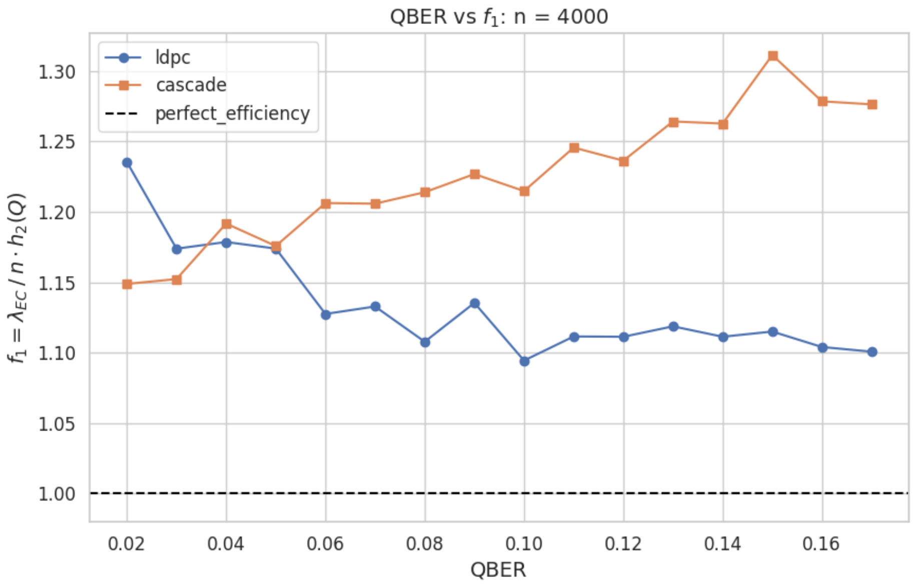


(a) QBER vs f_1

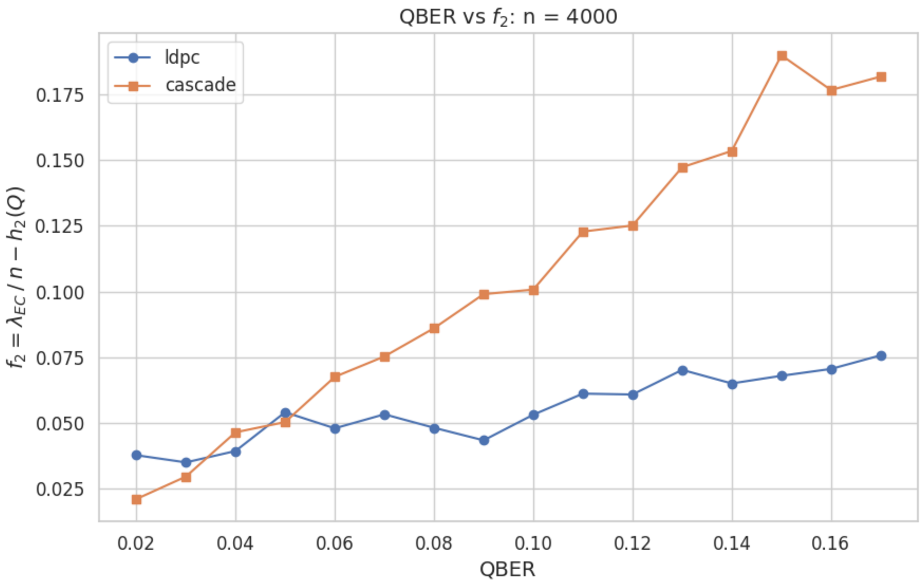


(b) QBER vs f_2

Figure 22: Comparative performance analysis: LDPC vs Cascade protocol in terms of f_1 and f_2 efficiency metrics, for n = 1944.



(a) QBER vs f_1



(b) QBER vs f_2

Figure 23: Comparative performance analysis: LDPC vs Cascade protocol in terms of f_1 and f_2 efficiency metrics, for n = 4000.

