HIGH THROUGHPUT PHOTON MATCHING ALGORITHM FOR QKD AND ITS RTL IMPLEMENTATION

A Thesis

by

Seçkin İpek

Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in
Computer Science

Özyeğin University
December 2022

HIGH THROUGHPUT PHOTON MATCHING ALGORITHM FOR QKD AND ITS RTL IMPLEMENTATION

Advisor: Prof. H. Fatih Uğurdağ, Özyeğin University
Coadvisor: Prof. Sezer Gören Uğurdağ, Yeditepe University

Approved by:

Prof. H. Fatih Uğurdağ, Advisor
Dept. of Electrical and Electronics Eng.
Özyeğin University

Asst. Prof. Kadir Durak
Dept. of Electrical and Electronics Eng.
Özyeğin University

Asst. Prof. Tacha Serif
Dept. of Computer Eng.
Yeditepe University

Date Approved: 29 December 2022

To those who were honest and fair with me

ABSTRACT

While means to communicate increase with the technological advancements, at the same time the methods to steal information during communication also advance. With these advancements, the need for a method to communicate without anyone stealing the shared information increases. Most of the communication methods used today are classical information channels, which involve sending data that can be very easily eavesdropped. On the other hand, quantum information channels use the principles of quantum mechanics that make the information secure even if a third party tries to listen using the best technology. One of the quantum communication methods is Quantum Key Distribution (QKD). With this method, an entangled photon source generates a few million photon pairs in a second and sends the photons of each pair to two terminals. Afterwards, these two terminals match the photons they received. This thesis explains the development process of a high throughput matching algorithm and its implementation in Python and Verilog. The algorithm developed has 3 parts, namely, finding entangled photon starting time for each terminal, entangled photon start time difference for the two terminals, and matching the photons. Our algorithm's Verilog implementation can match 30 million photon pairs per second and can be easily tuned for different protocols.

ÖZETÇE

Teknolojik gelişmelerle iletişim kurma araçları artarken aynı zamanda iletişim sırasında bilgi çalma yöntemleri de gelişmektedir. Bu gelişmelerle birlikte, paylaşılan bilgileri kimse çalmadan iletişim kurmak için bir yönteme olan ihtiyaç artıyor. Günümüzde kullanılan iletişim yöntemlerinin çoğu, kolayca dinlenebilecek verilerin gönderilmesini içeren klasik bilgi kanallarıdır. Öte yandan, kuantum bilgi kanalları, üçüncü bir taraf en iyi teknolojiyi kullanarak dinlemeye çalışsa bile bilgiyi güvenli kılan kuantum mekaniği ilkelerini kullanır. Kuantum iletişim yöntemlerinden biri de Kuantum Anahtar Dağıtımıdır. Bu yöntemle bir dolaşık foton kaynağı saniyede birkaç milyon foton çifti üretir ve her bir çiftin fotonlarını iki terminale gönderir. Daha sonra bu iki terminal aldıkları fotonları eşleştirir. Bu tez, yüksek hızlı bir eşleştirme algoritmasının geliştirme sürecini ve Python ve Verilog'daki gereklemesini açıklamaktadır. Geliştirilen algoritma her terminal için dolaşık foton başlama zamanını, iki terminal için dolaşık foton başlama zamanı farkını bulma ve fotonları eşleştirme olmak üzere 3 bölümden oluşmaktadır. Algoritmamızın Verilog uygulaması, saniyede 30 milyon foton çiftiyle eşleşebilir ve farklı protokoller için kolayca ayarlanabilir.

ACKNOWLEDGEMENTS

I would like to thank Professors H. Fatih Uğurdağ and Sezer Gören Uğurdağ for allowing me to work on this project and for guiding me in the right direction through the whole project.

Additionally, I would like to thank my friend M.P. for the support she has given me during the course of this project.

The work presented here is supported under the TÜBİTAK ARDEB-1003 subproject numbered 118E993, where Prof. Sezer Gören Uğurdağ is the principal investigator. On the other hand, Asst. Prof. Kadir Durak is the principal investigator of the main TÜBİTAK ARDEB-1003 project numbered 118E991.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Communications that take place through classical communication methods are prone to be intercepted by third parties. Despite technological advances to devise a completely secure classical communication method, once a third party intercepts a message, it is only a matter of time for the third party to decrypt the message, and if a quantum computer is involved, it would take the third party only seconds to decrypt messages.

Unlike the classical communication methods, quantum communication methods utilize the principles of quantum mechanics such as uncertainty principle, no-cloning theorem, and monogamy of entanglement, which result in a safer communication between two terminals [2]. Quantum Key Distribution (QKD), one of the quantum communication methods, utilizes the mentioned principles to distribute an encryption key to two terminals. Receiving a key with the QKD method enables two parties to communicate safely using this key, regardless of what kind of technological advancements the third party possesses.

1.1 A QKD System

A QKD system requires an Entangled Photon Generator (EPG) and two terminals that receive photons from the EPG, which is presumed to be between the two terminals [3]. EPG creates photon pairs and sends one photon of each pair to one of the terminals and the other photon to the other terminal as seen in figure 1. When receiving a photon, terminals detect the photon in their one of many detectors and depending on which detector they detected the photon they mark the photon with their arrival times, these marking operation is done by Time Stamp Units (TSUs).

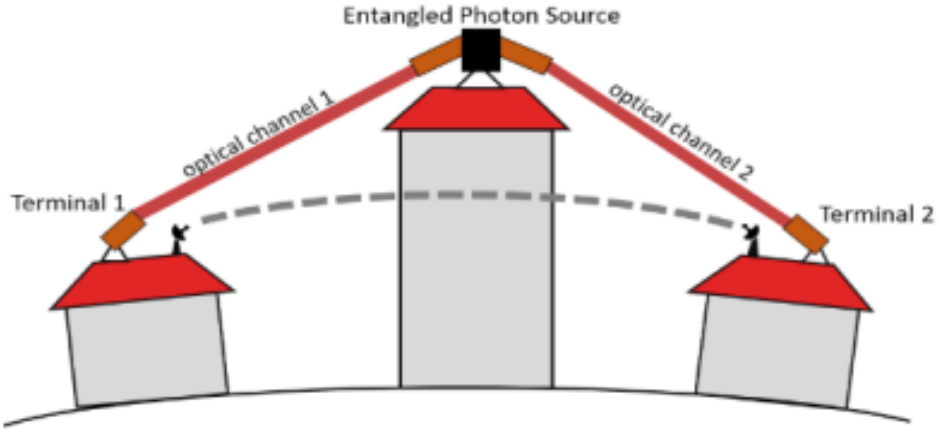


Figure 1: QKD system with a photon source and two terminals [1]

In a terminal, there can be 2n detectors and n bases. That is, a base consists of two detectors.

When photons are detected in a terminal, their information, which is their arrival time and detector number, is shared with the other terminal using classical communication methods, however, since these methods are non-secure, arrival times of photons are shared normally but their detector number is not shared. Barely enough information is shared to tell the other terminal which base the photon was detected. While terminals are receiving photons and exchanging the photon information with each other, the split up photons at the start are matched in each terminal. With each terminal having two bases, photons that arrived in base 1 of terminal 1 and 2 are matched with each other, and the photons that arrived in base 2 of terminal 1 and 2 are matched with each other. While this matching operation continues a key from each base is generated by each terminal. If the keys found at the end of matching is not satisfactory the keys get discarded and a new matching and key generation process is started over.

1.2 Problem Statement

The main challenge of the design is to be able to finish matching all the photons at the time a group of 5 million photons arrives. In the case that the found key is decided that it should not be used, a new group of 5 million photons is required to find another key, when this happens and matching takes longer time than the 1 second 5 million photon takes to arrive, each key finding finishes with more delay and more area is required to store the photons that are received and waiting to be matched.

The entangled photon pairs take a certain time to reach to each terminal, due to the distance differences between each terminal and EPS, the entangled photons arrive to each terminal with a different latency. In order to match the entangled photons this difference in latency needs to be found.

Additionally, while the entangled photons are being received, the detectors in the system can also detect photons from environment. The detection of the ambient photons make it harder to find the latency difference.

1.3 Related Work

As the need for a secure communication increases the interest in the quantum communication methods increases as well. Due to the lack of safety in the classical communication methods this problem can only be solved with advancing the quantum communication methods. With this field being relatively new, there only few researches as a result. With the main focus of our implementation being the matching speed, some of the other implementations and their matching speeds are analyzed.

An implementation done by M. Lucamarini [4] for QKD with BB84 protocol has a key generation speed of up to 2.2 million photon per second at optimum distance, reduced as the distance increasing.

An implementation done by Tian Zhong [5] for QKD with HDQKD protocol achieved key rate of 2.7 million bits per second.

An implementation done by Catherine Lee [6] for QKD with DO-QKD protocol achieved 23 million bits per second key generation rate in a back to back configuration, reduced to 5.3 million bits per second key generation rate with 43 km deployed fiber setup.

An implementation done by Mikołaj Lasota [7], when realized on a QKD system with BB84 protocol achieved a maximum key generation rate of 10 million bits per second at a distance of 120km, reduced exponentially as the distance increases to 280 km.

CHAPTER II

SOFTWARE IMPLEMENTATION

The initial part of the project was done in Python. Because python is a language which is easier to alter and optimize than Verilog, the necessary algorithms for the project first written in Python.

2.1 Time Stamp Sequence Generator

In order to design the algorithms the most important thing is to generate our own entangled photon pairs. For this purpose a time stamp sequence generator (TSSgen) is designed. The initial design of the TSSgen has only few features. In the real QKD system in one second five million entangled photon pairs arrives at the terminals, meaning on average once every two hundred nanosecond a photon pair arrives. With this information entangled photon pairs are generated with two consecutive pairs having a time delay of 50ns to 350ns between each other, on average each consecutive photon pairs having 200ns time delay between each other. The delays are randomly selected in the given range with a random seed.

In table 1 example entangled photons generated by TSSgen can be seen. When initially created these photons have their time stamp information, which detector detected the terminal 1 part of each pair, and which detector detected the terminal 2 part of each pair. With the entangled photons generated next step is to replicate the latency of the generated photons between generation and arriving to the detectors. Since two detectors can have different latencies, each photon pair is split and the part of the pairs that represent the photons that would arrive to terminal 1 are grouped and called as TSS1, and the part of the pairs that represent the photons that would arrive to terminal 2 are grouped and called as TSS2, TSS data after split can be

Table 1: Example entangled photons generated by TSSgen

photonnumber	*photondata*
1	[280, 1, 1]
2	[612, 1, 1]
3	[961, 1, 1]
4	[1039, 1, 1]
5	[1127, 2, 2]
6	[1473, 2, 2]
7	[1631, 1, 1]
8	[1971, 1, 1]
9	[2131, 2, 2]
10	[2428, 1, 1]

seen in table 2. With the photons of terminal 1 and terminal 2 are grouped, all time stamps of TSS1 photons are shifted by amount and all the time stamps of TSS2 photons are shifted by a different amount. The amount to shift both TSS1 and TSS2 data are randomly generated by selecting a random seed, TSS data after shift can be seen in table 3.

