UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science

Hendri

An Optimized Implementation of a Succinct Non-Interactive Zero-Knowledge Argument System

Master's Thesis (30 ECTS)

Supervisors: Helger Lipmaa, PhD
Danilo Gligoroski, PhD

Author: .... "....." 2013
Supervisor: .... "....." 2013
Supervisor: .... "....." 2013

Allowed to defence
Professor: .... "....." 2013

TARTU 2013

Abstract

In this thesis, we construct an implementation of succinct non-interactive zero knowledge argument system. A non-interactive zero knowledge argument system is a protocol for a party (usually known as Prover) to provide a proof of knowledge to the solution of a statement to other parties (usually known as Verifier). The argument system will be able to provide such proof without leaking any other information regarding the solution. The non-interactivity allows such argument system to be done without requiring interaction between the parties involved. The statement that is proven in this work is the circuit satisfiability problem. The circuit satisfiability problem is a problem of deciding whether there exists an input that can make the final output of a circuit to be true. The argument system is based on Lipmaa's work [Lip13] which uses span programs and linear error-correcting codes in its construction. We also try to give a very general explanation on zero knowledge argument system along the way in order to provide a simple concept to people encountering the notion for the first time. The argument system we attempt to construct is the non-adaptive version of the argument system. This version is useful for verifiable computation as pointed out by [PGHR13] apart from its zero knowledge behavior. We begin by giving an overview on non-interactive zero knowledge, followed by span programs. We then proceed to describe on how to represent the circuit satisfiability problem using the mentioned tool. We present our implementation afterwards, listing out the libraries and implementation details that matters. We conclude by providing a speed measurement and possible future improvements of this work.

Contents

Acknowledgements

I would like to thank my main supervisor Helger Lipmaa for his patience and effort in helping me understanding the construction. I would also like to thank my second supervisor Danilo Gligoroski for his feedbacks. Additionally, I would like to thank Mitsunari Shigeo for all his help in using the pairings library. Lastly, I would like to thank Monica Kezia who has been giving me a lot of moral support throughout all the hectic times.

1 Introduction

1.1 Problem Statement

In daily life, it is not uncommon for a certain party to possess information that is of interest to other parties. Knowing such information may make a person be regarded of bigger importance and thus treated with respect and luxury. However, as a result of this treatment there are always people claiming on knowledge of such information even though the afore mentioned does not really possess it. This leads to the interested parties demanding some kind of proof from the party claimed to have the information. The easiest and most straight-forward way to do this is to actually present such information to the interested parties by the one having the information. However, this would mean that the possession of this information is not exclusive to the party originally having it anymore, which might just lower the value of that information itself. This somehow leads to a stalemate. On one side, the interested parties needs proof to prevent any impostor from claiming to have the information. On the other side, the party proving is reluctant to give knowledge of such information as it might not be as valuable anymore. Impossible as it may seems to be, this is actually possible. The first result originates from [GMR85] shows that proving without giving any extra knowledge other than that the party actually possesses the information can be done. This leads to the notion of zero knowledge argument system.

Knowing that zero knowledge argument system is indeed possible, many research and studies has been done on this particular field. The first zero knowledge argument system was interactive [GMR85], where the prover and the verifier interact with each other until the verifier is convinced that the prover indeed possess certain knowledge. Moreover, the first zero knowledge argument systems provided soundness against omnipotent adversaries. Such argument systems are called proof systems. On the other hand, argument systems provide soundness only against probabilistic polynomial-time adversaries. This kind of system is more known as interactive zero knowledge, which comes from the interactive proof system. As good as it is, the interaction in the zero knowledge argument system tends to be impractical in many cases. One of such examples is when there are many verifiers interested in the argument, needing the prover to be available for each and everyone of the verification process. This leads to more study on how one could possibly remove such interaction and still retains the zero knowledge property. As an end result, several non-interactive versions of zero knowledge argument system was proposed [BFM88].

The ongoing research on non-interactive zero knowledge argument system has produced some interesting result such as [Gro10] and [Lip12]. More recently, [GGPR13] propose a non-interactive zero knowledge argument system using span programs. This was then further improved by [Lip13] with some modifications and the use of linear error correcting codes. This most notable result in this recent study is that the communication overhead is constant regardless of the problem statement size. Furthermore, the verifier only needs to do constant amount of pairings computation in [Lip13] the non-adaptive version. From this result, it would be of interest to see just how well does the non-interactive zero knowledge argument system perform in the practical sense and serve as a foundation on what possible improvement could be done on the future research. Compared to the adaptive version, in non-adaptive version the function/circuit needs to be known beforehand before running the argument system. The aim of this work is to implement such argument system. The non-interactive zero knowledge argument system will work on the circuit satisfiability as the problem statement to be proven. The non-adaptive version of this argument system has proven to be useful for practical usage such as verifiable computation shown on [PGHR13]. The performance measurement would be generally based on the computation time and the data size sent over the argument system's online phase.

1.2 Outline

The work in this thesis consists of three parts. First, preliminary knowledge required to understand the non-interactive zero knowledge argument system is provided. The explanation will start with definitions of zero knowledge argument system and span program. After the definitions, it will discuss how such span programs may be used to construct a zero knowledge argument system. From this idea, the explanation moves towards introducing the circuit satisfiability problem and how to represent it using span programs. This will lead to the definition of the gate checker span program, wire checker quadratic span program, and lastly the circuit checker. After laying out the necessary knowledge in the representation, some general details on how the span program will be used in the argument system is then provided. From here, further transformation of the span program into polynomials is introduced and backgrounds on why it would be advantageous for the argument system is explained. The explanation provided in this part is mainly based on [Lip13] but made more intuitive to understand for readers.

In the second part, the non-interactive zero knowledge argument system that is based on [Lip13] is introduced. This argument system uses all the preliminary knowledge that is discussed in the previous part including the polynomial transformation shown last. The discussion then proceed to the main contribution given by this thesis, which is the implementation on this argument system. The implementation was all done in C++. The external libraries used in the implementation are provided, such as the one based on [BGM$^+$10]. The work then proceed to explain some implementation details designed in order to achieve good efficiency on the argument system, such as the representation of sparse matrix using map, aggregating the gate and wire checkers, interpolation, and multiexponentiation function.

In the last part, the time measurement of the implementation such as the multiexponentiation time, the interpolation time, CRS generation time, prover's computation time, and verifier's computation time are provided. After that, some of the possible further improvement that could be done in the implementation are discussed. The thesis ends with a conclusion of the work that has been done.

1.3 Author's Contribution

The main contribution of this work is the implementation of the non-interactive zero knowledge based on [Lip13]. It also attempts to represent the knowledge needed to understand the argument system in a more general way so that it is easier to be understood. Other than that, this thesis has also listed the implementation details that may prove to be useful not only for this specific argument system design but also for other similar work in the future. It is hoped that this work may serve as a good representation of how non-interactive zero knowledge using span programs work in real life and become a good foundation for all the future work that might be related.

For the author, this work served as a good way to understand non-interactive zero knowledge argument system not only in a theoretical sense, but also in practical usage. The author started with minimal knowledge on zero knowledge argument systems. From there, the author has managed to read and understand [Lip13], benefitting from the help of the authors. After that, the author proceeded to think of a good design and tools in order to implement the argument system. With the help of the library's creator [BGM$^+$10], the author has managed to learn how to use the pairings library that is needed in the argument system. The author has also researched on several libraries to determine the best one for the interpolation. After much consideration, the author decided to implement its own layer of functions on top of the NTL library in order to achieve better efficiency. From this thesis, the author has managed to gather many interesting practical details that are needed in order to achieve the better result, ranging from data structure design to efficient computer algebra. Many iterations of optimization coupled with feedback from Lipmaa has improved the efficiency of the implementation by a lot. In the end, during the writing of this thesis the author has also managed to understand more theoretical details and learn to present the knowledge in a intuitive way. This will serve as a very good foundation into any other work that might be done by the author in the future.

2 Preliminaries

Before going into the argument system, some notions needed to understand how the argument system works are given. This will mostly consists of definitions of zero knowledge, non-interactive zero knowledge, span programs, and circuit satisfiability problem. After that, the way to represent circuit satisfiability problem using span programs are given. Lastly, polynomial transformation of span programs and background on such transformation are also given.

2.1 Basic Definitions

2.1.1 Zero Knowledge Argument System

Zero knowledge argument system was first introduced by [GMR85], in the form of an interactive proof system. In an interactive proof system setting, a proof of a theorem usually contains more knowledge than the fact that the theorem is true. What zero knowledge argument system tries to achieve is to be able to prove the correctness of the problem or theorem in question without giving any additional knowledge other than that correctness itself. A zero knowledge argument system should satisfy three properties, which is as follows:

1. **Completeness**: if the prover is honest then the verifier will always accept the proof.

2. **Soundness**: if the prover is dishonest then the verifier will reject the proof with very high probability.

3. **Zero knowledge**: no additional information is gotten by the verifier except from the fact that the statement being proven by the prover is true.

One of the more commonly known argument system is the sigma protocol [CDS94]. The sigma protocol itself is not completely zero knowledge, it is a honest verifier zero knowledge argument system. A honest verifier zero knowledge argument system is a zero knowledge argument system where the verifier is assumed to be honest, where in the sigma protocol case is to provide a "honest" challenge. If the challenge given by the verifier is somewhat crafted in some specific way, then the zero knowledge property is not guaranteed. Albeit not being completely zero knowledge, sigma protocol can however be used as a building block to construct a real zero knowledge argument system. This work will not go more into this argument system, more on this can be read on [CDS94]. Figure 1 shows the general step of the sigma protocol.

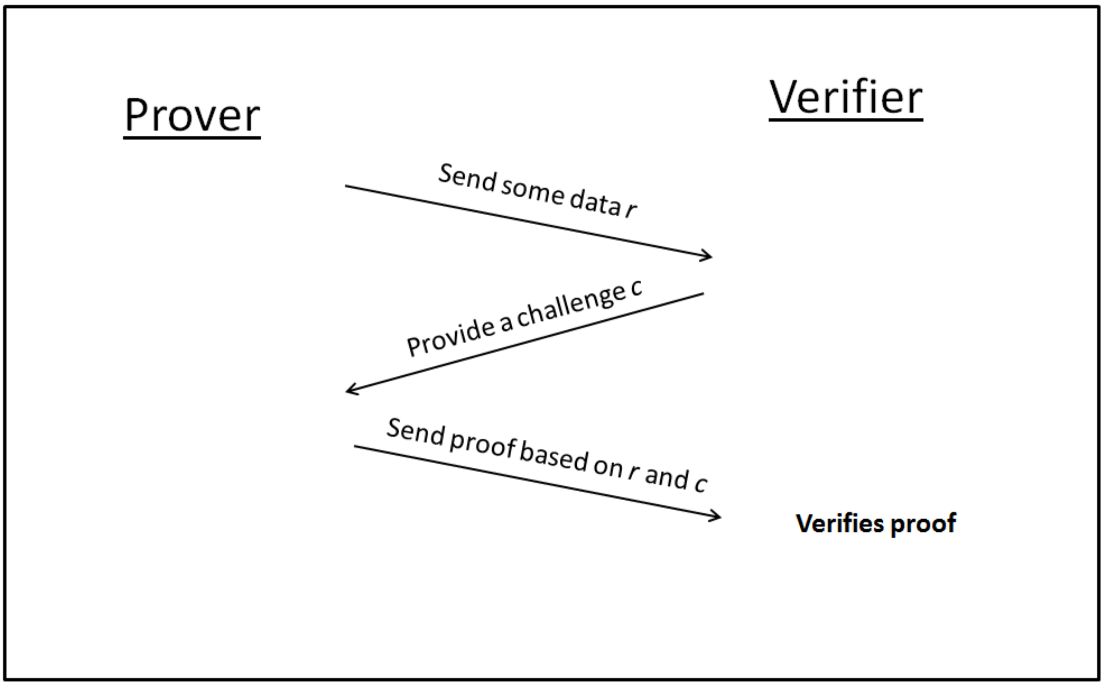


Figure 1: Sigma protocol general steps of interaction

2.1.2 Non-Interactive Zero Knowledge Argument System

Viewing from a practical value, there are times where the interactivity of the zero knowledge argument system explain earlier becomes a liability. For example, it might make sense in some settings where there are multiple parties that are interested in verifying the proof by prover. In this case, the interactive argument system would need the prover to always be there whenever a verifier wants to verify such proof. This leads to a new way of providing a zero knowledge argument system where a prover does not need to interact with the verifier whenever such verification takes place. The way to do this is to remove the interactivity that is needed in the argument system, which is why it is called the non-interactive zero knowledge argument system. One of the way that has been proposed includes a small modification from the general steps of the sigma protocol using random oracle model [BR95]. In the original setting, the prover would need to receive a challenge from the verifier and then compute the proof based on the challenge and the original data it sends to the verifier. In the new setting, first of all it is assumed that both the prover and the verifier has access to a random oracle. A random oracle can be viewed as a function that outputs a truly random output (chosen from its output domain) on every unique query. This means that if the random oracle is given the same query, then it will always give the same output. With the help of the random oracle, instead of getting the challenge from the verifier, the challenge in generated from the random oracle. By having the challenge generated from the random oracle, the prover can then generate its own initial data similar to the first in the original setting, while also computing the proof after that without having to interact with the verifier. The initial data, the challenge gotten from the random oracle, and the proof could then be sent over to the verifier all together. The rest of the verification follows that of the original setting. The concept of this argument system can be seen on figure 2.

