Cascade-Python

Release 0.0.1

Bruno Rijsman

Apr 20, 2020

CONTENTS:

CHAPTER

ONE

INTRODUCTION.

1.1 What is in this GitHub repository?

This GitHub repository contains a Python implementation of the Cascade information reconciliation protocol. Information reconciliation protocols in general, and the Cascade protocol in particular, are a small but important and complex classical post-processing step in quantum key distribution (QKD) protocols. They are intended to detect and correct inevitable bit errors in the distributed key.

This repository also contains Python scripts that analyze the Cascade protocol and reproduce the analysis results that were previously reported in the following academic papers:

- Jesus Martinez-Mateo, Christoph Pacher, Momtchil Peev, Alex Ciurana, and Vicente Martin. Demystifying the Information Reconciliation Protocol Cascade. arXiv:1407.3257 [quant-ph], Jul 2014.
- André Reis. Quantum Key Distribution Post Processing - A Study on the Information Reconciliation Cascade Protocol. Master's Thesis, Faculdade de Engenharia, Universidade do Porto. Jul 2019.

Finally, this repository contains extensive documentation describing the Cascade protocol, our implementation of the Cascade protocol, the findings of reproducing the Cascade analysis results from the academic literature, and lessons learned.

1.2 The broader context.

The code in this GitHub repository is just a small step in the larger project of adding full support for quantum key distribution (QKD) to OpenSSL. This larger project includes other GitHub repositories:

- The openssl-qkd GitHub repository contains a C implementation of a dynamically loaded engine for OpenSSL. This engine replace a classic Diffie-Hellman key exchange with a quantum key distribution (QKD) mechanism. The actual quantum key distribution protocol is not part of this repository. Instead, the engine invokes a stub implementation of application programmer interface (API) defined by European telecommunications standards institute (ETSI).
- The simulaqron-bb84-python GitHub repository contains a Python implementation of the BB84 quantum key distribution (QKD) protocol. It runs on top of the SimulaQron quantum network simulator.

All of these repositories are also just small steps working towards the overall goal adding full support for quantum key distribution to OpenSSL. Much work remains to be done, which is summarized at the end of this chapter.

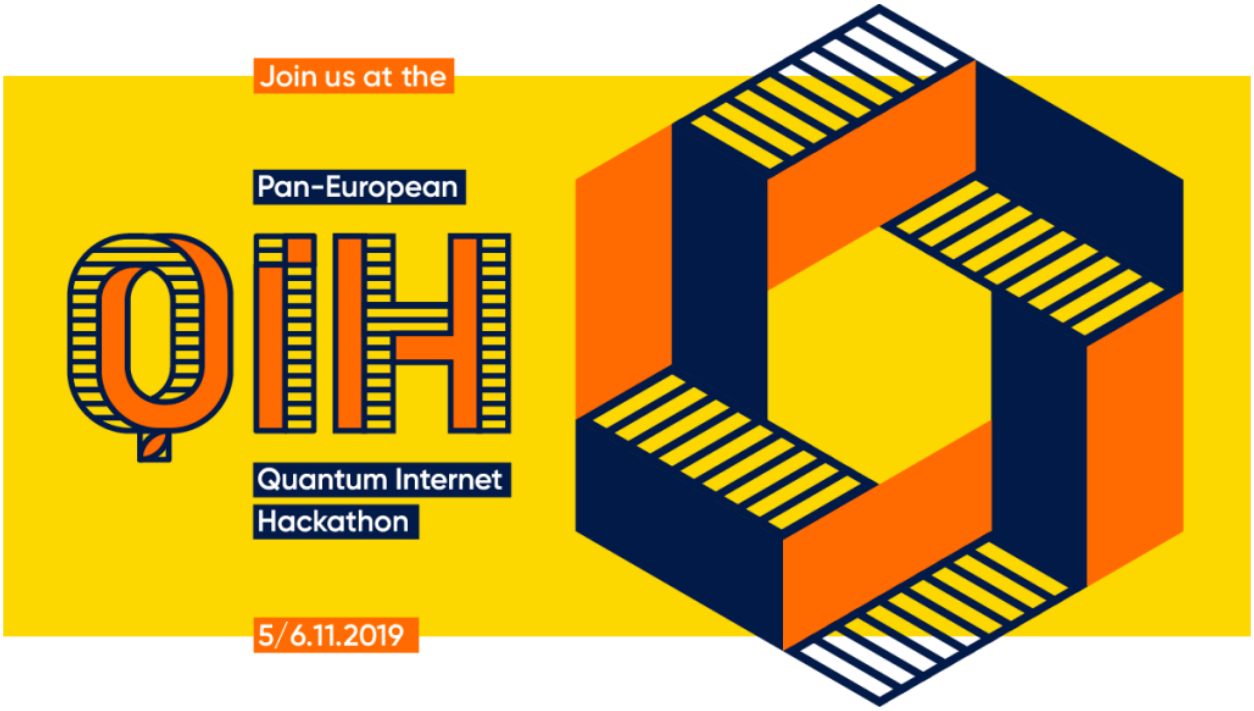
Once the OpenSSL library supports quantum key distribution, many applications that use OpenSSL (such as for example web servers and web clients) will be able to use quantum key distribution with little or no code changes to the application itself.

The initial goal is to support simulated quantum networks using simulators such as SimulaQron or NetSquid, both developed at QuTech. But by building on top of a well-defined application programming interface (namely the ETSI

QKD API) it is conceivable that our code will be able to interoperate with real quantum key distribution devices that are being developed in academia and by commercial vendors.

1.3 The pan-European quantum Internet hackathon.

This project has its roots in the Pan-European Quantum Internet Hackathon which took place on 5 and 6 November 2019 and which was organized by RIPE labs .



Participants from six geographically distributed locations (Delft, Dublin, Geneva, Padua, Paris, and Sarajevo) formed teams that worked on various projects related to the Quantum Internet.

I participated in Delft where the hackathon was hosted by QuTech, a world-leading quantum technology research and development office within the Delft University of Technology.

1.4 The OpenSSL integration challenge.

In Delft, I joined a team working on one of the challenges suggested by the hackathon organizers, namely the OpenSSL integration challenge.

This challenge was developed by Wojciech Kozlowski, a postdoctoral researcher at QuTech and one of the organizers of the Delft hackathon. He is also the main author of the Architectural Principles of the Quantum Internet document that is being developed in the Quantum Internet Research Group (QIRG) in the Internet Research Task Force (IRTF).

OpenSSL
Cryptography and SSL/TLS Toolkit

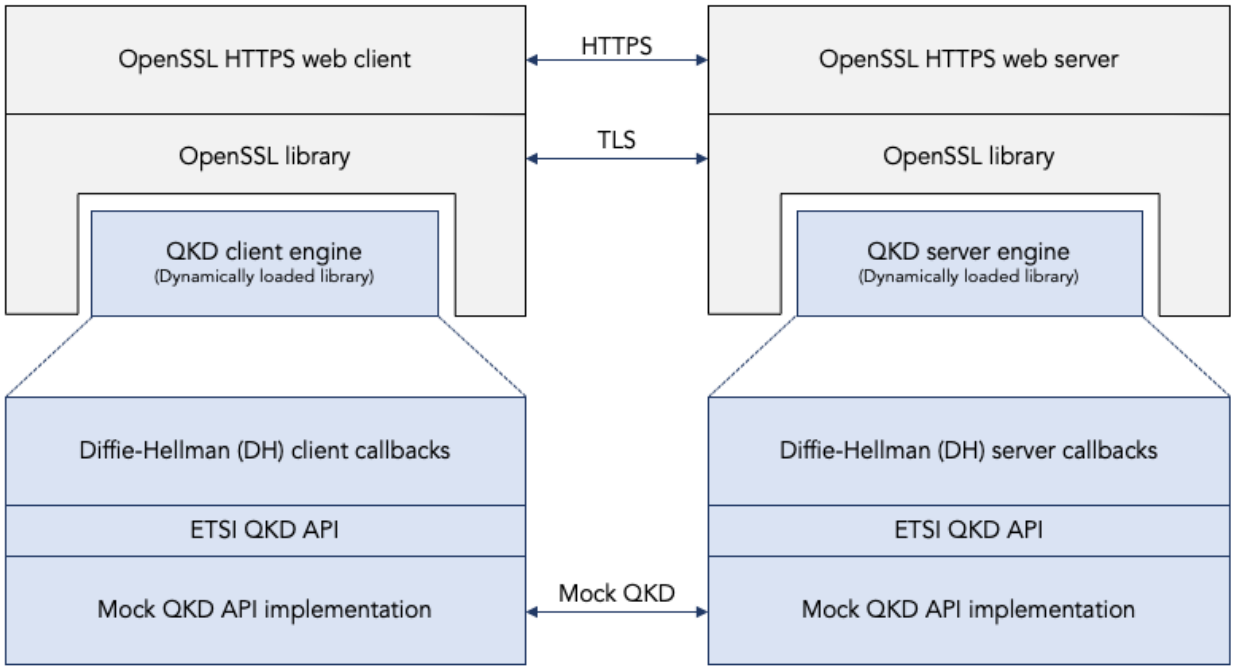
The OpenSSL integration challenge consists of two parts:

1. Enhance OpenSSL to be able to use Quantum Key Distribution (QKD) as a key agreement protocol. OpenSSL is an open source cryptography library that implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. OpenSSL is widely used in Internet applications such as web browsers and web servers.
2. Implement a specific quantum key distribution protocol, namely the Bennett and Brassard 1984 (BB84) protocol, on top of the SimulaQron quantum network simulator.

The end-goal of the challenge is to use an off-the-shelf browser (e.g. Chrome) and connect it to a secure HTTPS website hosted on an off-the-shelf web server (e.g. Apache), while using the BB84 quantum key distribution algorithm as the key agreement protocol (running a SimulaQron simulated quantum network), instead of the classical Diffie-Hellman protocol that is normally used in classical networks.

1.5 Integration of OpenSSL with the stub ETSI QKD API.

The following figure shows what was actually achieved soon after the end of the hackathon.



This is called the "upper half" of the solution for the OpenSSL integration challenge. The source code for this upper half implementation can be found in GitHub repository openssl-qkd and the documentation can be found on this page.

At the hackathon there was another team working on the "lower half" of the OpenSSL challenge. They were working on an implementation of the BB84 protocol running on SimulaQron. This BB84 implementation would provide a north-bound interface in the form of the ETSI QKD API.

The hope was that by the end of the hackathon the "upper half" (the OpenSSL engine that consumes the ETSI QKD API) could be integrated with the "lower half" (the BB84 implementation that provides the ETSI QKD API). We did not quite make that goal during the hackathon itself. We picked up the work where the hackathon left off.

1.6 Python implementation of BB84 on SimulaQron.

The GitHub repository simulaqron-bb84-python contains our Python implementation of BB84 running on SimulaQron. We essentially re-did the work that was done by the other hackathon team.

You can think of it as an exercise to get familiar with BB84 and with the CQC interface provided by SimulaQron. It is a fully functional implementation of BB84 that runs on SimulaQron. However, it is not very suitable as an implementation of the "lower half" that can be integrated with the "upper half" implementation in the openssl-qkd repository. This is because Python code can not easily be integrated with C code into a dynamically loaded library that can be used as an OpenSSL engine. Yes, it is technically possible, but we prefer to rewrite the Cascade code in C (or maybe C++ or Rust); we consider the Python code to be a prototype (we did prototyping in Python because it is much easier to experiment in Python than in C).

1.7 Python implementation of Cascade.

The openssl-qkd repository only contains code for the quantum phase of BB84; it does not contain any classical post-processing code: both the information reconciliation step and the privacy amplification step are missing.

This GitHub repository cascade-python contains a Python implementation of the information reconciliation step. Once again, it is a prototype and needs to be re-implementation in C or C++ or Rust to make it suiteable for integration into an OpenSSL engine in the form of a dynamically loaded library.

1.8 Next steps.

These are the remaining work-items for completing the work of implementing an OpenSSL engine that uses BB84 running on SimulaQron:

1. Implement a Python prototype for privacy amplification.
2. Implement one or more Python prototypes for other information reconciliation protocols, such as Golay codes.
3. Rewrite the Python implementation of BB84 into C or C++ or Rust and add a north-bound ETSI QKD API.
4. Rewrite the Python implementation of Cascade and the other information reconciliation protocols into C or C++ or Rust and integrate with the BB84 code.
5. Rewrite the Python implementation of privacy amplification into C or C++ or Rust and integrate with the BB84 code.
6. Demonstrate running Chrome and Apache on top of the QKD OpenSSL engine.

CHAPTER

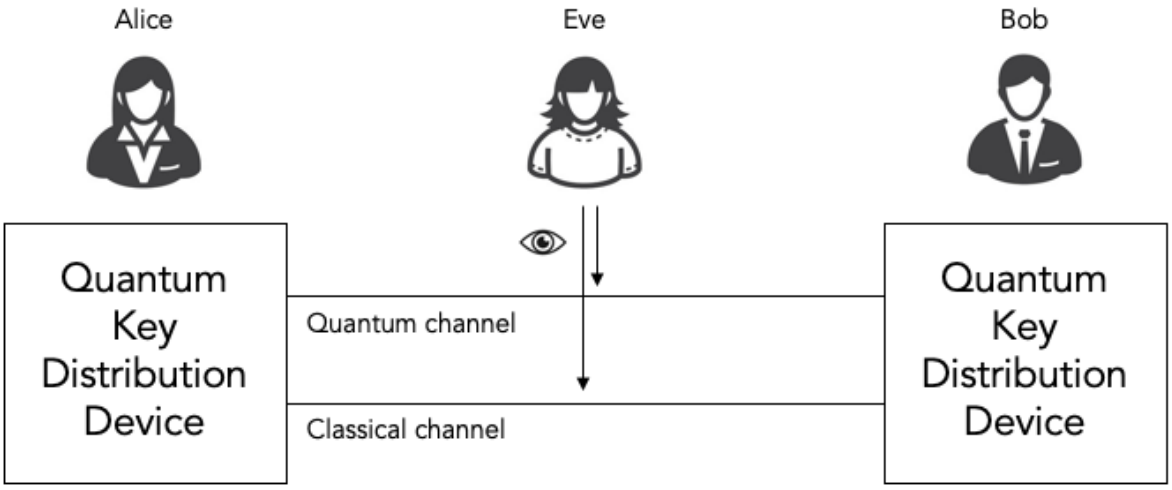
TWO

THE CASCADE INFORMATION RECONCILIATION PROTOCOL.

Tutorial by Bruno Rijsman

2.1 Quantum key distribution (QKD) protocols.

All quantum key distribution (QKD) protocols involve using a combination of quantum communications (qubits) and classical communications (classical bits) to allow two parties Alice and Bob to agree on a secret key in such a way that our nefarious eavesdropper Eve cannot observe what the secret key is without being detected by Alice and Bob.



There are multiple quantum key distribution protocols, including for example BB84 and B92. All of these protocols consist of both a quantum phase and a classical post-processing phase.

The quantum phase uses both the quantum channel and the classical channel to actually exchange the key.

The classical post-processing phase only uses the classical channel. The classical post-processing phase is further sub-divided into two parts:

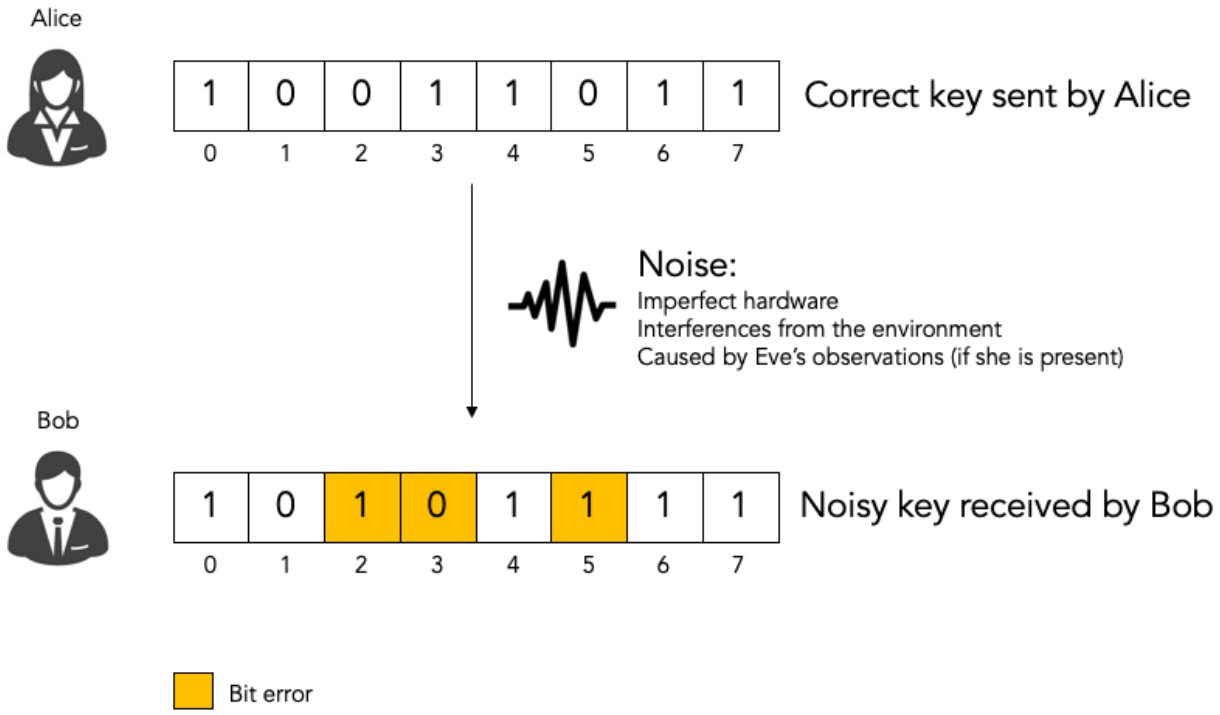
- Information reconciliation, which is responsible for detecting and correcting inevitable bit errors (noise) in the key that was exchanged during the quantum phase.
- Privacy enhancement, which is responsible for mitigating the information leakage during the information reconciliation step.

In this document we only discuss one specific information reconciliation protocol, namely the Cascade protocol.

We won't discuss privacy enhancement nor the quantum phase. Those interested in more details on the quantum phase can have a look at our simulaqron-bb84-python GitHub repository that contains our Python implementation of the quantum phase in the BB84 quantum key distribution protocol.

2.2 Key bit errors (noise).

Key distribution protocols always introduce some noise in the key. The key that Bob receives contains some noise (i.e. bit errors) as compared to the key that Alice sent. For that reason we refer to the key that Alice sent as the correct key and to the key that Bob received as the noisy key.



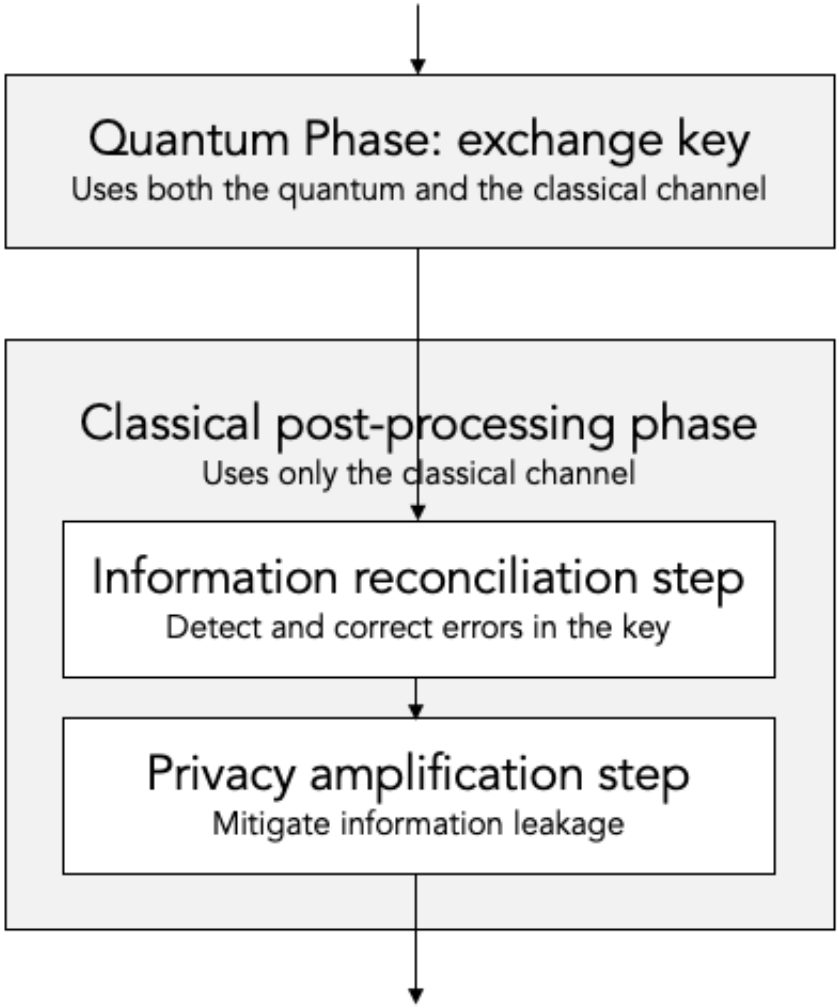
The noise can be introduced by imperfections in the hardware and by random fluctuations in the environment. Or the noise can be introduced by eavesdropper Eve observing traffic. Remember: in quantum mechanics observing a photon causes the photon to change and hence introduces detectable noise.

All quantum key distribution protocols provide an estimate of the noise level in the form of an estimated bit error rate. Bit error rate 0.0 means that no key bits have been flipped and bit error rate 1.0 means that all key bits have been flipped.

2.3 Classical post-processing.

If the estimated bit error rate is above some threshold we conclude that Eve is observing the traffic trying to determine the secret key. In that case, we abandon the key distribution attempt.

If the estimated bit error rate is below the threshold we perform classical post-processing, which consist of the two steps that we mentioned earlier. Both of these steps are classical protocols in the sense that they only involve classical communications and not any quantum communications.



2.3.1 Information reconciliation.

The first classical post-processing step is information reconciliation. Even if the bit error rate is below the threshold, it is not zero. There is still some noise: there are still bit errors in the noisy key that Bob received as compared to the correct key that Alice sent. The purpose of the information reconciliation step is to detect and correct these remaining bit errors.

There are multiple information reconciliation protocols. In this document we discuss only one specific protocol, namely the Cascade protocol.

The tricky part to information reconciliation is to avoid leaking (i.e. exposing) too much information about the key. Eve, the eavesdropper, can learn any information that we leak during the information reconciliation step. Even if she does not learn the entire key, learning any leaked partial information about the key simplifies her task of decrypting the encrypted traffic. Every bit of leaked key information halves the number of keys that Eve has to try during a brute force attack.