Table 2: Entangled photons being split

photonnumber	$TSS1$	$TSS2$
1	[280, 1]	[280, 1]
2	[612, 1]	[612, 1]
3	[961, 1]	[961, 1]
4	[1039, 1]	[1039, 1]
5	[1127, 2]	[1127, 2]
6	[1473, 2]	[1473, 2]
7	[1631, 1]	[1631, 1]
8	[1971, 1]	[1971, 1]
9	[2131, 2]	[2131, 2]
10	[2428, 1]	[2428, 1]

Next step is to add ambient photons to the system. In the real QKD system no matter how much the detectors are improved to only detect the entangled photons that are generated from the photon source, it is still possible for detectors to detect

Table 3: Entangled photons after being shifted

photonnumber	$TSS1$	$TSS2$
1	[20280, 1]	[25280, 1]
2	[20612, 1]	[25612, 1]
3	[20961, 1]	[25961, 1]
4	[21039, 1]	[26039, 1]
5	[21127, 2]	[26127, 2]
6	[21473, 2]	[26473, 2]
7	[21631, 1]	[26631, 1]
8	[21971, 1]	[26971, 1]
9	[22131, 2]	[27131, 2]
10	[22428, 1]	[27428, 1]

random photons from the environment, which is referred as ambient photons. Ambient photons can be detected at any frequency but it is known that they received far less than entangled photons. With this knowledge and adjustable frequency is set for ambient photons and they are generated to be received at 1/10th frequency of the entangled photons, resulting them to be generated in a range of 1000ns and 7000ns similar to the entangled photon generation. The difference in their generation is 2 different ambient photon series are generated for 2 terminals with different seeds and only 1 part from each of their photon pairs are used. The reason they are generated with different seeds is in order not to make each ambient photon in a terminal to have a pair on the other terminal, since the ambient photons are not pairs, they are only individual photons detected from environment. Example ambient photons can be seen in table 4. After ambient photons generated for each terminal, they are merged with the entangled photons of their respective terminals and sorted according to their time stamps. The TSS data with entangled and ambient photons merged can be seen in table 5.

The TSSgen so far generated photons with perfect detection rate, but in the real QKD system it is not possible to detect all of the photons without any miss. In order

Table 4: Example ambient photons generated by TSSgen

photonnumber	*terminal*1*ambient*	*terminal*2*ambient*
1	[3074, 1]	[5799, 1]
2	[9930, 1]	[9759, 1]
3	[15876, 1]	[16533, 2]
4	[22577, 1]	[22258, 1]
5	[25915, 2]	[23703, 2]
6	[28363, 2]	[28882, 2]
7	[29646, 1]	[33961, 2]
8	[35876, 1]	[38237, 1]
9	[40427, 2]	[39690, 2]

to replicate the missed photons some of the received photons are randomly deleted from TSS1 and TSS2, this random deletion operation is done with different seeds for TSS1 and TSS2 so that different photons from different orders would get deleted from both terminals. With the random deletion operation completed too, TSS1 and TSS2 data are ready to be matched. TSS data after deletion can be seen in table 6.

2.2 Posedge and Correlation Stages

The first implementation is a single process implementation that finds the shift amount and matches all the photons according to this shift amount. For shift amount to be calculated and photons to be matched, first it is required to implement a Time Stamp Sequence Generator (TSSgen) that generates random TSS data similar to the TSS data the photon detectors would generate in the real QKD system. This TSSgen is also needed to generate many data in short amount of time to use for verification of our implementations.

The first attempt to do matching for all TSS data was to correlate all the data, find a time shift, using the found time shift match all the photons that were generated from TSSgen. This method was used with both time domain and frequency domain correlations, but the time they required to calculate the shift was too high even with

Table 5: Entangled photons merged with ambient photons

$photonnumber$	$TSS1$	$TSS2$
1	[3074, 1]	[5799, 1]
2	[9930, 2]	[9759, 1]
3	[15876, 1]	[16533, 2]
4	[20280, 1]	[22258, 1]
5	[20612, 1]	[23703, 2]
6	[20961, 1]	[25280, 1]
7	[21039, 1]	[25612, 1]
8	[21127, 2]	[25961, 1]
9	[21473, 2]	[26039, 1]
10	[21631, 1]	[26127, 2]
11	[21971, 1]	[26473, 2]
12	[22131, 2]	[26631, 1]
13	[22428, 1]	[26971, 1]
14	[22577, 1]	[27131, 2]
15	[25915, 1]	[27428, 1]
16	[28363, 1]	[28882, 2]
17	[29646, 2]	[33961, 2]
18	[35876, 1]	[38237, 1]
19	[40427, 1]	[39690, 2]

small size TSS data. Due to the time problem another method to find the shift was needed, and in order to solve this issue sparse time dome correlation method was invented.

The TSS1 and TSS2 data has a static time shift between each other, if time difference between each possible photon pair was calculated and this time difference was incremented on a histogram, the address on the histogram with the biggest number should be the time shift between the entangled photon pairs. The TSS1 and TSS2 data are consist of ambient photons as well, but since they do not arrive to the terminal 1 and terminal 2 in pairs, in histogram they do not increment any specific address, but they increment random addresses. Additionally since ambient photons are not received as frequently as entangled photons, even in the low chance that they increment a specific address other than the actual shift address, they are unable to

Table 6: TSS1 and TSS2 after 10 percent random photon deletion

$photonnumber$	$TSS1$	$TSS2$
1	[3074, 1]	[5799, 1]
2	[9930, 2]	[9759, 1]
3	[15876, 1]	[16533, 2]
4	[20280, 1]	[22258, 1]
5	[20961, 1]	[23703, 2]
6	[21039, 1]	[25280, 1]
7	[21127, 2]	[25612, 1]
8	[21473, 2]	[25961, 1]
9	[21631, 1]	[26039, 1]
10	[21971, 1]	[26127, 2]
11	[22131, 2]	[26473, 2]
12	[22428, 1]	[26631, 1]
13	[22577, 1]	[26971, 1]
14	[25915, 1]	[27131, 2]
15	[28363, 1]	[28882, 2]
16	[29646, 2]	[33961, 2]
17	[35876, 1]	[38237, 1]
18	[40427, 1]	[39690, 2]

increment it too much.

The SparseTDC time complexity is reliant on the amount of data in the TSS1 and TSS2, while time domain correlation and frequency domain correlation time complexities rely on the maximum and minimum time in the TSS1 and TSS2. Due to this, when finding shift with large amount of data frequency domain correlation is faster, but when finding shift in smaller sized data SparseTDC is much faster. The current method uses all the TSS1 and TSS2 data in either correlation to find the time shift because it is not possible to differentiate ambient photons from entangled photons, so that we could filter out the ambient photons from entangled photons and find shift using only entangled photons in correlation, resulting in a faster correlation. However, instead of differentiating the ambient and entangled photons, it is possible to find the approximate time entangled photons start to arrive and stop arriving.

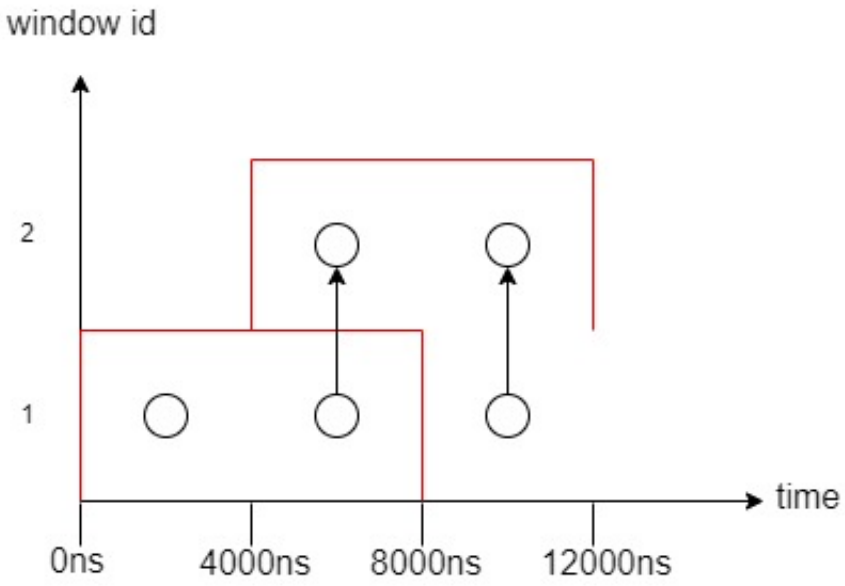


Figure 2: Sliding windows and the photons inside

2.2.1 Posedge Method

Entangled photons arrive to the detectors much more frequently than the ambient photons. With this information, if the amount of photons is checked in a time period, there should be few photons until entangled photons start to arrive, many photons after entangled photons start to arrive and until entangled photons finish arriving, and very few photons detected after entangled photons finish arriving. Knowing this, a sliding time window is used to check how many photons exist in a time period. For our system a sliding window with the width of 8000ns is used. The first window checks how many photons are detected between 0ns and 8000ns, if a photon with a time stamp higher than 8000ns arrives, the sliding window checks how many photons have been detected in its time frame, if enough photons are detected it is decided that entangled photons are started to arrive somewhere around the start of the sliding window. If sliding window did not detect enough photons in its time frame it shifts by 4000ns and until a new photons arrives with time stamp larger than its upper bound, then again checks how many photons have been detected in its time frame.

Figure 2 shows how the windows are slid and how the photons inside them are managed. With window 1 having a time range of 0ns and 8000ns, the first two photons that arrived is recorded in this windows. When the third photon arrives, its time stamp is greater than the upper bound of the first window, due to this, a new window created with upper and lower bounds shifted by 4000ns. The new window has a time range of 4000ns and 12000ns. While the third photons is added to the second window, the second photon that has arrived to the terminal is also in the time range of the second window. So the second photon is also added to the second window and the second window possesses two photons in total.

In order to decide detecting how many photons in a sliding window is enough, the arrival frequency of entangled photons are used. With 5 million photon arriving in 1 second, on average a terminal is receiving and entangled photon once every 200ns. With this average arrival frequency, in a sliding window with the width of 8000ns, on average 40 photon should be detected in a sliding window to decide that entangled photons have started to arrive. Although the 40 photon is the average, if the threshold to decide that the entangled photons have started to arrive were to be chosen too close to 40, even if it is less than 40 it is possible to not detect the starting time window if the entangled photons coincidentally arrive less frequently in that time window. Additionally if a very low threshold to be chosen, it is possible to have a time where ambient photons coincidentally arrive more frequently than average and result in the time window to assume that the entangled photons have started to arrive. Due to these issues, a threshold of 20 is selected, which is a number not too close to the 40 and not too low at the same time. Upon testing it was seen that the selected threshold was giving better results that the other possible threshold numbers.