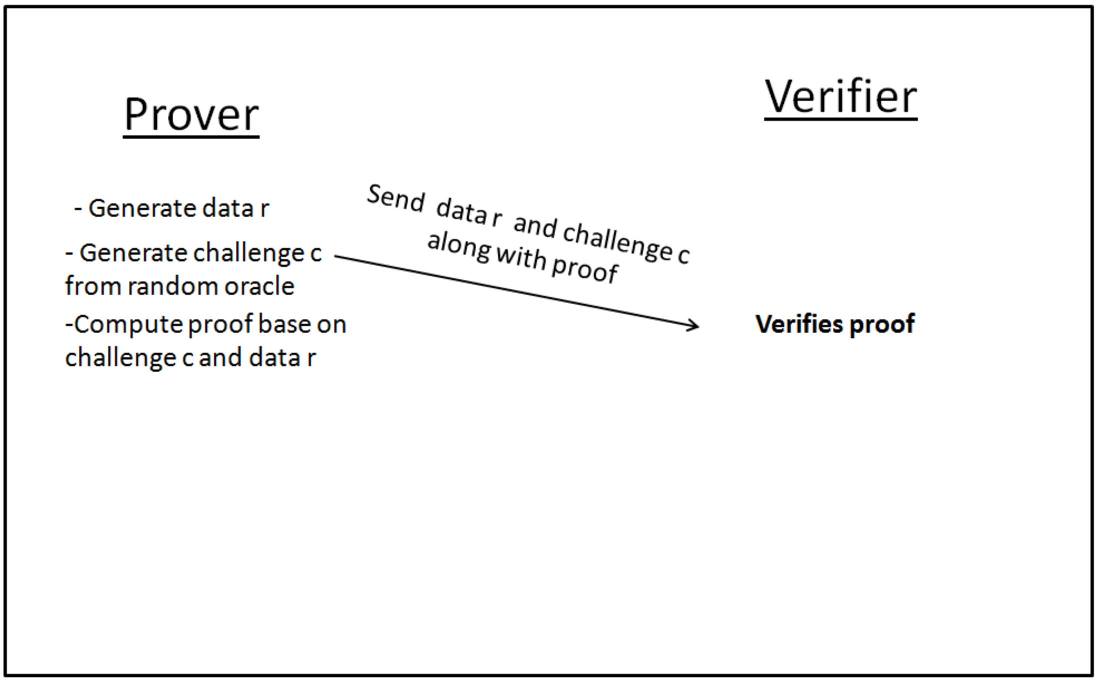

Prover
Verifier
- Generate data r
- Generate challenge c from random oracle
-Compute proof base on challenge c and data r
Send data r and challenge c along with proof
Verifies proof

Figure 2: Non interactive argument system with Random Oracle Model (ROM)

Unfortunately, the condition of this new setting is the existence of random oracles, which are pratically known to be non-existent [CGH98, BGI$^+$01, GT03]. This leads to a more profound idea of having a non-interactive zero knowledge argument system in the Common Reference String (CRS) model [BFM88]. In this model, it is assumed that there is another trusted third party aside from the prover and the verifier. This trusted third party would then generate some value which is called the CRS and send it over to both the prover and verifier. The prover would then compute its proof based on the CRS and send it to the verifier. The verifier herself would then verify the proof given by the prover also based on the CRS from the trusted third party. Note here that the argument system is designed such that the CRS does not need to be generated every time the prover needs to prove some statement but instead can be reused. Figure 3 shows a general outline of the CRS model non-interactive zero knowledge argument system.

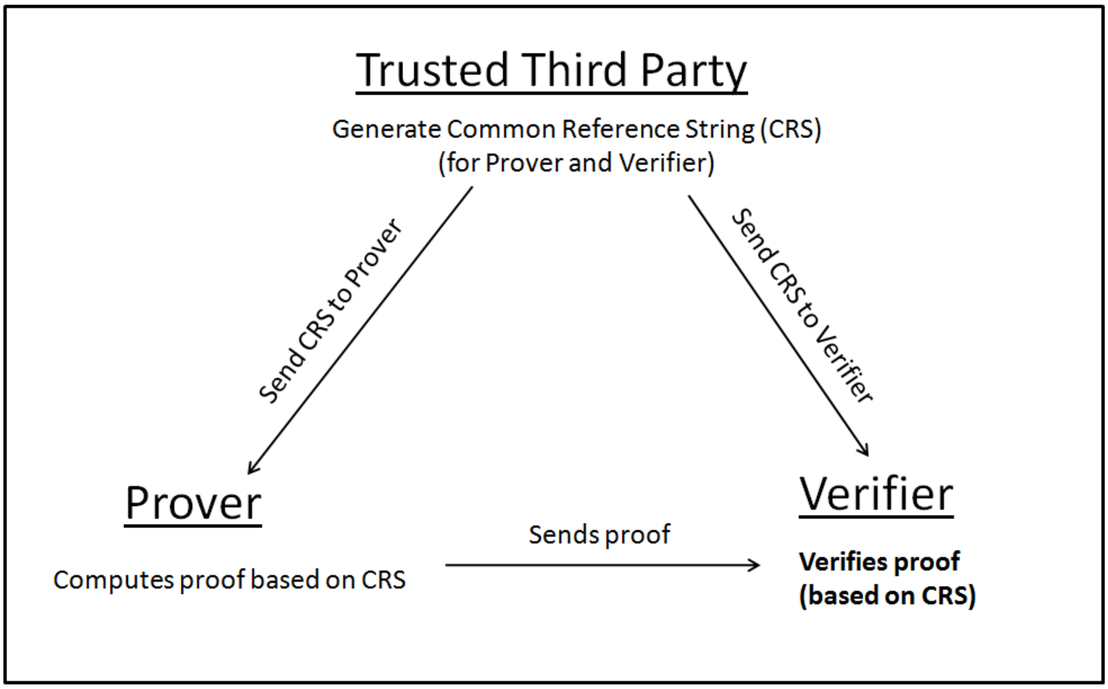


Figure 3: Non interactive argument system in CRS Model

2.1.3 Verifiable Computation

A verifiable computation scheme allows a computationally limited client to outsource to a worker the evaluation function F on input u. The client can then verify the correctness of the returned result $F(u)$ while performing less work than required for the function evaluation [PGHR13]. In real life where computational power are not symmetric, verifiable computation can be very useful. This scheme allows reliable allocation of heavier computation to more powerful worker so that even client who has weaker computational power is able to finish its computation faster. In that sense, the outsourced worker can't always be fully trusted, which explains why there would be a need for a way to verify that the worker computes the function correctly. What needs to be noted that for this to make sense, the verification should be of less effort compared to computing the function itself, or else the client can just compute the function itself. A good example of verifiable computation uses is the Amazon CPU time renting service (http://aws.amazon.com/ec2/).

The service allows user/client to run programs in their cloud server, at the cost of some price. As the client, it would make sense to be able to verify that the program actually runs smoothly without a hitch and this is where verifiable computation comes in.

It can be seen that verifiable computation scheme resembles the zero knowledge argument system to some degree. To match up the zero knowledge argument system's parties with verifiable computation, the outsourced worker here acts as a prover while the computationally limited client acts as the verifier. The difference here is that the "proof" given by the outsourced worker doesn't have to exhibit a zero knowledge behavior, i.e. the proof might leak some information about the output of the function itself (assuming that the output itself is not known/sent yet). The other behavior from the zero knowledge argument system, namely completeness and soundness applies to the verifiable computation scheme itself.

As stated before, the implementation of this work is the non-adaptive version of the zero knowledge argument system based on [Lip13]. The difference between the non-adaptive version and the adaptive version is that in the adaptive version, the CRS remains unchanged even when the problem (which in this case, the circuit) changes. This is due to the fact of the usage of universal circuit that takes in a circuit itself as the input (explained in a later section). This means by using this universal circuit approach the circuit/function itself need not to be known in the first place and the CRS can be still computed beforehand. On the other hand in the non-adaptive version, if the circuit changes then the CRS needs to be recomputed. The consequence is that then the function/circuit itself needs to be known first before any CRS construction can be done.

Although the implementation in this work is a non-adaptive version of zero knowledge argument system (which is less preferable compared to the adaptive version), it is sufficient for the usage of the verifiable computation scheme which as explained is a very common scenario nowadays. The reason that the non-adaptive version is sufficient is that the function is already known by the client/user. The implementation in this thesis extends the behavior of the verifiable computation scheme by adding the zero knowledge behavior on top of it. It is therefore very useful and hopefully can be used for such schemes as needed in the future.

2.1.4 Circuit Satisfiability Problem

Circuit satisfiability (SAT) problem is the problem of determining whether a boolean circuit has an input assignment that makes the value of the final output to be true. A boolean circuit is a mathematically precise computational model of the digital circuit where each input and output has a true or false value (in constrast with having a supply voltage or not in the digital circuit). A boolean circuit can be viewed as a directed acyclic graph where vertices corresponds to the gates and edges corresponds to the wire. In this work, the boolean circuit that is discussed will be restricted to circuits that have gates with maximum fan-in 2 and maximum fan-out 2, which means that every gate will have a maximum of two input and two output values. Viewing it again from graph point of view, this means that each vertex will have at most in-degree 2 and out-degree 2. Some example of the gates in the circuit are the AND gate, OR gate, and NAND gate. These gates represent the boolean function AND, OR, and NAND respectively. Note that instead of having one output as in the boolean function, a gate with fan-out 2 would just have two outputs with the same value that enter different gates.

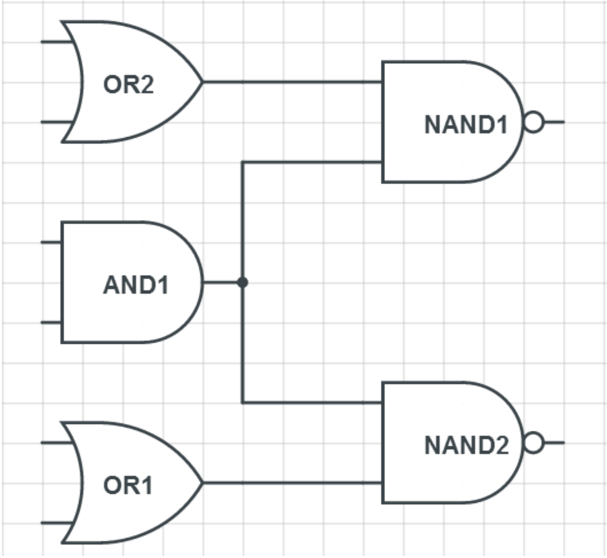


Figure 4: Boolean circuit example

The circuit SAT problem is one of the problems which is often used as the problem statement to be proven in a zero knowledge argument system. What the prover tries to prove in this case is that he/she knows the satisfying assignment that makes the final output of the circuit to be true. The reason behind it is that circuit SAT problem is known to be an NP-complete problem. An NP problem is a problem which can be solved in polynomial time by a non-deterministic Turing machine. An NP-complete problem is in turn an NP problem which is as hard as all the other problem in NP, in the sense that every NP problem can be reduced to the form of an NP-complete problem. By being able to construct a zero knowledge argument system to prove an NP-complete problem would mean that the same argument system would also work for any other NP problem (with some polynomial slowdown) by first reducing it into the NP-complete problem. The NP-complete problem that is chosen in this case is the circuit SAT problem. Furthermore, circuit SAT problem is also a natural language for the verifiable computation usage compared to other language in NP-complete class. The reasoning is that the function that is going to be verified in a verifiable computation protocol can be represented naturally in the form of a circuit. For each function, there exists circuits that will produce the same output as the function given the same input.

2.2 Representing Circuit SAT Problem using Span Programs

In this work, the circuit SAT problem is used as the problem statement to be proven. Before going into the argument system itself, circuit that is going to be proven needs to be encoded into appropriate form, in this case using span programs and quadratic span programs. There are two main component in representing the circuit. First, what the prover needs to do is to prove that each of the boolean gate in the circuit is consistent, meaning that the output of the gate is consistent with the boolean function applied to the input of the gate. Second, for each wire connecting up the output of a gate with the input of other gate, the prover needs to prove that those two value are actually the same. The first part of the proof is done by using a gate checker, while the second part is done by using a wire checker. Finally, both the gate checkers and the wire checkers are combined into one quadratic span program which is the circuit checker. Figure 5 shows illustration of the gate and wire checker. The upcoming explanation on the construction of the circuit checker is simplified for easier understanding. Interested reader should refer to [Lip13] for more background on this.

2.2.1 Gate Checker

The first part of the encoding of the circuit is constructing a gate checker for each of the gate in the circuit. In order to construct a gate checker, the argument system of [Lip13] uses span program [KW93] to represent the checker. A span program $P = (\mathbf{t}, \mathcal{V}, \varrho)$ contains a target vector $\mathbf{t}$ with d elements, an $m \times d$ matrix $\mathcal{V}$, and a labelling map ϱ which maps every row of $\mathcal{V}$ into a literal x_ι or $\bar{x}_\iota$ where ι is one of the indexes of the variable in the specified gate that is going to be checked by the gate checker. Figure 6 shows a span program example. The details of the components in this example are

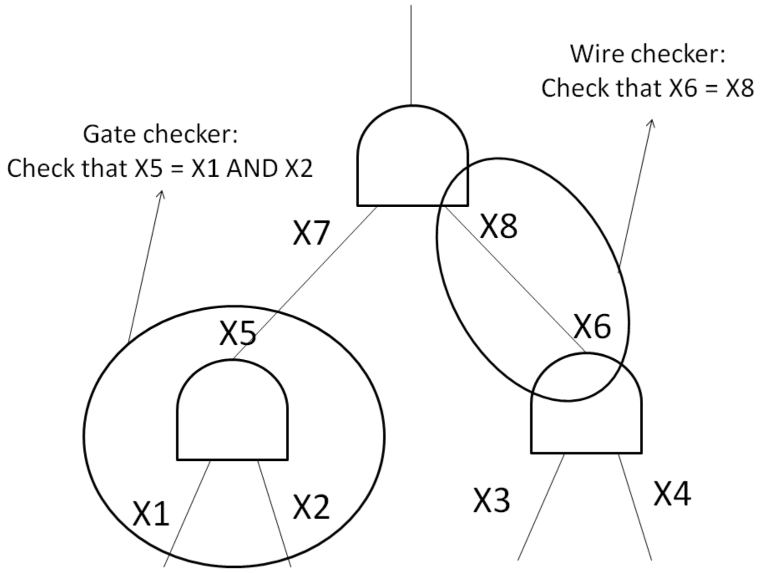


Figure 5: Gate and wire checker illustration

$$\begin{aligned} d &= 3 \\ m &= 6 \\ t &= (1,1,1) \\ \mathcal{V}_0 &= (1,0,0) \\ \mathcal{V}_1 &= (0,1,0) \\ \mathcal{V}_2 &= (1,1,0) \\ \mathcal{V}_3 &= (0,0,1) \\ \mathcal{V}_4 &= (0,0,1) \\ \mathcal{V}_5 &= (0,0,1) \end{aligned}$$

with labelling $\varrho = \{\mathcal{V}_0 \to x_1, \mathcal{V}_1 \to x_2, \mathcal{V}_2 \to x_3, \mathcal{V}_3 \to \bar{x}_1\, \mathcal{V}_4 \to \bar{x}_2, \mathcal{V}_5 \to \bar{x}_3\}$. The solution to the gate checker span program is directly translated from the value of the variables of the gate itself. For example, in Figure 6, the variables concerned are x_1, x_2, and x_3. An assignment $\mathbf{u}$ consists of the assignment of value for all the variables in the gate. Suppose that the assignment value from the example is $\mathbf{u} = \{x_1 = 1, x_2 = 0, x_3 = 1\}$. The assignment is said to be the solution of a span program iff $\mathbf{t}$ is in the span of the sum of row vector $\mathcal{V}_i$ based on the labelling ϱ

from the assignment $\mathbf{u}$. Based on the assignment $\mathbf{u} = \{x_1 = 1, x_2 = 0, x_3 = 1\}$, define the corresponding row vector set as $\mathcal{V}_{\mathbf{u}} = \{\mathcal{V}_0, \mathcal{V}_4, \mathcal{V}_2\}$. The row vector with the negation of the variable is chosen when the assignment variable is set to 0, while its non negation row vector is chosen otherwise. By definition of linear span, this means that

$$\exists a_i, \sum_{\mathcal{V}_i \in \mathcal{V}_{\mathbf{u}}} a_i \mathcal{V}_i = t$$

From the example, the sum of the row vector from $\mathcal{V}_{\mathbf{u}}$ by choosing $a_0 = 0, a_4 = 1, a_2 = 1$ yields the row vector (1, 1, 1), which is the target vector $\mathbf{t}$. Note that it is allowed to assign a_i to 0 for some or all the chosen row vector (meaning that this row vector itself isn't used in the end), but it is not allowed to use any other row vector than otherwise reflected in the assignment $\mathbf{u}$ in the sum itself.

$$\left(\begin{array}{c|ccc} \mathbf{t} & 1 & 1 & 1 \\ \hline x_1 & 1 & 0 & 0 \\ x_2 & 0 & 1 & 0 \\ x_3 & 1 & 1 & 0 \\ \bar{x}_1 & 0 & 0 & 1 \\ \bar{x}_2 & 0 & 0 & 1 \\ \bar{x}_3 & 0 & 0 & 1 \end{array}\right)$$

Figure 6: Span program example

After knowing the structure of a span program, the next step is to know how to construct the gate checker span program from a boolean gate. Notice that in the proof, the aim is to prove that the output of the boolean gate is consistent with the input. For example, if the gate is a NAND gate, then the checker should check whether the output is indeed the NAND of the two input. There is an exception to this which is in the last output gate of the circuit. As the proof is about the satisfiability of the circuit, instead of checking that the output has to be consistent with the input, what needs to be done is to check that the output is indeed 1 (which actually simplifies the span program more). The example span program shown earlier on Figure 6 is actually a gate checker for the NAND gate, where the span program only have a solution iff $x_3 = (x_1 \barwedge x_2)$. The case analysis is as follows.