That said, it is unavoidable that the information reconciliation protocol leaks some limited amount of information. This is okay as long as the amount of leaked information is bounded and known, so that we can compensate for it.

2.3.2 Privacy amplification.

The second classical post-processing step is privacy amplification. The purpose of privacy amplification is to compensate for the information leakage in the information reconciliation step. Privacy amplification introduces extra randomness at the cost of reducing the effective key size.

In this document we do not discuss privacy amplification any further.

2.4 The Cascade protocol.

The Cascade protocol is one example of an information reconciliation protocol. The purpose of the Cascade protocol is to detect and to correct any remaining bit errors in the noisy key that Bob received relative to the correct key that Alice sent.

Let's say that Alice and Bob are the two parties that have just run the quantum phase of a quantum key distribution such as BB84. Alice has the correct key and Bob has a noisy key, which is similar to Alice's correct key but which has some limited number of bit errors (i.e. noise). Alice and Bob will now run the Cascade protocol to detect and correct any remaining bit errors in the Bob's noisy key.

2.5 Cascade as a client-server protocol.

From a protocol point of view, it makes sense to describe the Cascade protocol in terms of a client-server protocol.

Bob takes the role of the client. As far as Cascade is concerned, he is the active party. He decides what needs to happen when. He does most of the computation. And he sends messages to Alice when he needs her to do something.

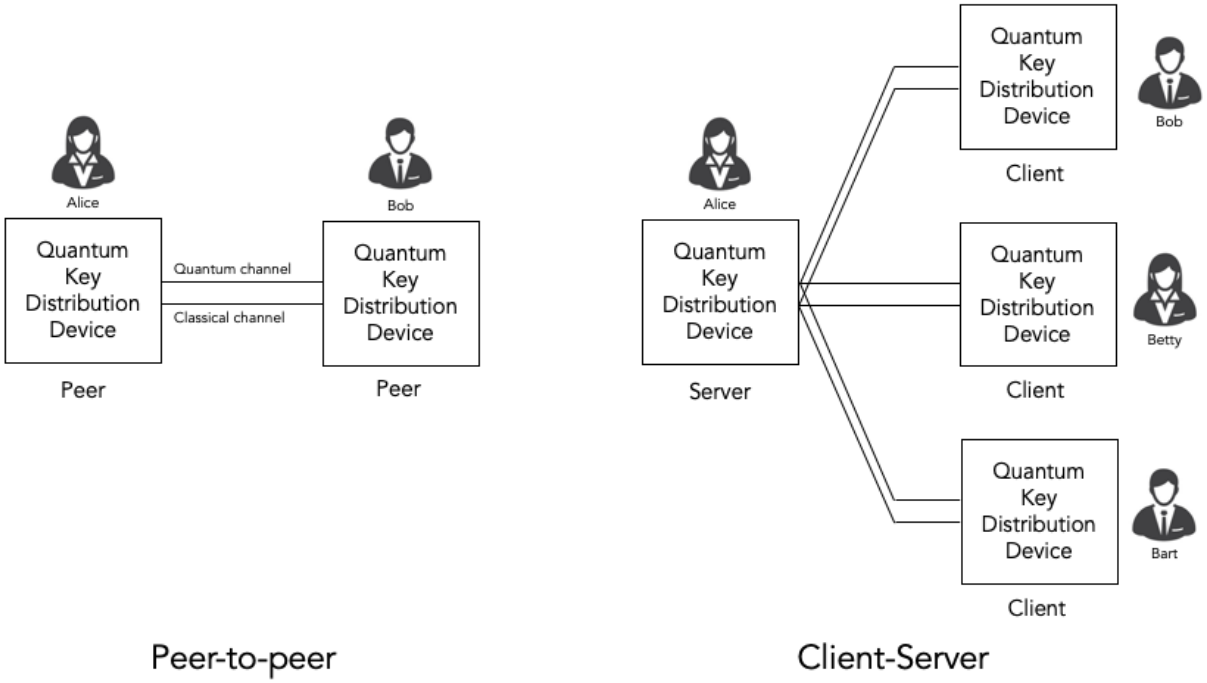
Alice takes the role of the server. As far as Cascade is concerned, her role is mostly passive. She waits for Bob to ask her simple questions and she provides the answers.

We will describe the Cascade protocol from the perspective of the client, i.e. from the perspective of Bob.

An interesting observation is that the Cascade protocol puts most of the complexity and most of the computational burden on the client. The server doesn't do much except compute simple parities when asked to do so by the client. This is a very nice property for a client-server protocol. The server could have many (thousands) of sessions to clients, so it is very desirable that each session is simple and light-weight. The client will typically have only a few sessions to a few servers, so it is okay if the sessions are more complex and heavy-weight.

It is fair to say that quantum key distribution is currently often (almost always, perhaps) used to secure point-to-point links with a quantum key distribution device on either end of the link. From that perspective it is natural to think of Cascade as a peer-to-peer protocol.

That said, quantum key distribution in general and Cascade in particular could very well be deployed in true client-server scenarios. One example scenario is secure web traffic where a web server (e.g. Apache) has many session to many different web clients (e.g. Chrome web browsers) using the HTTPS protocol.



2.6 The classical channel.

Cascade is a fully classical protocol. It only involves the exchange of classical messages. It does not involve any quantum communications.

We assume that there is a classical channel between Alice and Bob that allows Alice and Bob to exchange classical messages as part of the Cascade protocol. We rely on classical techniques to provide reliability, flow-control, etc. (for example, we could use TCP/IP).

We do not require that the classical channel is encrypted: we assume that eavesdropper Eve can observe all classical messages in the clear.

Any requirement that the classical channel be encrypted would introduce a chicken-and-egg problem: we would need a quantum key distribution protocol to encrypt the classical channel, but the quantum key distribution protocol would need an encrypted classical channel.

We do, however, require that the classical channel provides authentication and integrity. We assume that there is a mechanism that allows Alice and Bob to verify that all classical messages were actually sent by Bob and Alice and have not been forged or tampered with by Eve.

This is needed to avoid woman-in-the-middle attacks by Eve, where Eve intercepts all classical traffic and pretends to be Bob to Alice and pretends to Alice to Bob.

We do not discuss how the authentication and integrity are implemented nor does the code in this repository contain any authentication or integrity mechanisms.

This is consistent with most of the literature on quantum key distribution. Most literature barely mentions the need for an authentication and integrity on the classical channel. Details on how to do it are even less forthcoming. This might give you the impression that it is a trivial matter not worth discussing. Nothing could be further from the truth!

Yes, it is true that authentication and integrity are considered to be well-solved problems for classical protocols. For authentication, classical protocols typically use either public key infrastructure (PKI) or pre-shared keys. For

integrity, classical protocols typically use hash-based message authentication codes (HMAC) in combination with Diffie-Hellman or pre-shared keys to agree on the message authentication key.

But none of those options (pre-shared keys, public key infrastructure, Diffie-Hellman) are attractive options for quantum key distribution.

Public-key infrastructure and Diffie-Hellman are problematic because they are not quantum-safe: they rely on the assumption that factorization or modular logarithms are computationally difficult.

Pre-shared keys are somewhat acceptable for point-to-point connections, but they are really problematic in client-server scenarios where the server does not know a-priori which clients will connect to it. But more importantly, using pre-shared keys defeats the whole purpose of running a quantum key distribution protocol.

In summary: while the topic of authenticating the classical channel is usually glossed over, it is not at all obvious how to achieve it in the context of quantum key distribution.

2.7 Input and output of the Cascade protocol.

Let's start by looking at the Cascade protocol as a black box algorithm, and let's consider what the input and the output of the Cascade protocol are.

2.7.1 Input: noisy key and estimated quantum bit error rate (QBER).

Bob initiates the Cascade protocol after the quantum phase of the quantum key distribution has been completed.

At this point, Bob has the following information available to him, which is the input to the Cascade protocol.

Bob has the noisy key that he has received from Alice. Although a quantum key distribution protocol was used to agree on this key, there is nothing quantum about the key at this point. It is just a string of classical bits of a certain length (the key size).

As we described earlier, the quantum key distribution protocol introduces some noise when it delivers this key to Bob. Thus, Bob has a noisy key which has some bit errors compared to Alice's correct key.

Bob does not know exactly how many bit errors there are or which bits are in error, but the quantum key distribution protocol does provide an estimate of the bit error rate, which also known as the quantum bit error rate (QBER).

Thus, we have two inputs to the Cascade protocol: the noisy key and the estimated quantum bit error rate (QBER).

2.7.2 Output: reconciliated key and amount of leaked information.

It is the job of the Cascade protocol to determine which bits exactly are in error and to fix them.

It is important to understand that Cascade does not guarantee that all bit errors are corrected. In other words, Bob's reconciliated key is still not guaranteed to be the same as Alice's correct key. Even after the reconciliation is complete, there is still a remaining bit error rate. The remaining bit error rate is orders of magnitude smaller than the original bit error rate before Cascade was run. But it is not zero. That is why we prefer to use the term reconciliated key and not corrected key, although the latter is also often used.

Cascade per-se does not contain any mechanism to detect and report whether the reconciliation was successful. It will neither detect nor report that there are any remaining bit errors after reconciliation. Some mechanism outside of Cascade is needed to validate whether the reconciliated key is correct or not.

The Cascade protocol can also keep track of exactly how much information was leaked. Specifically, Cascade running at Bob can keep track of which parities he asked Alice to compute. We must assume that Eve will also know about those parities. We can express the amount of leaked information in terms of leaked key bits (this is a logical abstraction - it does not indicate which specific key bits were leaked, it only provides a measure of how much information was leaked).

The amount of leaked information may be used by the privacy amplification phase that runs after the information reconciliation phase to determine how much amplification is needed.

Thus, the output of Cascade are the reconciliated key and the amount of leaked information.

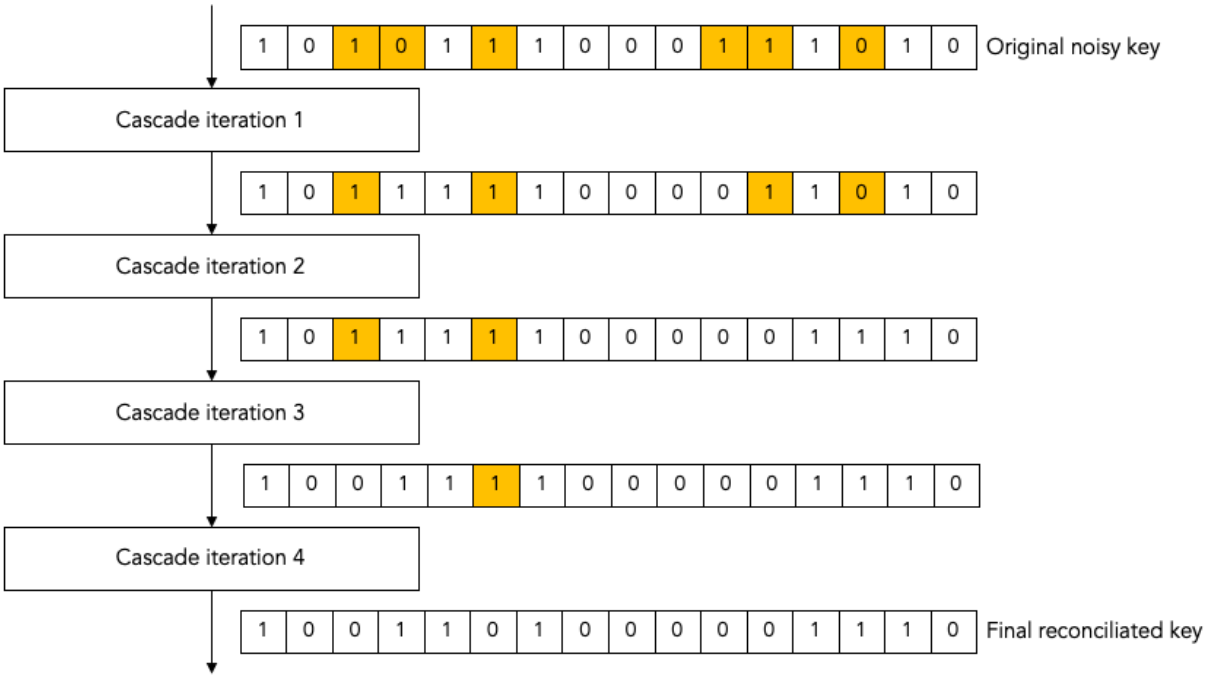
2.8 Cascade Iterations.

Now we are ready to start describing the guts of the Cascade protocol, i.e. to describe in detail how it actually works.

Let's define a single run of the Cascade protocol as Alice and Bob reconciliating (i.e. attempting to correct) a single key.

A single Cascade run consists of multiple iterations (these are also known as passes). Different variations of the Cascade protocol use different numbers of iterations. But we start by describing the original version of the Cascade protocol which uses four iterations.

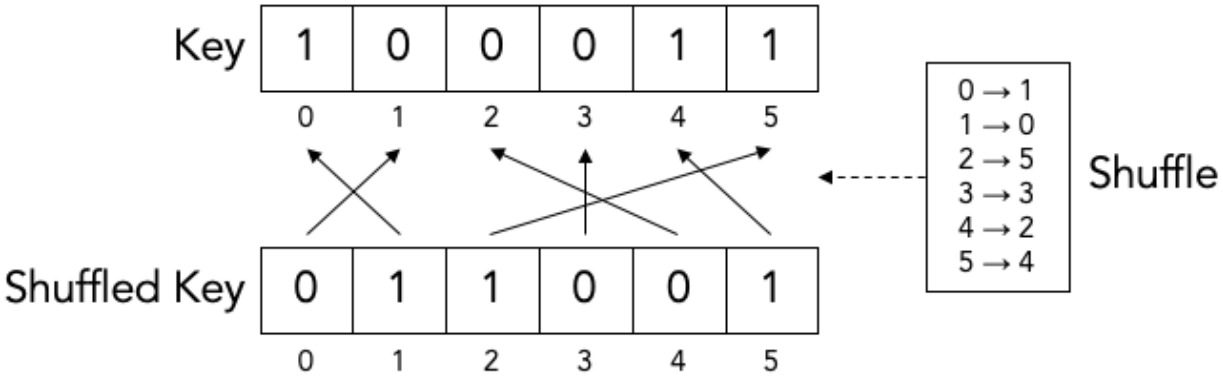
Each Cascade iteration corrects some of the bit errors in the key. It is very probable (but not entirely certain) that all bit errors will have been corrected by the end of the last iteration.



Note: for the sake of clarity, all of our diagrams show very small keys. In the above diagram, for example, we use 16-bit keys. In later diagrams we will use even smaller keys to make them fit in the diagram. In real life the keys can be much much larger: tens of thousands or even hundreds of thousands of bits.

2.9 Key shuffling.

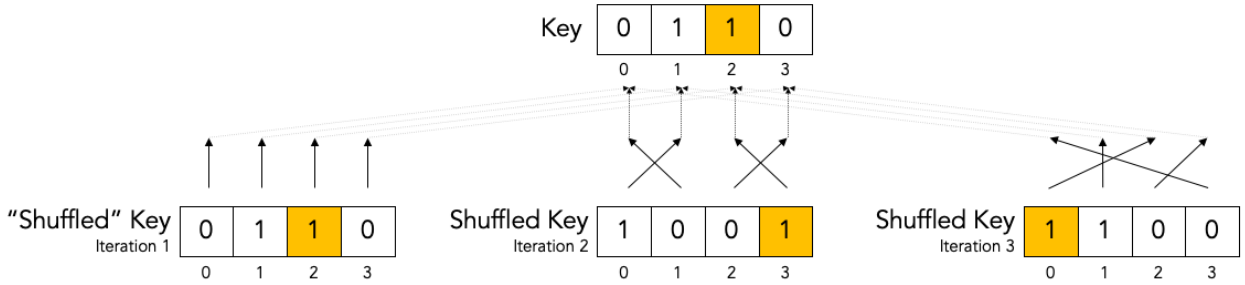
At the beginning of each iteration, except the first one, Bob randomly shuffles the bits in the noisy key. Shuffling means randomly reordering the bits in the key.



Later we will find out what the purpose of shuffling the key is. For now, we just point out that the shuffling is not intended to obfuscate the key for Eve. It is perfectly okay if the shuffling is only pseudo-random or even deterministic.

It is even okay if Eve knows what the shuffling permutation is (shown as the "Shuffle" in the above diagram) as long as the actual key values before ("Key") or after the shuffling ("Shuffled key") are not divulged. In fact, Bob needs to inform Alice what the shuffle permutation for each Cascade iteration is. It is no problem if the information about the shuffle permutation is sent in the clear and Eve can observe it.

As we mentioned, Bob re-shuffles his noisy key at the beginning of each iteration except the first one:



We put the word "shuffled" in quotes for the first iteration because they key is not really shuffled for the first iteration.

The important thing to observe is that any given bit in the original unshuffled key (for example bit number 2 which is marked in yellow) ends up in a different position in the shuffled key during each iteration.

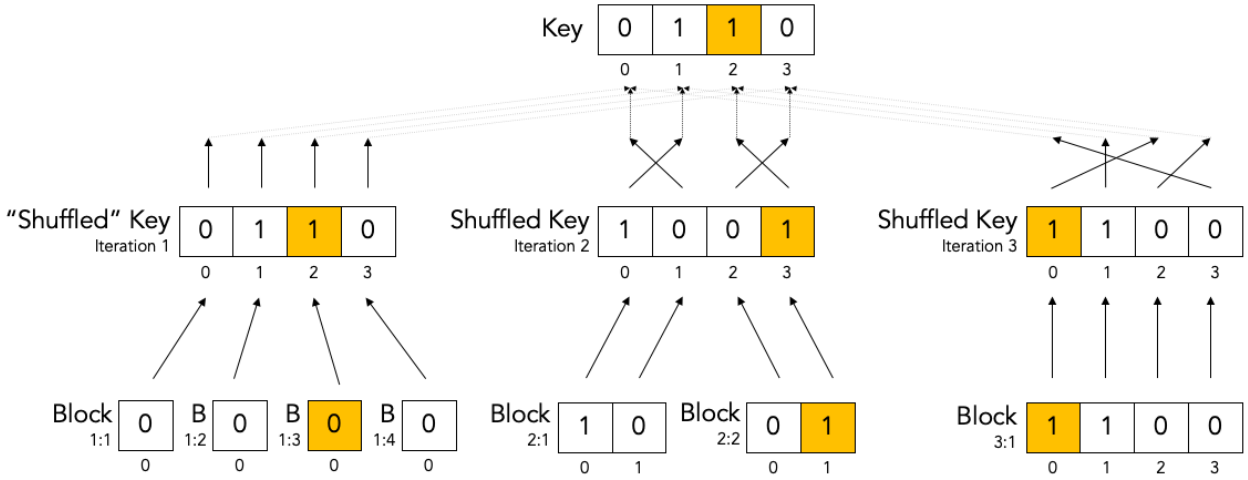
2.10 Creation of the top-level blocks.

During each iteration, right after shuffling the key, Bob divides the shuffled key into equally sized blocks (the last block may be a smaller size if the key is not an exact multiple of the block size).

We will call these blocks top-level blocks to distinguish them from other types of blocks (the so-called sub-blocks) that will appear later in the protocol as a result of block splitting.

The size of the top-level blocks depends on two things:

- The iteration number i. Early iterations have smaller block sizes (and hence more blocks) than later iterations.
- The estimated quantum bit error rate Q. The higher the quantum bitter error rate, the smaller the block size.



Note: to make things fit on a page the block sizes are extremely small in this diagram. In real life, top-level blocks are much larger. Specifically, we would never see a single-bit top-level block.

There are many variations of the Cascade protocol, and one of the main differences between these variations is the exact formula for the block size k_i as a function of the iteration number i and the quantum bit error rate Q.

For the original version of the Cascade protocol the formula is as follows:

$k_1 = 0.73 / Q$

$k_2 = 2 * k_1$

$k_3 = 2 * k_2$

$k_4 = 2 * k_3$

Without getting into the mathematical details behind this formula, we can build up some intuition about the reasons behind it.

Later on, we will see that Cascade is able to correct a single bit error in a block but is not able to correct a double bit error in a block.

If we pick a block size $1/Q$ for the first iteration, then each block will is expected to contain a single bit error on average. That is just the definition of bit error rate. If the bit error rate is 1 error per 100 bits, then a block of 100 bits will contain on average one error.

Now, if we use $0.73/Q$ instead of $1/Q$ then we will have slightly smaller blocks than that. As a result we will have more blocks with zero errors (which are harmless) and fewer blocks with two errors (which are bad because they cannot be corrected).

On the other hand, we don't want to make the blocks too small, because the smaller we make the blocks, the more information is leaked to Eve. Knowing the parity over more smaller blocks allows Eve know more about the key.

So, that explains the formula $0.73/Q$ for the first iteration. What about the doubling of the block size in each iteration?

Well, during each iteration Cascade corrects some number of errors. Thus the remaining quantum bit error rate for the next iteration is lower (i.e. fewer error bits). This allows us to use a bigger block size for the next iteration, and still have a low probability of two (uncorrectable) errors in a single block.

2.11 Detecting and correcting bit errors in each block.

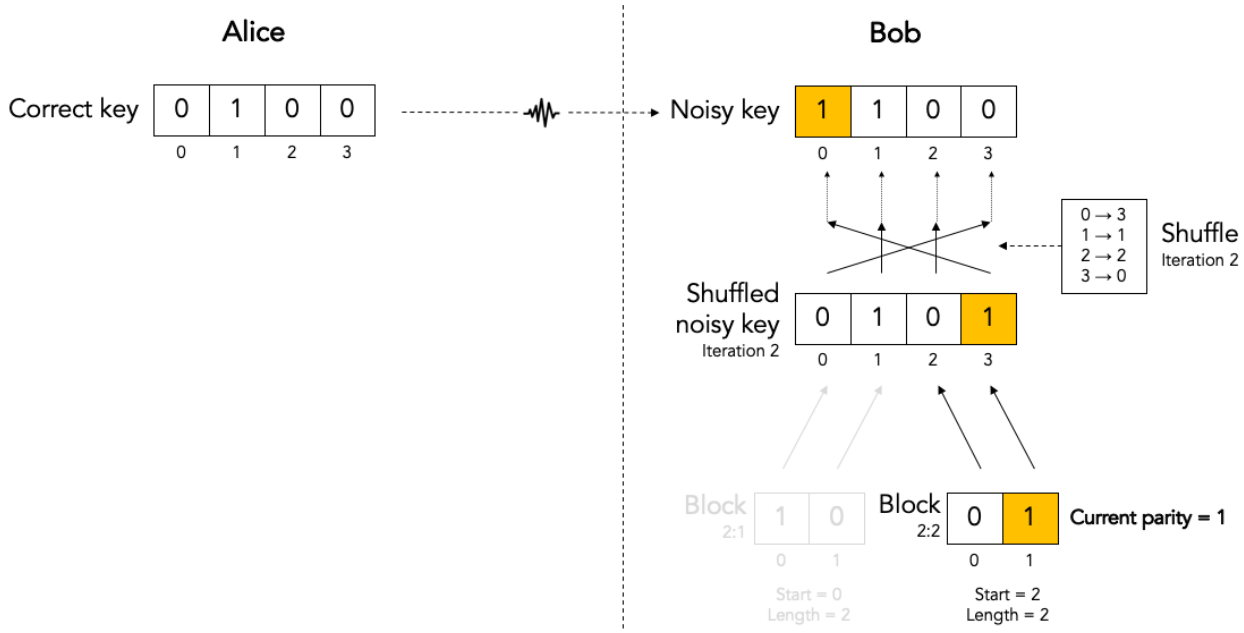
After having shuffled the key and after having split the key into blocks for a given iteration, Bob sets out on the task of determining, for each block, whether or not there are any bit errors in that block and, if so, to correct those bit errors.