With the starting time frame found, the window is continued to be slid with the same way. The sliding window after finding the starting frame searches for the ending frame. When ending frame is found too, the approximate times entangled started to

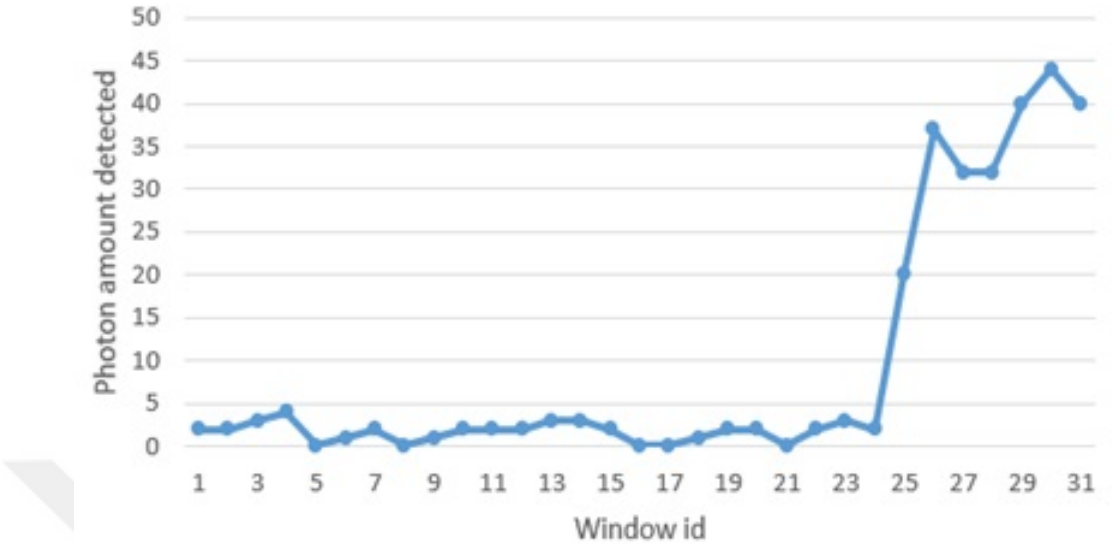


Figure 3: Sliding window id s and the detected photon amount in each window

arrive and finished to arrive are found. Because this method is not perfectly accurate, for starting time the lower bound of the window that was prior to the found starting window is used. By taking the starting time one time window earlier, in the case that there were any entangled photons in the prior window but not enough to pass the threshold, they will not missed. Similarly the ending time is taken as the upper bound of the first time window that has less photon than the threshold after starting window is found is taken as the ending time to not lose any photons near the ending time.

In figure 3 it is possible to see how many photons detected in each sliding window in an example data. In this example sliding windows possess less than 5 photons until the 25th window, with the 25th window having 20 photons. With our threshold being 20, the 25th window is assumed to be the starting time window in the terminal. When calculated, the starting time window have a range of 0ns and 8000ns, and the windows after that is slid by 4000ns, so the 25th window should have a range of 96000ns and 104000ns. For this example, the entangled photons were starting to arrive to the terminal with 100000ns latency, so the latency is found correctly in the

first window satisfying the threshold. Due to how the photon density increase in the sliding windows look on the graph, this sliding window method will be referred to as Posedge method.

With Posedge method now it is possible to remove the ambient photons before and after the entangled photons started and finished to arrive. Since the SparseTDC had a time complexity with the amount of photons in the system, removing some of the ambient photons result in a faster correlation time. However, still the speed of the correlation is much slower than the required speed.

So far, because it was not possible to differentiate entangled and ambient photons, the correlation was using either all the photons in the system, or the photons in the system after some of the ambient photons at the start and finish are removed. However, since it is possible to know the approximate time entangled photons started to arrive, as an alternative, it is possible to do the correlation with using only some of the photons that are detected after the found approximate starting time. When doing the correlation using all the photons between found starting and finish times, a mix of ambient and entangled photons are used but because ambient photons do not arrive in pairs and they arrive less frequently than the entangled photons, a shift can be found. Similarly, when only few photons after the found approximate starting time are used for correlation, it is possible to find a shift amount by using far less amount of photons in comparison.

With this new method a problem is to decide how many photons should be used starting from the approximate starting time. As explained earlier, the starting time taken from the lower bound of the previous sliding window to not miss any entangled photons that might be in the previous window but not in last window. Due to this, if too few photons were used for the correlation, it is possible that the correlation would mainly have ambient photons and too few entangled photons, and result in a higher chance of getting wrong result.

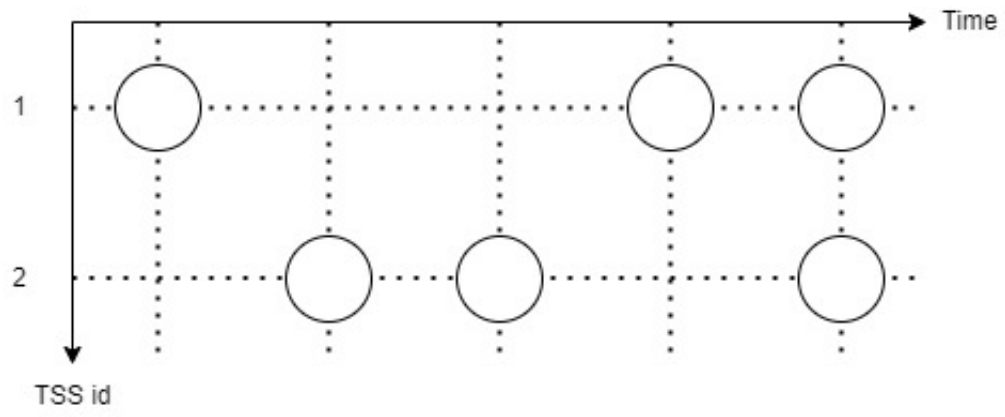


Figure 4: Example photon time comparison with zero shift difference

With each sliding window having on average 40 entangled photons, the first 40 photons are taken after the found approximate starting time and they are used in correlation. After correlation is completed a time shift is found and with this information it is possible to match entangled photons to find a key.

2.2.2 Correlation

The initial matching process is simple, the time stamps of the first photons from TSS1 and TSS2 is compared. If their time difference is greater than the found shift, the photon from TSS1 is deleted and the next photon from the TSS1 is compared with the TSS2 photon. Similarly, if their time difference is less than the found shift, the photon from TSS2 is deleted and the next photon from the TSS2 is compared with the TSS1 photon. The reason these photons are deleted is because if the time difference between the photons is greater than time shift, comparing the TSS1 photon with next TSS2 photons will result in the difference being even greater than the previous difference. Similarly, if the time difference between the photons is less than time shift, comparing the TSS2 photon with next TSS1 photons will result in the difference being even lesser than the previous difference.

In figure 4 TSS1 and TSS2 both have 3 photons each, in this example there is no shift difference between the two terminals. When 1st photon of TSS1 and 1st photon

of TSS2 is compared and a shift difference is found, if the 2nd photon of TSS2 were to be compared the difference would be certainly larger than the previous one. That is why instead of deleting the TSS2 photon, TSS1 photon should be deleted. The photons not matching indicated one of the few possibilities:

- TSS1 photon is an ambient photon

- TSS2 photon is an ambient photon

- Both TSS1 and TSS2 photons are ambient photons

- TSS1 photon is an entangled photon but lost its pair, TSS2 photon can be either entangled or ambient photon

- TSS2 is an entangled photon and TSS1 photon is an ambient photon, if TSS1 photon is deleted the next photon from TSS1 might be the pair of the TSS2 photon

Due to the last two possibilities, the next TSS1 photon should be compared with the TSS2 photon to see if they are a pair. In this example 2nd TSS1 and 1st TSS2 will not match either, in this case because time difference between the two photons is less than the shift, TSS2 photon will be deleted and the next TSS2 photon is compared with the TSS1 photon. The 2nd TSS2 photon will not match with the 2nd TSS1 photon either, and because again the time difference between the two photons is less than the shift, TSS2 photon will be deleted and the next TSS2 photon is compared with the TSS1 photon. When the 2nd TSS1 and 3rd TSS2 photons are compared their time difference will be greater than the shift so TSS1 photon will be deleted and next TSS1 photon will be compared with TSS2. On this comparison 3rd TSS1 and 3rd TSS2 photons will have zero time difference, which is equal to the found time shift difference, they are decided to be a pair and their detector numbers are recorded to create a key. Because they matched, both 3rd TSS1 and 3rd TSS2 photons are deleted and the next TSS1 and TSS2 photons are compared. The matching procedure continues with this logic until no more TSS1 and TSS2 photons are left in the system. Step by step visualization of this matching can be seen in figure 5.