- $x_1 = x_2 = x_3 = 0$ does not give a solution,
- $x_1 = x_2 = 0$ and $x_3 = 1$ gives a solution with $a_2 = 1$, a_3 arbitrary, and $a_4 = 1 - a_3$,

$$\left(\begin{array}{r|ccc} \mathbf{t} & 1 & 1 & 1 \\ \hline x & 0 & 1 & 0 \\ y_1 & 0 & 0 & 1 \\ y_2 & 1 & 0 & 0 \\ \bar{x} & 1 & 0 & 0 \\ \bar{y}_1 & 0 & 1 & 0 \\ \bar{y}_2 & 0 & 0 & 1 \end{array}\right)$$

Figure 7: Fork gate checker

- $x_1 = x_3 = 0$ and $x_2 = 1$ does not give a solution,
- $x_1 = 0$ and $x_2 = x_3 = 1$ gives a solution with $a_2 = 1$, $a_1 = 0$, $a_3 = 1$,
- $x_1 = 1$ and $x_2 = x_3 = 0$ does not give a solution,
- $x_1 = x_3 = 1$ and $x_2 = 0$ gives a solution with $a_0 = 0$, $a_2 = 1$, $a_4 = 1$,
- $x_1 = x_2 = 1$ and $x_3 = 0$ gives a solution with $a_0 = 1$, $a_1 = 1$, $a_4 = 1$,
- $x_1 = x_2 = x_3 = 1$ does not give a solution.

In this work, it is assumed that all the gates of the circuit will be of NAND gates (this does not reduce the power of the argument system, as it is well known that all circuit with different gates can be transform into circuit with only NAND gates). Apart from that, a fork gate that computes $y_1 \leftarrow x$, $y_2 \leftarrow x$ is also needed to reduce the fanout. The reasoning behind the reduction of fanout has to do with the size of the wire checker later on, which in turn will affect the efficiency of the final argument system. By having a small fanout, the efficiency of the argument system will be better. For more details on this, see [Lip13]. Note that in his paper, Lipmaa stated that limiting the circuit to have fanout at most 3 might be more efficient in some cases. Although this thesis focuses on circuit with fanout at most 2, future work should also test out the efficiency of the implementation on circuit with different fanout. The span program of the fork gate checker is presented in Figure 7.

After constructing the gate checker for each of the gate in the circuit, all the gate checkers need to be aggregated into one span program. The aggregation is done by doing an AND composition between the gate checker. Suppose that there are two span programs $P_1 = (\mathbf{t}^1, \mathcal{V}^1, \varrho^1)$ and $P_2 = (\mathbf{t}^2, \mathcal{V}^2, \varrho^2)$ with size m_1, d_1 and m_2, d_2 respectively. The AND composition of these two span program is $P = (\mathbf{t}, \mathcal{V}, \varrho)$ with size $m = m_1 + m_2, d = d_1 + d_2$. $\mathbf{t}$ will just be a concatenation of the two target vectors $\mathbf{t}^1$ and $\mathbf{t}^2$. Next, define $\mathcal{V}$ as follows

$$\mathcal{V}_i = \begin{cases} (\mathcal{V}_i^1, 0_{d_2}) & 0 \leq i < m_1 \\ (0_{d_1}, \mathcal{V}_{i-m_1}^2) & m_1 \leq i < m_1 + m_2 \end{cases}$$

where 0_x represents x amount of zeros. The new labelling ϱ will be consistent with the previous labelling, meaning that if ϱ^1 has a mapping $\mathcal{V}_i^1 \rightarrow x_\iota$, then ϱ has a mapping $\mathcal{V}_i \rightarrow x_\iota$, and if ϱ^2 has a mapping $\mathcal{V}_i^2 \rightarrow x_{\iota'}$ then ϱ has a mapping $\mathcal{V}_{m_1+i} \rightarrow x_{\iota'}$.

2.2.2 Wire Checker and Quadratic Span Program

After defining a gate checker for each of the gate in the circuit, a wire checker for each of the wire also needs to be constructed in order for the prover to prove that the assignments it makes are consistent. As this work only focuses on gates with fan-out at most 2, this means that each wire will have at most 3 different variables associated with it. Previously on Figure 5, the gate that is shown has fan-out 1. Figure 10 shows the illustration of wire checker with fan-out 2.

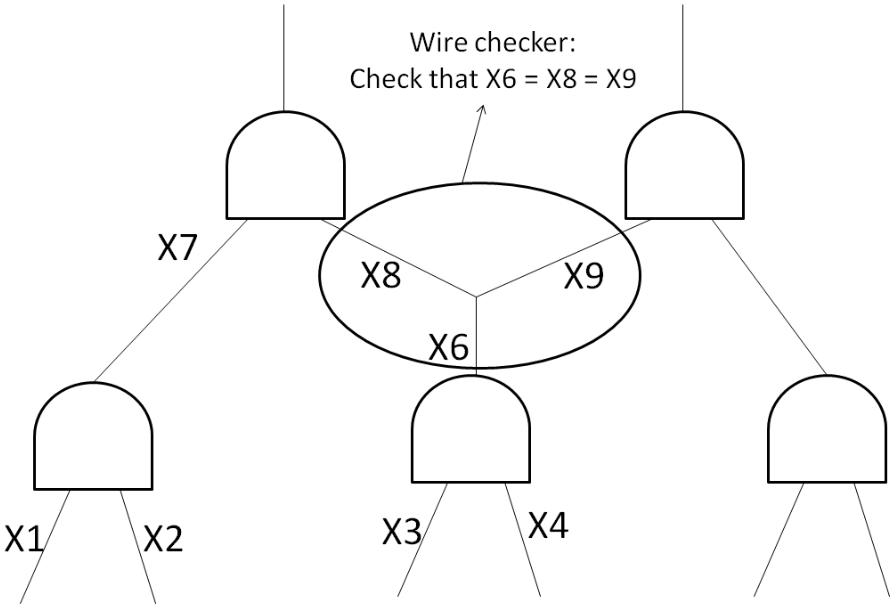


Figure 8: Wire checker illustration for gates with fan-out 2

A quadratic span program is used in order to construct the wire checker. A quadractic span program $P = (\mathbf{v_0}, \mathbf{w_0}, \mathcal{V}, \mathcal{W}, \varrho)$ [GGPR13] consists of two target

$$\left(\begin{array}{c|c|cccccc}
 & \mathbf{v_0} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
 & x_1 & 1 & 0 & -1 & 0 & 0 & 0 \\
\mathcal{V} & x_2 & 0 & 1 & 2 & 0 & 0 & 0 \\
 & \bar{x}_1 & 0 & 0 & 0 & 1 & 0 & -1 \\
 & \bar{x}_2 & 0 & 0 & 0 & 0 & 1 & 2 \\ \hline
 & \mathbf{w_0} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
 & x_1 & 0 & 0 & 0 & 1 & 0 & -1 \\
\mathcal{W} & x_2 & 0 & 0 & 0 & 0 & 1 & 2 \\
 & \bar{x}_1 & 1 & 0 & -1 & 0 & 0 & 0 \\
 & \bar{x}_2 & 0 & 1 & 2 & 0 & 0 & 0
\end{array}\right)$$

Figure 9: Quadratic span program example

vectors $\mathbf{v_0}$ and $\mathbf{w_0}$ with d elements respectively, two $m \times d$ matrices $\mathcal{V}$ and $\mathcal{W}$, and a labelling ϱ which maps every row of $\mathcal{V}$ and $\mathcal{W}$ into a literal x_ι or $\bar{x}_\iota$ where ι is one of the indexes of the variable in the specified wire that is going to be checked by the wire checker. It can be seen that the components of the quadratic span program are very similar to that of the span program, whereas instead of having only one target vector and matrix, the quadratic span program has two of them. An assignment $\mathbf{u}$ is said to be the solution for a quadractic span program iff the corresponding row vector $\mathcal{V}_\mathbf{u}$ and $\mathcal{W}_\mathbf{u}$ chosen based on the assignment $\mathbf{u}$ (defined similarly as in span program) satisfies

$$\exists a_i, b_i, ((\sum a_i \mathcal{V}_i) - \mathbf{v_0}) \circ ((\sum b_i \mathcal{W}_i) - \mathbf{w_0}) = 0$$

where $\mathbf{x} \circ \mathbf{y}$ denotes the pointwise (Hadamard) product of $\mathbf{x}$ and $\mathbf{y}$. The pointwise (Hadamard) product of $\mathbf{x} = (x_1, \cdots, x_n)$ and $\mathbf{y} = (y_1, \cdots, y_n)$ is the vector $\mathbf{z} = (x_0.y_0, \cdots, x_n.y_n)$. Figure 9 shows a quadratic span program example, which in fact is also a wire checker for gates with fan-out 1. The size of are $m = 4$ and $d = 6$ respectively. As both target vectors are all zero vectors, the previous equation simplifies to

$$\exists a_i, b_i, (\sum a_i \mathcal{V}_i) \circ (\sum b_i \mathcal{W}_i) = 0$$

Notice that both $\mathcal{V}$ and $\mathcal{W}$ are in the form of

$$\begin{pmatrix} G & 0 \\ 0 & G \end{pmatrix}$$

where each element is a sub-matrix of size $\dfrac{m}{2} \times \dfrac{d}{2}$. In fact, G is a generator matrix of the $[2D-1, D, D]$ systematic Reed-Solomon linear error correcting code where

$$\left(\begin{array}{cc|ccccccccc}
 & \mathbf{v_0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
 & x_1 & 1 & 0 & 0 & 1 & 3 & 0 & 0 & 0 & 0 \\
 & x_2 & 0 & 1 & 0 & -3 & -8 & 0 & 0 & 0 & 0 \\
\mathcal{V} & x_3 & 0 & 0 & 1 & 3 & 6 & 0 & 0 & 0 & 0 \\
 & \bar{x}_1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 3 \\
 & \bar{x}_2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -3 & -8 \\
 & \bar{x}_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 & 6 \\
\hline
 & \mathbf{w_0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
 & x_1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 3 \\
 & x_2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -3 & -8 \\
\mathcal{W} & x_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 3 & 6 \\
 & \bar{x}_1 & 1 & 0 & 0 & 1 & 3 & 0 & 0 & 0 & 0 \\
 & \bar{x}_2 & 0 & 1 & 0 & -3 & -8 & 0 & 0 & 0 & 0 \\
 & \bar{x}_3 & 0 & 0 & 1 & 3 & 6 & 0 & 0 & 0 & 0
\end{array}\right)$$

Figure 10: Wire checker for gates with fan-out 2

$D = 2$ [Rot06]. What this means is that every resulting vector (that has size 3) that is generated by multiplying any non-zero initial vector (that has size 2) with the generator matrix will have non-zero values in at least 2 positions. For example, suppose that there are two initial vectors $\mathbf{a} = (0, 0)$ and $\mathbf{b} = (0, 1)$. The resulting vector by multiplying $\mathbf{a}$ with the generator matrix G is $\mathbf{R_a} = (0, 0, 0)$. The resulting vector gotten from the second vector $\mathbf{b}$ is then $\mathbf{R_b} = (0, 1, 2)$. The claim that the resulting vector will have non zero values in at least $D = 2$ positions holds true in this case. In fact, this holds true for any other two arbitrary initial vectors $\mathbf{a}$ and $\mathbf{b}$. This work will not discuss the theory behind the error correction and proof of correctness. Interested reader should refer to [Lip13]. Figure 10 shows the wire checker for gates with fan-out 2.

Now, one might wonder how does using this generator matrix helps the proof of the wire checker. Notice that if the assignment of the variables are indeed consistent, then all of the variables in one wire checker should have the same value (i.e. all are 0 or all are 1). When observed from the quadratic span program that is given above, this means that either the first $\frac{m}{2}$ rows or the last $\frac{m}{2}$ rows are chosen for each of $\mathcal{V}$ and $\mathcal{W}$. Either way, this would mean that if the assignments are actually consistent, either the first or the last $\frac{d}{2}$ elements of $\sum a_i\mathcal{V}_i$ are 0. For example, if the quadratic span program on Figure 9 has a consistent assignment, then the value of x_1 should be the same as x_2. Let's assume that both value are true. If both are true, then only the first two row from each $\mathcal{V}$ and $\mathcal{W}$ is chosen. Then, no matter what value a_i and b_i is, the last 3 elements of $\sum a_i\mathcal{V}_i$ will be 0 while the first 3 elements of $\sum b_i\mathcal{W}_i$ will be 0. The pointwise (Hadamard) product of those two will then be the zero vector. As one might have noticed, $\mathcal{W}$ is just a copy of $\mathcal{V}$ with each of the labelling between $\mathcal{V}_i \rightarrow x_\iota$ and $\mathcal{V}_j \rightarrow \bar{x}_\iota$ swapped (and reordered). This means that if $\sum a_i\mathcal{V}_i$ has the first $\frac{d}{2}$ elements being 0, then $\sum b_i\mathcal{W}_i$ would have the last $\frac{d}{2}$ elements being 0, and vice versa. By multiplying those two together, then one would indeed have 0 as the end result.