The process of doing so is a bit complex because Bob needs to do it in such a way that he leaks a minimum of information to eavesdropper Eve who is watching his every move.

2.12 Computing the error parity for each top-level block: even or odd.

2.12.1 Computing the current parity.

Bob locally computes the current parity of each top-level block. This is a parity over some subset of bits in the shuffled noisy key that Bob has received from Alice. In the following example, Bob computes the parity of the 2nd top-level block in the 2nd iteration. That block has value 01 so its current parity is 1.



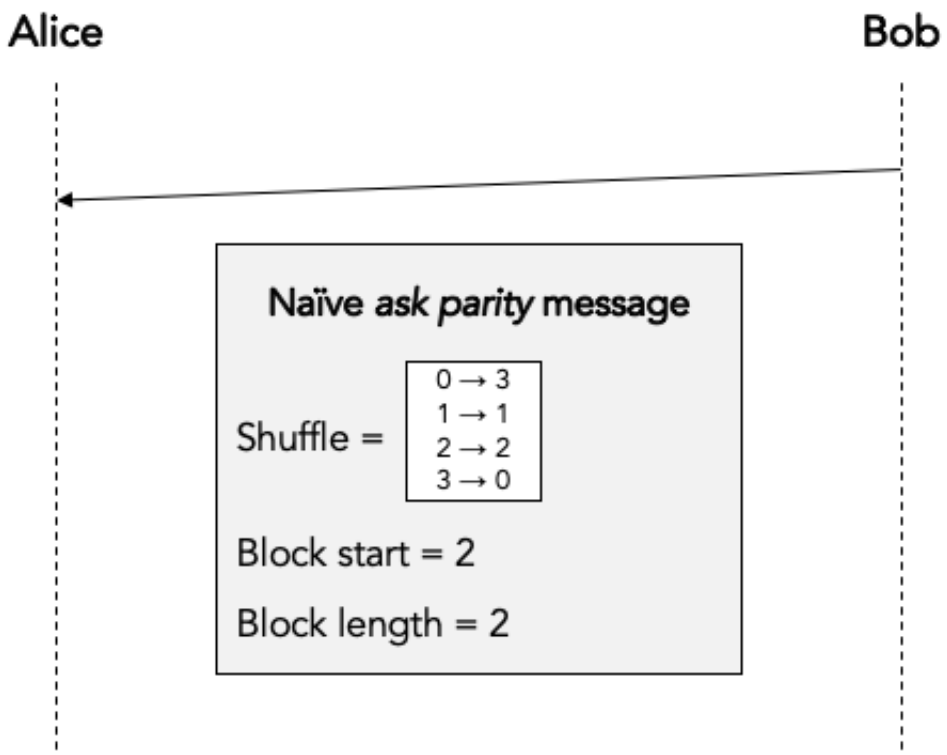
2.12.2 Computing the correct parity.

Next, Bob wants to know Alice's perspective on the block parity. He already knows the "current parity" of the block in his own noisy key, but now he wants to know the "correct parity" of the same block in the Alice's correct key.

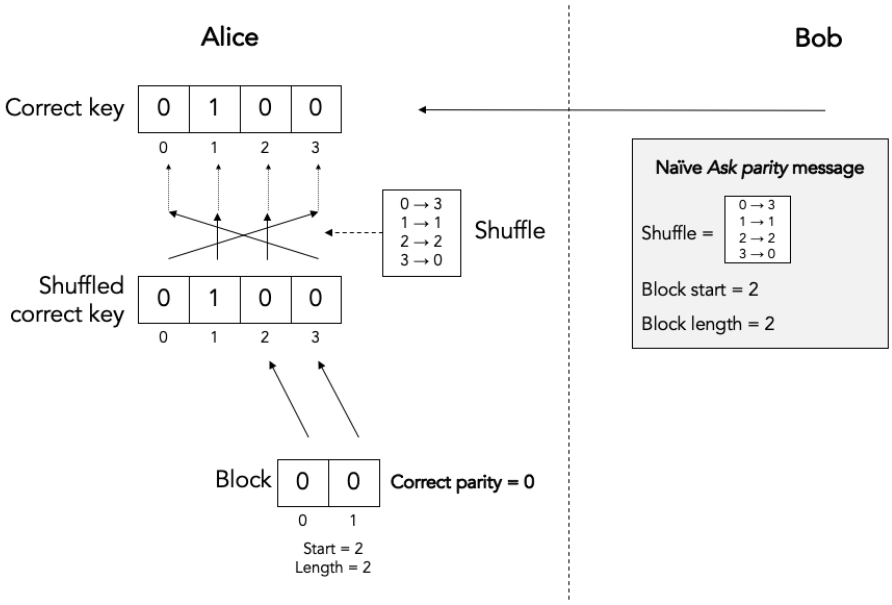
There is no way for Bob to compute the correct parity himself. Bob does not have access to the correct key, only Alice does. Actually, that statement is a little bit too strong. It turns out that there is an exception to this statement. Hold on until we discuss block parity inference (BPI) near the end of this tutorial.

The solution is simple: Bob simply sends an *ask parity* message to Alice. The purpose of this message is to ask Alice to compute the correct parity.

Let's first look at a very naive way of implementing the *ask parity* message, which is very inefficient but which makes the concept very clear:

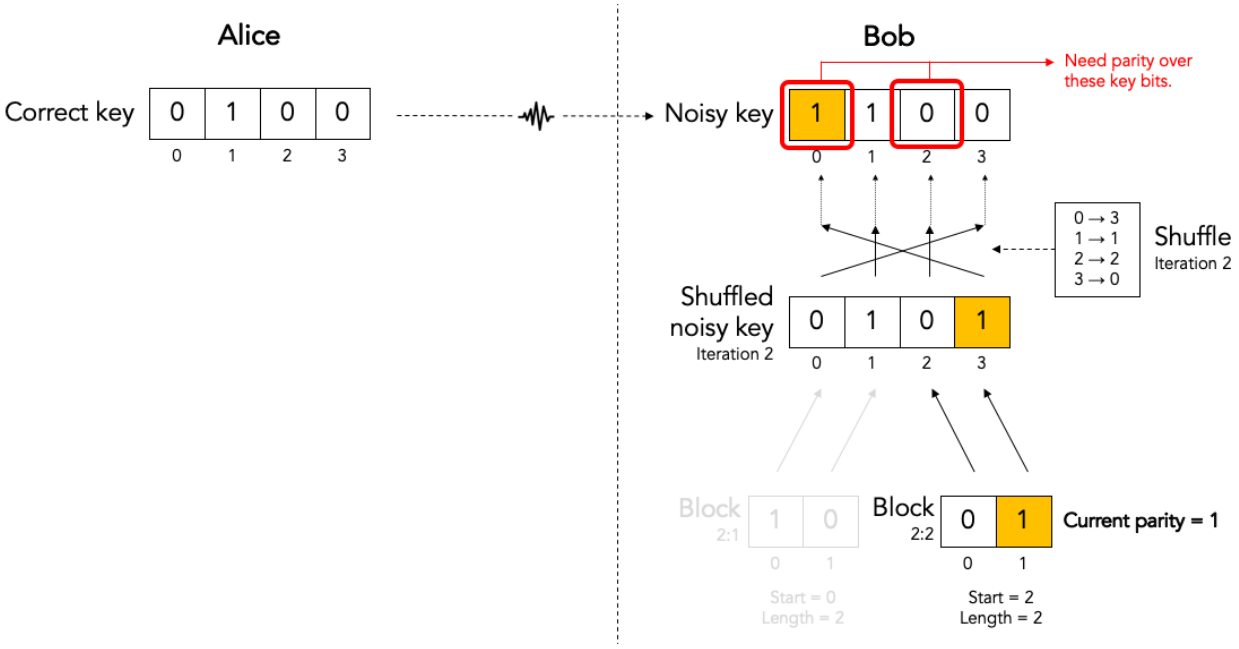


In this implementation Bob literally provides all the information that Alice needs to reconstruct the block and compute the correct parity:

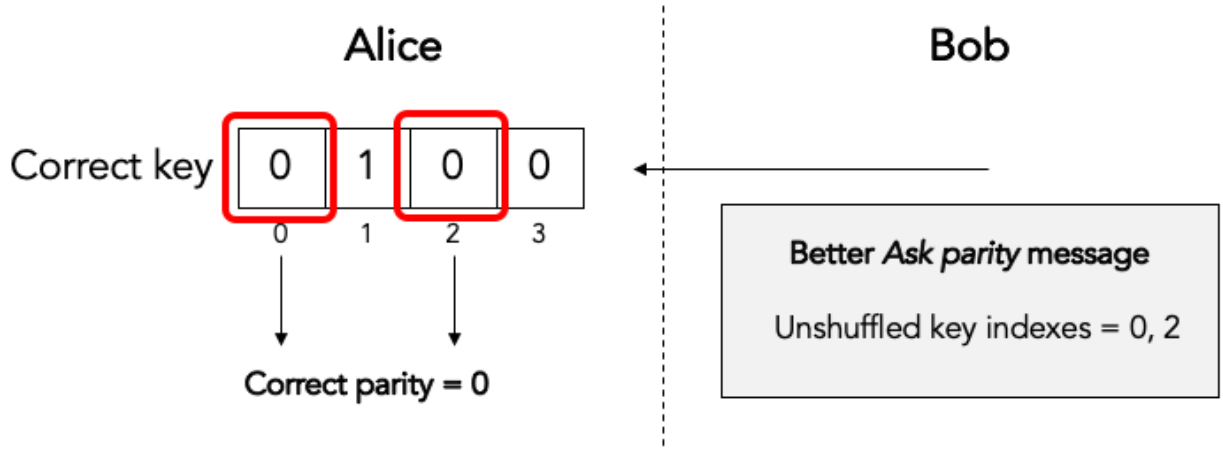


This is an inefficient way of computing the correct the parity. For one, the *ask parity* message can get very large because the shuffle permutation can get very large: here it is N numbers, where N is the key size (but it is easy to see that we could reduce N to the block size). Secondly, it requires Alice to spend processing time on reconstructing the shuffled key and the block.

An obvious optimization is for Bob to just cut to the chase and list the actual unshuffled key indexes over which Alice must compute the parity:



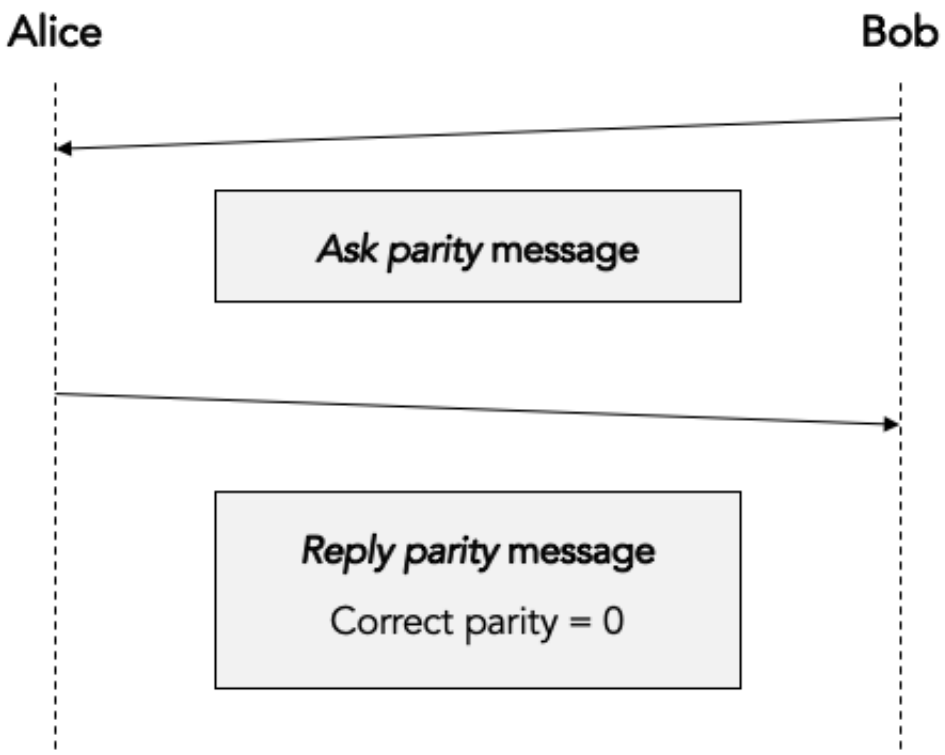
This allows Alice to just compute the correct parity without wasting CPU cycles on reconstructing the shuffled key and block:



In both cases the *ask parity* message does not leak any information about the key (yet): it does not contain the value of any key bit or any other information about the key bits themselves.

It turns out that there are even more efficient ways of implementing the *ask parity* message. These rely on the fact that the key is only shuffled once per iteration and we ask for block parities many times per iteration. These optimizations are described in the implementation guide.

The only thing left to do is for Alice to send the correct parity back to Bob in a *reply parity* message:



In any real implementation there would be additional fields in the *reply message* to associate the reply parity message with the corresponding ask parity message, but we gloss over those details here.

Although neither Alice nor Bob ever divulge any actual key bits, the divulgence of the correct parity in the *reply parity* message does leak a little bit of information to Eve. This is easy to understand if we look at the number of values Eve has to try out in a brute force attack. If Eve knows nothing about N bits, she has to try out 2N values in a brute force attack. But if she knows the parity of those N bits, she only has to try out 2N-1 values.

2.12.3 Inference the error parity from current parity and the correct parity.

At this point Bob knows both the correct parity and the current parity of the block.

Can Bob determine which bits in the block are in error? Well, no, he cannot. Can Bob at least determine whether there are any errors in the block or not? Well, no, he cannot determine even that.

What can Bob determine then? Well, Bob can determine whether there are an even or an odd number of errors in the block (the so-called error parity), by using the following table:

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

If the error parity is odd, then Bob knows that there is at least bit one error in the block. He doesn't know exactly how many bit errors there are: it could be 1 or 3 or 5 or 7 etc. And he certainly doesn't which which key bits are in error.

If the error parity is even, then Bob knows even less. Remember that zero is an even number. So, there could be no (zero) errors, or there could be some (2, 4, 6, etc.) errors.

2.13 Correcting a single bit error in top-level blocks with an odd number of bits.

When Bob finds a block with an even number of errors, Bob does nothing with that block (for now).

But when Bob finds a block with an odd number of errors, Bob knows that there is at least one remaining bit error in the block. Bob doesn't know whether there is 1 or 3 or 5 etc. bit errors, but he does now there is at least one bit error and that the number is odd. For such a block, Bob executes the Binary algorithm. We will describe the Binary algorithm in the next section. For now, suffice it to say that the Binary algorithm finds and corrects exactly one bit error in the block.

Let's summarize what we have done so far.

In each iteration (except the first) Bob first shuffles the noisy key. Then he takes the shuffled key and breaks it up into blocks. Then he visits every block and determines the error parity for that block. If the error parity is even, he does nothing. If the error parity is odd, then he runs the Binary algorithm to correct exactly one bit errors.

So, at the end of the iteration, Bob ends up with a list of blocks that all have an even error parity.

Some blocks already had an even error parity at the beginning of the iteration and Bob did not touch them.

Some blocks had an odd error parity at the beginning of the iteration and Bob ran the Binary algorithm to correct exactly one bit error. If you start with a block with an odd number of bit errors, and you correct exactly one bit error, then you end up with a block with an even number of bit errors.

Does this mean that we have removed all errors during this iteration? No it does, not. We only know that each block now contains an even number of errors. It could be zero errors. But it could also be 2, 4, 6, etc. errors.

During this iterations there is nothing more Bob can do to find or correct those remaining errors. But that doesn't mean those remaining error won't get corrected. Later we will see how a combination of reshuffling in later iterations and the so-called cascading effect will (with high probability) find and correct those remaining errors.

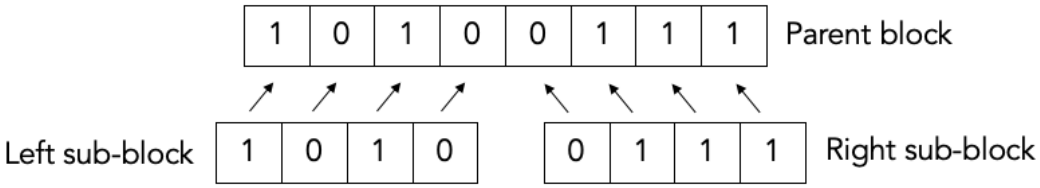
2.14 The Binary algorithm.

The Binary algorithm takes as input a block that has an odd number of errors. It finds and corrects exactly one bit error.

Bob is only allowed to run the Binary algorithm on blocks that have an odd number of errors. Bob is not allowed to run the Binary algorithm on a block that has an even number of errors (it is a useful exercise to figure out why not).

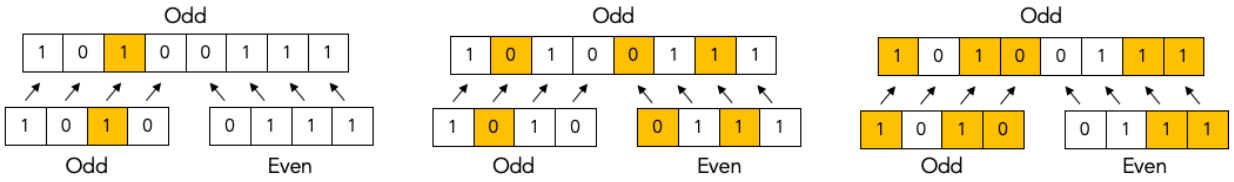
2.14.1 Split block.

The first thing Binary does is to split the block into two sub-blocks of equal size. We call these sub-blocks the left sub-block and the right sub-block. And we call the block that was split the parent block. If the parent block has an odd size, then the left sub-block is one bit bigger than the right sub-block.

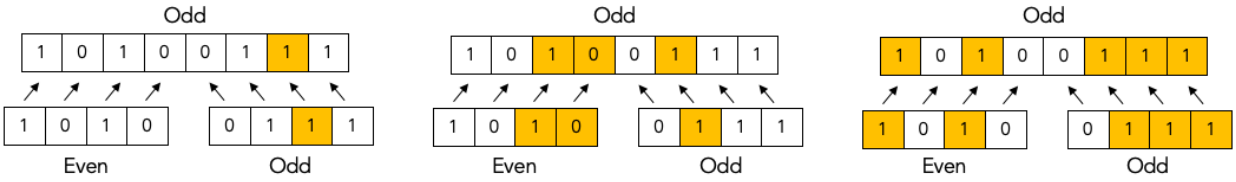


Given the fact that we know for certain that the parent block has an odd number of errors, there are only two possibilities for the sub-blocks.

Either the left sub-block has an odd number of errors and the right sub-block has an even number of errors, as in the following examples:



Or the left sub-block has an even number of errors and the right sub-block has an odd number of errors, as in the following examples:



It is simply not possible that both sub-blocks have an even number of errors and it is also not possible that both sub-blocks have an odd number of errors.

2.14.2 Determine error parity of sub-blocks.

Bob doesn't know which it is: Bob doesn't know whether the left sub-block or the right sub-block has an odd number of errors. All Bob knows at this point is that the parent block has an odd number of errors.

In order to find out, Bon sends an *ask parity* message to Alice to ask the correct parity for the left sub-block (only the left sub-block, not the right sub-block). When Alice responds with the correct parity for the left sub-block, Bob can compute the error parity (odd or even) for the left sub-block: he just needs to combine the locally computed current parity with the correct parity provided by Alice.

If the error parity of the left sub-block turns out to be odd, then Bob immediately knows that the error parity of the right sub-block must be even.

On the other hand, if the error parity of the left sub-block turns out to be even, then Bob immediately knows that the error parity of the right sub-block must be odd.

Either way, Bob knows the error parity of both the left sub-block and the right sub-block. Bob only asked Alice to give the correct parity for the left sub-block. Bob never asked Alice to provide the correct parity for the right sub-block.

Bob can infer the correct parity and hence error parity for the right sub-block (and so can Eve, by the way).

By the way, this inference trick only works if Bob knows for a fact that the error parity of the parent block is odd. That is why Bob is not allowed to run the Binary protocol on a block with even error parity.