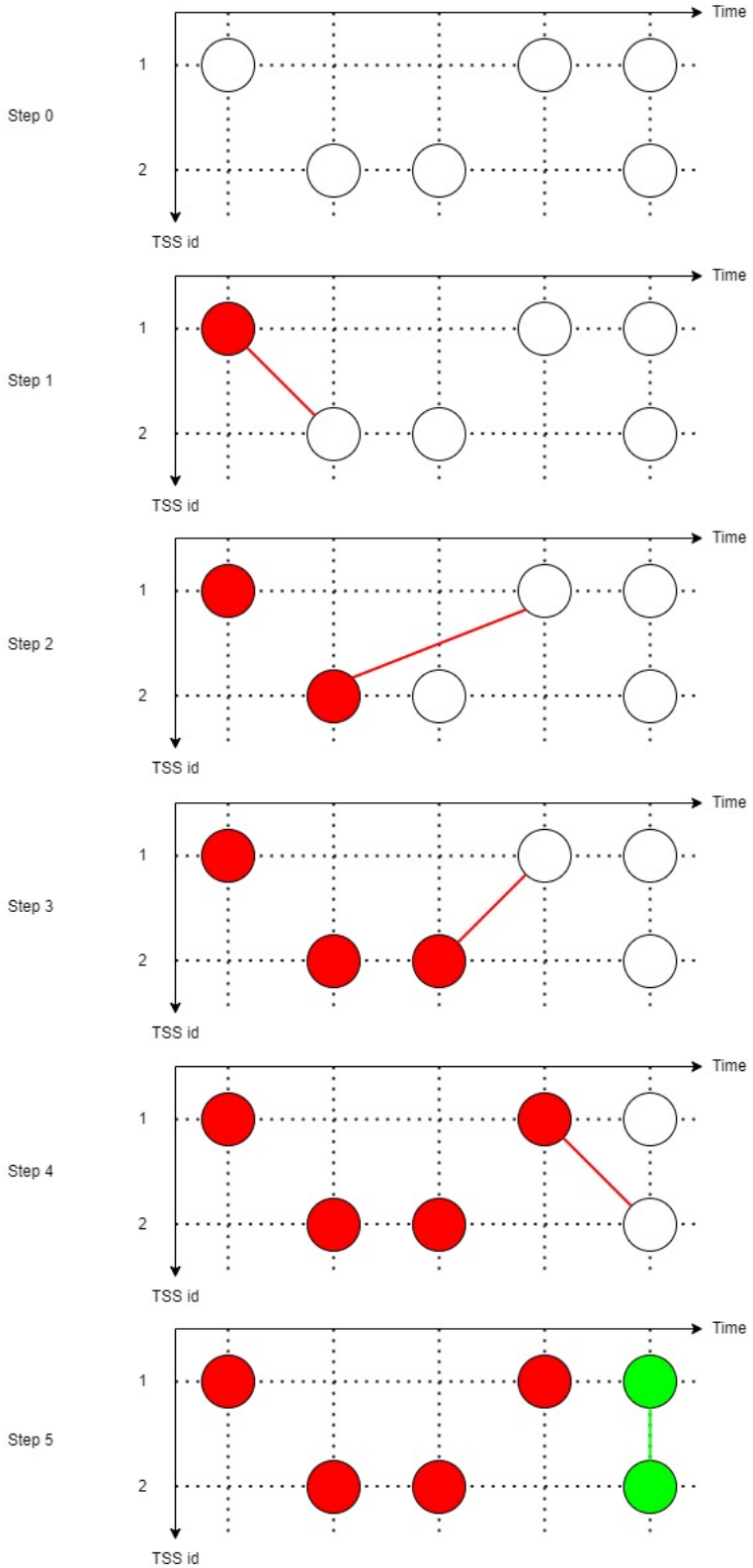


Figure 5: Step by step visualization of matching process

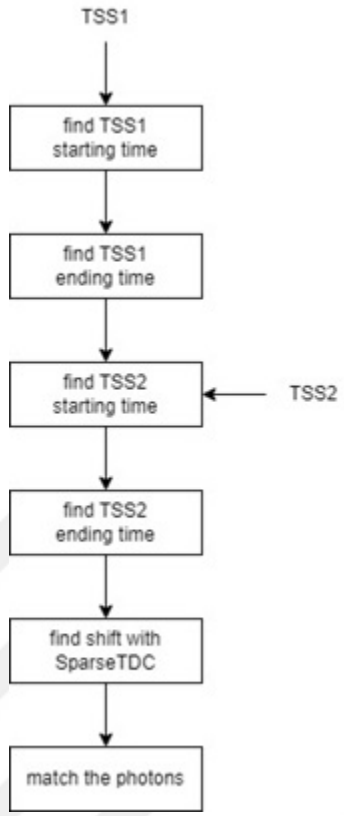


Figure 6: State machine of the linear system

With the method currently used the system uses a linear approach to matching the photons.

As seen in figure 6:

- The system first takes TSS1 photons

- Finds starting time of TSS1 photons

- Finds ending time of TSS1 photons

- Takes TSS2 photons

- Finds starting time of TSS2 photons

- Finds ending time of TSS2 photons

- Find shift with SparseTDC

- Match the photons

The system does many operations in a linear order and this results in a time loss

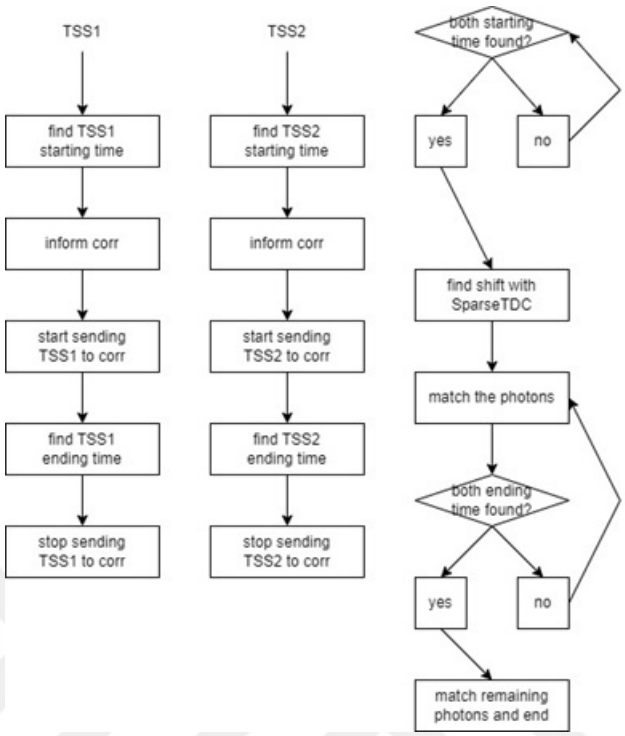


Figure 7: Posedge1, Posedge2 and Corr state machines

and additionally requires more area to be used to record all the photons from TSS1 and TSS2 until all of the photons are matched or deleted. As an alternative, many of these operations can be done in parallel to save time and area.

2.3 Matching Stage

In the earlier design, entangled photon start time of TSS1 and TSS2 were found in series, but there is no need to find one after the other, because TSS1 and TSS2 are not related to each other. Instead of doing these two operations in series it is possible to do them at the same time. The parallelization of the processes utilizes the multiprocessing library of the Python. Multiprocesses in Python is similar to the functions, but instead of calling a function and halting the main code until the function is completed, in multiprocessing a process is started in the main code and the main code continues. If there are more than one processes to be started, after the first one is started, the main code continues to the next lines and if another process

is to be started on those lines, those processes get started in parallel as well.

By using the multiprocessing library, at this stage of the implementation there are 3 processes working in parallel and their state machines can be seen in figure 7:

- Posedge1, finding the entangled photon starting and ending time of TSS1

- Posedge2, finding the entangled photon starting and ending time of TSS2

- Corr, finding the shift difference in TSS1 and TSS2, then match the photons and find the key

After Posedges find the entangled photon starting times of TSS1 and TSS2, they inform the Corr module that the starting times are found and start to send TSS data to Corr module.

In Python, public variables can be accessed by different function and they can be read or altered. In multiprocessing public variables can be read by the different processes, but when a public variable is read, a local copy of the variable is made and the changes are recorded locally, so other processes can't see the changes done to a public variable in other process.

In order to make the processes communicate with each other, queue library from multiprocessing library is used. Queue is basically a software FIFO, for Posedges to send TSS1 and TSS2 to Corr, they write their data to a queue each and Corr module reads from these FIFOs as long as they are not empty. For single bit like signal communications, Posedges uses queue as well, but instead of reading the data inside them, Corr only checks if they are empty or not. For example, when Posedge1 finds the entangled photon starting time for TSS1, Posedge1 writes a small data to a queue called Posedge1found connected to the Corr. Same operations is done in Posedge2 for TSS2, and connected to Corr with a queue called Posedge2found. Corr checks if Posedge1found and Posedge2found queues empty or not, and when both of them are not empty, Corr starts to receive data from TSS1 and TSS2 queues. The connection of the processes and FIFOs can be seen in figure 8.

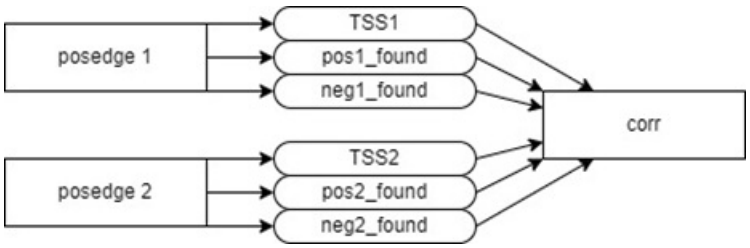


Figure 8: Processes and software FIFO connections

At this stage of the design, Posedge modules were reading TSS data from text files. However, for a better testing environment it was important for modules to read TSS data from over TCP/IP because reading text files is significantly faster than sending and receiving data over a TCP/IP communication, and this speed difference in how the TSS data is received were causing the data receiving part to be unrealistically fast, because in the real system the design will be receiving TSS data from TSUs over an internet or similar connection.

For this purpose, another Python code called "feeder.py" was written to feed TSS data to main Python code. Instead of Posedges, feeder reads the text files generated by TSSgen, and sends the TSS1 and TSS2 data over TCP/IP from different ports.

In the main Python code, Posedges receive the data sent by feeder, but both receiving data and running the Posedge algorithms at the same time results in Posedges to work much slower than before and become the bottleneck of the design. In order to solve this issue, the data receiving function got separated from the Posedges and receiver processes created. Two receiver processes work to receive data from feeder and send them to Posedges over queues. With receiving and Posedge algorithms parallelized, design works faster than before. Module and FIFO connection with receivers can be seen in figure 9.

One last addition for the data receiving is so far the main Python code was designed to work on a single computer while receiving data from both TSUs. However, in actual system two computers run the algorithms, and receive TSS data from one

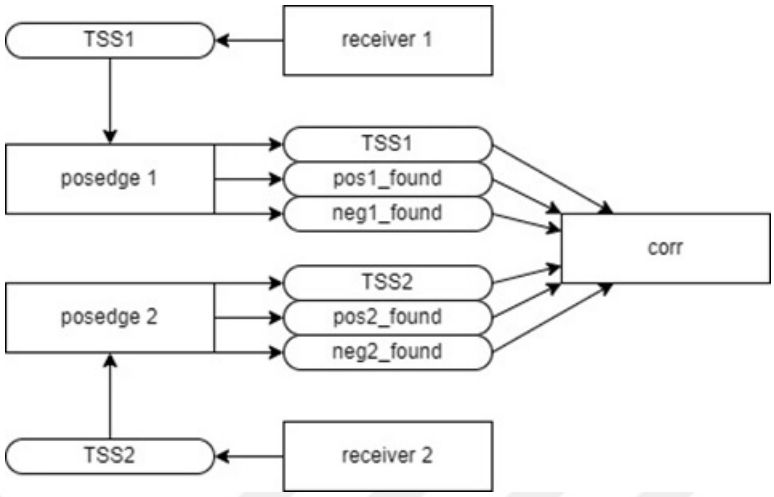


Figure 9: Receivers, Posedges, Corr and queue connections

TSU each, then send the TSS they received to each other. To add this function to the design as well another process called sender got added to the design. Sender module receives TSS data from receiver 1 and sends it to the other computer, and other computer receives these TSS data from its receiver 2 process, while receiver 1 modules receive their data from the feeder module. The module connection with sender can be seen in figure 10.

The main Python code has a variable at start, and with this variable the system can work as running as a single computer with one code running and receiving both TSS data from feeder, or work as one of two computers with an id that decides which computer each main code will work as, this is important for port connections. The module connection when codes ran on two computer can be seen in figure 11.

2.4 Drift and Jitter

With the main algorithms and design completed, next step is to include two problems that are present in the actual QKD system. These problems are clock drift and detector jitter.