On the other hand, suppose that the assignment of the variables are not consistent. This means that there is at least one row vector $\mathcal{V}_i$ and $\mathcal{W}_i$ chosen for $\mathcal{V}_\mathbf{u}$ and $\mathcal{W}_\mathbf{u}$ at both the first $\frac{m}{2}$ rows and the last $\frac{m}{2}$ rows. Notice that, in the example before, when the initial vector is (0, 0) the resulting vector are also an all zero vector (0, 0, 0). By the definition that all resulting vector will have at least $D = 2$ different values, this means that all the other resulting vectors will have non zero value in at least 2 positions. As $\mathcal{V}_\mathbf{u}$ and $\mathcal{W}_\mathbf{u}$ has row vector in both the first $\frac{m}{2}$ rows and the last $\frac{m}{2}$ rows, this means that for each part, the initial vector is not an all zero vector. Each of the resulting vector will have non zero value in at least 2 out of 3 positions, and therefore when multiplied, there will be at least one position that will stay non zero. This means that the equation for the quadratic span program cannot be satisfied, thus not an assignment that is the solution for the wire checker. One might think that it is still possible to cheat by setting the proper coefficient a_i or b_i to 0 instead, but it will be seen later that this is not possible as the wire checker will be combined with the gate checker with consistent labelling.

After constructing wire checker for each of the wire, all the wire checkers also need to be aggregated into one quadratic span program. The method of aggregating is similar to that of the gate checker, which is by doing an AND composition on the wire checkers. The only difference here is that instead of just one target vector and one matrix, each wire checker has two target vector and two matrix. The rest of the aggregating process stays the same.

2.2.3 Circuit Checker

After having an aggregated gate checker and an aggregated wire checker for the circuit, the next step is to combine them both to become a full circuit checker. The resulting circuit checker is also a quadratic span program. The step to construct such a circuit checker is as follows. Suppose that the resulting aggregated gate checker is $P_g = (\mathbf{t}_g, \mathcal{V}^g, \varrho_g)$. Construct another span program $P'_g = (\mathbf{t}_g, \mathcal{W}^g, \varrho'_g)$ where $\mathcal{W}^g = \mathcal{V}^g$. ϱ'_g will have the "swapped" version of ϱ_g, meaning that if ϱ_g has a mapping $\mathcal{V}_i \rightarrow x_\iota$, then ϱ'_g has a mapping $\mathcal{W}_i \rightarrow \bar{x}_\iota$, and if ϱ_g has a mapping $\mathcal{V}_i \rightarrow \bar{x}_\iota$, then ϱ'_g has a mapping $\mathcal{W}_i \rightarrow x_\iota$. Suppose also that the resulting wire checker is $P_w = (\mathbf{0}, \mathbf{0}, \mathcal{V}^w, \mathcal{W}^w, \varrho_w)$. Construct the quadratic span program circuit checker $P = (\mathbf{v_0}, \mathbf{w_0}, \mathcal{V}, \mathcal{W}, \varrho)$ as

$$\left(\begin{array}{c} \mathbf{v_0} \\ \mathcal{V} \\ \hline \mathbf{w_0} \\ \mathcal{W} \end{array}\right) = \left(\begin{array}{ccc} \mathbf{t}_g & \mathbf{1} & \mathbf{0} \\ \mathcal{V}^g & 0_{m_g \times d_g} & \mathcal{V}^w \\ \hline \mathbf{1} & \mathbf{t}_g & \mathbf{0} \\ 0_{m_g \times d_g} & \mathcal{W}^g & \mathcal{W}^w \end{array}\right) .$$

The labelling ϱ is constructed from the proper reordering of the labelling between the gate checker and the wire checker. This is done by combining/concatenating the row with the same literal mapping. For example, if ϱ_g has a mapping of $\mathcal{V}_i^g \to x_\iota$ and ϱ_w has a mapping of $\mathcal{V}_i^w \to x_\iota$, then the new labelling ϱ will have a mapping $(\mathcal{V}_i^g, 0_{d_g}, \mathcal{V}_i^w) \to x_\iota$. For proof behind the construction, see [Lip13].

2.3 From Span Programs to Polynomials

After knowing how to represent a circuit into one circuit checker, it can be seen that the circuit checker is just one big quadratic span program. The question now is what the prover sends to the verifier in this new encoding in order to prove that he knows the satisfying assignment. Remind that an assignment $\mathbf{u}$ of the circuit is also a solution for a quadratic span program $P = (\mathbf{v_0}, \mathbf{w_0}, \mathcal{V}, \mathcal{W}, \varrho)$ iff the chosen row vector satisfies

$$\exists a_i, b_i, ((\sum a_i \mathcal{V}_i) - \mathbf{v_0}) \circ ((\sum b_i \mathcal{W}_i) - \mathbf{w_0}) = 0$$

It is worth to note that this requirement is automatically enforced in the constructed circuit checker, meaning as long as one is able to find $\mathbf{a}$ and $\mathbf{b}$ that makes the circuit checker to accept, the requirement will be satisfied for sure. What the prover needs to do now is then to show that he/she knows a_i's and b_i's such that the above equation holds. However, this means that the amount of data that the prover needs to send is propotional in terms of the size of the span program, m. The final goal of the argument system is to bring the amount of data that needs to be sent to a constant, meaning that the size of the proof is always the same regardless of the size of the circuit, while still preserving the security property and privacy of the argument system.

In order to do this, the circuit checker needs to be modified. Instead of looking each of the target vector and row vector just as a vector, each of them are instead transformed into a polynomial $p(x)$. The polynomial $p(x)$ is interpolated from a set of points that are constructed based on the vector. For example, suppose there's a row vector $\mathcal{V}_i = (\mathcal{V}_{i0}, ..., \mathcal{V}_{id-1})$. Define the polynomial $p(x)$ as the polynomial interpolation of $\mathcal{V}_i$ where $p(r_j) = \mathcal{V}_{ij}$ for all $j \in [0, ..., d-1]$. r_j can be chosen arbitrarily, but smart choice of r_j will affect the efficiency of the arithmetics in the computation. By doing this for every target vector and row vector $\mathcal{V}_i$, $\mathcal{W}_i$, the new polynomial span program is then $P = (\mathbf{v_0}(x), \mathbf{w_0}(x), \mathcal{V}(x), \mathcal{W}(x), \varrho)$.

The idea behind how representing the span program as polynomials helps in compacting the proof is as follows. Instead of sending the value if a_i's and b_i's, compute the new polynomial $v(x) = \sum a_i \mathcal{V}_i(x)$, and $w(x) = \sum b_i \mathcal{W}_i(x)$. Notice that it now holds that $(v(x) - \mathbf{v_0}(x)) \cdot (w(x) - \mathbf{w_0}(x)) = \hat{\mathbf{Z}}(x)h(x)$ for some polynomial $h(x)$ where $\hat{\mathbf{Z}}(x) = \prod(x - r_j)$. Then, instead of sending $v(x)$ and $w(x)$ as the proof (which still has size that depends on the size of the circuit), the prover can instead send an encoding of $v(\sigma)$ and $w(\sigma)$ where σ is a secret evaluation point. The new element $v(\sigma)$ and $w(\sigma)$ is just one element each, meaning that no matter what degree the polynomial $v(x)$ and $w(x)$ is, they will stay the same size. However, with this modification there are other things that needs to be checked, such as making sure that $v(\sigma)$ is indeed from the span of $\mathcal{V}_i(x)$, sending the information needed about $h(x)$ and so on. The zero knowledge property also needs to be maintained. The detailed explanation on the whole process can be found in [Lip13].

3 Argument System and Implementation

This section provides the non-interactive zero knowledge argument system based on [Lip13]. The argument system provided is the non-adaptive version where the circuit is not the universal circuit. After providing the argument system, the implementation of the argument system is then given. Several details to improve the efficiency of the implementation are explained.

3.1 Notation

Some of the notations used in the argument system are explained as follows.

1. var $\leftarrow \mathbb{F}$, means that the variable is assigned a value randomly chosen from a chosen finite field. In the argument system, $\mathbb{F}$ is equal to $\mathbb{Z}_q$, where q is the order of the group.

2. $[\![a]\!]_i$, means an encoding of a in group $\mathbb{G}_i$ which in this implementation is represented by an exponentiation, $[\![a]\!]_i = g_i^a$ for a fixed g_i.

3. $\hat{\mathbf{e}}(a, b)$, means a bilinear pairing of a and b. A pairing [BF01] is a map $\hat{\mathbf{e}} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_3$. A bilinear pairing has the property $\forall a, b \in \mathbb{G}_1, \mathbb{G}_2 : \hat{\mathbf{e}}(aP, bQ) = \hat{\mathbf{e}}(P, Q)^{ab}$ where P and Q is the generator in group $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. In addition, a cryptographically useful bilinear map is assumed to satisfy certain security assumptions. See [Lip13] for the assumptions required there.

3.2 Non-Interactive Zero Knowledge Argument System

The argument system mainly consists of 3 part, the CRS generation, the prover's computation, and finally the verifier's verification. The argument system is directly taken from [Lip13] by removing the adaptive part from the argument system. The non-interactive zero knowledge argument system is defined formally as the triple of algorithms:

- CRS generation $\mathcal{G}(1^\kappa, n)$, with input κ as the security parameter and circuit size n. The output of this algorithm is the prover's CRS crs and verifier's CRS vcrs.

- Prover's computation $\mathcal{P}(\mathsf{crs}; C, \mathbf{u})$, with input of the prover's CRS crs, the circuit description C, and the circuit's satisfying assignment $\mathbf{u}$. The output of this algorithm is the zero knowledge argument π.

- Verifier's verification $\mathcal{V}(\mathsf{vcrs}; C, \pi)$, with input of verifier's CRS vcrs, the circuit description C, and argument π. The output of this algorithm is a boolean value signifying whether the verification is successful.

The details of each of the algorithm are as follows. Comments are added on some descriptions to highlight the total amount of computation needed in that specific operation. See [Lip13] for more details.

3.2.1 CRS generation $\mathcal{G}(1^\kappa, n)$

```
Let $C_n$ be a circuit of size $|C| = n$;
Let $P := (\mathbf{v_0}, \mathbf{w_0}, \mathcal{V}, \mathcal{W}, \varrho)$ be the circuit checker for $C_n$ with size $m$ and $d$, $\mathcal{V} = (\mathcal{V}_\mathbf{1}, \ldots, \mathcal{V}_\mathbf{m})$, and
    $\mathcal{W} = (\mathcal{W}_\mathbf{1}, \ldots, \mathcal{W}_\mathbf{m})$;
Choose $d$ roots $r_i$ for $i \in [1, d]$;
Compute $\hat{\mathbf{v}}_0(X) = \sum_{j=0}^{d-1} \bar{v}_{0j} X^j$ by polynomial interpolation from $\mathbf{v_0}$;    /* $\Theta(d \log^2 d)$ $\mathbb{F}$-ops */
Compute $\hat{\mathbf{w}}_0(X) = \sum_{j=0}^{d-1} \bar{w}_{0j} X^j$ by polynomial interpolation from $\mathbf{w_0}$;    /* $\Theta(d \log^2 d)$ $\mathbb{F}$-ops */
$\alpha, \sigma, \beta_v, \beta_w, \gamma \leftarrow \mathbb{F}$;
for $i \leftarrow 0$ to $m$ do compute $(\llbracket \hat{\mathbf{v}}_i(\sigma) \rrbracket_2, \llbracket \hat{\mathbf{w}}_i(\sigma) \rrbracket_2)$;
$\hat{\mathbf{Z}}(\sigma) \leftarrow \prod_{i=1}^{d} (\sigma - r_i)$;
$(V_0, V_0^*) \leftarrow (\llbracket \hat{\mathbf{v}}_0(\sigma) \rrbracket_2, \llbracket \alpha \hat{\mathbf{v}}_0(\sigma) \rrbracket_2)$;
for $i \leftarrow 1$ to $2$ do $(W_0[i], W_0^*[i]) \leftarrow (\llbracket \hat{\mathbf{w}}_0(\sigma) \rrbracket_i, \llbracket \alpha \hat{\mathbf{w}}_0(\sigma) \rrbracket_i)$;
for $i \leftarrow 1$ to $2$ do $(Z[i], Z^*[i]) \leftarrow (\llbracket \hat{\mathbf{Z}}(\sigma) \rrbracket_i, \llbracket \alpha \hat{\mathbf{Z}}(\sigma) \rrbracket_i)$;
Let
    $$\mathsf{crs} \leftarrow \begin{pmatrix} P, (r_i)_{i \in [d]}, \left(\llbracket \sigma^j \rrbracket_i, \llbracket \alpha \sigma^j \rrbracket_i\right)_{i \in \{1,2\}, j \in [0, d-1]}, \hat{\mathbf{v}}_0(X), \hat{\mathbf{w}}_0(X), V_0, V_0^*, W_0[1], W_0^*[1], Z[1], Z^*[1], \\ Z[2], Z^*[2], (\llbracket \beta_v \hat{\mathbf{v}}_i(\sigma) \rrbracket_2, \llbracket \beta_w \hat{\mathbf{w}}_i(\sigma) \rrbracket_2)_{i \in [m]}, \llbracket \beta_v \hat{\mathbf{Z}}(\sigma) \rrbracket_2, \llbracket \beta_w \hat{\mathbf{Z}}(\sigma) \rrbracket_2 \end{pmatrix};$$
$\mathsf{vcrs} \leftarrow \left(P, (\llbracket 1 \rrbracket_i, \llbracket \alpha \rrbracket_i)_{i \in \{1,2\}}, V_0, W_0[1], Z[1], \llbracket \gamma \rrbracket_1, \llbracket \beta_v \gamma \rrbracket_1, \llbracket \beta_w \gamma \rrbracket_1\right)$;
The trapdoor is $(\sigma, \alpha, \beta_v, \beta_w)$;
```