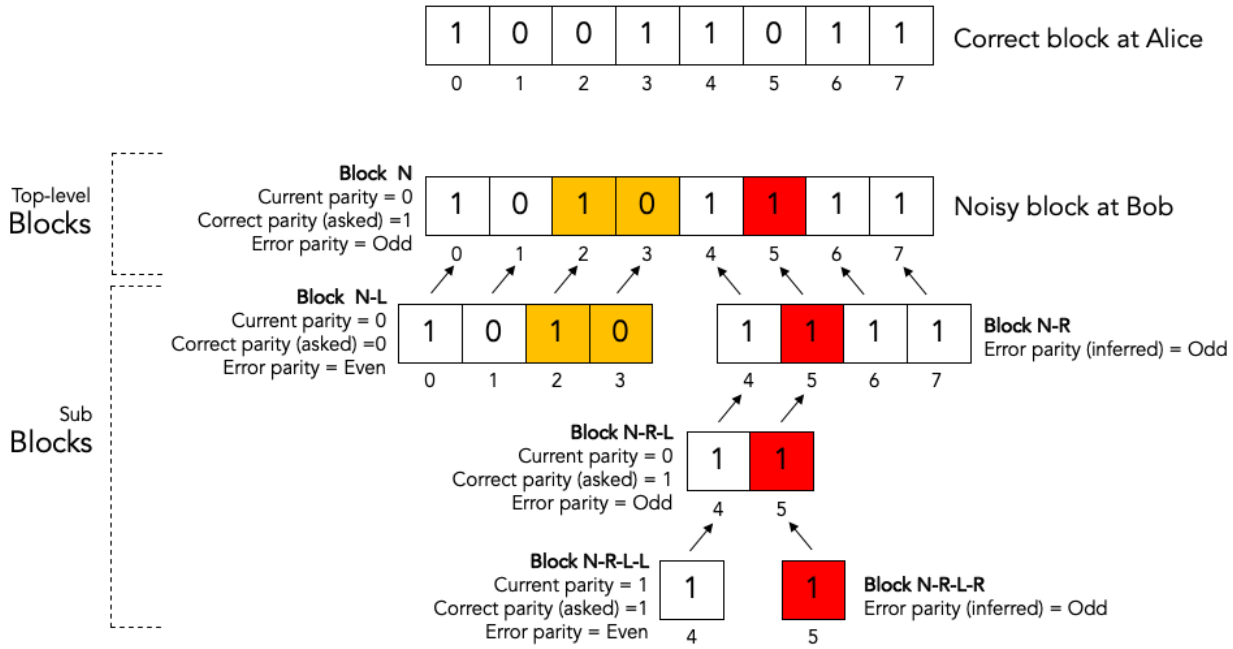
2.14.3 Recursion.

Once Bob has determined whether the left sub-block or the right sub-block contains an odd number of errors, Bob can recursively apply the Binary algorithm to that sub-block.

The Binary algorithm will keep recursing into smaller and smaller sub-blocks, until it finally reaches a sub-block that has a size of only a single bit.

If we have a sub-block whose size is one single bit and we also know that that same sub-block has an odd number of errors, then we can conclude that the single bit must be in error. We have found our single error bit that we can correct!

Let's look at a detailed example to get a better feel for how this works in practice:



Bob has received a noisy key from Alice, he has shuffled that key, and he has split the shuffled key into top-level blocks. The block labeled "noisy block at Bob" is one of those blocks. Let's just call it block N.

For the sake of clarity we have show corresponding block in the correct key at Alice as well. This is the block labeled "correct block at Alice".

As we can see, there are three bit errors in the noisy top-level block, namely the colored blocks at block indexes 2, 3 and 5.

We will now show how the Binary algorithm will detect and correct exactly one of those errors, namely the red one at block index 5.

The other two errors, the orange onces at block index 2 and 3, will neither be detected nor corrected by the Binary algorithm.

Here are the steps:

1. Bob splits top-level block N into two sub-blocks: the left sub-block N-L, and the right sub-block N-R.

2. Bob determines the error parity for the blocks N-L and N-R as follows:

   2a. Bob computes the current parity over block N-L and finds that it is 0.

   2b. Bob asks Alice for the correct parity over block N-L and gets the answer that it is 0.

   2c. Since the current parity and the correct parity for block N-L are the same, Bob concludes that the error parity must be even.

   2d. Bob infers that block N-R must have odd error parity.

3. Bob recurses in the to sub-block with odd error parity, which is block N-R.
4. Bob splits sub-block N-R into two sub-sub-blocks: the left sub-sub-block N-R-L, and the right sub-sub-block N-R-R.
5. Bob determines the error parity for the blocks N-R-L and N-R-R as follows:

   5a. Bob computes the current parity over block N-R-L and finds that it is 0.

   5b. Bob asks Alice for the correct parity over block N-R-L and gets the answer that it is 1.

   5c. Since the current parity and the correct parity for block N-R-L are the different, Bob concludes that the error parity must be odd.

   5d. Bob doesn't care about block N-R-R because he has already found his block to recurse into.

6. Bob recurses in the to sub-sub-block with odd error parity, which is block N-R-L.
7. Bob splits sub-sub-block N-R-L into two sub-sub-sub-blocks: the left sub-sub-sub-block N-R-L-L, and the right sub-sub-sub-block N-R-L-R.
8. Bob determines the error parity for the blocks N-R-L-L and N-R-L-R as follows:

   8a. Bob computes the current parity over block N-R-L-L and finds that it is 1.

   8b. Bob asks Alice for the correct parity over block N-R-L-L and gets the answer that it is 1.

   8c. Since the current parity and the correct parity for block N-R-L-L are the same, Bob concludes that the error parity must be even.

   8d. Bob infers that block N-R-L-R must have odd error parity.

9. Bob notices that block N-R-L-R has a size of only one bit. Bob has found an error and corrects that error by flipping the bit!

2.15 What about the remaining errors after correcting a single bit error?

Now consider what happens after Bob has used the Binary protocol to correct a single bit error in a block.

Before the correction the block had an odd number of errors, which means that after the correction the block will contain an even number of errors. It may be error-free (have 0 remaining errors), or it it may not yet be error-free (have 2, 4, 6, etc. remaining errors).

There is no way for Bob to know whether there are any errors left, and even if he did, Bob could not run the Binary algorithm on the same block again since the Binary algorithm can only be run on blocks with odd error parity. There is nothing left for Bob to do with the block, at least not during this iteration.

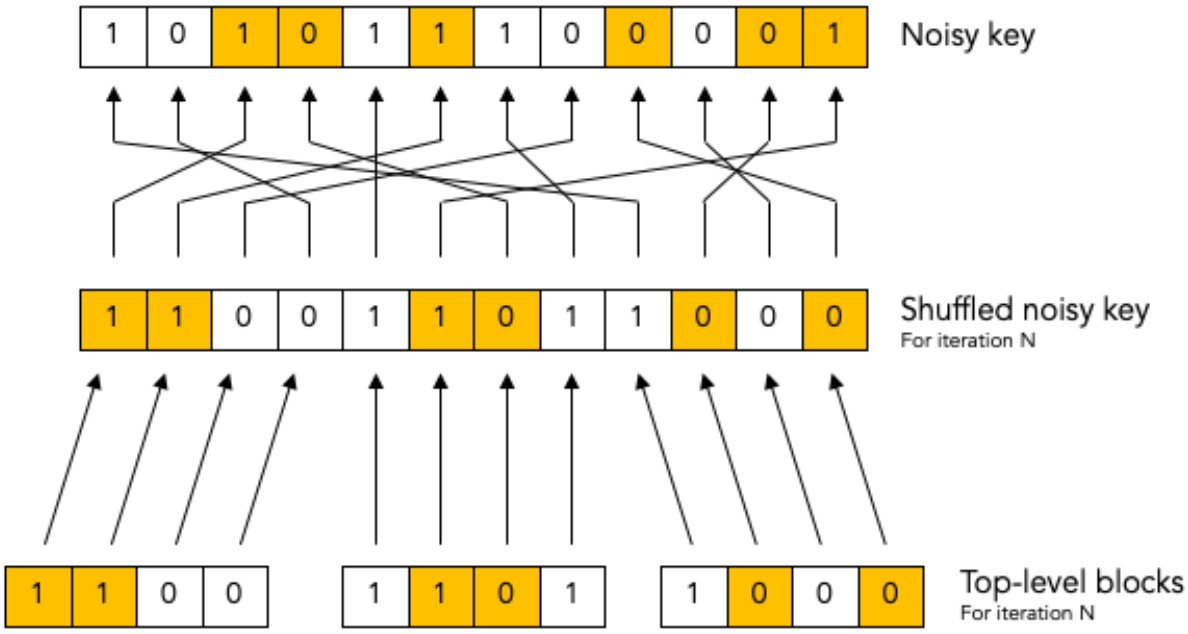
So what about the remaining errors in the block (if any)? How will they get corrected? There are two mechanisms:

1. Reshuffling in later iterations.
2. The cascading effect.

We will now discuss each of these mechanisms in turn.

2.16 The role of shuffling in error correction.

The following diagram show the situation that Bob might find himself in at the end of some iteration, say iteration number N:

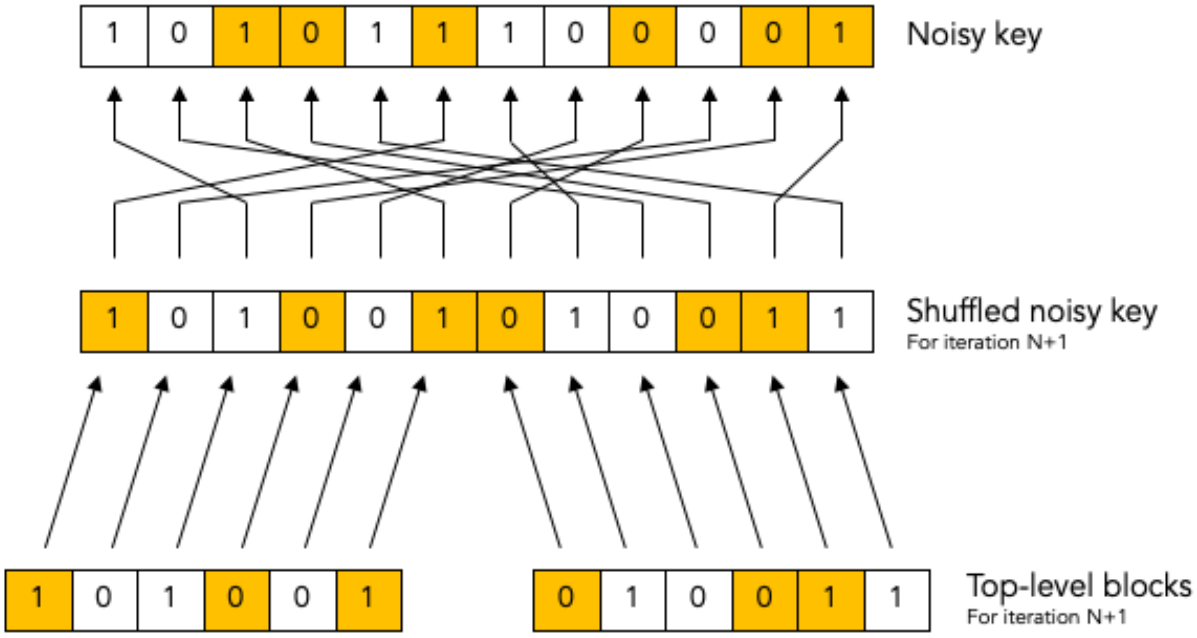


Maybe Bob already corrected a bunch of errors, but there are still six remaining errors left to correct.

Unfortunately, every top-level block contains an even number of remaining errors, so Bob is not able to make any progress during this iteration.

Bob has no choice but to move on to the next iteration N+1. In the next iteration, Bon reshuffles the keys (using a different shuffling order). Then he breaks up the reshuffled key into top-level blocks again, but using bigger blocks this time.

We might up with something like this at the beginning of iteration N+1:



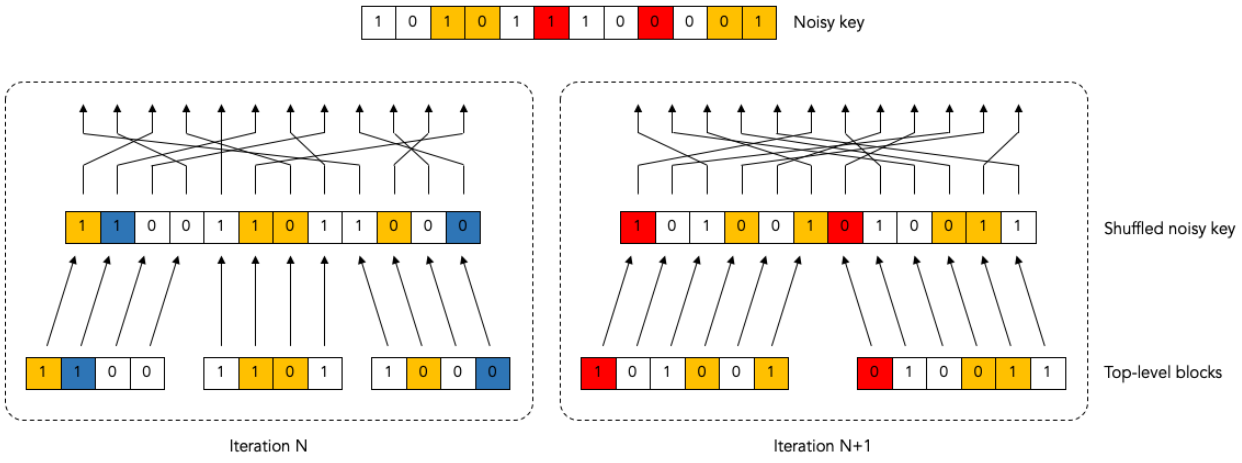
It is possible that at the beginning of iteration N+1 Bob ends up with some blocks that have an odd number of errors. Indeed, in this example Bob is quite lucky both remaining blocks have an odd number of errors (3 errors in each block).

Now Bob can make progress again: he can run the Binary algorithm on each block and remove exactly one error in each block. At the end of iteration N+1 there will be 4 errors remaining (2 in each block).

2.17 The Cascade effect.

Let's keep going with same example a little bit more.

If you go through the steps of the Binary protocol, you will see that Bob will end up correcting the two bits that are marked in red during iteration N+1:

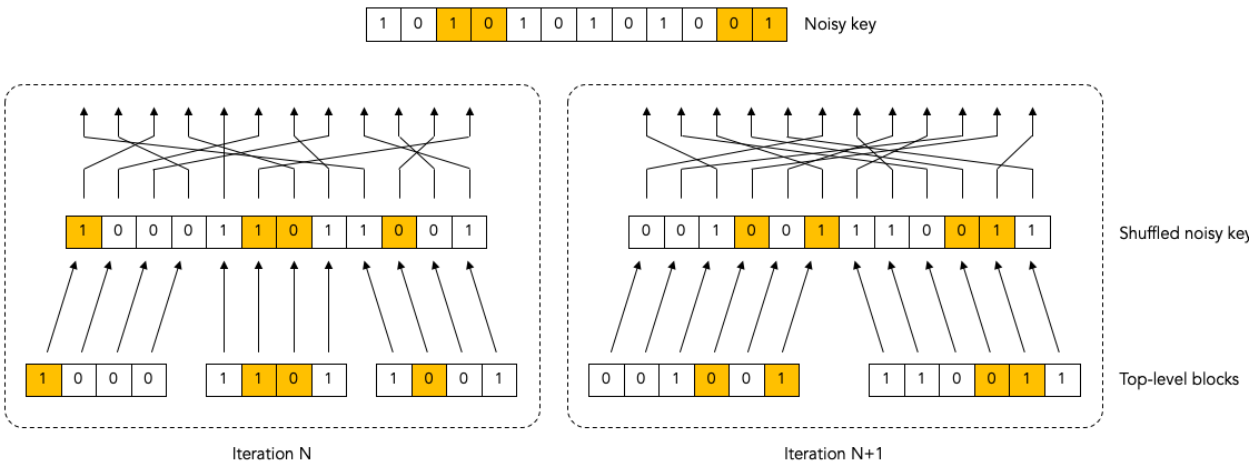


If you follow the arrows from the red corrected bit in the shuffled blocks back to the top you can see which bits in the underlying unshuffled noisy key will end up being corrected (these are also marked in red).

But wait! If are going to be flipping bits (correcting errors) on de underlying unshuffled noisy key, then this is going to have a ripple effect on the shuffled blocks from earlier iterations.

In this particular example, we can see that flipping the red bits in iteration N+1 will cause the blue bits in iteration N to be flipped as a side-effect.

After all the red and blue bit flipping is done, we end up with the following situation:



As we discussed before, we have corrected two bit errors in iteration N+1, and now there are 4 bit errors remaining.

And, as expected, we are now stuck as far as iteration N+1 is concerned. We can make no further progress in iteration N+1 because each block has an even number of errors.

But look! Because of the ripple effect on the previous iteration N, we now have two blocks in iteration N that now have an odd number of errors! Bob can go back to those iteration N blocks and re-apply the Binary protocol to correct one more error in them.

This ripple effect is what is called the cascade effect that gives the Cascade protocol its name.

The cascade effect is very profound and much stronger than it seems from our simple example.

Firstly, fixing an error in iteration N does not only affect iteration N, but also iterations N-1, N-2, . . . , 1.

Secondly, consider what happens when the cascade effect causes us to go back and revisit a block in an earlier iteration and fix another error there. Fixing that error in the earlier block with cause yet another cascade effect in other blocks. Thus, when we correct a single error, the cascade effect can cause a veritable avalanche of other cascaded error corrections.

2.18 Parallelization and bulking.

Every time Bob asks Alice to provide the correct parity for a block he sends an *ask parity* message and then waits for the *reply parity* response message.

If Alice is in Amsterdam and Bob is in Boston they are 5,500 km apart. The round-trip delay of the *ask parity* and *reply parity* messages will be 110 milliseconds (the speed of light in fiber is 200,000 km/sec) plus whatever time Alice needs to process the message.

During the reconciliation of a single large key Bob can ask Alice for many parities (hundreds of ask parities for a 10,000 bit key, for example).

If Bob processes all blocks serially, i.e. if Bob doesn't start working on the next block until he has completely finished the Binary algorithm for the previous block, then the total delay will be very long. If we assume 200 *ask parity*

messages, it will add up to at least a whopping 22 seconds. That is clearly too slow.

5,500 km was a bit extreme, just to make a point. But even for more realistic distances for quantum key distribution, say 50 km, the round-trip time delay is significant.

Luckily Bob does not have to process all blocks sequentially; he can do some parallel processing of blocks.

The lowest hanging fruit to to parallelize the processing of the top-level blocks. At the beginning of each iteration, Bob shuffles the key and splits it up into top-level blocks. Bob can then send a single "bulked" *ask parities* (plural) message asking Alice for all the parities of all the top-level blocks in that iteration. Alice sends a single *reply parities* (plural) message with all correct parities. Then Bob can start processing all the top-level blocks.

But to get the full effect of parallelization Bob must do more. When Bob, in the process of running the Binary algorithm, gets to the point that he needs to ask Alice for the parity of a sub-block, Bob should not block and do nothing, waiting for the answer from Alice. Instead, Bob should send the *ask parity* message and then go work on some other block that has an odd number of errors "in parallel" while waiting for the answer to the first message. Working on multiple sub-blocks "in parallel" greatly reduces the total latency for an iteration.

If, in addition to reducing the latency, Bob also wants to reduce the number of *ask parity* messages, Bob can do "bulking" of messages. When Bob needs to ask Alice for the parity for some block B1, Bob can already start working on some other block B2. But instead of immediately sending the *ask parity* message for block B1, Bob can hold off for some time in anticipation of probably having to ask to parity for some other parities as well.

Note that the bulking of messages reduces the number of messages bit it does very little to reduce the volume (i.e. total number of bytes) of messages.

In the extreme case, Bob can hold off sending any *ask parities* message until he can absolutely positively not make any more progress before he gets an answer. But that would increase the latency again because it would force Bob to sit idle not doing anything.

The sweet spot is probably to hold off sending *ask parity* messages for only a fixed delay (similar to what the Nagle algorithm does in TCP/IP).

2.19 Information leakage.

TODO: Quantitative discussion of information leakage.

2.20 Variations on the Cascade Protocol.

TODO: Discuss variations on Cascade protocol.

TODO: Different number of iterations.

TODO: Different functions for calculation block size.

TODO: BICONF.

TODO: Sub-block re-use.

TODO: Block parity inference (BPI).

TODO: Deterministic vs random shuffle.

TODO: Discarding of corrected bits.

CHAPTER

THREE

RAW COMPARISON OF RESULTS WITH LITERATURE.

3.1 Comparison with "Demystifying the Information Reconciliation Protocol Cascade"

Here we compare the results of our Python Cascade implementatin with the results reported in the following paper:

Demystifying the Information Reconciliation Protocol Cascade. *Jesus Martinez-Mateo, Christoph Pacher, Momtchil Peev, Alex Ciurana, and Vicente Martin.* arXiv:1407.3257 [quant-ph], Jul 2014.

3.1.1 Figure 1

Original figure in paper:

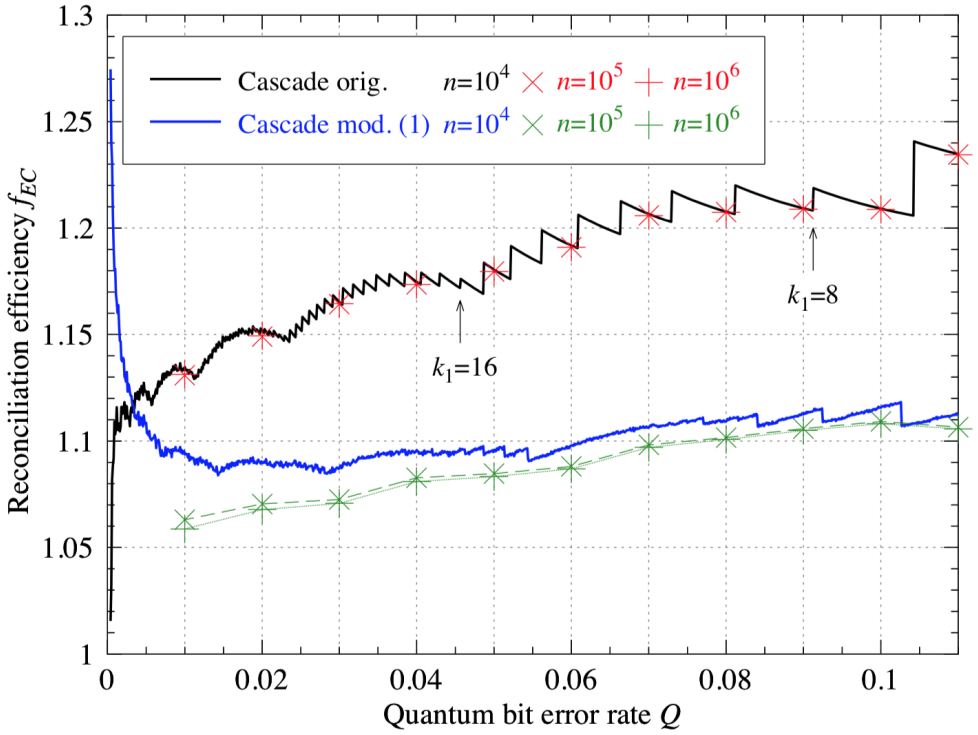
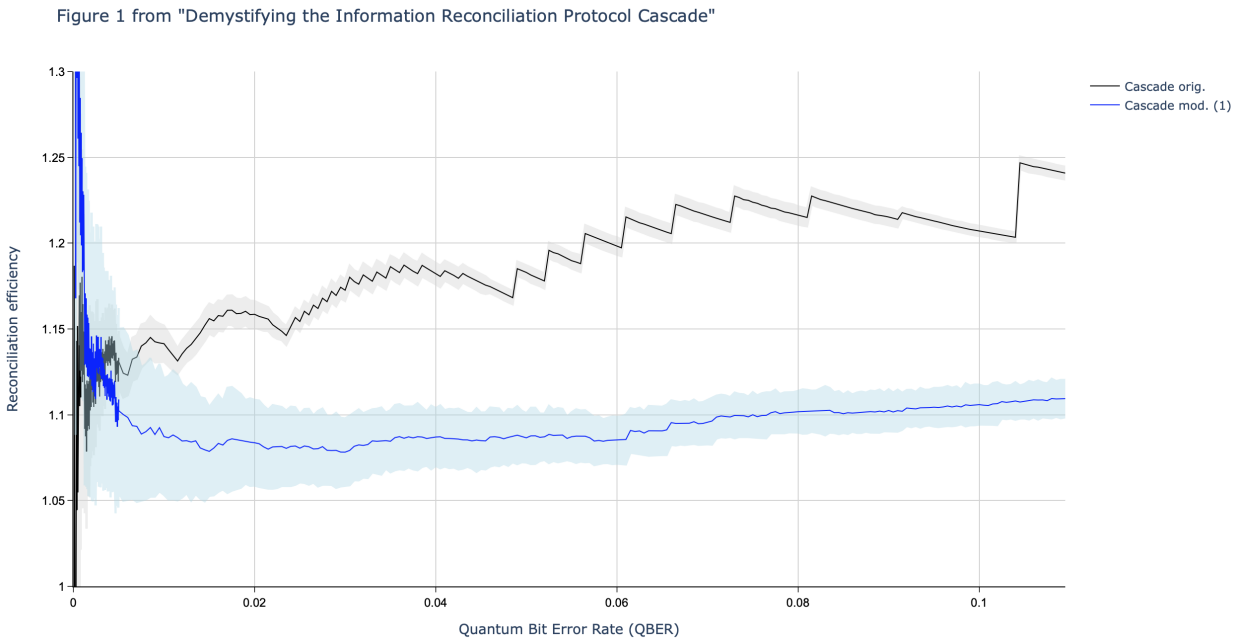


Figure 1: Comparison of the average reconciliation efficiency as a function of the quantum bit error rate Q for the original `Cascade` protocol [6] and the version of [10] (i.e., modified version (1) throughout this paper). The length of the frames used is denoted by n.