The two TSUs normally working in synchronization, with both of their time moving forward in the same pace. However, in some instances it is possible for timers of

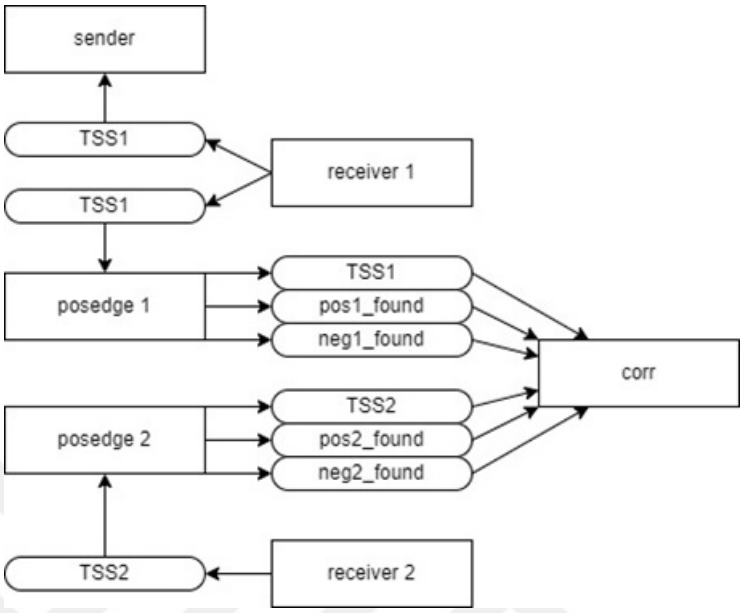


Figure 10: Module connections with sender

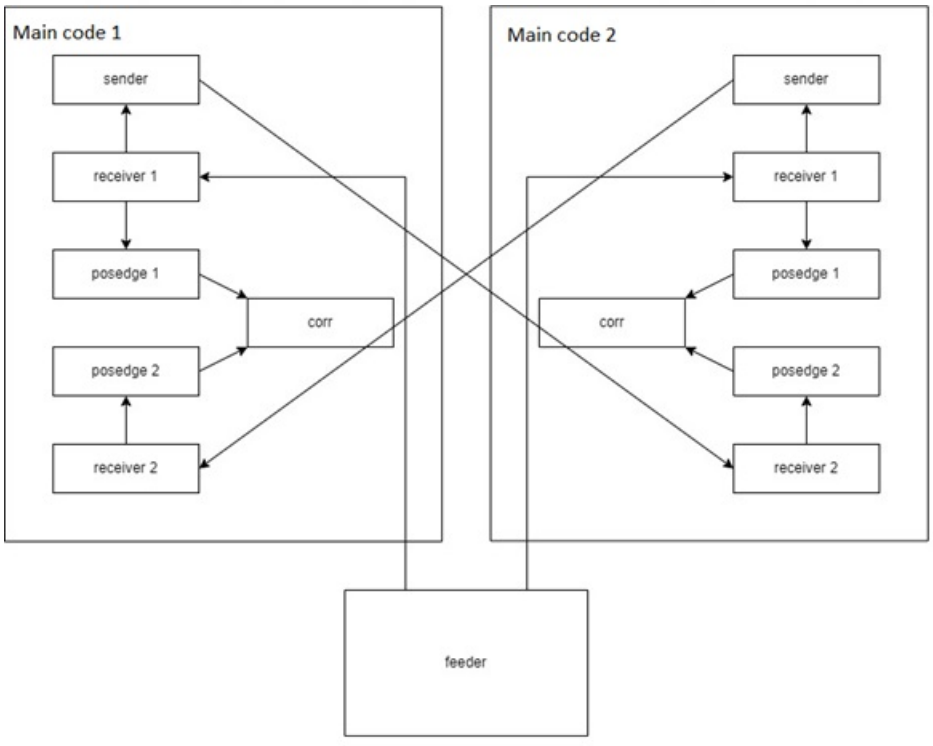


Figure 11: Two main codes and their modules connections

the TSUs to become unsynchronized in time [8]. While this error could be negligible in larger time frames, because we are working as small as nanosecond levels this issue effects matching. The clocks of the TSUs can slowly drifts towards forward and backwards but in different speeds, or towards different directions.

The other issue is the detectors not detecting the photon arrival times perfectly. In a setup where two terminals receive photon pairs with 505ns delay difference from each other, it is not possible for detectors to detect all pairs with exact same delay difference. The photon detection times are susceptible to errors in a range, as a result the 505ns delay difference is also ends up not being static but change in a small range such as 500ns to 510ns randomly for every photon pair. This issue is referred to as jitter, because of the jittery detector time stamping [9].

In order to add these two problems to our system, first the TSSgen needs to be updated with according to clock drift and detector jitter to create TSS data with these two problems. Clock drift addition is straight forward, after TSS1 and TSS2 data are created, all time stamps of the TSS2 data are scaled. Usually the clock drift happens at a rate of once every 50.000ns [10]. With this information all the time stamps of the TSS2 data are scaled by a rate of 50.001/50.000ns.

Jitter addition takes in a parameter to decide at what range the time stamps would jitter forwards and backwards in time, after TSS1 and TSS2 are finished with this parameter all time stamps are shifted randomly in the range of this parameter.

The addition of clock drift causes the shift difference to be no longer static in a system. If the shift difference is 500ns at the time entangled photons started to arrive, this shift difference will increase by 1 once every 50.000ns passes, and with entangled photons arriving once every 200ns, shift difference will change on average once every 250 entangled photons arrive.

In order to solve this issue, after the first correlation, periodically new correlations are done to update the shift difference. Knowing that clock drift happens once every

50.000ns, after the first correlation the Corr process continues to do only matching for a long period. After this period passes, Corr process starts to save the TSS1 and TSS2 data it receives and after recording this photons for a short period, the matching part of the Corr process pauses until the next shift difference is found. After the new shift difference is found, matching continues until another longer period passes, afterwards for a short period the received photons are recorded for the next shift difference to be found. This operation repeats until matching finishes. A problem with this method is that, if a new shift is found before or after the data with more drift starts to arrive, some photons can be missed from matching or the previous shift difference can be found as the new shift difference. This problem will be solved with the additions to the design for jitter.

The first problem jitter causes in the system is finding a shift difference with correlation. Without jitter all the entangled photons having the same shift difference between their photons result in a spike in the histogram correlation is recording.

Table 7: Sample TSS1 and TSS2 data without jitter and their histogram

$TSS1timestamps$	$[105, 115, 125, 135, 145, 155, 165, 175, 185]$
$TSS2timestamps$	$[205, 215, 225, 235, 245, 255, 265, 275, 285]$
$Histogramfrom0to200differences$	$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,$ $1,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,$ $3,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,$ $5,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,0,$ $7,0,0,0,0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,$ $9,0,0,0,0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,$ $7,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,0,$ $5,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,$ $3,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,$ $1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$

As seen in table 7, an example TSS1 and TSS2 data without jitter taken into correlation has the biggest value at the location 100, indicating that the shift difference is 100ns.

Table 8: Sample TSS1 and TSS2 data with jitter and their histogram

$TSS1timestamps$	$[105, 115, 125, 135, 145, 155, 165, 175, 185]$
$TSS2timestamps$	$[201, 212, 223, 234, 245, 256, 267, 278, 289]$
$Histogram from 0 to 200 differences$	$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,$ $0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,1,0,$ $0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,1,1,1,$ $1,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,$ $1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,$ $1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,0,0,0,1,1,$ $1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,$ $1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,$ $0,0,1,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,$ $0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$

As seen in table 8, when the same correlation operation is done to a data with jitter it results in histogram not having any highest value. In this example all the non 0 locations of the histogram have value of only 1, it is possible for some locations to have more than 1 and become the highest value, but this value can be on the very edge of one side of the jitter, resulting in photons pairs at the other side of the jitter to be not detected later in matching. Additionally, when histogram have a highest value which is very close to 0, it is possible for ambient photons to coincidentally make a spike at a very wrong location, which would result in none of the entangled photons to be matched.

This problem is solved with a method that will be called as blur method. Because incrementing the differences during correlation one by one result in no location to have a spike, during correlation addition to incrementing a location in the histogram, the neighboring location of that location is also incrementing. With this method the center addresses are incremented more from the blur method and end up having the highest value in the histogram.

As seen in table 9, as blur amount increases the histogram starts to get more blurred locations. For this example the best results achieved with the blur amounts

of 7 and 8. For the blur amount of 7 a single peak found at the location 103 is found with 9 increments, for the blur amount of 8 two peaks found at the location 103 and 104 with 9 increments. Both location 103 and 104 is very close to the average shift amount of 100, so this amount can be used for matching. As can be seen in table 9, setting blur amount too little give a similar result without blur with many peaks with small amount of increments, on the other hand setting blur too high result in a lot of histogram locations to merge together and give no peak at all. For this example after testing the best blur amount turns out to be 7, but for the TSS data generated by TSSgen a different amount is used.

With the blur correlation, a shift difference very close to the actual shift can be found but afterwards another problem due to jitter comes up during matching. Previously two photon were compared and if their time stamp difference was equal to the found shift difference, there were accepted as photon pairs and matched. However, when time stamps with jitter compared very few of them can be exactly equal to the found shift difference

As seen in table 10, only the 8th pair's time stamp difference is equal to the shift difference. With the previous matching method, both photons of the pairs other than the 8th pair would be deleted, and only the 8th pair would be matched. In order to solve this problem, an acceptable error amount is selected. If the absolute value of the time stamp difference between two photons is less than a certain threshold, these two photons are accepted as a pair and matched. This solution brings the possibility of a photon having more than one possible matches. In order to fix this issue, while two photons are compared, photon from TSS1 is also compared with the next TSS2 photons, and the TSS2 photons is also compared with the next TSS1 photon. If TSS1 photon is also a match with the next TSS2 photon, the three photons are deleted since there is no way to tell which photon from TSS2 is the actual pair and which photon is an ambient photon. A similar operation is done when TSS2 photon match with

both current and next TSS1 photons. In the cases both TSS1 and TSS2 match with the next TSS2 and TSS1 photons, all 4 photons are deleted from the system.

2.5 Key Negotiation

So far the design mainly is done for a terminal or a computer to receive both TSS1 and TSS2 data. When the key is generated from the matched photons, the detector number of the TSS1 photon of the pair is taken into consideration and the detector number of TSS2 photon is ignored. In the cases with two terminals, terminal 1 generates a key taking TSS1 detectors numbers into consideration, and terminal 2 generates a key taking TSS2 detector numbers into consideration. Normally when two photons of a pair is detected in the same base, both photons should be detected on the same detector. However, although rarely, it is possible for two photons of a pair to be detected in the same base but on different detectors. If this happens, since two terminal 1 generate key from TSS1 and terminal 2 generate key from TSS2, the two terminals generate two slightly different keys.

When the two terminals generate their keys, the key terminal 1 generated is accepted as the correct key and terminal 2 changes the different bits of its keys. In order to change the different bits of the key terminal 2 found, terminal 2 needs to know the key terminal 1 found, but exchanging the keys they found with each other is not safe, therefore a key negotiation algorithm called cascade error correction algorithm is used.

Instead of exchanging all the bits of the keys they found, the terminals compare the parity of certain range of bits in their keys, if their parity do not match, the left half and the right half of the ranges' parity is compared. This operation is continues until a 1 bit range is found with not matching parity, in this case terminal 2 changes this bit, and continues to compare the parities in the remaining ranges.