3.2.2 Prove $\mathcal{P}(\mathsf{crs}; C, \mathbf{u})$

```
Evaluate $C_n$ to obtain $\mathbf{a}, \mathbf{b} \in \mathbb{F}^m$;
$\mathbf{v} \leftarrow \sum_i a_i \mathbf{v_i}$, $\mathbf{w} \leftarrow \sum_{i=1}^m b_i \mathbf{w_i}$
Compute $\hat{\mathbf{v}}(X) = \sum_{j=0}^{d-1} \bar{v}_j X^j$ by polynomial interpolation from $\mathbf{v}$;      /* $\Theta(d \log^2 d)$ $\mathbb{F}$-ops */
Compute $\hat{\mathbf{w}}(X) = \sum_{j=0}^{d-1} \bar{w}_j X^j$ by polynomial interpolation from $\mathbf{w}$;      /* $\Theta(d \log^2 d)$ $\mathbb{F}$-ops */
$\hat{\mathbf{v}}^\dagger(X) \leftarrow \hat{\mathbf{v}}_0(X) + \hat{\mathbf{v}}(X)$; $\hat{\mathbf{w}}^\dagger(X) \leftarrow \hat{\mathbf{w}}_0(X) + \hat{\mathbf{w}}(X)$;      /* $\Theta(d)$ additions */
$\hat{\mathbf{Z}}(X) \leftarrow \prod_{i=1}^d (X - r_i)$;
$\hat{\mathbf{h}}(X) \leftarrow \hat{\mathbf{v}}^\dagger(X) \cdot \hat{\mathbf{w}}^\dagger(X) / \hat{\mathbf{Z}}(X) \in \mathbb{F}^{d-2}[X]$;      /* $\Theta(d \log d)$ $\mathbb{F}$-operations */
$(V, V^*) \leftarrow \prod_{j=0}^{d-1} ([\![\sigma^j]\!]_2^{\bar{v}_j}, [\![\alpha\sigma^j]\!]_2^{\bar{v}_j})$;      /* 2 $d$-wide multiexponentiations */
For $i \in \{1, 2\}$: $(W[i], W^*[i]) \leftarrow \prod_{j=0}^{d-1} ([\![\sigma^j]\!]_i^{\bar{w}_j}, [\![\alpha\sigma^j]\!]_i^{\bar{w}_j})$;      /* 4 $d$-wide multiexponentiations */
$(H, H^*) \leftarrow \prod_{j=0}^{d-2} ([\![\sigma^j]\!]_2^{h_j}, [\![\alpha\sigma^j]\!]_2^{h_j})$;      /* 2 $(d-1)$-wide multiexponentiations */
$r_v, r_w \leftarrow \mathbb{F}$;
$(\pi_v, \pi_v^*) \leftarrow (V, V^*) \cdot (Z[2], Z^*[2])^{r_v}$;
$(\pi_w, \pi_w^*) \leftarrow (W[1], W^*[1]) \cdot (Z[1], Z^*[1])^{r_w}$;
$(\pi_h, \pi_h^*) \leftarrow (H, H^*) \cdot (W_0[2]W[2], W_0^*[2]W^*[2])^{r_v} (V_0 V_c V_u, V_0^* V_c^* V_u^*)^{r_w} \cdot (Z[2], Z^*[2])^{r_v r_w}$;
$\pi_y \leftarrow \prod_{i=1}^m [\![\beta_v \hat{\mathbf{v}}_i(\sigma)]\!]_2^{a_i} \cdot \prod_{i=1}^m [\![\beta_w \hat{\mathbf{w}}_i(\sigma)]\!]_2^{b_i} \cdot [\![\beta_v \hat{\mathbf{Z}}(\sigma)]\!]_2^{r_v} [\![\beta_w \hat{\mathbf{Z}}(\sigma)]\!]_2^{r_w}$;      /* 2 $m$-wide multiexp-s */
$\mathcal{P}$ outputs $\pi = (\pi_v, \pi_v^*, \pi_w, \pi_w^*, \pi_h, \pi_h^*, \pi_y)$;
```

3.2.3 Verify $\mathcal{V}(\mathsf{vcrs}; C, \pi)$

```
Parse $\pi$ as $\pi = (\pi_v, \pi_v^*, \pi_w, \pi_w^*, \pi_h, \pi_h^*, \pi_y) \in \mathbb{G}_2^2 \times \mathbb{G}_1^2 \times \mathbb{G}_2^3$; Abort if this is not the case;
Confirm that the following equations hold:
    1. $\hat{\mathbf{e}}(W_0[1]\pi_w, V_0 V_c \pi_v) = \hat{\mathbf{e}}(Z[1], \pi_h)$;
    2. $\hat{\mathbf{e}}([\![\alpha]\!]_1, \pi_v) = \hat{\mathbf{e}}([\![1]\!]_1, \pi_v^*)$;
    3. $\hat{\mathbf{e}}(\pi_w, [\![\alpha]\!]_2) = \hat{\mathbf{e}}(\pi_w^*, [\![1]\!]_2)$;
    4. $\hat{\mathbf{e}}([\![\alpha]\!]_1, \pi_h) = \hat{\mathbf{e}}([\![1]\!]_1, \pi_h^*)$;
    5. $\hat{\mathbf{e}}([\![\gamma]\!]_1, \pi_y) = \hat{\mathbf{e}}([\![\beta_v \gamma]\!]_1, \pi_v) \cdot \hat{\mathbf{e}}([\![\beta_w \gamma]\!]_1, \pi_w)$;
```

3.3 Implementation

In order to implement the argument system, this work uses two main libraries: the encryption/pairings library from [BGM$^+$10] and the NTL 5.5.2 library (available at http://www.shoup.net/ntl/). The first library is used to do the encryption and pairings, while the second library is used to do the polynomial arithmetics. The encryption and pairings are done over a Barreto-Naehrig curve [BN06] over 256-bit prime field F_p, where $p = 36z^4 + 36z^3 + 24z^2 + 6z + 1, z = -(2^{62} + 2^{55} + 1)$. The group $\mathbb{G}_2$ in the argument system is a point in F_p (256 bit), while the group $\mathbb{G}_1$ is represented as a point in F_p^2 (512 bit). The NTL library is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. NTL's polynomial arithmetic is one of the fastest available anywhere, and has been used to set "world records" for polynomial factorization and determining orders of elliptic curves. The main class of NTL used in this work is the ZZ_pX class which is a representation of a polynomial with coefficients in finite field.

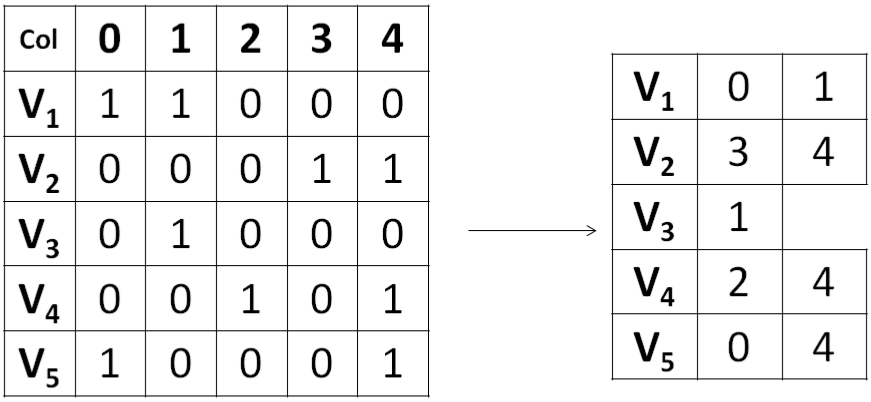
3.3.1 Sparse Matrix Representation

In order to have efficient memory consumption, there needs to be a good representation of the circuit checker. The circuit checker is a quadratic span program which in turns consists of aggregation of many checkers. It can be seen that the circuit checker will have a lot of zero value entries due to the AND composition of span programs and the nature of the gate checkers used itself. Therefore, the representation of the quadratic span program should not represent all those zero values, but still have a representation such that the arithmetics such as vectors addition will have a good complexity. To be more precise, the new representation should be made such that the arithmetics of the span program will have complexity proportional to the number of non-zero entries.

The representation of the quadratic span program is done by representing each of the row vectors (and target vectors) separately. Furthermore, in each of those vectors the data that is stored is a list of pair of positions with its value where the entries are not zero. To further optimize, for the gate checker the value itself is not stored, as it can be seen the entries in the span programs are either 1 or 0. Figure 11 shows an illustration of the new sparse matrix representation.

3.3.2 Ordered vs Unordered Map

To represent the labelling of the span programs, the implementation uses the map from the standard template library extensively. Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order. For readers familiar with hash table data structure, map have somewhat similar structure to it. However, it is known that the lookup time of map in C++ is not constant, but logarithmic. This is due to the fact that the map data structure in C++ uses red-black trees to store its mapping instead of hashing. In order to have a constant time lookup, one can instead use the unordered_map data structure. It has to be noted that this data structure is



Col	0	1	2	3	4
V_1	1	1	0	0	0
V_2	0	0	0	1	1
V_3	0	1	0	0	0
V_4	0	0	1	0	1
V_5	1	0	0	0	1

V_1	0	1
V_2	3	4
V_3	1	
V_4	2	4
V_5	0	4

Figure 11: Sparse matrix representation example

not available for all version of compilers and therefore the official version shown on this work retains the usage of the usual map. In order to use the unordered_map data structure, the compiler must have C++0x support. A light amount of work is needed in order to use the unordered_map data structure instead for efficiency. The improvement between using ordered or unordered_map is not significant, as the logarithmic time is logarithmic in terms of the size of each individual span program and the total number of checkers. Furthermore, the lookup operation is not the main operations that takes time (it is overshadowed by other operation such as exponentiation etc). In spite of that, it is always good to know that if at some point in the future the size of the span program grows larger that the complexity becomes significant enough, the implementation can still handle it by using this data structure.

3.3.3 Aggregating the Checkers

Aggregating checkers involves doing an AND composition between two span programs. The new aggregated span program between two span program will have double the size of the original. However, the top right and bottom left part of the matrix will just contain zero entries. Therefore, instead of simulating the real aggregation, the implementation only keeps track of which span programs are being aggregated. When the aggregated checker needs to be accessed, the entries will be created on the fly based on the need of the program.

3.3.4 Multipoint Evaluation and Interpolation

In this work, the polynomials are interpolated using the ZZ_pX class in NTL library where the underlying arithmetics such as multiplication, division, addition, and modular arithmetics are highly optimized. Some of the algorithms used in the polynomial arithmetics are Fast Fourier Transform (FFT), Karatsuba multiplication, and Chinese Remainder Theorem. However, after researching the interpolation function of the library, it turns out that the interpolation function uses usual quadratic Lagrange interpolation. Initial test on the function in the argument system turned out to be unsatisfactory, where the interpolation function would actually dominate over the encryption/pairings time by a lot. As interpolation is one of the core operation in the argument system, we decided to design its own interpolation function in the end.

In the implementation, we built a new layer on top of the NTL library. As NTL library already has very good underlying arithmetics functions, what we did was to build the interpolation function on top of these arithmetics. There are two main functions on top of the layers, the multipoint evaluation function and the interpolation function itself. The new interpolation function works in quasilinear $O(M(n) \log n)$ time, where $M(n)$ is the time to do a polynomial multiplication. As NTL uses FFT in the polynomial multiplication, the complexity of the multiplication is $O(n \log n)$, resulting in total complexity of $O(n \log^2 n)$ for the interpolation function. This is a significant improvement over the function NTL provides. The two main functions are based on [BS12] with some slight corrections. Section 4 will show the improvement of the new function compared to the one provided by NTL.

Algorithm 1 shows the pseudocode of the multipoint evaluation function that is implemented. In the function, a precomputation of a matrix P is needed, where P is computed recursively as follows

$$P_{0,j} = (x - u_j) \text{ and } P_{i+1,j} = P_{i,2j} \cdot P_{i,2j+1}.$$

The multipoint evaluation function takes in a function f and a set of points to be evaluated $(u_0, \cdots, u_{n-1})$. The function assumes that n is a power of two, which can be easily fixed for the argument system usage by adding extra elements into the vectors. In order to evaluate a set of points, one would call **MultiPointEval**$(f, u, 0, n-1, P)$ and expect $n-1$ return values which is the values of f evaluated on $(u_0, \cdots, u_{n-1})$. The multipoint evaluation function works in quasilinear $O(T(n) \log n)$ time, where $T(n)$ is the time to do a single polynomial modulo operation. The total complexity is then $O(n \log^2 n)$.

```
Precompute $P_{i,j}$ for all $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$;
**Function MultiPointEval(**$f, u, a, b, P$**):**
    **Data**: A function $f$, a set of points $u$, interval $a$ and $b$, $P$
    **Result**: A set of values $(f(u_a), \cdots, f(u_b))$
Let $n = (b - a)$;
**if** $n = 1$ **then**
    **return** $(f(a))$;
**else**
    Let $k = \log_2(n)$;
    Let $\text{idx} = \lfloor a/2^{k-1} \rfloor$;
    Let $r_0 = f \mod P_{k-1,\text{idx}}$ and $r_1 = f \mod P_{k-1,\text{idx}+1}$;
    Let $(y_a, ..., y_{(a+b)/2}) =$ **MultiPointEval(**$r_0, u, a, (a+b)/2, P$**)**;
    Let $(y_{(a+b)/2+1}, \cdots, y_b) =$ **MultiPointEval(**$r_1, u, (a+b)/2+1, b, P$**)**;
    **return** $(y_a, \cdots, y_b)$;
**end**
```

Algorithm 1: Multipoint evaluation

In order to implement the interpolation function, this work make use of the multipoint evaluation function. As one know, the polynomial interpolation is the problem of finding the polynomial f where $f(u_i) = v_i$ for two vectors u and v of same length. Here, it is also assumed that the length of the vectors n is a power of two. The Lagrange interpolation formula constructs f as follows

$$f(x) = \sum_{i=0}^{n-1} v_i \cdot \prod_{j=0, j \neq i}^{n-1} \frac{x - u_j}{u_i - u_j}.$$

One can rewrite the above equation as

$$f(x) = \sum_{i=0}^{n-1} v_i s_i \cdot \prod_{j=0, j \neq i}^{n-1} (x - u_j),$$

$$s_i = \prod_{j=0, j \neq i}^{n-1} \frac{1}{u_i - u_j}.$$

Notice that s_i is somewhat similar to the definition of P defined previously in algorithm 1. To be more precise, $s_i^{-1} = P^{'}_{k,0}(u_i)$ where $P^{'}_{k,0}$ is the formal derivative of the polynomial $P_{k,0}$. This means that in order to find s_i one just need to do multipoint evaluation of $P^{'}_{k,0}$ at points $(u_0, \cdots, u_{n-1})$ to obtain $s_0^{-1}, \cdots, s_{n-1}^{-1}$. After that, it is easy to compute $v_i^{'} = v_i/s_i^{-1} = v_i s_i$. The problem now simplifies to the following: given two vectors u and v compute the polynomial f where $f(u_i) = v_i$ as follows

$$f(x) = \sum_{i=0}^{n-1} v_i^{'} \cdot \prod_{j=0, j\neq i}^{n-1} (x - u_j).$$

The polynomial f has the following structure: $f = r_0 P_{k-1,1} + r_1 P_{k-1,0}$ where

$$r_0(x) = \sum_{i=0}^{n/2-1} v_i^{'} \cdot \prod_{j=0, j\neq i}^{n/2-1} (x - u_j),$$

$$r_1(x) = \sum_{i=n/2}^{n-1} v_i^{'} \cdot \prod_{j=n/2, j\neq i}^{n-1} (x - u_j).$$

which then again suggests a recursive algorithm as shown in Algorithm 2. In order to use the function, one would call **Interpolate**$(u, v, 0, n-1, P)$ and expect the polynomial f as explained earlier. Similar as the multipoint evaluation function, the interpolation function has a quasilinear $O(n \log^2 n)$ running time.

In addition to the multipoint evaluation and the polynomial interpolation function, there are several speed up that is applicable in this argument system.

- By using Lagrange interpolation, all the basis polynomials can be precomputed in the offline phase, as r_j is chosen independent of the span program content. Compared to using a library's interpolation function, the basis polynomial is always recomputed for every single row which is redundant.