Reproduced figure from this code:



The original and the reproduced figure match very well.

The original graph has more detail because they executed more runs per data point (this is true for all graphs, so we

won't repeat this point.)

The original graph does not have any indication of the standard deviation (this is true for all graphs, so we won't repeat this point).

In order to get my results to match well with the results reported in the literature, I had to use a very unrealistic definition of reconciliation efficiency. This is discussed in more detail in the comparison conclusions section.

3.1.2 Figure 2

Original figure in paper:

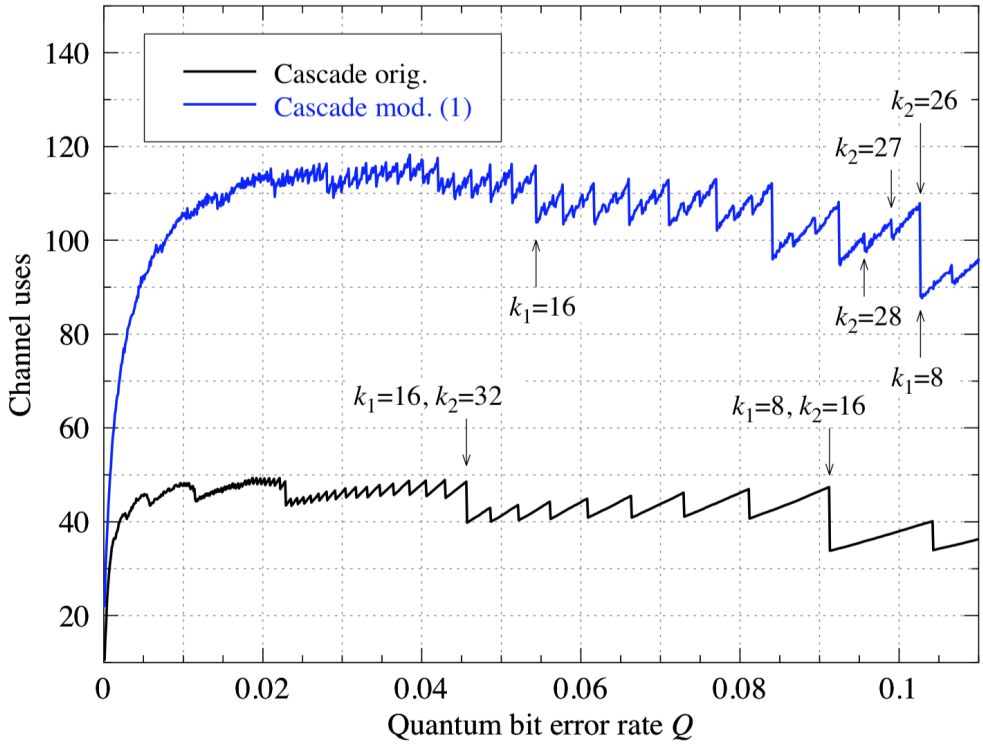
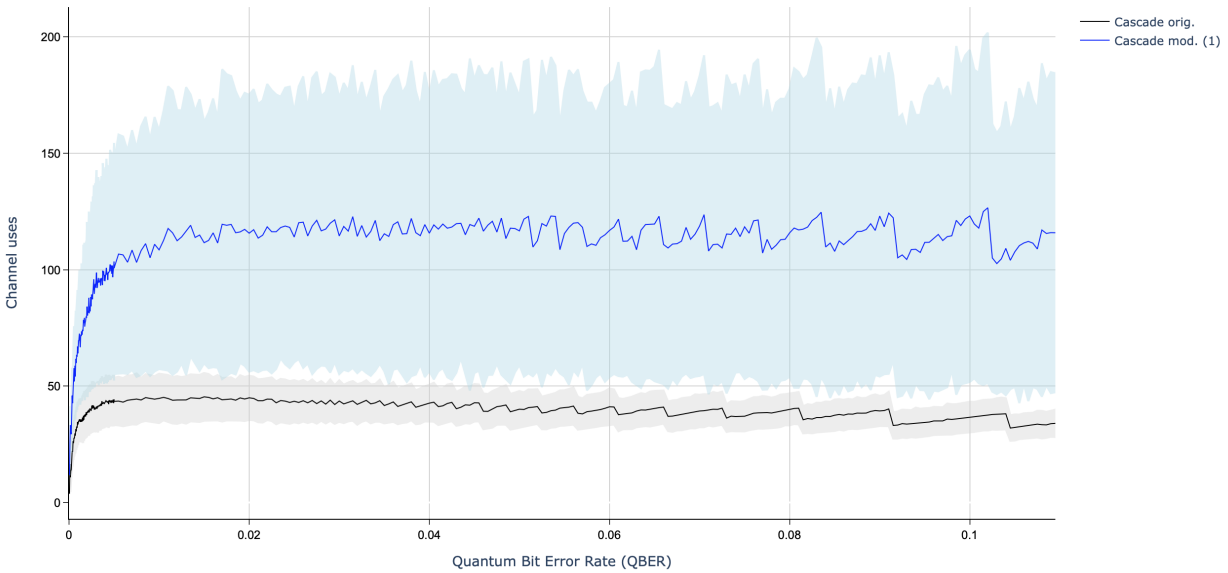


Figure 2: Comparison of the number of channel uses as a function of the quantum bit error rate Q for the original `Cascade` protocol [6] and the version (mod. (1)) of [10].

Reproduced figure from this code:

Figure 2 from "Demystifying the Information Reconciliation Protocol Cascade"



At first blush the original and the reproduced figure match are quite similar.

However, both the original algorithm (black line) and the modified algorithm (blue) line have a more distinct drop-off towards the right side of the figure.

For example, in the figure from the original paper, as the x-axis increases from bit error rate 0.03 to 1.10, the jigsaw shape of the blue line starts to have bigger "jigsaws" and also distinctly slopes down. In the reproduced graph, we do see the bigger "jigsaws" but we don't see the downward slope: the line stays essentially flat. I currently have no explanation for this difference.

3.1.3 Figure 3

Original figure in paper:

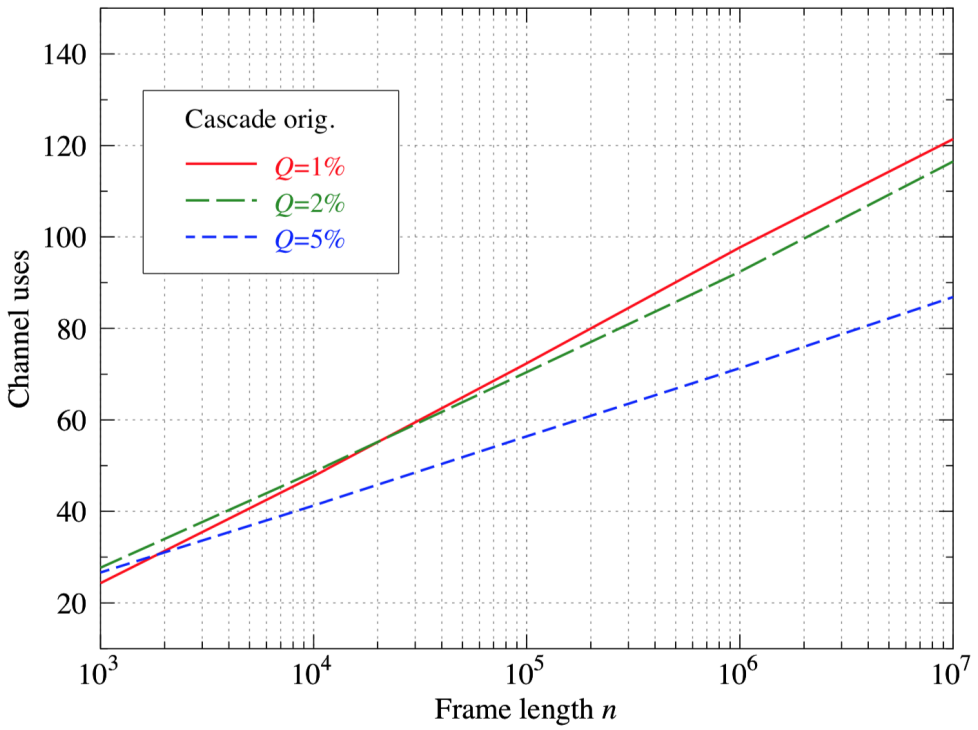
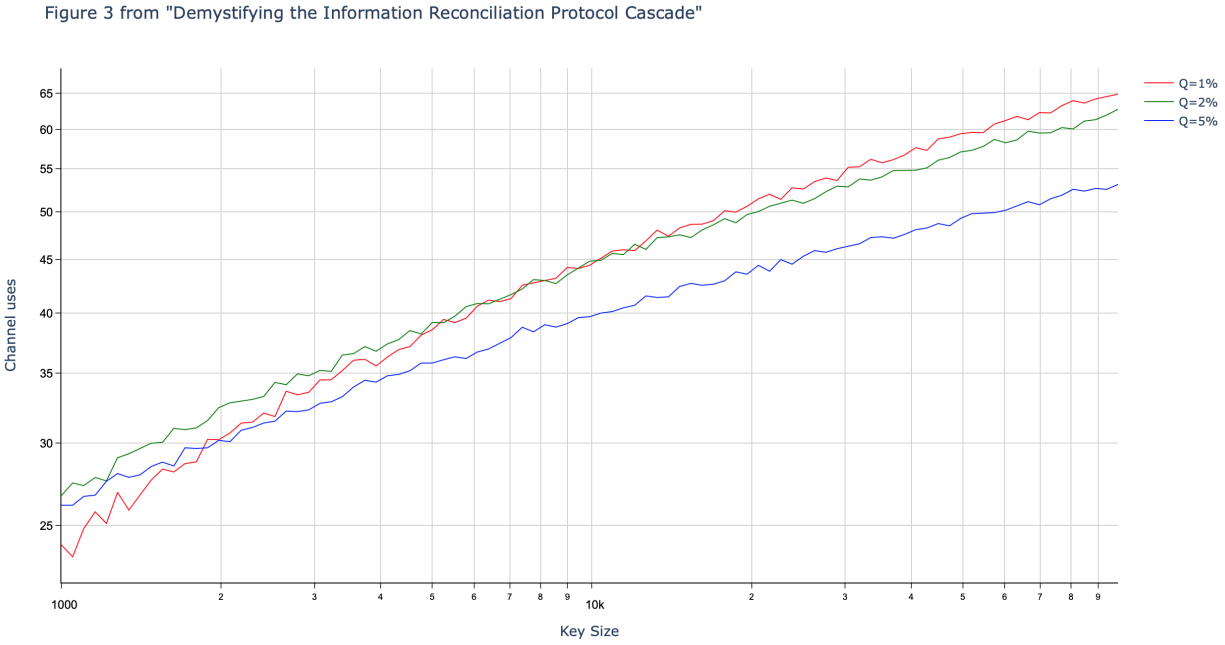


Figure 3: Channel uses (communication rounds) as a function of the frame length n for the original `Cascade` protocol [6].

Reproduced figure from this code:



In the original figure, the frame length (the x-axis) ranges from 10^3 to 10^7. In the reproduced figure the frame length only ranges from 10^3 to 10^5. This is because my Python code was too slow to run many iterations for key lengths 10^6 or 10^7.

In the original figure, the lines are perfectly straight. In the reproduced figure the lines are very slightly curved.

Other than those minor differences, the original and reproduced figure match very well: they shapes are very similar and the lines cross over at the same points.

3.1.4 Figure 4

Original figure in paper:

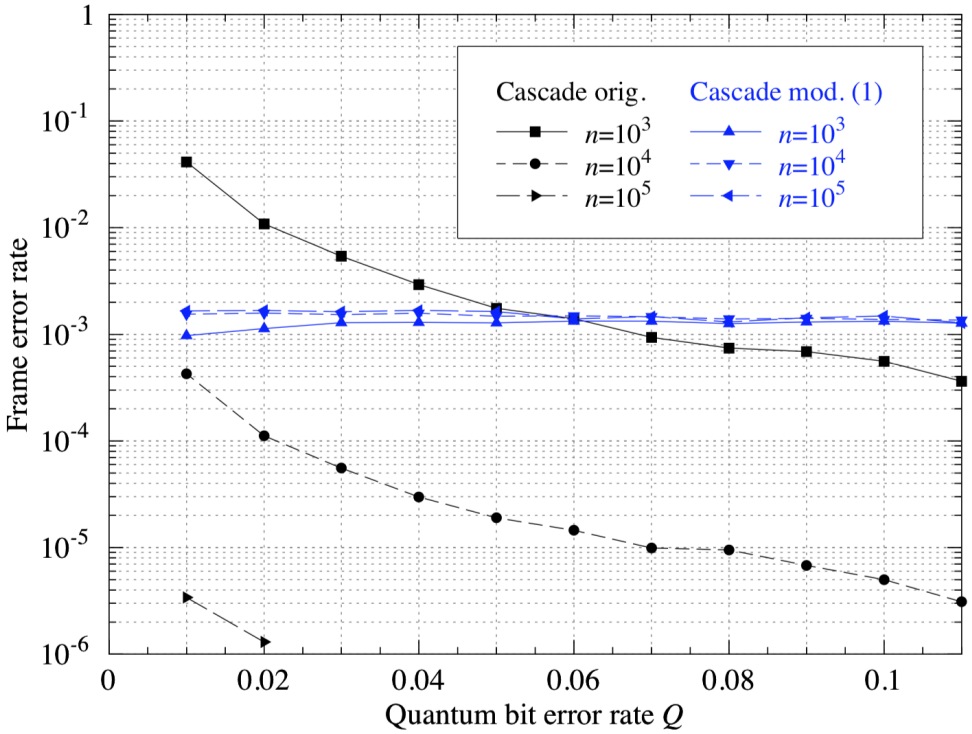
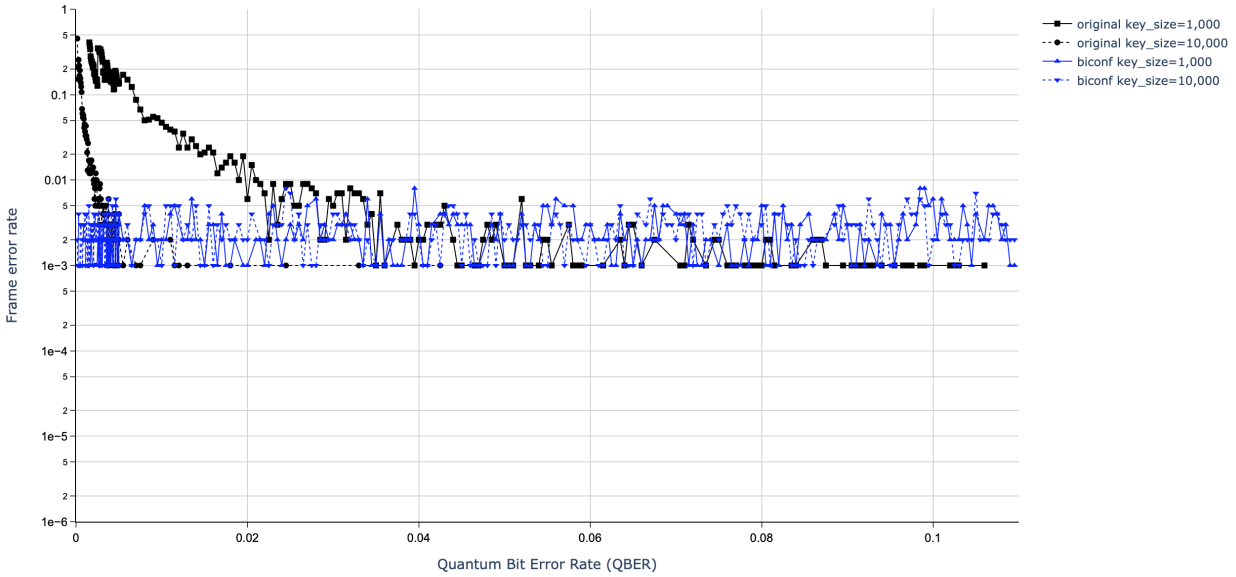


Figure 4: Frame error rate (failure probability) as a function of the quantum bit error rate for the original `Cascade` protocol [6] and the version (mod. (1)) of [10].

Reproduced figure from this code:

Figure 4 from "Demystifying the Information Reconciliation Protocol Cascade"



The reproduced figure is mostly useless: for small frame error rates (below 10^-3) the reproduced figure falls apart. This is because I only did 1,000 iterations per data point. The Python code was too slow to do more iterations per data point. To detect frame errors below 10^-n we need to do at least 10^n iterations. Hence, running only 1,000 iterations it is only to be expected that we cannot detect frame error rates below 10^-3.

Hopefully, when we have a faster C++ implementation we will be able to study lower error rates.

3.1.5 Figure 5

Original figure in paper:

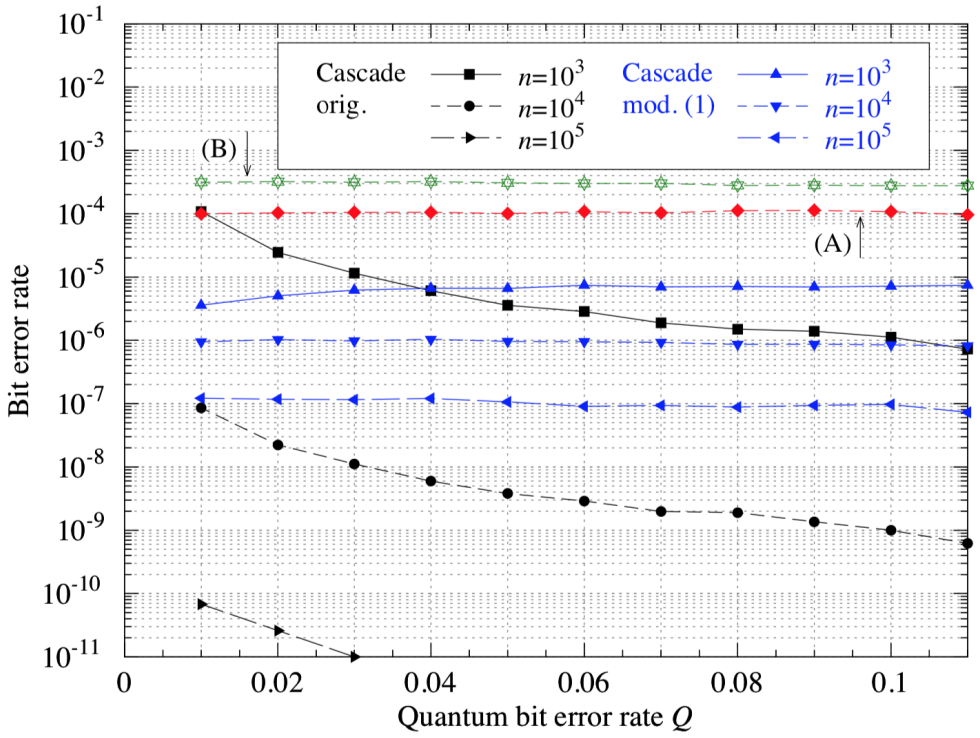
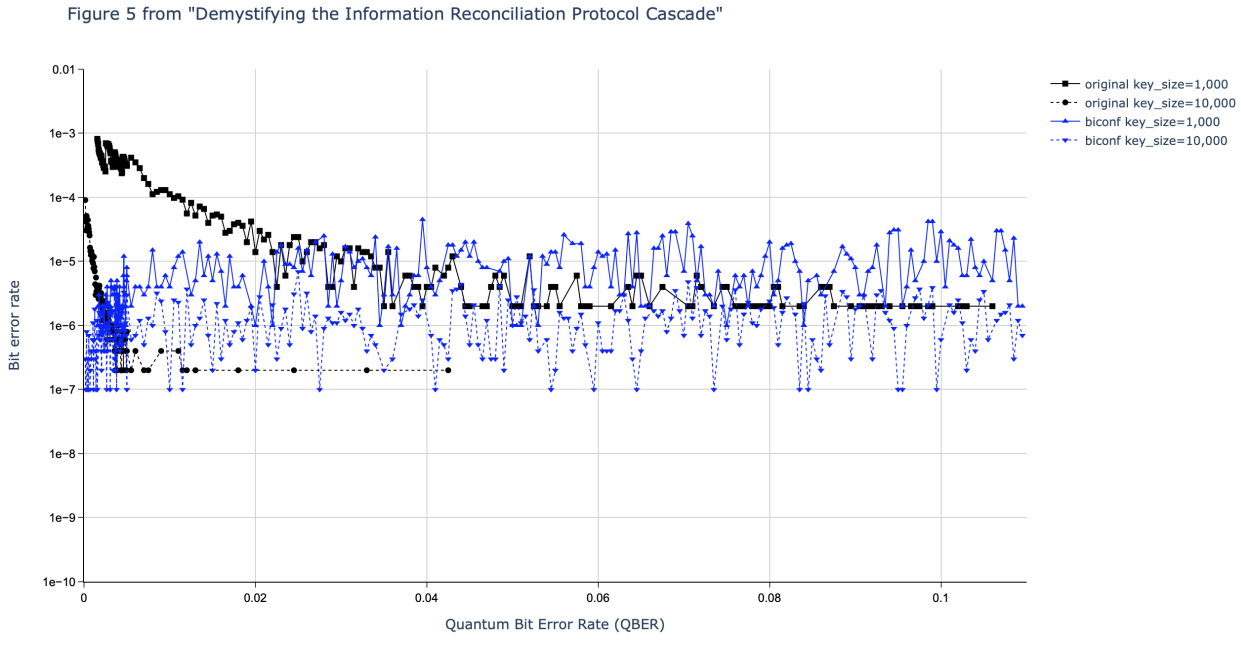


Figure 5: Bit error rate (residual error) as a function of the quantum bit error rate for the original `Cascade` protocol [6] and the version (mod. (1)) of [10]. The curves labeled (A) and (B) correspond to the bit error rate after the first two passes of the original `Cascade` (A) and two passes in the modified one without `BICONF` (B).