As seen in table 11, the differences of two keys with 8 bits are reduced with the

cascade algorithm. The two key has different bits at the location 2, 4, 6 and 7. The operations done in cascade is as follows:

- The parity of bits from 0 to 3 are checked.

- In the range of 0 to 3 key1 has even parity and key 2 has odd parity

- Because the parity is not matching, parity of the left and right half of the range is compared

- The parity of bits from 0 to 1 are checked

- In the range of 0 to 1 both keys have even parity

- Because the parity is matching the bits in the range of 0 to 1 are assumed to be same

- The parity of bits from 2 to 3 are checked

- In the range of 2 to 3 key1 has even parity and key 2 has odd parity

- Because the parity is not matching, parity of the left and right half of the range is compared

- The bits at the location 2 are compared

- Because the bits at location 2 do not match, the bit at location 2 in key 2 is changed.

- The bits at the location 3 are compared

- Because the bits at location 2 match, the bit at location 2 in key 2 is unchanged.

- The parity of bits from 4 to 7 are checked.

- In the range of 4 to 7 key1 has even parity and key 2 has odd parity

- Because the parity is not matching, parity of the left and right half of the range is compared

- The parity of bits from 4 to 5 are checked

- In the range of 4 to 5 key 1 has even parity and key 2 has odd parity

- Because the parity is not matching, parity of the left and right half of the range is compared

- The bits at the location 4 are compared

- Because the bits at location 4 do not match, the bit at location 4 in key 2 is changed.

- The bits at the location 5 are compared

- Because the bits at location 5 match, the bit at location 5 in key 2 is unchanged.

- The parity of bits from 6 to 7 are checked

- Because the parity is matching the bits in the range of 6 to 7 are assumed to be same

At the end of the cascade algorithm the different bits in the location 2 and 4 are changed in key 2, but the differences in the bits 6 and 7 could not be fixed. The solution to this problem is, after each time cascade algorithm is completed, the two key shuffles in the same order, and the cascade algorithm is ran again.

An example shuffle after the first cascade can be seen in table 12, with this shuffle it is possible to correct the wrong bits with the next cascade. It was possible for shuffle to arrange wrong bits in a way that cascade algorithm couldn't correct again, in order to minimize this possibility, the two keys shuffled multiple times and cascade algorithm is ran after each shuffle for two keys.

2.6 Further Optimization

At the current stage of the implementation, the design was still working slower than the intended speed, and when algorithms reviewed again, it is found out that certain parts can be improved.

When the initial correlation is done to find the shift difference, all of possible shift differences are recorded on histogram between the TSS1 and TSS2 photons. From Posedge 1 and Posedge 2 processes, the approximate entangled photon start time of TSS1 and TSS2 are known. With this information it is not needed to compare all of the possible matches from TSS1 and TSS2. With the known approximate start

times, an approximate shift difference can be found too, and the real shift difference should be close this approximate shift difference. Knowing this, when TSS1 and TSS2 time stamps are compared and written on a histogram, the possible matches too far from the approximate shift difference are skipped and only photons that give results in a range around the approximate shift difference are written on the histogram. With this new method the initial correlation significantly, and non-initial correlation slightly becomes faster.

The previous logic utilized even more for the non-initial correlation. Knowing that the drift happens once every 50.000ns, the shift difference after the first shift difference should be very close to the first shift difference. Knowing this, instead of recording photons and stopping matching to find a new correlation periodically, it is possible not do any correlation for a period again and then do correlation in parallel to matching with matched photons to see how much the shift difference is between the matched photons compared to the current shift difference. Due to the blur matching the photons with drift will continue to match but their average shift difference will be changing due to the drift. Recording this change in the average shift difference is enough to update our shift difference to continue matching photons with more drift.

At the end of the implementation and the attempts for further optimization, the speed of the Python code was still too slow compared to the speed required to work with the QKD system. Therefore it was decided to implement the algorithms in Verilog RTL due to FPGAs having faster performance than GPUs and multicore devices for many cases [11]. While GPUs can have the same computational speed as FPGAs in some cases, using FPGAs is an advantage due to the smaller memory size of GPUs [12].

2.7 Verification

The main result of the implementation is the found key. In order to verify that the found key is correct TSSgen marks photons when created. Entangled photon pairs receive id numbers in the order they are created, while ambient photons receive id of zero. Additionally since TSSgen does the operation such as random photon deletion, if a pair's photon or photons are deleted, their ids are recorded, and with these information, an expected key is generated from the TSSgen.

When implementation matches the photons with their time stamps, it also records the photons it matched with their time stamps and ids, also when photons are deleted their time stamp and id are also recorded. When a test data is ran and a key found, a "checker" modules reads these recordings, and compare the key which is generated by the implementation with the key TSSgen expects. If there is nothing wrong with the recordings and keys match, it is decided that the implementation found the key correctly.

Additionally, entangled photon start times are also recorded in the implementation, and compared with the parameters TSSgen uses for entangled photon start time.

One last thing checked is the initial and the periodic shift differences found with the correlation are compared with the parameters used in TSSgen to create a shift difference in the test data.

If all of these results are matching with the TSSgen it is concluded that the implementation is working correctly with the given test data.

Table 9: Sample two TSS data with jitter and their histogram with blur method

$TSS1timestamps$	$[105, 115, 125, 135, 145, 155, 165, 175, 185]$
$TSS2timestamps$	$[201, 212, 223, 234, 245, 256, 267, 278, 289]$
$Histogramfrom0to200$ $differencesWith1blur$	$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,$ $0,0,0,0,0,1,2,2,1,0,0,0,0,0,0,1,2,3,2,1,$ $0,0,0,0,0,1,2,3,3,2,1,0,0,0,0,1,2,3,3,3,$ $2,1,0,0,0,1,2,3,3,3,3,2,1,0,0,1,2,3,3,3,$ $3,3,2,1,0,1,2,3,3,3,3,3,3,2,1,1,2,3,3,3,$ $3,3,3,3,2,1,1,2,3,3,3,3,3,3,2,1,0,1,2,3,$ $3,3,3,3,2,1,0,0,1,2,3,3,3,3,2,1,0,0,0,1,$ $2,3,3,3,2,1,0,0,0,0,1,2,3,3,2,1,0,0,0,0,$ $0,1,2,3,2,1,0,0,0,0,0,0,1,2,2,1,0,0,0,0,$ $0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$
$Histogramfrom0to200$ $differencesWith7blur$	$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,$ $1,1,1,1,0,1,2,2,2,2,2,2,2,2,1,1,2,3,3,3,$ $3,3,3,3,2,2,2,3,4,4,4,4,4,4,3,3,3,3,4,5,$ $5,5,5,5,4,4,4,4,4,5,6,6,6,6,5,5,5,5,5,5,$ $6,7,7,7,6,6,6,6,6,6,6,7,8,8,7,7,7,7,7,7,$ $7,7,8,9,8,7,7,7,7,7,7,7,7,8,8,7,6,6,6,6,$ $6,6,6,7,7,7,6,5,5,5,5,5,5,6,6,6,6,5,4,4,$ $4,4,4,5,5,5,5,5,4,3,3,3,3,4,4,4,4,4,4,3,$ $2,2,2,3,3,3,3,3,3,3,2,1,1,2,2,2,2,2,2,2,$ $2,1,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]$
$Histogramfrom0to200$ $differencesWith8blur$	$[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,$ $1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,3,3,3,$ $3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,4,4,5,$ $5,5,5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,$ $6,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,$ $8,8,8,9,9,8,8,8,8,8,8,8,8,8,8,8,7,7,7,7,$ $7,7,7,7,7,7,7,6,6,6,6,6,6,6,6,6,6,6,5,5,$ $5,5,5,5,5,5,5,5,5,4,4,4,4,4,4,4,4,4,4,4,$ $3,3,3,3,3,3,3,3,3,3,3,3,2,2,2,2,2,2,2,2,$ $2,2,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0]$

Table 10: Sample TSS1 and TSS2 data with the time stamp difference of each pair

$TSS1timestamps$	$[105, 115, 125, 135, 145, 155, 165, 175, 185]$
$TSS2timestamps$	$[201, 212, 223, 234, 245, 256, 267, 278, 289]$
$Timestampdifferenceofeachpair$	$96, 97, 98, 99, 100, 101, 102, 103, 104$

Table 11: Cascade used for two keys with 8 bits

		0	1	2	3	4	5	6	7
$Before$	$key1$	1	1	1	1	1	1	1	1
	$key2$	1	1	0	1	0	1	0	0
$After$	$key1$	1	1	1	1	1	1	1	1
	$key2$	1	1	1	1	1	1	0	0

Table 12: Shuffle used for two keys with 8 bits

		0	1	2	3	4	5	6	7
$Before$	$key1$	1	1	1	1	1	1	1	1
	$key2$	1	1	1	1	1	1	0	0
$After$	$key1$	1	1	1	1	1	1	1	1
	$key2$	1	0	1	1	0	1	1	1

CHAPTER III

FPGA IMPLEMENTATION

Previously the TSSgen was creating memory files for TSS1 and TSS2 in an array format. Since Verilog can only read binary data, TSSgen was updated to generate binary versions of the TSS data. The TSS data should be recorded over the course of one second, which is equal 1.000.000.000ns, and this number in binary is equal to 111011100110101100101000000000. This binary number has 30bits, depending on the detector amount the time stamps can be generated in 31 bits formats or more.

3.1 RTL Implementation

The two main processes in Python code, Posedge and Corr, needs to be implemented first as they are the main algorithms of the design. These two processes are designed as separate Verilog modules connected with FIFOs. The design can be seen in figure 12.

3.1.1 Posedge Implementation

The Posedge method uses sliding windows that record photons in them. In Python these windows were implemented as dynamic arrays, two arrays are used in total: one array to record arriving photons called current window, and another window to save the last completed window called previous window. When a current window needs to slide, it becomes the previous window and the photons that should be in next window transferred to the current window.

Verilog do not have dynamic arrays and due to the memory writing issues in Verilog, transferring photons between windows takes a clock cycle for each photon, and this is time consuming. Instead, a circular block ram is used. Knowing that on

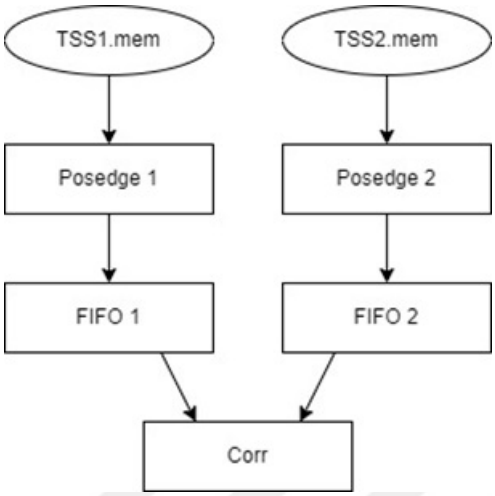


Figure 12: Modules of FPGA

average in a window there are 40 photons, and we need to record 1.5 windows, on average we need to record 60 photons. In order to not miss any photons a circular block ram with 128 locations is created, and instead of transferring photons between dynamic arrays, pointers on circular block ram is held. These pointers record the start and end addresses of current window and previous window. When a photon with a time stamp higher than the upper limit of the current window arrives, the pointers of the two windows are updated and the difference between the location numbers are checked. If the difference between the ending and the starting location of the current window is more than the threshold, it is decided that the entangled photon starting time is found.