- The precomputation of each basis polynomial itself can be further optimized in the offline phase by using the sliding window technique to avoid whole recomputation of the denominator of the polynomial. For example, if $(x_0, \cdots, x_n) = (1, \cdots, n)$ then the denominator for the first basis polynomial is $d_0 = (1-2)(1-3)\cdots(1-n) = (-1)(-2)\cdots(-n+1)$. The denominator for the second basis polynomial is $d_1 = (2-1)(2-3)\cdots(2-n) = (1)(-1)\cdots(-n+2) = d_0 \cdot 1/(-n+1)$.

```
Pre-compute $P_{i,j}$ for all $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$;
Pre-compute $v_i^{'}$ for $0 \leq i < n$;
**Function Interpolate**$(u, v, a, b, P)$**:**
Data: a set of points $u$ and $v$, interval $a$ and $b$, $P$
Result: A polynomial $f$ where $f(u_i) = v_i$ for $a \leq i \leq b$
Let $n = (b - a)$;
if $n = 1$ then
    return $v_a^{'}$;
else
    Let $k = \log_2(n)$;
    Let idx $= \lfloor a/2^{k-1} \rfloor$;
    Let $r_0 =$ Interpolate$(u, v, a, (a+b)/2, P)$;
    Let $r_1 =$ Interpolate$(u, v, (a+b)/2+1, b, P)$;
    return $(r_0 \cdot P_{k-1,idx+1}) + (r_1 \cdot P_{k-1,idx})$;
end
```

Algorithm 2: Polynomial Interpolation

3.3.5 Pairings

The pairings function in this work is the function *opt_atePairing <Fp >(result, element1, element2)*, where *element1* is in group $\mathbb{G}_2$ and *element2* is in group $\mathbb{G}_1$. The pairings function as stated is from the library by [BGM+10]. This pairings function has been optimized by the author of the library, and runs considerably fast compared to other pairings libraries. One of the comparison is with Aranha library [AKL+11] where Aranha library uses 10545 unreduced multiplications producing double precision result and 5028 modular reduction of double precision integers. This library uses 1033 unreduced multiplications producing double precision result and 4925 modular reduction of double precision integers for the pairings. More information of the benchmark could be found in [BGM+10].

3.3.6 Multiexponentiation

In order to have a better efficiency in the cryptographic process of the argument system, this thesis has implemented another function on top of the encryption library, which is the multiexponentiation function. The multiexponentiation algorithm is known as the Straus algorithm and was later rediscovered as Shamir's Simultaneous Squaring Multiexponentation algorithm, and it is also discussed in [Str64]. The pseudocode of the algorithm is shown in algorithm 3. Note that the exponents described in the pseudocode are represented in bits.

```
Data: base number $g_1, \cdots, g_h$, exponents $e_1, \cdots, e_h$ where
    $e_j = (e_{j,k-1}, \cdots, e_{j,0})$
Result: $g_1^{e_1} \times \cdots \times g_h^{e_h}$
Let $R = 1$ ;
for $i = k - 1 \rightarrow 0$ do
    $R = R^2$;
    $R = R \times (g_1^{e_{1,i}} \times \cdots \times g_h^{e_{h,i}})$ for all $e_{j,i} \neq 0$
end
return $R$;
```

Algorithm 3: Multiexponentiation Algorithm

Apart from the algorithm shown above, this thesis has also used the optimization suggested by [PGHR13] which is to precompute exponent table using sliding window technique. The essence mainly is to precompute $(g_1^{e_{1,i}} \times \cdots \times g_k^{e_{k,i}})$ for all possible (boolean) values of $e_{j,i}$ where k is the window size.

4 Comparison and Time Measurement

This section gives the time measurement result of the implementation done on the argument system. The time measurement mostly consists of the computation time of each party (CRS, prover, and verifier), single interpolation time, and also the multiexponentiation function.

4.1 Parameters

All the time measurement was conducted on a Thinkpad T420, 4GB RAM, i7 core @ 2.7GHz. The operating system is Windows 7 64 bit edition, and the implementation was run on Cygwin. The compiler used is x86_64-w64-mingw32-g++ (GCC) 4.5.3. The compilation was optimized by using `-O3 -fomit-frame-pointer -msse2 -mfpmath=sse -march=native` compile options. No multithreading was used in the measurement. In order to have a good accuracy, the measurement was done on the same parameter. A function to generate a circuit was constructed specifically for the measurement. The function $GenCircuit(level)$ takes the parameter $level$ which then generates a circuit with size $2^{level} - 1$ where the size here is the number of gates. The structure of the circuit resembles to that of a perfect binary tree where each node corresponds to a gate, and each edge corresponds to the wire. Furthermore, in order to minimize any time difference due to the randomness of the gate type, all gates are set as NAND gate. Figure 12 shows an example of the circuit output from the function. With this systematic generation of circuit, the number of gates, wires, and variables can be pre-determined. Note that the implementation itself is built to be able to accept circuit description that is far more flexible than this. The purpose of this generation is merely to measure the timings of the operation accurately as it is hard to generate a systematic circuit for fanout 2 for this purpose. Furthermore, the result timing will still

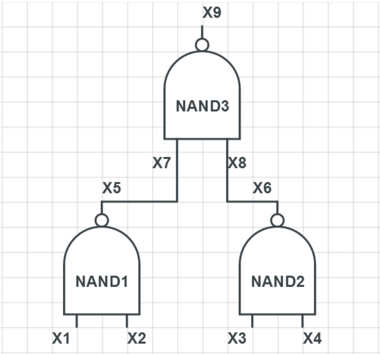


Figure 12: Circuit output with $level = 2$

serve as a good benchmark as the only part that is affected is the computation of $[\![\hat{\mathbf{v}}_\mathbf{i}(\sigma)]\!]_i, [\![\hat{\mathbf{w}}_\mathbf{i}(\sigma)]\!]_i$ in CRS and $\mathbf{v}, \mathbf{w}$ calculation from Prover. Both of these are not the part that affect the timing the most. The timing per dimension, multiexponentiation, and interpolation time will not be affected at all as they solely depend on the dimension of the circuit checker.

We denote by $|C|$ = number of gates, $|E|$ = number of wires, and $|X|$ = number of variables. From Figure 12, it can be seen that the $|X| = 3\ |C|$, whereas the number of wires $|E|$ (which needs to be checked using wire checker) $= |C| - 1$. With the same reasoning, the final size (with m = number of rows/size, and d = number of columns/dimension) of the quadratic span program can also be determined beforehand due to the fact that all gates are NAND gates. The (m, d) value is computed as follows. As every gate is a NAND gate, $m = m_1 \cdot |C| + 1$ where $m_1 = 6$ is the number of rows/size of the NAND checker. The $+1$ is from the target vector. We note that in [Lip13] the target vector is not counted towards m. The value of d is $2d_1 \cdot |C| + d_2 \cdot |E|$ where $d_1 = 3$ is the dimension of the NAND checker and $d_2 = 6$ is the dimension of a wire checker for fanout 1. The value of d is then simplified to $6|C| + 6(|C| - 1) = 12|C| - 6 = 12(2^{\text{level}} - 1) - 6$. These formulas are gotten from the circuit checker that is explained in Section 2.2.3. The measurement was done on several levels, namely on $level = 3, ..., 13$. In the following sections, the horizontal axis would be representing the span program size and dimension, which can be seen from its (m, d) value. Note that the m value given is only the V part of the circuit checker span program, which means the total rows should be two times larger. The (m, d) values are (43, 78), (91, 174), (187, 366), (379, 750), (763, 1518), (1531, 3054), (3067, 6126), (6139, 12270), (12283, 24558), (24571, 49134), (49147, 98286) respectively. More details on the parameters can be seen on Table 1. We note that the (m, d) listed on the table are the original values generated from the circuit. In the argument system implementation, the values might get bigger in order to be suitable for some functions, such as the interpolation function and multiexponentiation function. For the interpolation function the input vector should be a power of two, so additional zeros are added to the vector until the number of elements is a power of two. For the multiexponentiation, as we used window size 8, we pad them out by putting dummy base exponent where its corresponding exponent is equal to zero so it does not affect the end result.

4.2 Interpolation Time

Figure 13 shows the interpolation time measurement in logarithmic (base 2) scale. The interpolation time is measured based on the interpolation of one vector in the span program, which means that in this case the first parameter m in the horizontal axis does not affect the time. However, the parameter is still given on the figure in order to provide consistency over the following figures. As explained

Level	Gates	Wires	Variables	m	d
3	7	6	21	43	78
4	15	14	45	91	174
5	31	30	93	187	366
6	63	62	189	379	750
7	127	126	381	763	1518
8	255	254	765	1531	3054
9	511	510	1533	3067	6126
10	1023	1022	3069	6139	12270
11	2047	2046	6141	12283	24558
12	4095	4094	12285	24571	49134
13	8191	8190	24573	49147	98286

Table 1: Test parameters based on level

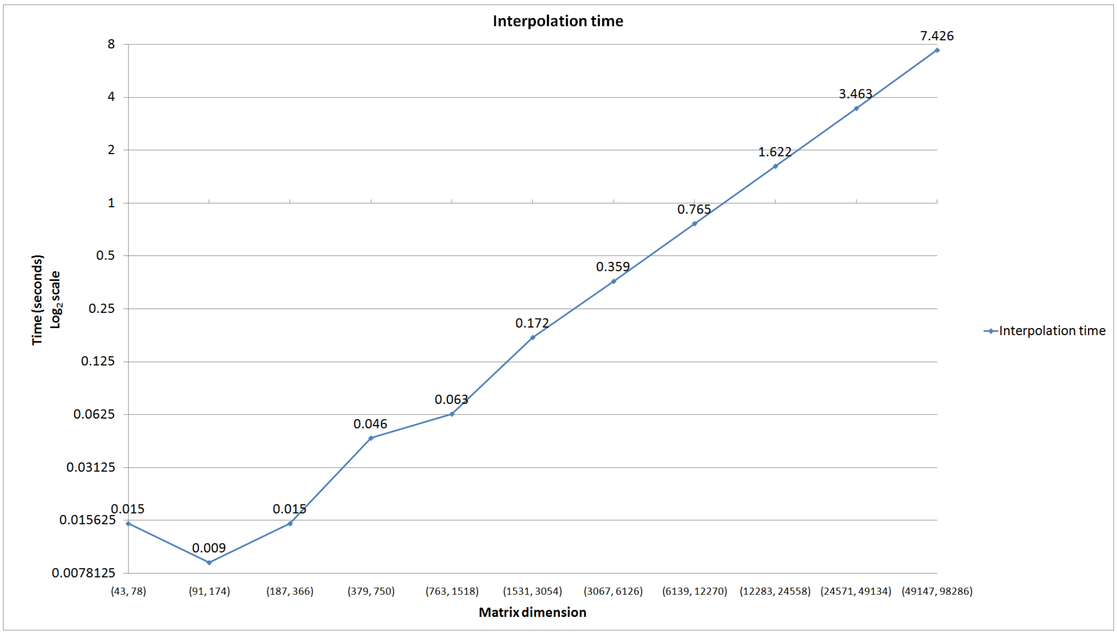


Figure 13: Interpolation time for 1 vector in $\log_2$ scale

before, the interpolation function was remade instead of reusing the interpolation function from the NTL library. In the implementation, some optimization such as pre-computation of all basis polynomials beforehand are done (only once, because all basis polynomials remain the same throughout the argument system). The time measurement shows a very satisfactory result. It can be seen that even on the highest level tested, only 7.426 seconds was needed to interpolate one vector. In total, there are only 4 vectors that need to be interpolated throughout the whole argument system (two by CRS and another two by Prover) so the total time is very reasonable.

In order to demonstrate how much the new time measurement of the interpolation time improves over the default NTL interpolation time, Figure 14 shows the comparison between the new and the default interpolation function on the same parameters. The details is shown on Table 2.

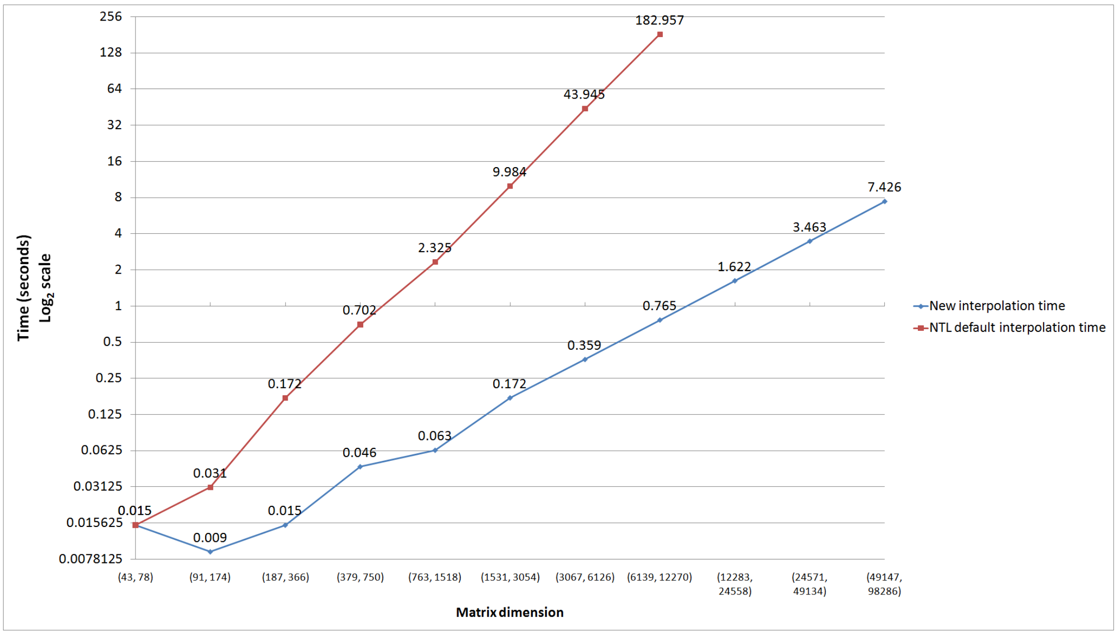


Figure 14: Comparison between NTL and new interpolation function in $\log_2$ scale

We initially used the default interpolation function from NTL. However, the resulting time was not as good as expected, since the interpolation time strongly dominated the encoding/pairings time. This forced the author to look into the interpolation function of NTL, which indeed uses a quadratic interpolation function. A new interpolation function was then designed on top of the basic arithmetics to achieve better efficiency. The new result shows a much better time and adheres to the initial guess that the interpolation time should be dominated by the encryption/pairings time for practically relevant values of n.

Dimension	NTL interpolation function	New interpolation function
78	0.015	0.015
174	0.031	0.009
366	0.172	0.015
750	0.702	0.046
1518	2.325	0.063
3054	9.984	0.172
6126	43.945	0.359
12270	182.957	0.765
24558	-	1.622
49134	-	3.463
98286	-	7.426

Table 2: Interpolation function time comparison

4.3 Multiexponentiation Time

As explained in the previously, we implemented a multiexponentiation function based on [Str64], also known as Shamir's Simultaneous Squaring algorithm. Figure 15 shows time measurement and comparison between several versions of multiexponentiation and the usual exponentiation algorithm. The usual exponentiation algorithm here means the naive way where every single element is taken to the exponent, and then multiplied together. The difference between each multiexponentation version is the radix that was used, where 1, 2, and 4 bit(s) are used as the radix respectively. The time measurement was the average of repeating it for 50 times in order to get a more accurate time. As seen on the figure, the fastest version is the multiexponentiation that uses bit by bit (basis 2, single bit) squaring, which is the best in a computer setting. It performs quite well, having almost a 3x speed up compared to the naive version. Note that the speed of this function is important due to the fact that a big part of the prover's computation is calculating multiexponentiation where the number of exponents is proportional to the size of either m or d. More details on the exact timing can be seen on Table 3.