As can be seen in figure 22, in terms of f_1 efficiency metric, LDPC codes exhibit superior performance compared to the Cascade protocol. This advantage is particularly pronounced for moderate to high QBER conditions. However, it's noteworthy that Cascade outperforms LDPC codes at low QBER levels. This behavior can be attributed to the interplay of error correction capacity (λ_{EC}), key length (n) and the binary entropy of the QBER ($h_2(Q)$) in the f_1 definition. The higher efficiency of Cascade at low QBER may be associated with its specific error correction mechanisms and adaptability to varying QBER levels. Instead, for the f_2 efficiency metric, Cascade consistently achieves better performance compared to LDPC codes across different QBER levels, as can be retrieved in figure 23. The definition of f_2 involves subtracting the function $h_2(Q)$ from the error correction capacity per bit. This indicates that, in the context of privacy amplification, Cascade provides more effective protection against information leakage and achieves higher efficiencies. The specific characteristics of Cascade, such as its inherent privacy amplification mechanisms, contribute to its superior performance in scenarios where preserving information security is of utmost importance. Additionally, two other crucial points are to be specified. Firstly, Cascade proves significantly faster as a protocol compared to the use of LDPC codes for both key lengths. This is naturally consistent with the simplicity of the Cascade protocol compared to the bipartite graphs of LDPC codes. However, the number of asked parity bits for error correction in Cascade is enormous compared to the `com_iters_mean` of LDPC: this indicates that the Cascade protocol requires an enormous number of communications between the two parties (that's also why Winnow by Buttler et al. [2] was introduced), while in the case of LDPC, the number of iterations is substantially lower.

5 Conclusions

In our tests, LDPC codes with a block length of 1944 bits were analyzed, extending observations to a 4000 bits scenario. For the lowest rate, minimal differences were noted, except for puncturing, resulting in a more favorable starting efficiency, while the `com_iters_mean` increased with higher QBER (especially in puncturing case). Similar trends were observed for higher rates, with better starting efficiencies, particularly with puncturing. Puncturing in the 0.75 rate enhanced efficiency, but higher rate codes exhibited decreased performance at elevated QBER. The highest rate demonstrated optimal performance at low QBER but significant efficiency loss at higher QBER, with time-consuming simulations. Comparing LDPC codes and the Cascade protocol, LDPC codes excelled in f_1 efficiency, especially at moderate to high QBER. Cascade outperformed at low QBER levels, attributed to error correction capacity, key length, and binary entropy. For f_2, Cascade consistently performed better across QBER levels, showcasing its effectiveness in privacy amplification. Cascade's simplicity and faster execution were noted, accompanied by a need for extensive communication due to asked parity bits. These findings shed light on the trade-offs between LDPC codes and the Cascade protocol under different conditions. If there is an interest in delving into the developed code, it can be found at the following link: `QC&S-Lab-02-QKD-IR-Report`.

References

[1] G. Brassard, and L. Salvail. Secret-Key Reconciliation by Public Discussion. International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology, EUROCRYPT, pp. 410–423. 1994.
URL: https://link.springer.com/chapter/10.1007/3-540-48285-7_35;

[2] W. T. Buttler, S. K. Lamoreaux, J. R. Torgerson, G. H. Nickel, C. H. Donahue, and C. G. Peterson. Fast, efficient error reconciliation for quantum cryptography. Physical Review A, vol. 67, no. 5, p. 052303, May 2003.
URL: http://link.aps.org/doi/10.1103/PhysRevA.67.052303;

[3] C. H. Bennet, and G. Brassard. Quantum cryptography: Public-key distribution and coin tossing. Proceedings IEEE Int. Conf. on Computers, Systems and Signal Processing, Bangalore, India, pp. 175–179, December 1984.
DOI: 10.1016/j.tcs.2014.05.025.
URL: http://dx.doi.org/10.1016/j.tcs.2014.05.025;

[4] Russian Quantum Center, and M. Avesani. Symmetric information reconciliation for the QKD post-processing procedure, 2016.
URL: https://github.com/marcoavesani/QKD_LDPC_python.