One other issue with Verilog implementation of the Posedge is, when a photon with a time stamp higher than the upper limit of the current window arrives, the current window is shifted by a certain amount. However, it is possible for a photon to arrive after long delay and the window needs to be shifted more than once. This issue in Python is solved by checking how many times the windows should be shifted in a while loop. This is not possible in Verilog, so instead if it is seen that shifting the window once is not enough for a photon, the Posedge module goes to a state where

is shifts its windows every clock cycle until the windows are shifted by the correct amount.

After entangled photon starting time is found, the photons are written to a FIFO starting from the start address of the previous window until the last address of the current window. While Posedge module continues to shift the windows the TSS data are also written to a FIFO in parallel. This operation continues until entangled photon ending time is found. Afterwards, Posedge stops the windows and also no longer writes any data to the FIFOs.

The Posedge module takes a validIn signal and 32bit TSS data from the test bench, and outputs wrEn signal and 32bit TSS data to a FIFO. Posedge modules also send the entangled photon starting times and signals to inform that entangled photon starting times are found. Additionally when the ending time is found each Posedge sends a signal to Corr.

3.1.2 Correlation and Matching Implementation

Similar to the issue in Posedge with the dynamic arrays not existing, Corr module also has two rotating block rams with 128 locations, for TSS1 and TSS2 data. Pointers used in Corr module as well to keep track of which photons will be matched next, for this case instead of deleting photons the pointers just skip them since due to the nature of the rotating block rams, these locations will be overwritten later on.

One of the design problems in the Verilog is the blur method used in the correlation. Incrementing the neighboring addresses in histogram in Python is a rather simple operation, but in Verilog it is not possible to increment more than one memory location at the same time, and incrementing an address and its neighbors one by one each clock cycle takes too much time.

In order solve this issue, blur method is changed. To get a similar result to what was used previously, instead of incrementing multiple locations, the differences

found during correlation can be compressed to fewer addresses, and at the end of the correlation the address with the highest value can be scaled according to the compression amount. This method gives similar results to the previous blur method, but takes less time and takes less area.

During correlation an address of the histogram is incremented each clock cycle. For this increment, the previous value of the address is checked and an incremented value is written to the address. Reading and writing requires two clock cycles in total, but this operation can be pipelined and each clock cycle an address is read and another address is written. The only problem with this pipelining is if an address needs to be incremented twice in a row, the pipeline can get the wrong value for the second increment. In order to solve this issue a forwarding method is used when writing to the histogram.

After the correlation a shift difference is found and the matching operation starts. After the first correlation is completed the histogram used for the correlation is filled with numbers. It is required to reset this histogram for the non-initial correlation. As mentioned before, there is a time delay between two correlations, in this time delay the histogram location are reset to zero.

3.2 Hardware Verification

The verification of the implementation is done in two steps, first verification with simulations, then verification on a board. Since debugging on simulation is significantly easier compared to testing on a board, the algorithms were first tested by simulation.

3.2.1 Verification by Simulation

The verification of the Verilog implementation is rather simple compared to the Python verification. Because of the Python implementation it is possible to know exactly what results to be expected from the Verilog simulation. Some of the design changes such as pointers and blur correlation change is also added to the Python

implementation in order to run exact same algorithm in both Python and Verilog implementation. When a test data is created, from a test bench the binary TSS data is fed to the Posedge modules of the Verilog design. After the Corr module finds a key, some of the results of the implementation is compared to the results of the Python design. These results are entangled photon start and end time for TSS1 and TSS2, initial and non-initial shift differences found, key size and every bit of the key.

After several tests both Verilog and Python implementations find the same results. For the implementation Atlys Spartan-6 FPGA Trainer Board was chosen, and when the timing analysis is done a worst case clock frequency of 70MHz was found. With this worst case clock frequency, when a theoretical matching speed is calculated for the Corr module, a matching speed of 48.5 million photon pairs per second is found. When the matching speed is measured from the test cases, a matching speed of 50 million photon pairs per second on average was found.

3.2.2 Verification on a Board

With the simulation stage completed, the next step was to run the algorithms on an FPGA. In order to run the algorithms on an FPGA board, the FPGA needs to receive TSS data from the computer for testing. For sending TSS data to the FPGA an open source Ethernet interface for Atlys Spartan-6 FPGA Trainer Board is used.

The Ethernet interface is connected with the top module of the Verilog implementation. The Ethernet interface receives data with 125MHz and sends the top module of the implementation in 8bit slices. Since Corr and Posedge modules can't work at 125MHz it is required for Ethernet and algorithms to work with different clocks. However, this clock difference causes Posedges to take data much slower than the Ethernet sends. In order to solve this issue two FIFO, working with different clocks for reading and writing, is connected between Ethernet interface and the top module, but this solution only works with small size TSS data. When larger sized TSS data

is used, the two FIFO between Ethernet and top module overflows in time.

As an alternative, a merger module is added between Ethernet interface and Posedges. When Ethernet interface sends data to top module, the merger module receives the data and after receiving 4 data, sends it to a FIFO, working with different clocks for reading and writing, connected to the Posedge. Due to the simplicity of the merger module, it can work at the fast clock frequency of the Ethernet interface. Since the merger module works at the same clock frequency with the Ethernet interface, there is no data pile up between Ethernet and top module. Additionally with merger module sending data to FIFO once it receives four 8 bit data, the highest speed it can write to the FIFO is 31.25MHz, which is significantly slower than the clock frequency Posedge and Corr modules work with, due to this, it is not possible for data to pile up between merger and Posedge modules. However, still it is required to use a small FIFO between mergers and Posedges for data transfer since the two modules work with different clock frequencies. The final version of the modules and FIFOs can be seen in figure 13.

With this final version of the design a theoretical matching speed of 42MHz is found, which is higher than the highest speed Corr module can receive data, meaning it is possible to match photons while receiving them without any overflow happening.

For the verification of the design on FPGA, initially the key length is displayed over the LEDs on the FPGA and it is compared with the key size found in simulation. After confirming that the key sizes are correct for several test cases, the key bits are checked for verification. In order to check the bits of the key, a FIFO after the Corr module is added. Corr module writes the bits of the key to this FIFO in 8 bit slices. In order to read the key slices in this FIFO, a UART interface is added to the system to display the key bits on a computer. Several test cases are run and the found key is compared with the key found in simulation. After these tests it is concluded that the design can find key correctly.

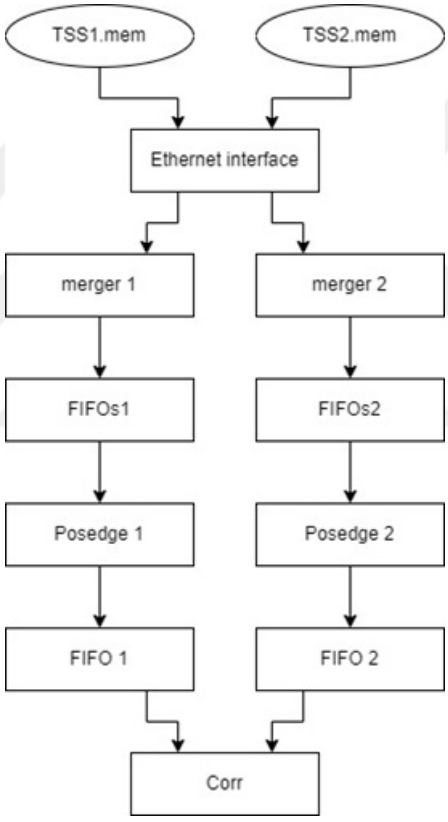


Figure 13: The final version of the modules and FIFOs

CHAPTER IV

RESULTS

The Python implementation of the designed algorithms was able to match photons and find keys successfully. However, the photon matching speed of the Python implementation was significantly slower than the target speed of 5 million photons per second.

Table 13: Python and Verilog matching speeds

$DataSize$	$PyTime$	$PySpeed(pho/sec)$	$RTLtime$	$RTLspeed(pho/sec)$
500	$0.66sec$	757.58	$215us$	$4.35M$
1000	$0.77sec$	1298.70	$312us$	$3.21M$
5000	$1.00sec$	5000.00	$1114us$	$4.49M$
10000	$1.38sec$	7246.38	$2116us$	$4.73M$
50000	$4.42sec$	11312.22	$10118us$	$4.94M$

As seen in table 13, the Python speed is significantly slower that the target speed. After the Python implementation, the Verilog RTL implementation gave better speed results as seen in table 13. The reason matching speed is slower than 5 million photons per second is due to the 100us delay of entangled photons starting to arrive. If this delay is ignored the speed is as in table 14.

Table 14: Verilog matching speed measured after entangled photons start to arrive

$DataSize$	$RTLtimeafterentangledstart$	$RTLspeed(pho/sec)$
500	$115us$	$4.35M$
1000	$212us$	$4.72M$
5000	$1014us$	$4.93M$
10000	$2016us$	$4.96M$
50000	$10018us$	$4.99M$

When entangled photons start to arrive, first the Posedge needs to be found and then the shift difference needs to be calculated. These operations causes a delay before matching starts. The delay Posedge and correlation causes can be seen in table 15.

Table 15: Posedge and Correlation time delay in different test cases

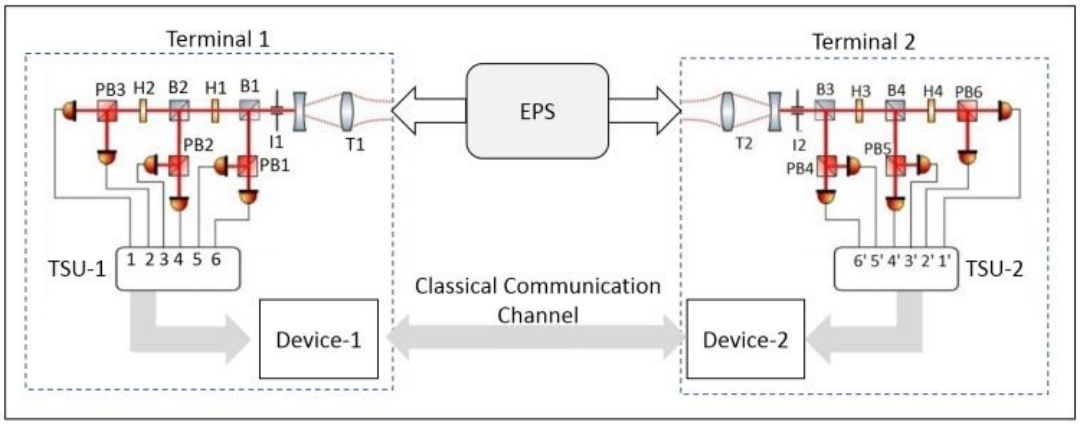
$TestID$	$PosedgeTimeDelay$	$correlationTimeDelay$
1	$4252ns$	$6648ns$
2	$4520ns$	$6228ns$
3	$4352ns$	$7416ns$
4	$4472ns$	$10752ns$
5	$4592ns$	$9048ns$

As seen in table 15, Posedge causes a delay of on average 4437ns, while correlation causes a delay on average 8018ns. These delays cause on average 62 photons to pile up, but afterwards the high throughput matching algorithm of the implementation matches the piled up photons faster than the rate the entangled photons received from the EPS. In table 16 the matching speed of the piled up photons can be seen.