Apart from the radix, as stated before, we used the optimization proposed by [PGHR13] which is to precompute exponent table using sliding window technique. After multiple tests, we decided to use window size of 8 in order to achieve the best timing performance while still retaining a reasonable memory usage. The details on the timings of the sliding window optimization can be seen on Table 4.

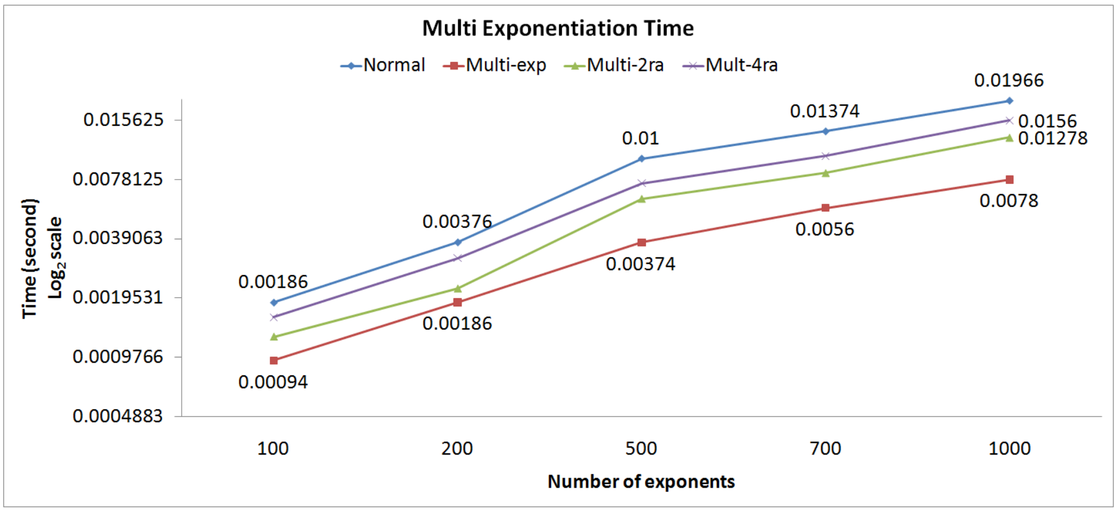


Figure 15: Normal and multiexponentiation average time ($\log_2$ scale)

Exponents	Normal	Multi-exp	Multi-2ra	Mult-4ra
100	0.02852	0.01554	0.02326	0.05666
200	0.05486	0.03182	0.04536	0.11828
500	0.11884	0.07078	0.10668	0.33172
700	0.20932	0.12242	0.1778	0.44368
1000	0.29706	0.17412	0.25904	0.6388

Table 3: Multiexponentiation average timing (in seconds)

(m, d)	Window size 2	Window size 4	Window size 8
(43, 78)	0.011	0.01	0.01
(91, 174)	0.026	0.016	0.016
(187, 366)	0.041	0.016	0.016
(379, 750)	0.086	0.046	0.046
(763, 1518)	0.173	0.093	0.094
(1531, 3054)	0.36	0.203	0.203
(3067, 6126)	0.742	0.406	0.436
(6139, 12270)	1.541	0.873	0.874
(12283, 24558)	3.058	1.856	1.747
(24571, 49134)	6.14	3.495	3.573
(49147, 98286)	12.185	7.504	7.254

Table 4: Multiexponentiation timing on different window size

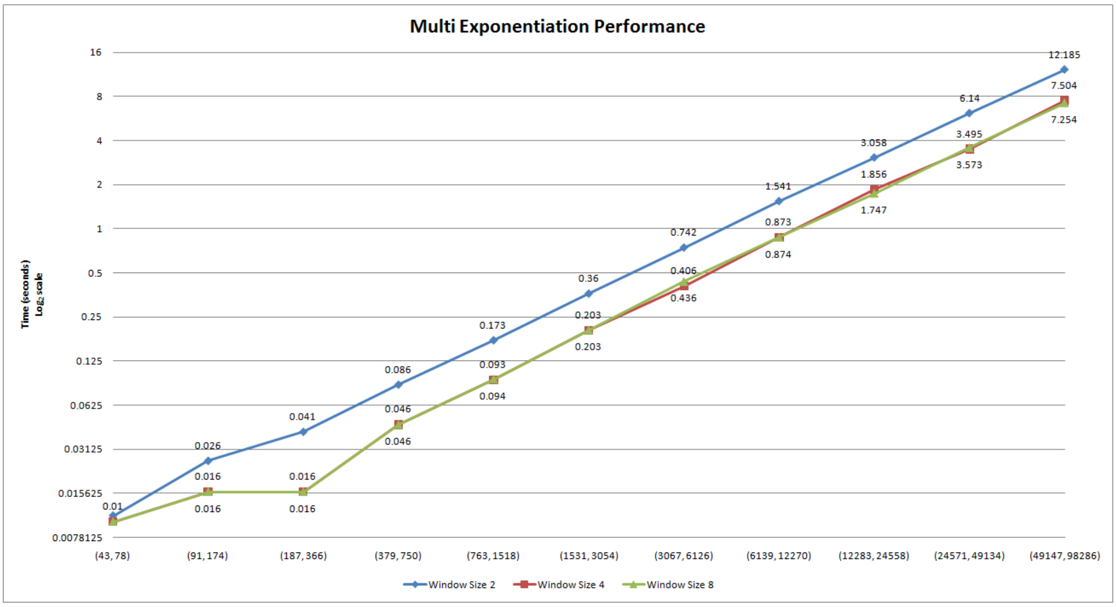


Figure 16: Window size timing ($\log_2$ scale)

4.4 CRS Generation Time

The CRS generation time consists mainly of two parts, interpolating each v_i and w_i from the circuit checker and computing the encoding of elements which is gotten by evaluating the aforementioned polynomials on a secret point. The result of the time measurement shows that the CRS generation time is dominated by the time of the encoding (exponentiation) of elements. The reason behind this is simply because the amount of encoding operation needed is more than that of interpolation. The amount of time needed for 1 multiexponentiation and 1 interpolation is roughly the same as seen later in this section. Although the complexity of the multiexponentiation is $O(d)$ and the complexity of interpolation is $O(d \log^2 d)$, the multiexponentiation takes more time in practical value of degree size. This is because it takes another logarithmic factor of time in terms of the exponent to do the multiexponentiation. The exponent is an element in the field which is 256/512 bits in size. Compared to the degree, the size of the exponent is obviously much bigger. Taking this into account, the $O(\log^2 d)$ factor in the interpolation is much smaller compared to the $\geq \log 2^{256}$ steps of multiexponentiation. The $O(\log^2 d)$ factor can be brought down even further to $O(\log d)$ (discussed later in Section 5.6) which makes the factor even smaller. In the CRS interpolation, only v_0 and w_0 are truly interpolated in its polynomial form, while the rest of v_i and w_i are interpolated directly into the evaluation of the secret point σ. With that, the resulting time of the CRS interpolation time is consistent with the result on single interpolation, as in CRS interpolation two vectors are interpolated in polynomial form and d doubles on each level, resulting in the increase shown on Figure 17. The CRS exponentiation time also exhibits a normal behavior as the level goes up (the size of the matrix doubles, and the time of the CRS computation time goes up proportionally to it, with some constant multiplier). As the CRS is usually generated by a trusted third party and can be reused, this measurement isn't of much importance compared to the prover's and verifier's time. The prover's interpolation time as can be seen later on has a comparable time with the CRS interpolation time, meanwhile the verifier's total computaion time is significantly smaller. We note here that if the algorithm supports an adaptive version, the CRS only needs to be generated once. Figure 17 compares the interpolation and the exponentiation time of the CRS generation.

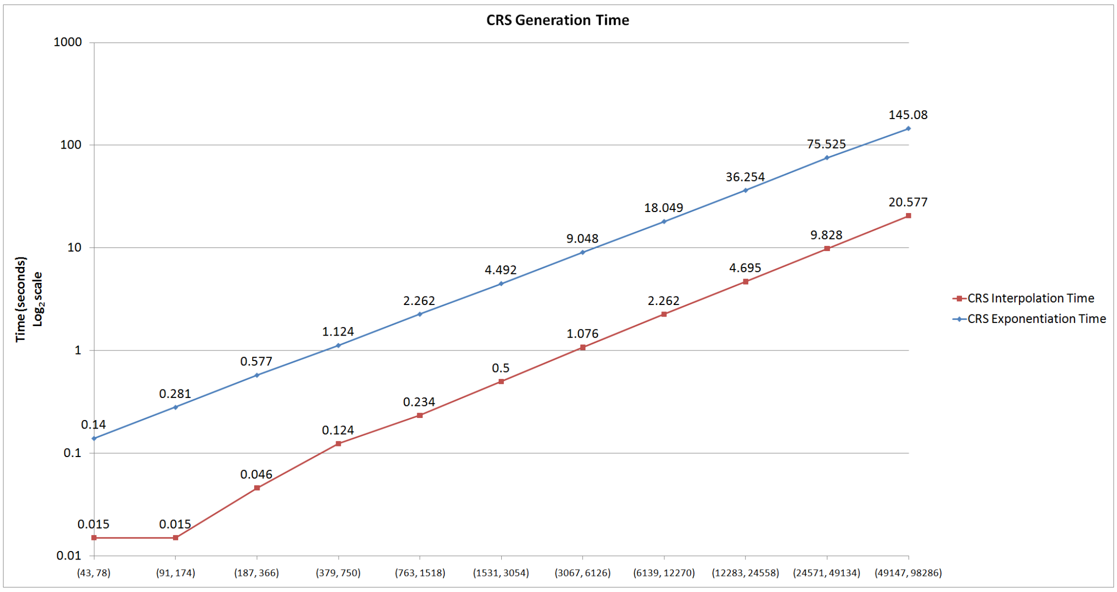


Figure 17: CRS interpolation and computation time ($\log_2$ scale)

4.5 Prover's and Verifier's Computation Time

This section shows the prover's and verifier's computation time. Note that as stated the timing here is the non-adaptive version of the argument system. The time needed for the prover is also dominated by the encoding of the elements (i.e. multiexponentiation) as opposed to the interpolation time. From the prover's side interpolation is still needed, but the number of interpolations is constant. The verifier only needs to do constant amount of pairings in the non-adaptive version, which makes the time needed to be much smaller than the prover's time

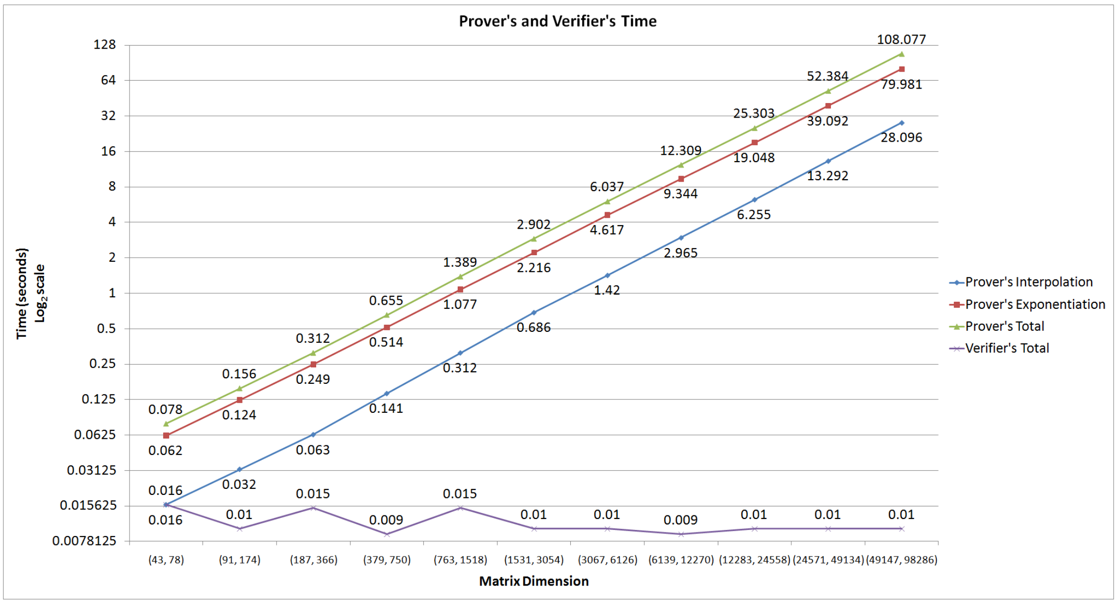


Figure 18: Prover's and Verifier's computation time in $\log_2$ scale

(which is already very good). Figure 18 shows the time from both the prover and the verifier. Figure 19 shows more details on verifier's time as the scale is much smaller compared to prover's total time. The verifier's time graph in Figure 19 is shown in linear scale. The reason behind the irregular time in the verifier's total time is due to the fact that the time needed is constant and so insignificant that it can easily change with just a little noise from the test environment. It can be seen that the prover's total time in the argument system is lower than the CRS time. The verifier's total time is in turn significantly lower than the prover's total time due to constant number of pairings regardless of the circuit size.

4.6 Overall Timings and Details

This section lists a wrap up of all the important timings and details of the implementation. Table 5 shows all the important timings and argument size of the implementation. The timings are: time to interpolate 1 vector of dimension d, time to do 1 multiexponentiation of dimension d, CRS interpolation time, CRS exponentiation time, CRS generation total time, Prover's interpolation time, Prover's multiexponentiation time, Prover's total time, Verifier's total time, interpolation time per dimension, multiexponentiation time per dimension, ratio between multiexponentiation and interpolation time, and argument size. All the timings are in seconds, except for Interpolate/Dimension (in microseconds), MultiExp/Dimension (in microseconds), MultiExp/Interpolate (in percent). CRS size is in megabytes,

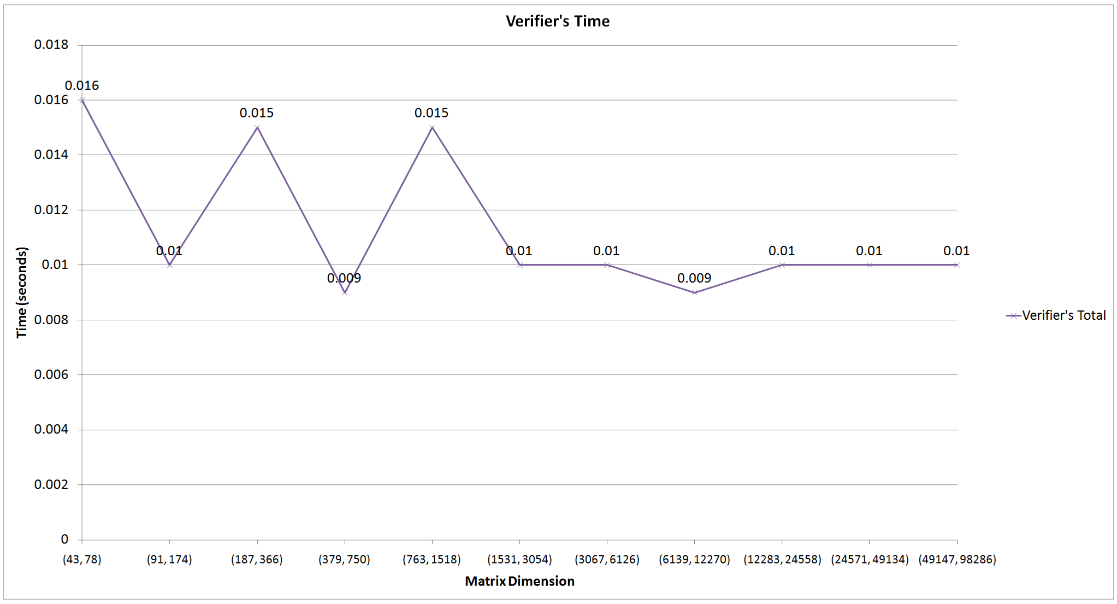


Figure 19: Verifier's computation time (linear scale)

VCRS size and argument size are in bytes. The CRS and VCRS size includes all the elements except for the span program P in order to show the better contrast between the two. It can be seen that the timing ratio of interpolation over dimension and multiexponentiation over dimension stays stable. Furthermore, the ratio between 1 multiexponentiation time and 1 interpolation time is roughly 1:1. The time measurement was not done on larger circuit size due to insufficient memory on the machine, which causes the operating system to be unstable making the timing to be tremendously inaccurate.