Reproduced figure from this code:



Once again, the reproduced figure is mostly useless, for similar reasons to figure 4.

Due to the slow Python code, we only did 1,000 (10^3) iterations per data point. For key size 1,000 (10^3) this means we cannot bit error rates below 10^-6. And for key siz 10,000 (10^4) this means we cannot detect bit error rate below 10^-7.

As expected, the reproduced figure 5 falls apart below these 10^-6 (black line) and 10^-7 (blue line) bit error rates.

Again, hopefully, when we have a faster C++ implementation we will be able to run more iterations per data point and hence study lower error rates.

3.1.6 Figure 6

Original figure in paper:

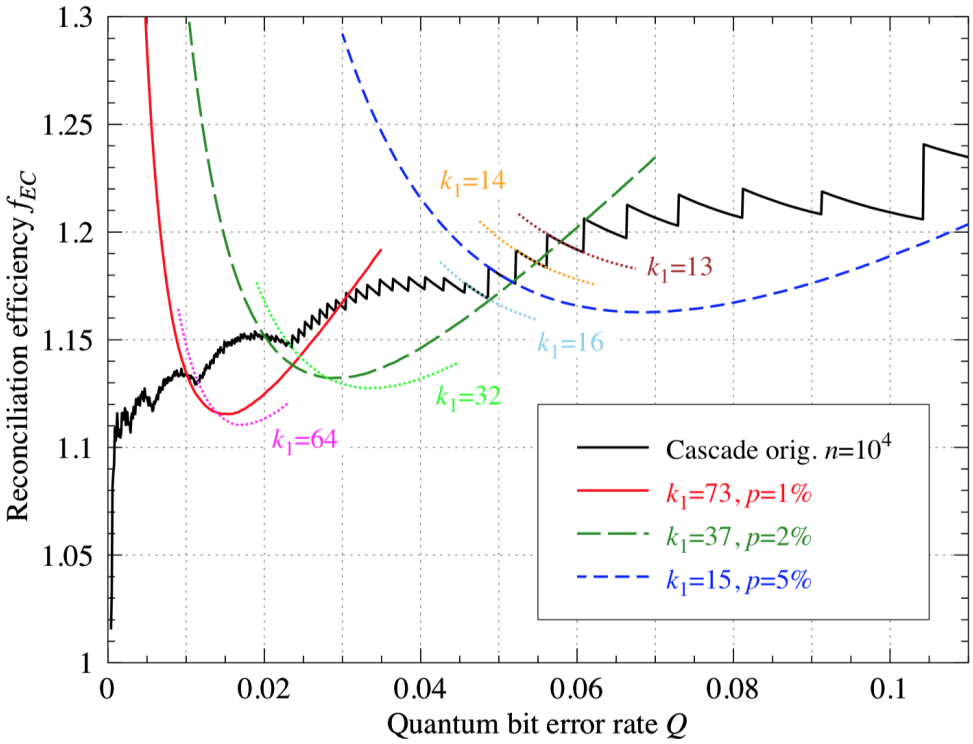


Figure 6: Average reconciliation efficiency, f_{EC}, as function of the quantum bit error rate when k_1 is fixed over a larger interval of Q than originally proposed [6]. For comparison also the efficiency of the original `Cascade` protocol [6] is shown.

This figure is not (yet) reproduced by the code.

3.1.7 Figure 7

Original figure in paper:

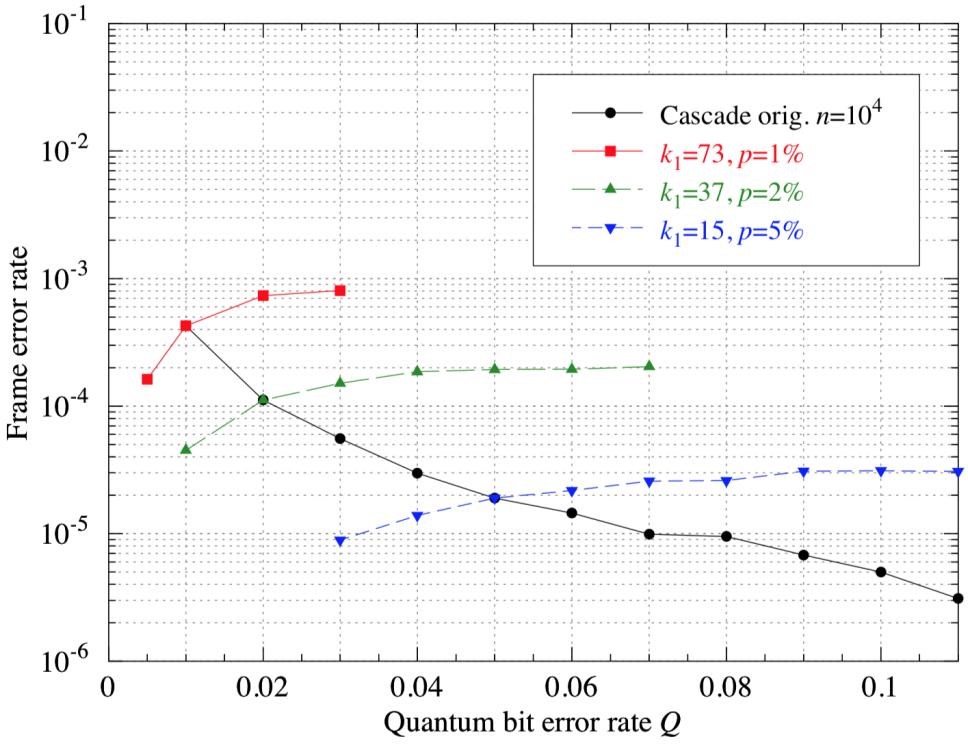


Figure 7: Frame error rate (failure probability) as a function of the quantum bit error rate when k_1 is fixed over a larger interval of Q than originally proposed [6]. For comparison, also the frame error rate of this original `Cascade` protocol is shown.

This figure is not (yet) reproduced by the code.

3.1.8 Figure 8

Original figure in paper:

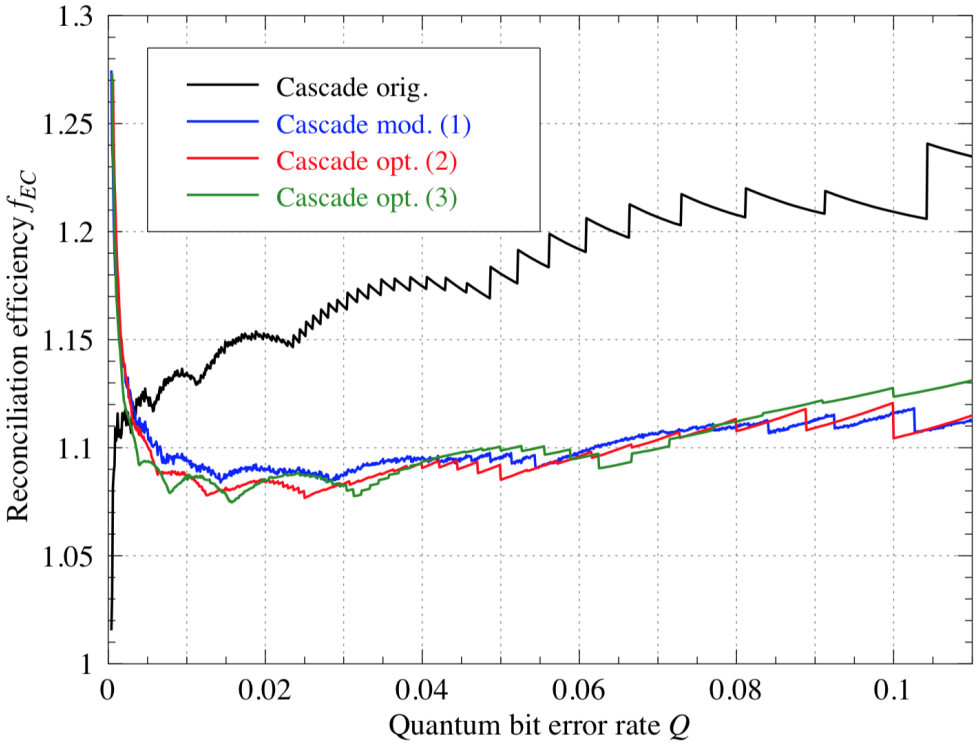
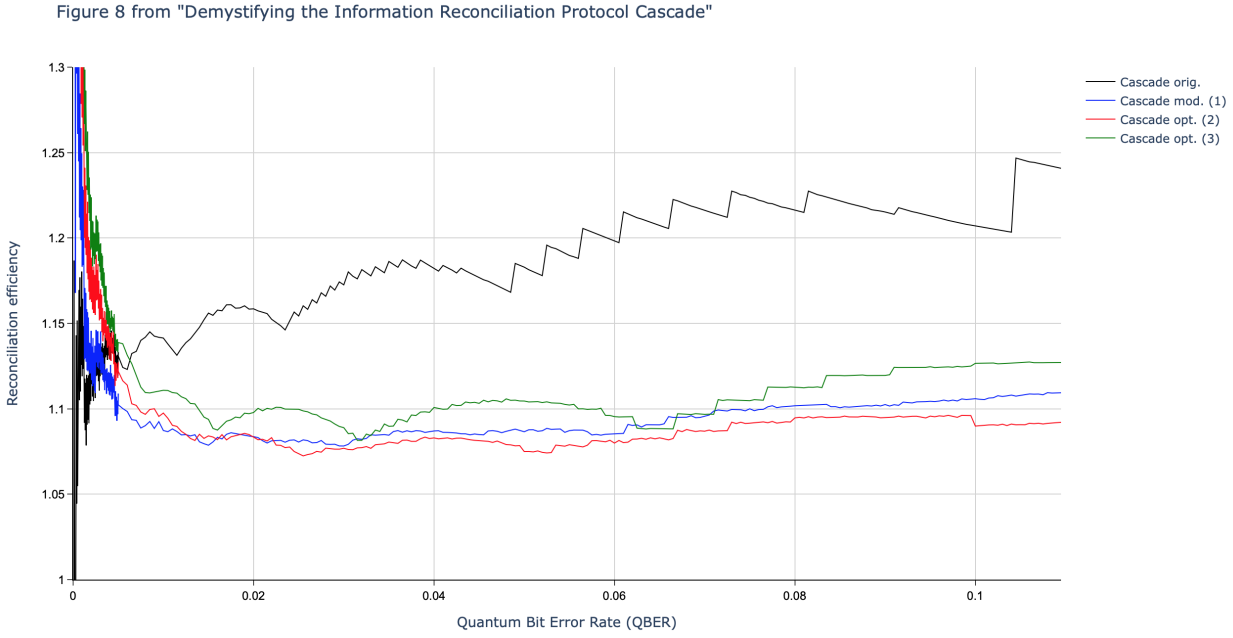


Figure 8: Average reconciliation efficiency, f_{EC}, of the original `Cascade` (black) and three modified versions: (1) the modified protocol proposed in [10] (blue), (2) the version using the optimized parameters suggested in [19] (red), and (3) the version proposed here using 16 passes (green). More details are given in the text.

Reproduced figure from this code:



The original and the reproduced figure match very well.

Once again, the original graph has more detail because they executed more runs per data point.

3.1.9 Figure 9

Original figure in paper:

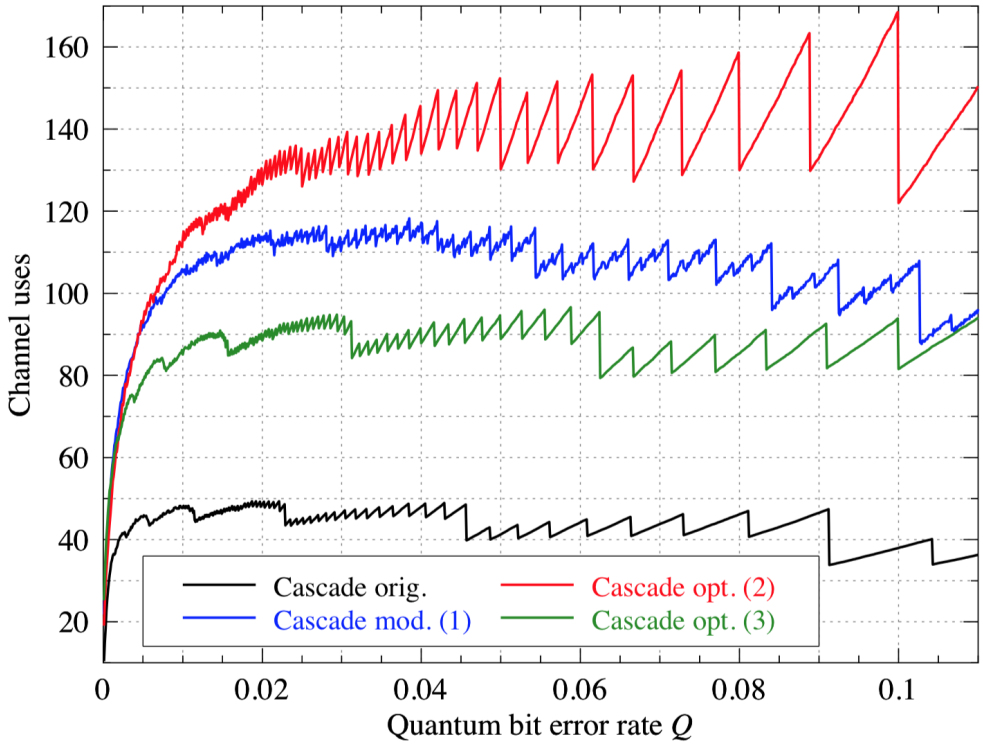
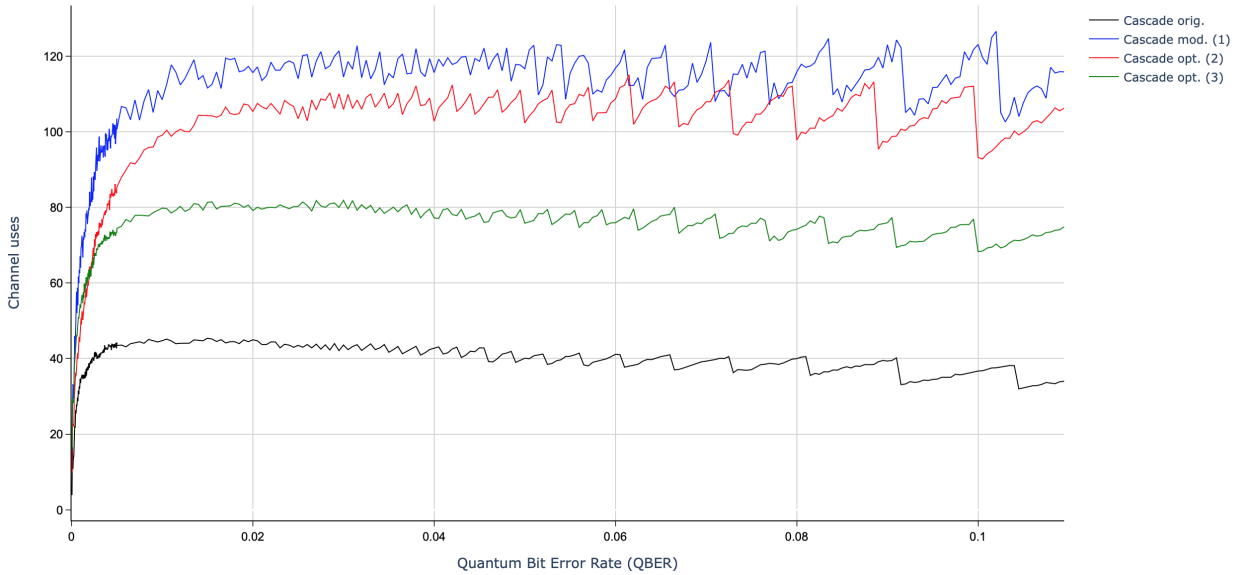


Figure 9: Channel uses of the original `Cascade` (black) and three modified versions: (1) the modified protocol proposed in [10] (blue), (2) the version using the optimized parameters suggested in [19] (red), and (3) the version proposed here using 16 passes (green). More details are given in the text.

Reproduced figure from this code:

Figure 9 from "Demystifying the Information Reconciliation Protocol Cascade"



There are few noticeable differences between the original figure and the reproduced figure.

There is of course the fact that the original figure has more detail than the reproduced figure, because we run fewer iterations per data point.

The black, green, and blue graphs match reasonably well in the original and the int reproduced figure. They have very similar values and similar shapes including the obvious saw-teeth. There are a few differences in the details though.

We already observed the first difference in figure 2. In the original graph the blue graph clearly slopes down towards the end. In the reproduced graph the blue line saw-tooths around a flat trend instead of trend that slopes down.

In the green and the black graphs, we also see a difference. In the original graph we see a lower frequency secondary wave pattern on top of the higher frequency saw-teeth. For example, there are 6 “waves” in the green graph and lots of saw-teeth within each “wave”. In the reproduced green and black graphs, we do not see these “waves”.

The last and biggest difference is in the red graph. This graph is completely different in the original vs reproduced figure. In the original figure the red graph is much higher (above the blue graph) and has much bigger saw-teeth.

I currently do not have an explanation for any of these observed differences.

3.1.10 Figure 10

Original figure in paper:

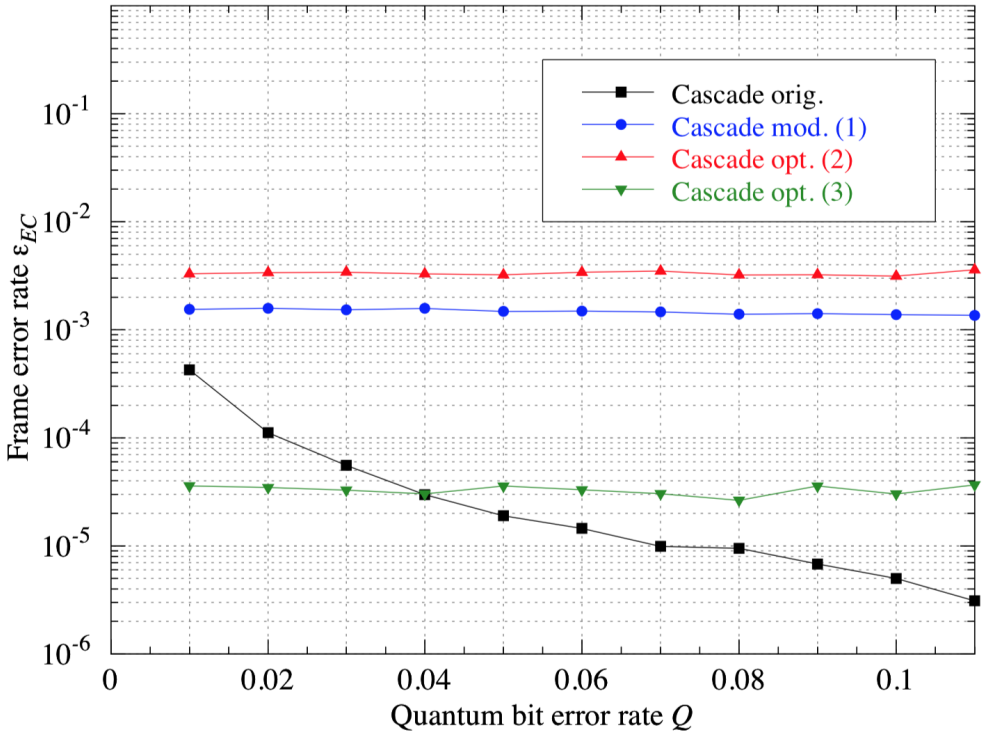
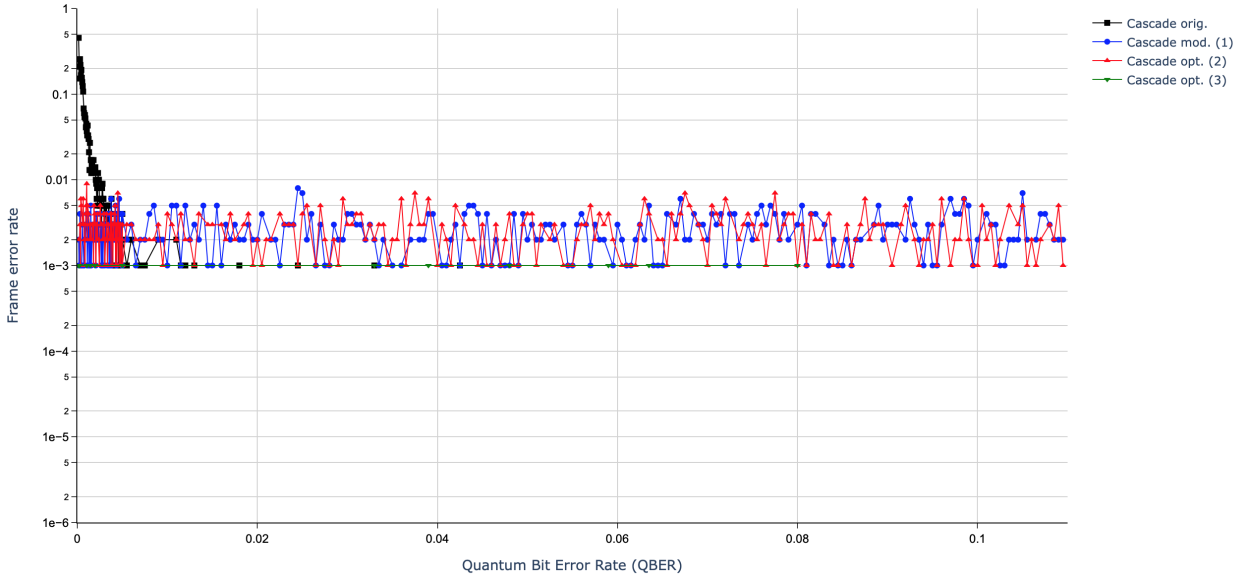


Figure 10: Frame error rate (failure probability) of the original `Cascade` (black) and three modified versions: (1) the modified protocol proposed in [10] (blue), (2) the version using the optimized parameters suggested in [19] (red), and (3) the version proposed here using 16 passes (green). More details are given in the text.