Table 16: Matching speed when there are piled up photons in different test cases

$TestID$	$Time$	$PhotonMatched$	$Speed$
1	$1464ns$	45	$30.74(Mphoton/sec)$
2	$1224ns$	38	$31.05(Mphoton/sec)$
3	$1608ns$	50	$31.09(Mphoton/sec)$
4	$2112ns$	62	$29.36(Mphoton/sec)$
5	$1752ns$	53	$30.25(Mphoton/sec)$

While there are enough photons in the system it is possible to match them at an average 30.5 million photons per second rate, which is a much higher speed than they are generated. After the piled up photons are matched, the matching speed is lowered due to the photon generation speed being 1/6th of the maximum matching speed. This speed difference results in large data sizes to have almost 5 million photons per



$$E = \frac{(N_{++}) - (N_{+-}) - (N_{-+}) + (N_{--})}{(N_{++}) + (N_{+-}) + (N_{-+}) + (N_{--})}$$

$$S = E(a,b) - E(a,b') + E(a',b) + E(a',b')$$

Figure 14: E91 protocol

second matching speed as seen in table 14 with 50.000 data size.

With the target matching speed achieved, it is possible to alter the implementation for different QKD protocols. A simple example is E91 protocol where a S value is calculated depending on which bases photons of a pair was detected as seen in figure 14.

For this protocol the implementation can be altered to record E values and a key while matching, then calculate the S value when matching ends.

Another protocol to use the implementation with is the bbm92 protocol. This protocol has 2 detectors and no S value is calculated. The detector layout can be seen in figure 15.

For this protocol each terminal has 2 bases and photons are matched between base 1 and between base 2 only. Additionally the base 1 and 2 have different shift differences. For this reason 2 different shift difference is calculated and matching is done separately for each base. The modules for bbm92 can be seen in figure 16.

In this implementation the Posedge modules send the photons to different FIFOs

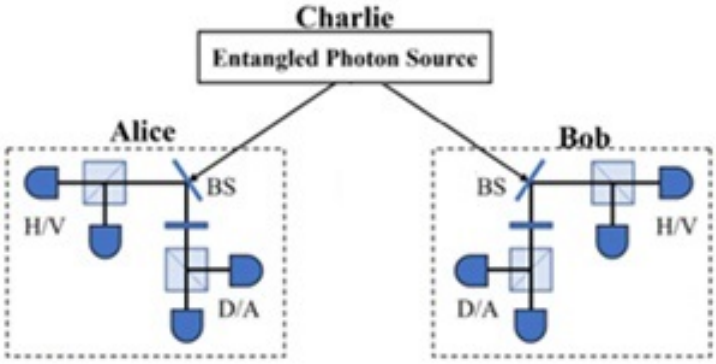


Figure 15: BBM92 protocol

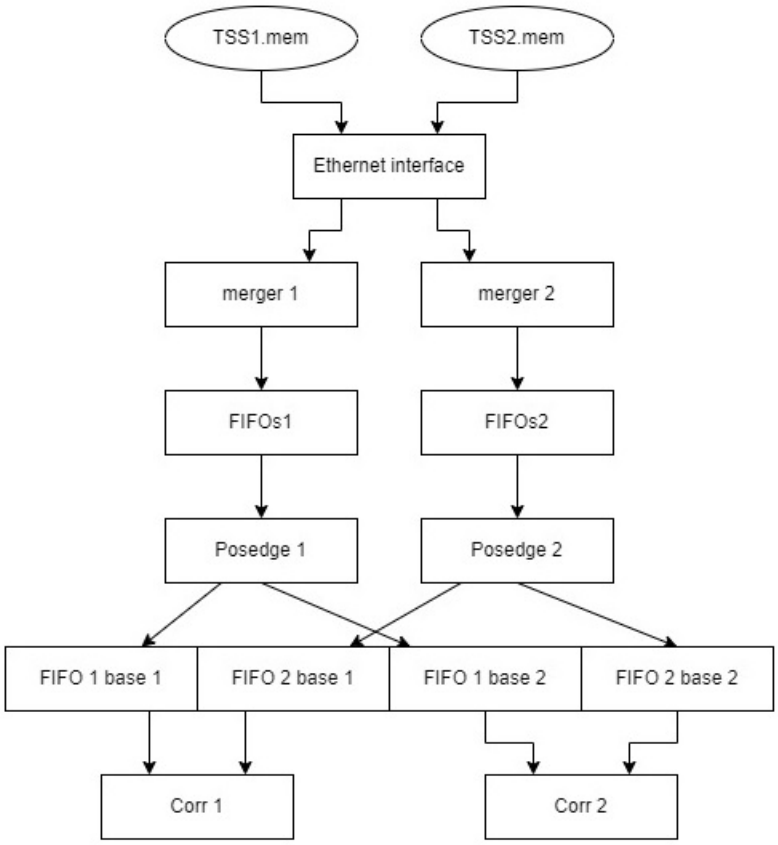


Figure 16: FPGA modules for BBM92 protocol

depending on whether they are detected on base 1 or base 2. The two correlation modules find shifts for base 1 and base 2 photons and match them separately and find two keys from the two bases.

CHAPTER V

CONCLUSION

With the python implementation, the Posedge algorithm to find the entangled photon starting time for the two terminals, the sparse time domain correlation algorithm to calculate the entangled photon starting time difference between the two terminals, and a matching algorithm that matches photons with the time difference correlation calculated were designed and optimized. These algorithms were designed to work in parallel while sharing required information between them.

The python implementation was able to achieve a maximum matching speed of 11313 photon pairs per second. Although this matching speed of the Python implementation was not able to reach the required speed, due to Python being easy to alter, it was useful to design and optimize the algorithms. Additionally, the Python implementation was also useful to verify the Verilog implementation during its testing.

Unlike the Python implementation, the Verilog implementation was able to achieve the target speed. When entangled photons start to arrive, the Verilog RTL implementation has an average delay of 4437ns to find the starting time of entangle photons, and afterwards another delay to find the entangled photon starting time difference between the two terminals with an average delay of 8018ns. These two delays on average causes a total delay of 12455ns, which results in 62 photons to pile up in FIFOs on average. After both entangled photon starting times for both terminals and the entangled photon starting time difference are found, the matching starts and it has an average speed of 30.5 million photon pairs per second. With this speed, even though photons continue to arrive with a rate of 5 million photons per second, since the matching speed is much higher than the generation speed, the piled up photon

amount is reduced until FIFOs are empty. Afterwards, matching continues with a speed equal to photon generation speed without any photon piling up.

With the matching speed satisfied, the implementation can be altered for different QKD protocols, and it could also work with systems that generate entangled photons as fast as 30 million photon pairs per second.

REFERENCES

[1] Y. Bilgin, S. W. Tesfay, S. Ipek, H. F. Uğurdağ, K. Durak, and S. Gören, "PYNQ-based rapid FPGA implementation of quantum key distribution," in *Proceedings of the International Conference on Computer Science and Engineering (UBMK)*, pp. 483–488, 2021.

[2] S. Mishra, A. Biswas, S. Patil, P. Chandravanshi, V. Mongia, T. Sharma, A. Rani, S. Prabhakar, and R. Singh, "BBM92 quantum key distribution over a free space dusty channel of 200 meters," *Journal of Optics*, vol. 24, paper id: 074002, pp. 1–7, 2021.

[3] E. Waks, A. Zeevi, and Y. Yamamoto, "Security of quantum key distribution with entangled photons against individual attacks," *Physical Review A*, vol. 65, no. 5, pp. 1–28, 2000.

[4] M. Lucamarini, K. Patel, J. Dynes, B. Fröhlich, A. Sharpe, A. Dixon, Z. Yuan, R. Penty, and A. Shields, "Efficient decoy-state quantum key distribution with quantified security," *Optics Express*, vol. 21, pp. 24550–24565, 2013.

[5] T. Zhong, H. Zhou, R. Horansky, C. Lee, V. Verma, A. Lita, A. Restelli, J. Bienfang, R. Mirin, T. Gerrits, S. Nam, F. Marsili, M. Shaw, Z. Zhang, L. Wang, D. Englund, G. Wornell, J. Shapiro, and F. Wong, "Photon-efficient quantum key distribution using time–energy entanglement with high-dimensional encoding," *New Journal of Physics*, vol. 17, paper id: 022002, pp. 1–10, 2015.

[6] C. Lee, D. Bunandar, Z. Zhang, G. Steinbrecher, P. Dixon, F. Wong, J. Shapiro, S. Hamilton, and D. Englund, "High-rate field demonstration of large-alphabet quantum key distribution," *arXiv preprint arXiv:1611.01139*, 2016.

[7] M. Lasota and P. Kolenderski, "Optimal photon pairs for quantum communication protocols," *Scientific Reports*, vol. 10, no. 1, pp. 1–12, 2019.

[8] H. Marouani and M. Dagenais, "Internal clock drift estimation in computer clusters," *Journal of Computer Systems, Networks, and Communications*, vol. 2008, pp. 1–7, 2008.

[9] L. ou, X. Yang, Y. He, W. Zhang, D. Liu, W. Zhang, L. Zhang, L. Zhang, X. Liu, S. Chen, Z. Wang, and X. Xie, "Jitter analysis of a superconducting nanowire single photon detector," *AIP Advances*, vol. 3, paper id: 072135, pp. 1–6, 2013.

[10] A. Monot, N. Navet, and B. Bavoux, "Impact of clock drifts on CAN frame response time distributions," in *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–9, 2011.

[11] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in

Proceedings of the ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA), pp. 47–56, 2012.

[12] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 126–131, 2009.

VITA

Seçkin İpek started his Bachelor's degree in Electrical and Electronics Engineering in 2015 at Özyeğin University. In Summer 2018, he started to work in embedded systems field within his first internship. In Fall 2018, he started doing a minor in Computer Science. In Spring and Summer 2019, he worked on his graduation project. In January 2020, he graduated from Electrical and Electronics Engineering. Right after graduating, he started his master's in Computer Science in February 2020 at Özyeğin University. In Fall 2020, he finished his minor in Computer Science.