4.7 Comparison with Pinocchio

This work tries to compare the implementation with the Pinocchio's implementation from [PGHR13]. Even though Pinocchio solves a different problem, there are some similarities in the implementation itself.

- Both Pinocchio and we uses asymmetric pairings which is faster compared to symmetric pairings (with the same level of security). This is because the calculation can be pushed more into the cheaper base curve compared to the twist curve.

- The size of the proof in this work is the same as Pinocchio's, 288 bytes. The proof is the statement sent by the prover to the verifier. The prover sends 5 elements from $\mathbb{G}_2$ and 2 elements from $\mathbb{G}_1$ where each element of $\mathbb{G}_2$ is 256 bits

(m, d)	(3067, 6126)	(6139, 12270)	(12283, 24558)	(24571, 49134)	(49147, 98286)
Interpolate 1	0.359	0.765	1.622	3.463	7.426
MultiExp 1	0.436	0.874	1.747	3.573	7.254
CRS Interpolate	1.076	2.262	4.695	9.828	20.577
CRS Expo	9.048	18.049	36.254	75.525	145.08
CRS Total	10.124	20.311	40.949	85.353	165.657
Prover Interpolate	1.42	2.965	6.255	13.292	28.096
Prover Expo	4.617	9.344	19.048	39.092	79.981
Prover Total	6.037	12.309	25.303	52.384	108.077
Verifier Total	0.01	0.009	0.01	0.01	0.01
Dimension	6126	12270	24558	49134	98286
Interpolate/Dimension (μs)	58.603	62.347	66.048	70.481	75.555
MultiExp/Dimension (μs)	71.172	71.231	71.138	72.720	73.805
MultiExp/Interpolate (percent)	121.448	114.248	107.707	103.176	97.684
CRS size (MB)	1.373	2.749	5.502	11.007	22.017
VCRS size (bytes)	544	544	544	544	544
Argument Size (bytes)	288	288	288	288	288

Table 5: Timing and argument size

and each element of $\mathbb{G}_1$ is 512 bits. The total is then $(5 \times 256) + (2 \times 512) = 2304$ bits = 288 bytes. It can be noted that this can only be achieved using some optimization pointed out in [PGHR13] which is to store the points in affine form and further compress it to only store the abscissa (and finding the pair ordinate as needed).

- The polynomial interpolation function implemented in this work is faster than the one shown on Pinocchio. Pinocchio shows that in order to interpolate a polynomial of degree 1000, its optimized function needs 331.1 ms whereas we need 93 ms to interpolate polynomial of degree around 1500. The reason behind might be due to the fact that some precomputation on the basis polynomial is done beforehand in this work which makes it to run faster than Pinocchio's function. Other factors that affect the performance apart from the interpolation/multipoint evaluation itself is the polynomial arithmetics underneath that are highly optimized by NTL.

- The multiexponentation function in Pinocchio seems to be of the same speed compared to this work. As there are no really details given on the parameter on the multiexponentiation function, this work tries to compare the optimization used and the speed up achieved compared to the naive version. As both this thesis and Pinocchio uses the table precomputation, they both achieve a 3-4x speed up

- In the Pinocchio paper, they claimed that using Quadratic Arithmetic program (QAP) is much faster compared to Quadratic Span Program (QSP). Instead of having input as boolean value, a Quadratic Arithmetic Program assumes each input to be an element in the field $\mathbb{F}$. Futhermore, instead of boolean gates, QAP works with arithmetic gates (multiplication, addition, etc). We haven't look much into the details in their implementation, but from their theoretical explanation adding a gate into the circuit increases the QSP size by 12 and dimension by 9. This result was referred from [GGPR13]. In [Lip13], adding a gate is done by using a span program (not QSP). The final circuit checker QSP will indeed increase in terms of dimension and size. Focusing on size increase by 12, this means that the span program initially has size 6. The two span program that Lipmaa proposed which has size 6 are the fork gate and NAND gate. However, using either of those two gate checker will only increase the dimension of the circuit checker by 6, not 9. Furthermore, the other gate checkers proposed (OR, AND, XOR, EQUAL) all have smaller size (in terms of rows and dimension) compared to the NAND and fork gate checker. In addition to that, the Pinocchio's verification time is not constant. They also used less precise bounds on the circuit checker size. Interested reader should refer to both [Lip13] and [PGHR13] for comparison. By that reasoning, even though QSP might indeed become slower due to larger resulting circuit checker (still needs to be directly compared with [Lip13]), we feel that the slowdown factor is not as big as claimed in the Pinocchio paper.

Even though not all of this work implementation can be compared directly with Pinocchio, it may be worth noted that Pinnochio's implementation is highly optimized over longer time for industrial purpose. The author believes that this initial implementation of work serves as a good start and can be improved much more over time as a foundation.

5 Future Work and Possible Improvements

This section discusses some of the possible future work and improvements resulting from this work. Some of the improvements listed here were noticed by the author during the work process. Unfortunately, due to time constraint not all of the optimizations and features were added to this work. The author hopes that this section would help anyone that would be interested in continuing this work in the near future.

5.1 Universal Circuit

The argument system is this work is the non-adaptive version of the argument system based on [Lip13]. In his paper, Lipmaa has also managed to design an adaptive version of the same argument system. This version is better in the sense that there is no need for the function/circuit to be known first before the CRS generation. This is achieved by using a universal circuit which in turn takes in the circuit to be proven and the assignment itself as its input. The universal circuit is a boolean circuit that can be made to compute any boolean function by setting its specially designated set of control inputs to appropriate fixed values [Val76]. Although the adaptive version is logarithmically slower than the non-adaptive version due to the usage of universal circuit, the "non-interactive" part of the argument system looks much nicer as the non-adaptive version somehow needs the trusted third party to be there for every single new problem to be proven (although in the non-adaptive version the CRS can still be reused if the function/circuit stays the same).

5.2 Circuit Fanout Reduction

The implementation done on this work assumes that the input of the circuit has a maximum fanout 2. In [Lip13], it was also explained that any circuit with fanout more than 2 can be transformed into a circuit with maximum fanout 2. Although it can be argued that this is sufficient, it would be better if there is a feature where given any circuit of any fan out, another circuit which computes the same function is outputted where this later circuit has a maximum fanout of 2. The reason that the circuit with higher fanout should be first transformed is due to some efficiency issue which is explained in [Lip13].

5.3 $h(X)$ optimization

In [Lip13], Lipmaa proposed an optimized way to compute $h(X)$ by computing $\hat{\mathbf{Z}}^{rev}(X) = X^d\hat{\mathbf{Z}}(1/X)$ and then using it to calculate $h(X)$. With this optimization, the Prover essentially needs to do two multiplication. This optimization is not implemented in this thesis.

5.4 Fanout 3 implementation

As stated in earlier section, [Lip13] claimed that limiting the fanout to 3 is more efficient than limiting the fanout to 2 in some cases. It would be of interest to have such implementation to see the time comparison between the two.

5.5 Input Circuit Generation

The time measurement was done by feeding a systematically generated circuit that has a perfect binary tree structure into the program, which means that all gates will have fanout 1. An implementation of circuit generation with fanout at most 2 would be good to see how much does it affect the time measurement.

5.6 Futher Optimization on Interpolation Function

As stated in the earlier section, the complexity of the new interpolation function in this work is $O(n \log^2 n)$. This can be further reduced to $O(n \log n)$ by using inverse FFT and FFT respectively. This however requires the roots r_i to be the roots of unity, meaning that the field itself needs to support such choices. This part remains to be explored, and successful implementation of inverse FFT and FFT on the field will decrease the interpolation time by a factor of $O(\log n)$.

There is another optimization that the author realized during the end of this thesis, but not implemented. As the QSP of the wire checker uses Reed-Solomon code, it introduces redundancy. This means that not all the elements are needed to be given to the interpolation function to interpolate the polynomial. For example, for fanout 1 only the first two out of the three columns of the generator are needed. This will reduce the number of additions and multiplications (of the polynomial) by the size of wires.

6 Conclusion

The main result of this work is the implementation of the non-adaptive version of the non interactive zero knowledge argument system based on [Lip13]. In the process, we also managed to improve the interpolation function for field polynomials which to author's knowledge hasn't been implemented by many libraries. Based on the implementation and the resulting measurement, there are several conclusions to be made:

- In order to achieve maximum efficiency, a good sparse matrix representation must be used to optimize the memory usage while still considering the operation time trade off
- For the CRS generation and prover's side of computation, the multiexponentiation of elements dominates the total time of computation as opposed to the interpolation time
- The prover and verifier computation takes a reasonable amount of time. This shows as a good sign to use the non interactive zero knowledge argument system in a practical setup.
- In order to remove the dependency with the trusted third party, one can attempt to build a universal circuit setup on top of the existing implementation in order to avoid generating the CRS every time.

Although there are rooms for improvement, the author believes that the existing implementation serves as a very good signal on how non interactive zero knowledge can actually be used as a building block in real life. The running time of the argument system looks reasonable enough for the main parties involved to prove problems that are not trivial.

Lakoonilise nullteadmusargumendisüsteemi optimiseeritud implementatsioon

Magistritöö (30 EAP)

Hendri

Resümee

Käesolevas töös üritame konstrueerida lakoonilise mitteinteraktiivse nullteadmustõestuste süsteemi implementatsiooni. Mitteinteraktiivne nullteadmustõestuste süsteem on protokoll, milles üks osapool, keda kutsutakse tõestajaks, tõestab teistele osapooltele, keda kutsutakse verifitseerijateks, et mingi verifitseerijale esitatud väide on tõene. Nullteadmusprotokoll peab muuhulgas garanteerima, et vastav tõestus ei lekita väite kohta muud informatsiooni peale väite kehvituse. Antud töös käsitleme tõeväärtusskeemide kehtestatavuse probleemi. Tõeväärtusskeemi kehtestatavuse probleem on küsimus selle kohta, kas leidub sisend, millel antud tõeväärtusskeem saab väljundiks väärtuse tõene. Implementeeritud tõestusskeem põhineb Helger Lipmaa tööl [Lip13], mis kasutab tõestuse konstrueerimiseks lineaarkatte programme *(span program)* ja lineaarseid veaparanduskoode. Töös anname ka kerge ülevaate nullteadmustõestuste üldisest olemusest, et ülejäänud töö olemust paremini selgitada.

Me konstrueerime mitteadaptiivse versiooni tõestussüsteemist. Lisaks nullteadmustõestusele iseloomulikele omadustele on see versioon kasulik ka verifitseeritava arvutamise saavutamiseks, nagu käsitletud näiteks artiklis [PGHR13]. Töö algab ülevaatega mitteinteraktiivsest nullteadmusest ning lineaarkatte programmidest. Edasises kirjeldame, kuidas esitada tõeväärtusskeemi kehtestatavuse probleemi kasutades mainitud lineaarkatte programme. Lõpuks kirjeldame oma implementatsiooni, keskendudes olulistele detailidele ning kasutatud teekidele. Töö kokkuvõtteks on jõudlustulemused ning suunad edasisteks täiendusteks.

References

[AKL$^+$11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, EUROCRYPT'11, pages 48–68, Berlin, Heidelberg, 2011. Springer-Verlag.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 213–229, London, UK, UK, 2001. Springer-Verlag.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. ACM.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, UK, 2001. Springer-Verlag.

[BGM$^+$10] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto-naehrig curves. In *Proceedings of the 4th international conference on Pairing-based cryptography*, Pairing'10, pages 21–39, Berlin, Heidelberg, 2010. Springer-Verlag.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag Berlin/Heidelberg, 2006. `http://cryptojedi.org/papers/#pfcpo`.

[BR95] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols, 1995.

[BS12] Markus Bläser and Chandan Saha. Computational Number Theory and Algebra, Lecture 6, 2012.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '94, pages 174–187, London, UK, UK, 1994. Springer-Verlag.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 209–218, New York, NY, USA, 1998. ACM.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, Athens, Greece, May, 26–30 2013.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[Gro10] Jens Groth. Short Non-interactive Zero-Knowledge Proofs. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477, pages 341–358, Singapore, December, 5–9 2010.

[GT03] Sha Goldwasser and Yael Taumann. On the (in)security of the fiat-shamir paradigm. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 102–115. IEEE Computer Society Press, 2003.

[KW93] M. Karchmer and A. Wigderson. On span programs. In *In Proc. of the 8th IEEE Structure in Complexity Theory*, pages 102–111. IEEE Computer Society Press, 1993.

[Lip12] Helger Lipmaa. Progression-Free Sets and Sublinear Pairing-Based Non-Interactive Zero-Knowledge Arguments. In Ronald Cramer, editor, *TCC 2012*, volume 7194, pages 169–189, Taormina, Sicily, Italy, March, 19–22 2012.

[Lip13] Helger Lipmaa. Succinct Non-Interactive Zero Knowledge Arguments from Span Programs and linear Error-Correcting Codes. Technical Report 2013/121, February 28, 2013. Available at http://eprint.iacr.org/2013/121.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[Rot06] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA, 2006.

[Str64] Ernst G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1964. URL: `http://cr.yp.to/bib/entries.html#1964/straus`.

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the eighth annual ACM symposium on Theory of computing*, STOC '76, pages 196–203, New York, NY, USA, 1976. ACM.

Non-exclusive licence to reproduce thesis and make thesis public

I, Hendri

(date of birth: 14th October 1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

An Optimized Implementation of a Succinct Non-Interactive Zero-Knowledge Argument System,

supervised by Helger Lipmaa, PhD,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 20th May 2013