Reproduced figure from this code:

Figure 10 from "Demystifying the Information Reconciliation Protocol Cascade"



3.1.11 Figure 11

Original figure in paper:

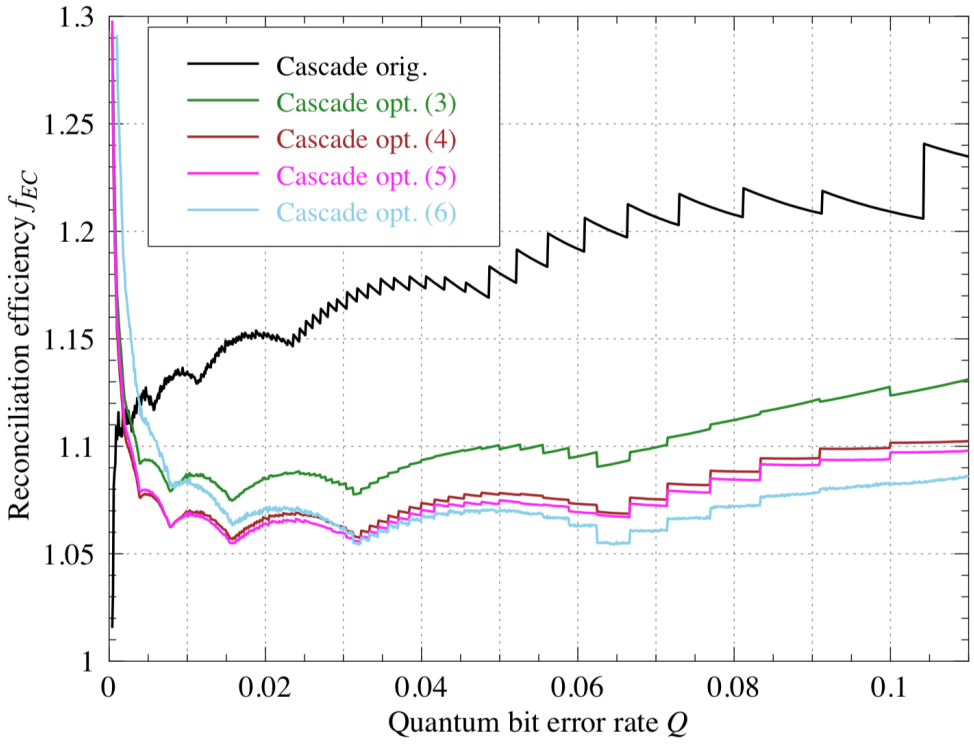
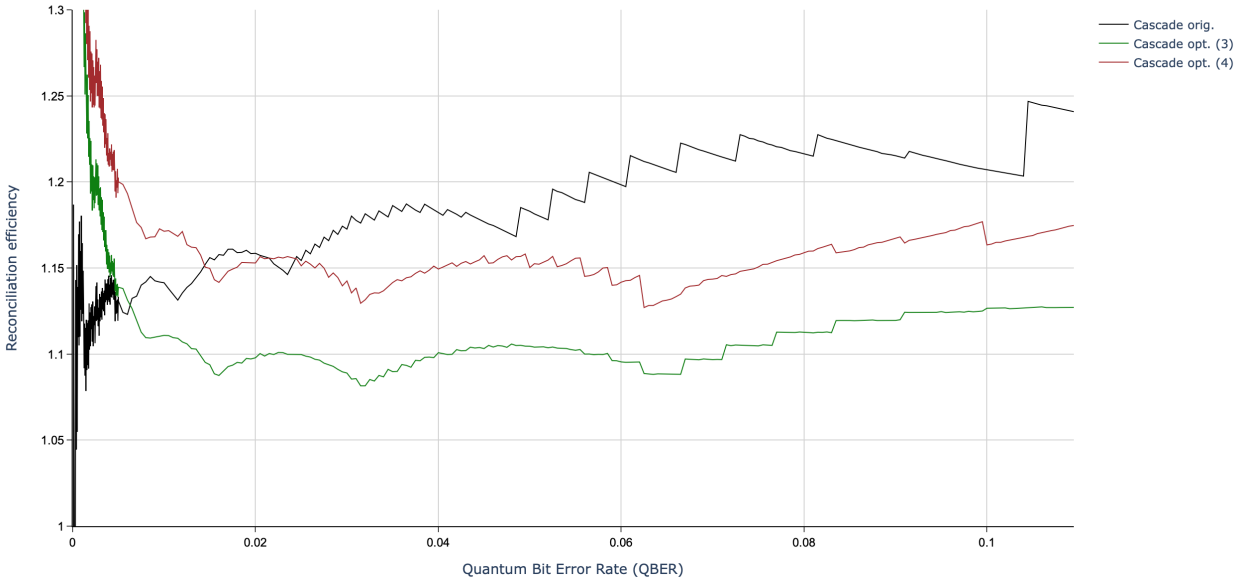


Figure 11: Average reconciliation efficiency, f_{EC}, of the original `Cascade` (black) and for optimized versions: (3) the version using 16 passes, proposed above and presented in the previous figures (green), (4) same as (3) but leveraging in addition the idea of block reuse as suggested by [19] (brown), (5) same as (4) but replacing the random shuffling between passes (magenta), and (6) same as (3) but discarding singleton blocks after each pass (sky blue). More details are given in the text.

Reproduced figure from this code:

Figure 11 from "Demystifying the Information Reconciliation Protocol Cascade"



3.1.12 Figure 12

Original figure in paper:

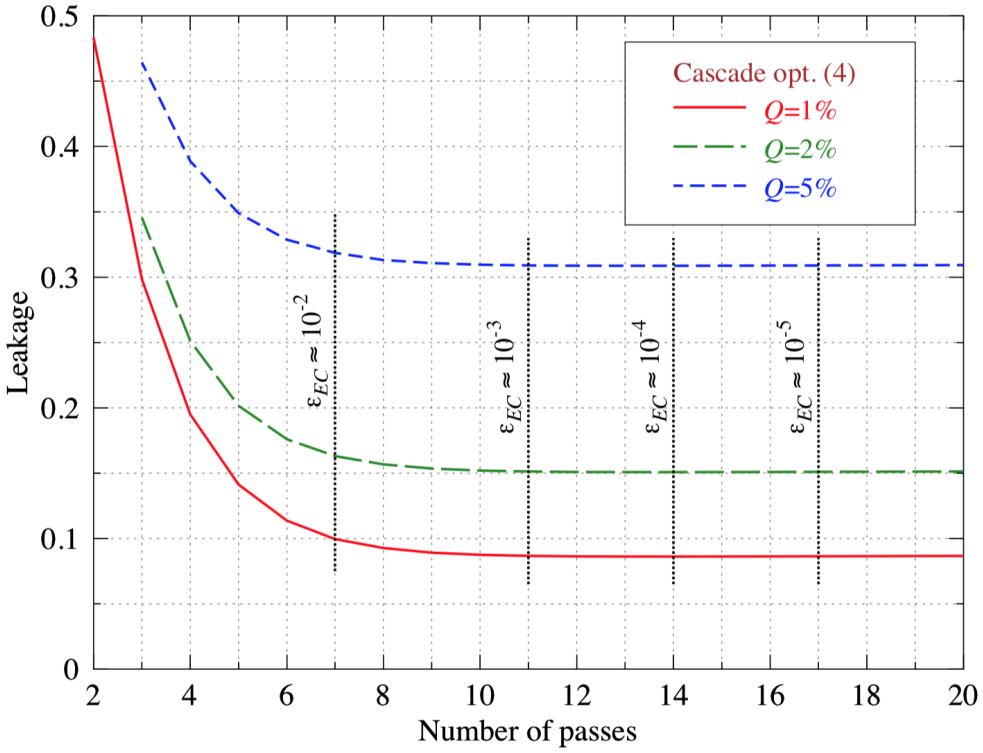


Figure 12: Information leakage as a function of the number of passes in the proposed modification of `Cascade` utilizing subblock reuse, labeled with (4) in Fig. 11.

This figure is not (yet) reproduced by the code.

3.1.13 Figure 13

Original figure in paper:

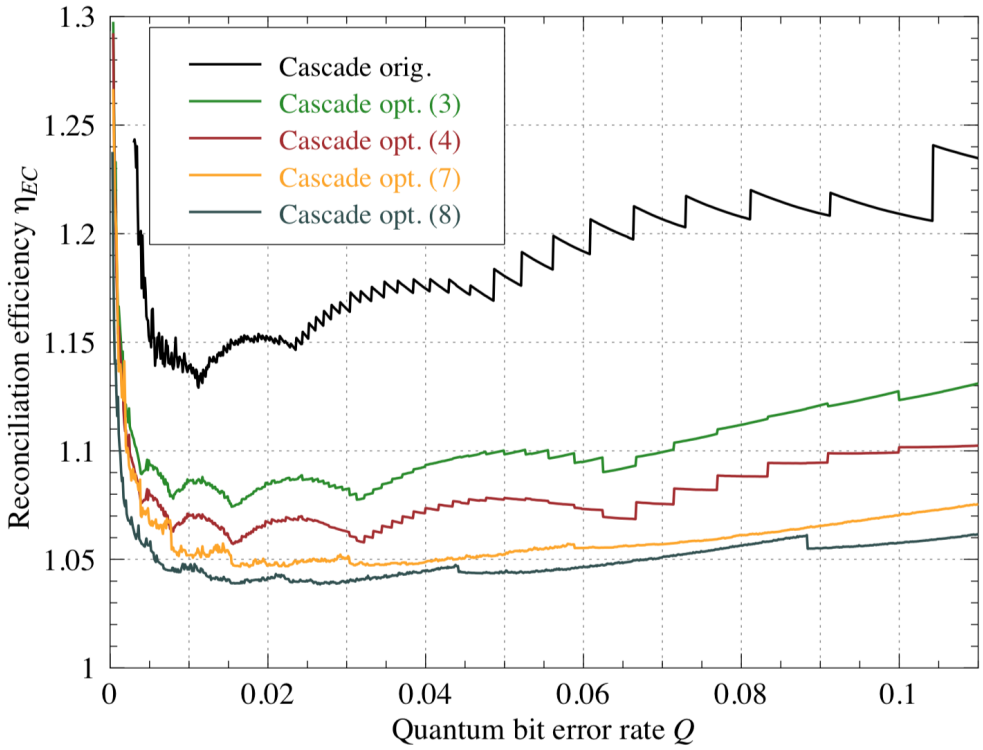
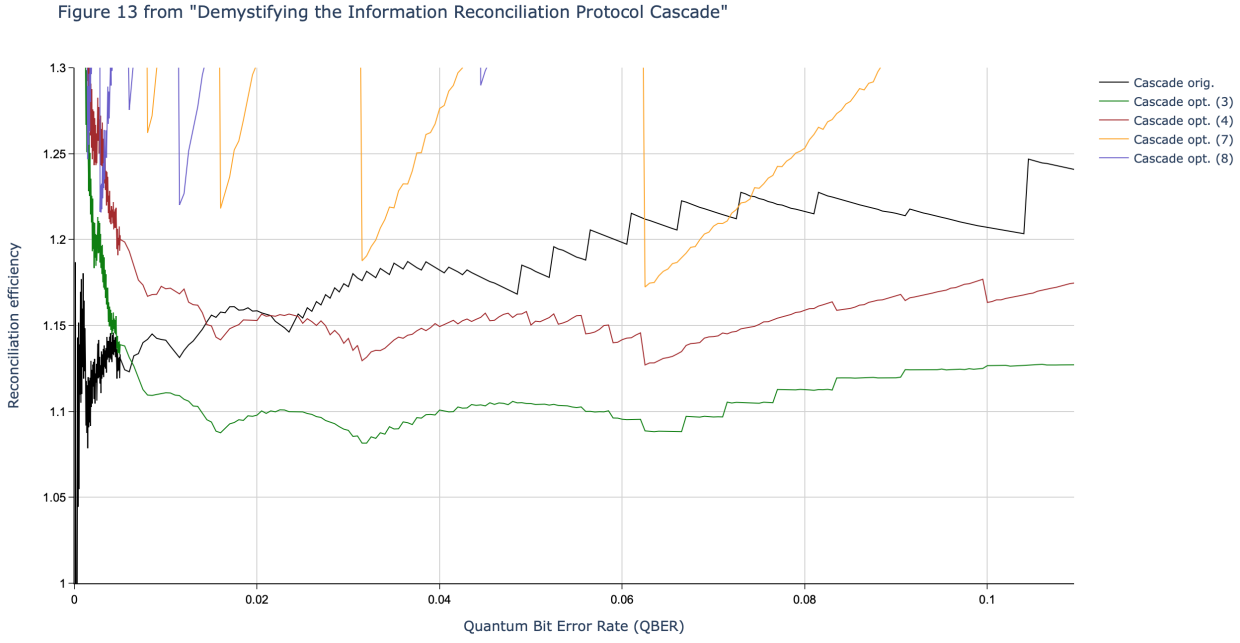


Figure 13: Average reconciliation efficiency, η_{EC}, of the original `Cascade` (black) and for optimized versions: (3) the first version proposed above and presented in the previous figures using 16 passes (green), (4) same as (3) but leveraging in addition the idea of block reuse (brown), (7) same as (4) but optimizing the first and second block sizes and using 14 passes (orange), and (8) same as (7) but also optimizing the third block size and using a power of two value for the frame length $n = 2^{14}$ (dark gray). More details are given in the text.

Reproduced figure from this code:



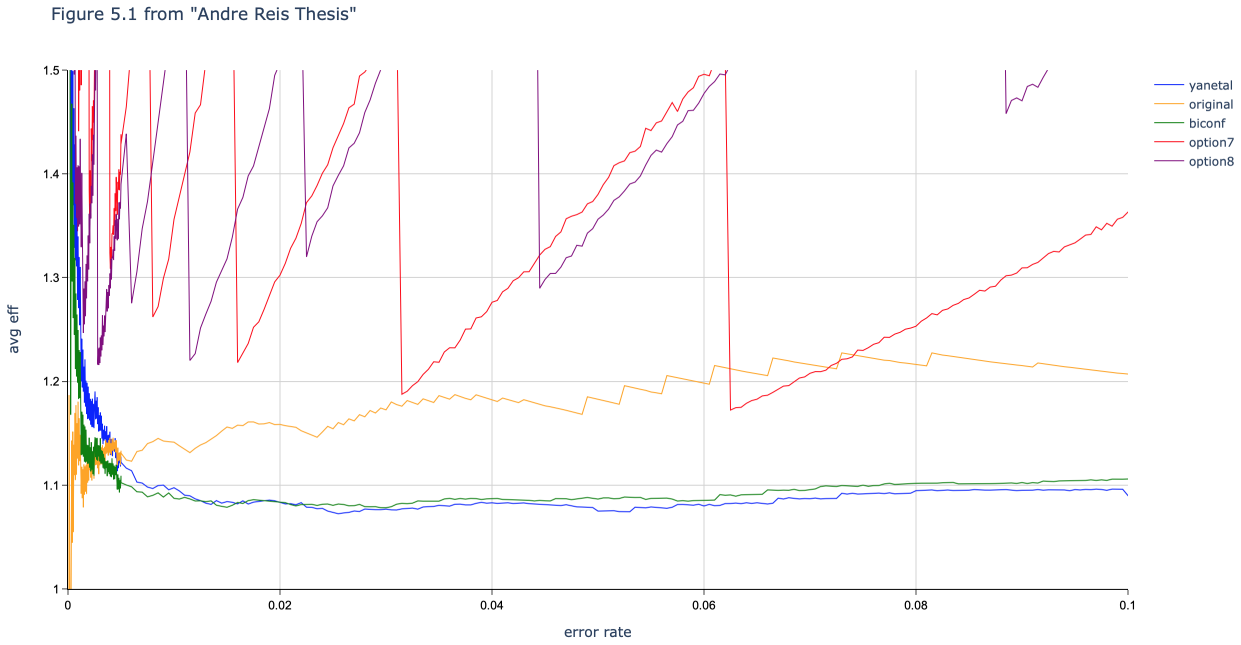
3.2 Comparison with “Andre Reis Thesis”

3.2.1 Figure 5.1

Original figure in thesis:

Figure 5.1: First experiment reconciliation efficiency by error rate on keys with 10000 bits

Reproduced figure from this code:



3.2.2 Figure 5.2

Original figure in thesis:

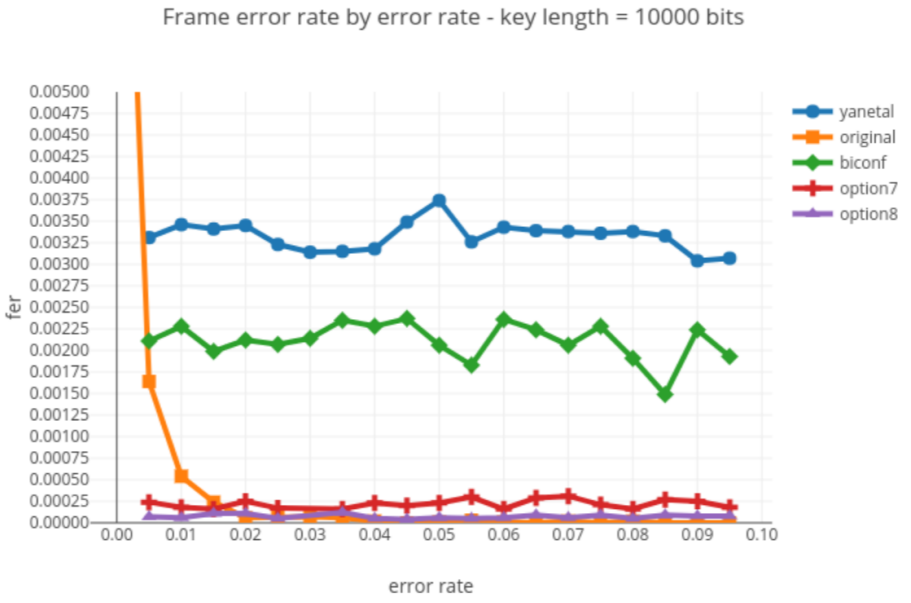
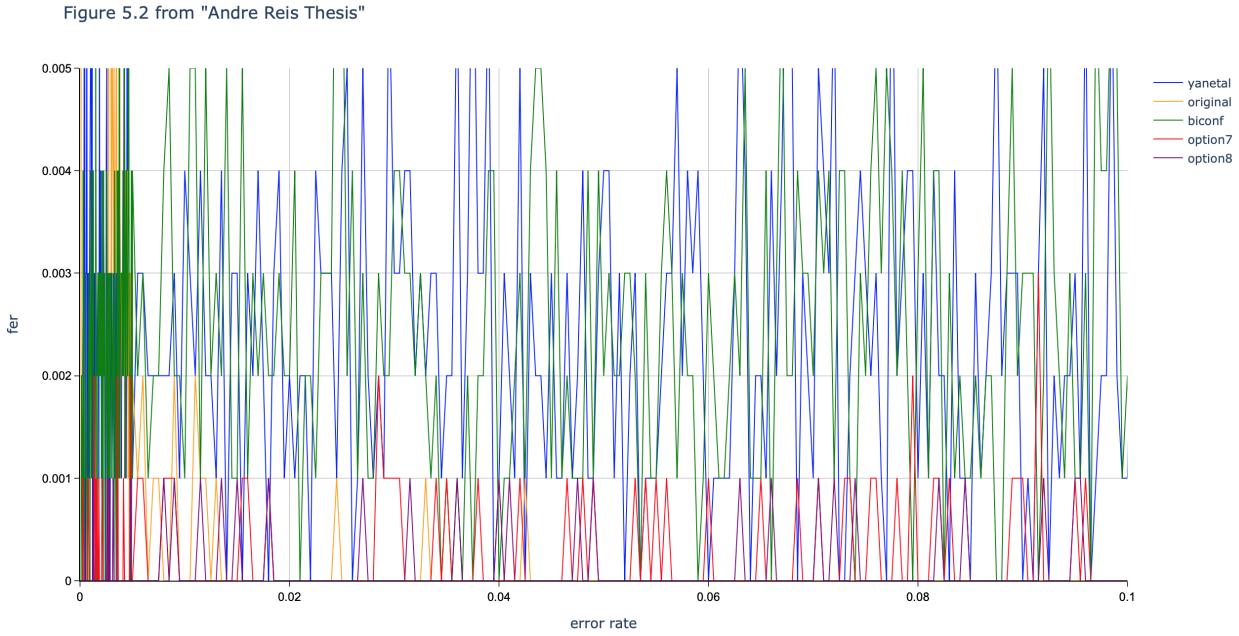


Figure 5.2: First experiment frame error rate by error rate on keys with 10000 bits

Reproduced figure from this code:



3.2.3 Figure 5.3

Original figure in thesis:

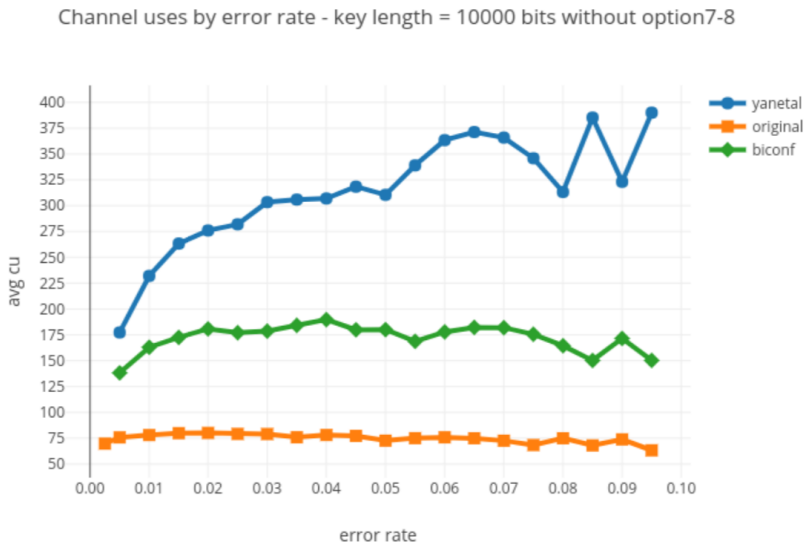
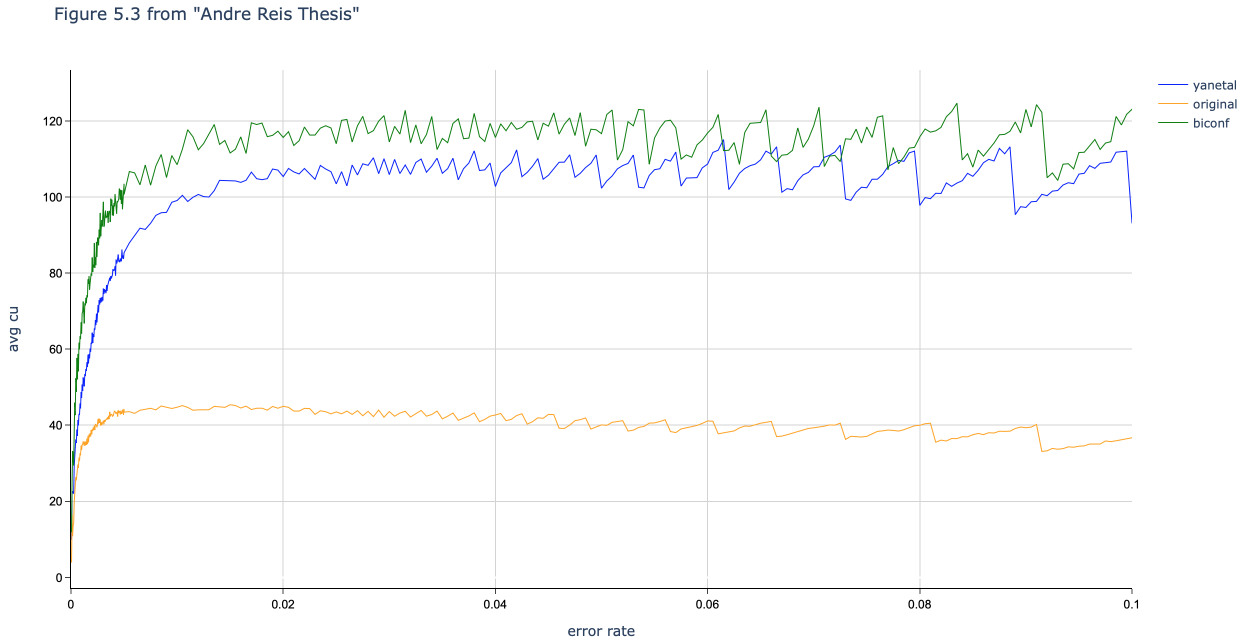


Figure 5.3: First experiment number of channel uses by error rate on keys with 10000 bits

Reproduced figure from this code:



3.2.4 Figure 5.4

Original figure in thesis:

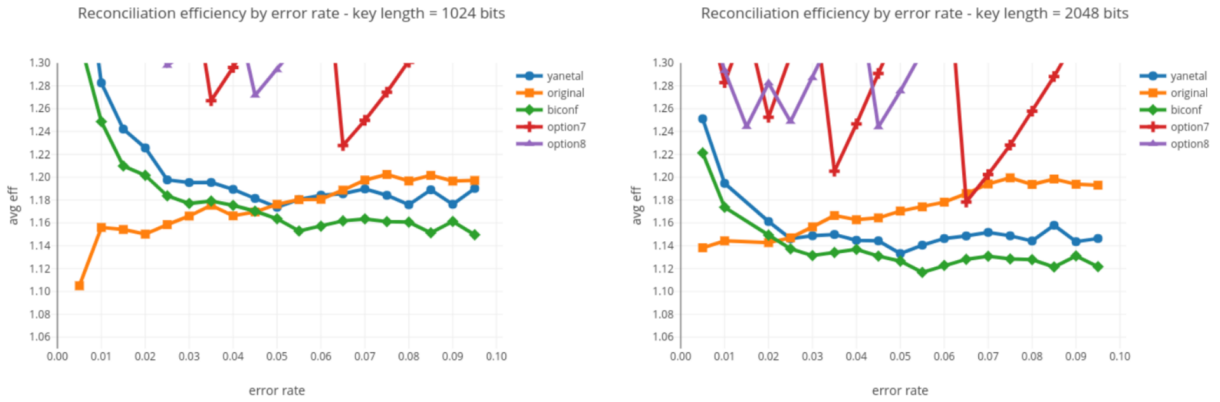


Figure 5.4: Reconciliation efficiency by error rate for keys with 1024 and 2048 bits

This figure is not (yet) reproduced by the code.

3.2.5 Figure 5.5

Original figure in thesis:

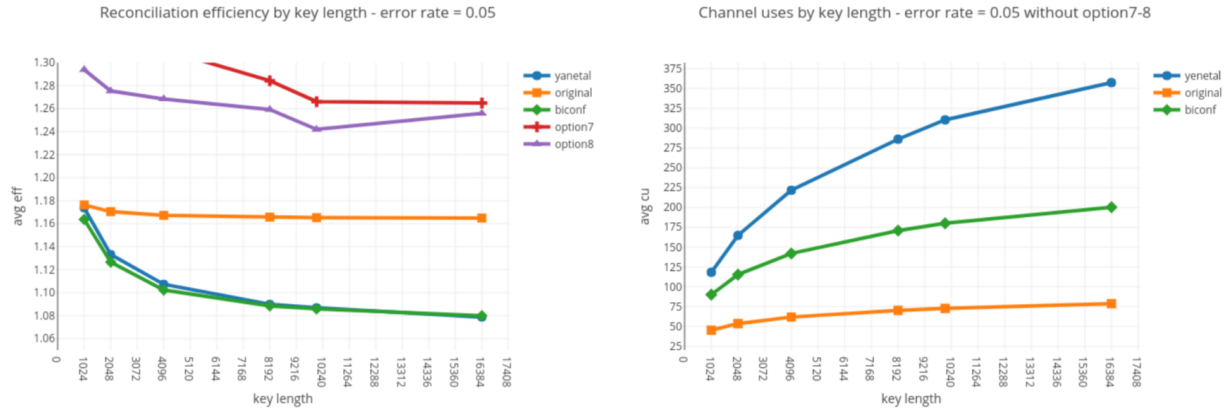


Figure 5.5: Reconciliation efficiency and channel uses by key length

Reproduced figure from this code:

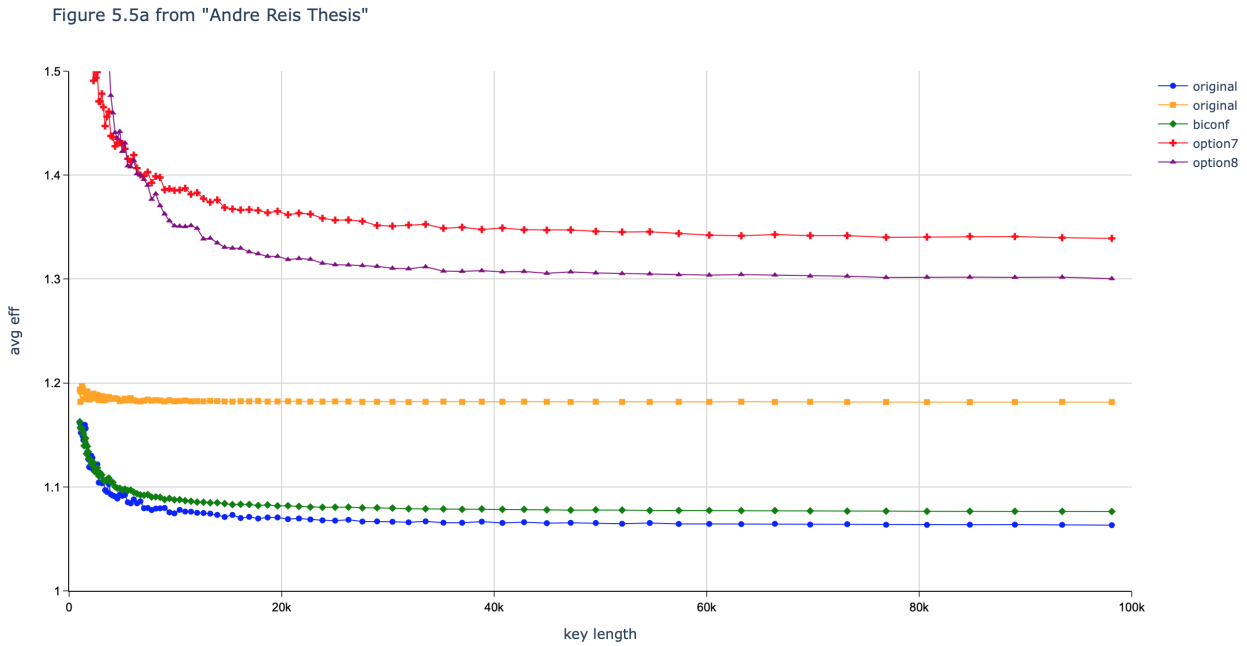


Figure 5.5b is not (yet) reproduced by the code.

3.2.6 Figure 5.6

Original figure in thesis:

Figure 5.6: Reconciliation efficiency by error rate on keys with 16384 bits

This figure is not (yet) reproduced by the code.

3.2.7 Figure 5.7

Original figure in thesis:

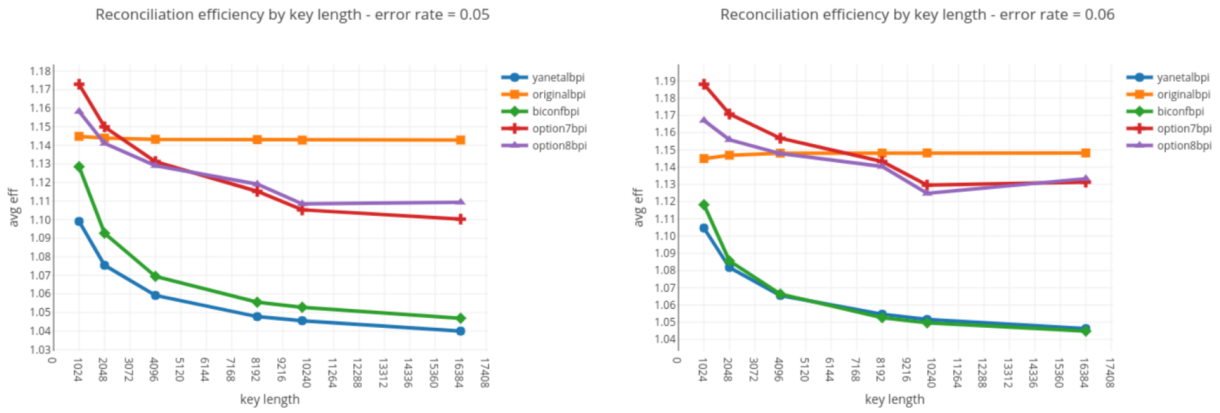


Figure 5.7: Reconciliation efficiency by key length with 5% error rate and 6% error rate

This figure is not (yet) reproduced by the code.

3.2.8 Figure 5.8

Original figure in thesis:

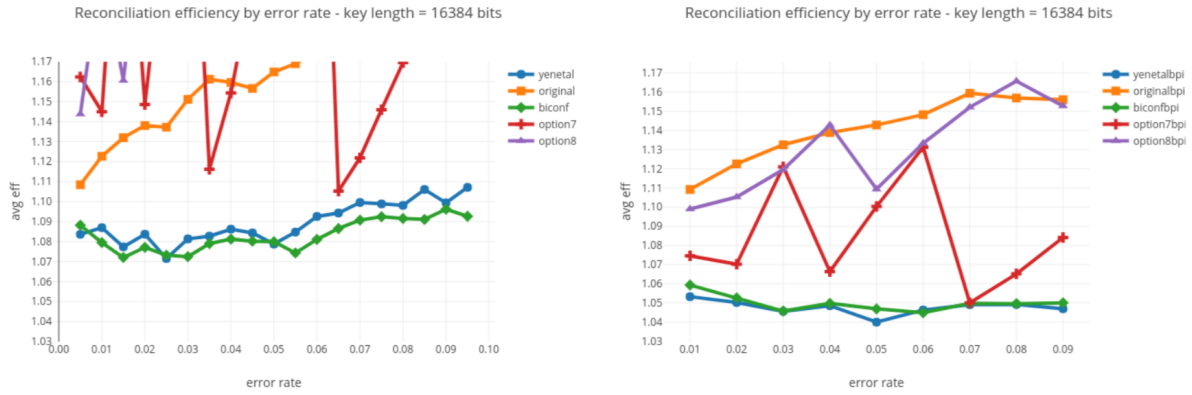


Figure 5.8: Reconciliation efficiency of the first experiment (on the left) and second (on the right)

This figure is not (yet) reproduced by the code.

3.2.9 Figure 5.9

Original figure in thesis:

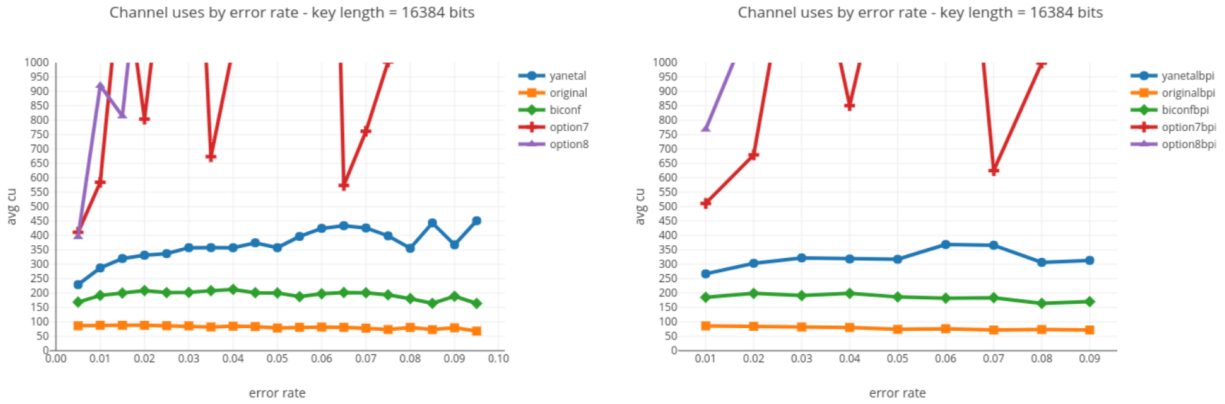


Figure 5.9: Channel uses of the first experiment (on the left) and second (on the right)

This figure is not (yet) reproduced by the code.

3.2.10 Figure 5.10

Original figure in thesis:

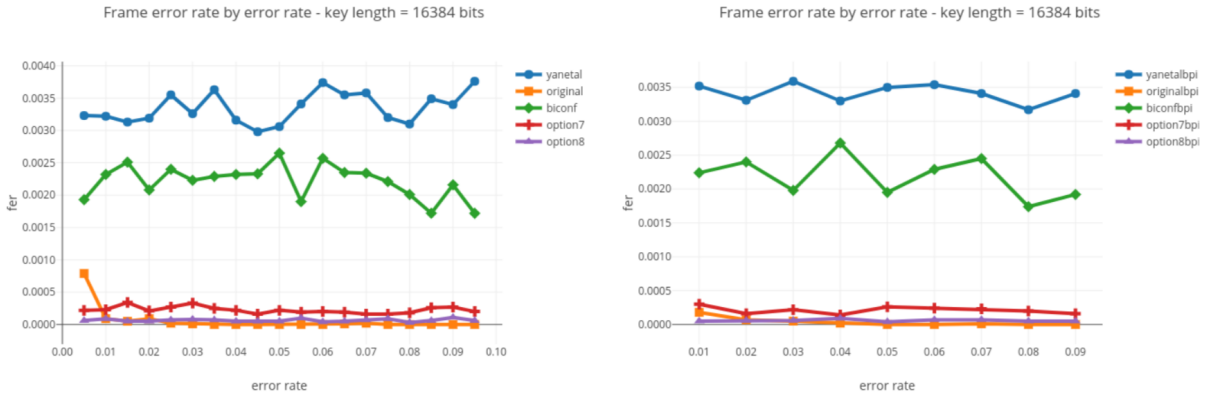


Figure 5.10: Frame error rate of the first experiment (on the left) and second (on the right)

This figure is not (yet) reproduced by the code.

CHAPTER

FOUR

CONCLUSIONS FROM COMPARISON OF RESULTS WITH LITERATURE.

Here we summarize some of the differences that we observed in the reproduced figures as compared to the literature.

4.1 Unrealistic efficiency.

My cascade-python code produced "efficiency vs bit error rate" graphs that matched the results in the literature very well.

However, I could only obtain a good match between my results and the literature results if I used an extremely unrealistic definition of efficiency (this is called "unrealistic_efficiency" in the code).

The "spirit" of the definition of "reconciliation efficiency" it to compare the actual number of bits exchanges between Alice and Bob during the reconciliation protocol and compare it with the theoretical minimum number of bits.

For example, if the theoretical minimum is 200 bits and we actually exchanged 220 bits, then the reconciliation efficiency is 220/200 = 1.10, meaning that we used 10% more bits than the theoretical minimum.

Every time Bob asks Alice to compute and reveal the correct parity for some block, some bits are exchanged. To compute the reconciliation efficiency we need to add up all these exchanged bits and compare the total number of exchanged bits with the theoretical minimum. This is what I call the "realistic_efficiency" in my code. However, if I do this, I end up with efficiencies that are way worse by several orders of magnitude than the ones reported in the literature.

To accurately reproduce the results reported in the literature I had to make a very unrealistic assumption: every time Bob asks Alice to compute and reveal the correct parity of a block, I count that as a single bit of information exchanged. When I do that, my results and the results from the literature match almost perfectly.

Clearly, this is not realistic. Bob asking Alice for the correct parity actually uses much more than one bit: Bob has to identify the block and the shuffling order which takes way more than one bit. Hence, I call this "unrealistic_efficiency" in my code.

4.2 Less detail.

The graphs in the "demystifying the Information Reconciliation Protocol Cascade" paper are much more detailed because they executed 10,000 (or sometimes even 100,000 or 1,000,000) runs per data point, whereas we only ran 1,000 runs per data point. Our Python implementation was is not fast enough to execute more than 1,000 runs per data point. It already took us about 5 days to run all the experiments on 2xlarge m5 instance on AWS with just 1,000 runs per data point. We are in the process of re-implementing Cascade in C++ which will hopefully allow for more runs per data point.

4.3 Standard deviation.

The graphs in the original papers do not have any indication of the standard deviation (i.e. no error bars). We have that information for all experiments, although we don't show it in all graphs - we omit it if it would make the graph too noisy.

4.4 Differences in the detailed shape of the channel use graph.

In many respects, most of my reproduced channel use figures match the figures in the original literature quite well. The overall shape matches quite well. The numerical value matches quite well. And the appearance of saw-tooth patterns matches quite well.

In some other aspects, however, there are some striking differences between the original and reproduced channel use figures as well.

In some of the figures in the original literature (e.g. figure 2 in the demystifying paper) the channel uses graph clearly slopes down as the error rate increases. In my reproduced figures, this downward slope is missing.

In other figures in the original literature (e.g. the black and green graphs in figure 9 in the demystifying paper) we see a "wave" pattern on top of the "saw tooth" pattern. This "wave" pattern is missing in my reproduced graphs.

4.5 Channel use graph for Cascade opt. (2) is different.

The original channel use graph for algorithm "Cascade opt. (2)" is quite different from the reproduced graph: the original values are higher and have much more wildly swinging saw-teeth.

CHAPTER

FIVE

FURTHER READING.

5.1 Papers.

Demystifying the Information Reconciliation Protocol Cascade. *Jesus Martinez-Mateo, Christoph Pacher, Momtchil Peev, Alex Ciurana, and Vicente Martin.* arXiv:1407.3257 [quant-ph], Jul 2014.

Towards an Optimal Implementation of Cascade. *Jesús Martínez Mateo, Christoph Pacher, Momtchil Peev, Alex Ciurana Aguilar, Vicente Martín Ayuso.* 2014.

Secret-Key Reconciliation by Public Discussion *Charles H. Bennett and Gilles Brassard.* Proceedings of IEEE International Conference on Computers,Systems and Signal Processing, pages 175–179, Sep 1984.

An analysis of error reconciliation protocols used in Quantum Key Distribution systems. *James S Johnson, Michael R Grimaila, Jeffrey W. Humphries, and Gerald B Baumgartner.* The Journal of Defense Modeling & Simulation 12(3):217-227, Jul 2015.

High Performance Information Reconciliation for QKD with CASCADE. *Thomas Brochmann Pedersen and Mustafa Toyran.* arXiv:1307.7829 [quant-ph], Jul 2013.

Key Reconciliation Techniques in Quantum Key Distribution. *Al-Janabi, Sufyan & Rabiee, Ruqaya.* 2011.

Information Reconciliation Protocol in Quantum Key Distribution System. *Hao Yan, Tienan Ren, Xiang Peng, Xiaxiang Lin, Wei Jiang, Tian Liu, and Hong Guo.* Proceedings - 4th International Conference on Natural Computation, ICNC 2008, 2008.

A Probabilistic Analysis of BINARY and CASCADE. *Ruth II-Yung Ng.* 2014.

5.2 Thesis.

Quantum Key Distribution Post Processing - A study on the Information Reconciliation Cascade Protocol. *Andre Reis.* Master's Thesis, Faculdade de Engenharia da Universidade do Porto, Jul 2019.

On Experimental Quantum Communication and Cryptography. *Chris Erven.* PhD Thesis, University of Waterloo, 2012.

Quantum Key Distribution Data Post-Processing with Limited Resources: Towards Satellite-Based Quantum Communication. *Nikolay Gigov.* Master's Thesis, University of Waterloo, 2013.

An Empirical Analysis of the Cascade Secret Key Reconciliation Protocol for Quantum Key Distribution. *Timothy I. Calver, Captain, USAF.* Master's Thesis, Department of the Air Force, Air University, Air Force Institute of Technology.

5.3 Implementations.

GitHub repository andrebreis/cascade-study.

GitHub repository mdskrzypczyk/QChat.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search