UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

Computer Science & Artificial Intelligence Laboratory

BACHELOR'S DEGREE THESIS

ACCELERATING ZERO KNOWLEDGE PROOFS

Author

Laia Pujol Ventosa

Supervisor (MIT)

Daniel Sánchez

Tutor (UPC)

Antonio González

In partial fulfillment of the requirements for the

Bachelor's degree in Mathematics

Bachelor's degree in Informatics Engineering

October 2023

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Abstract

Zero-knowledge proofs represent a highly promising cryptographic tool that enables the validation of a statement's correctness without revealing any supplementary information. These proofs find utility in applications demanding both computational integrity and privacy, such as validating outsourced computation results, where confidential input values may be involved. However, a significant challenge hinders their practical adoption: the potentially time-consuming process of generating proofs. Consequently, this project investigates the feasibility of accelerating zero-knowledge proofs using hardware, aiming to overcome this critical hurdle.

Keywords: verifiable computation, zero-knowledge proofs, zk-SNARKs, learning with errors, lattice-based cryptography, accelerator

MSC2020: 94A60

Resum

Les proves de coneixement zero són una eina criptogràfica altament prometedora que permet demostrar que un predicat és correcte sense revelar informació addicional sobre aquest. Aquestes tipus de proves són útils en aplicacions que requereixen tant integritat computacional com privadesa, com ara verificar la correcció dels resultats d'una computació delegada a una altra entitat, on hi poden haver involucrats valors d'entrada confidencials. Tanmateix, té un impediment que obstaculitza la seva adopció pràctica: el procés potencialment lent de generació de les proves. Així doncs, aquest projecte explora la viabilitat d'accelerar les proves de coneixement zero mitjançant hardware, amb l'objectiu de superar aquest obstacle crític.

Paraules clau: computació verificable, proves the coneixement zero, zk-SNARKs, aprenentatge amb errors, criptografia basada en reticles, accelerador

MSC2020: 94A60

Resumen

Las pruebas de conocimiento cero representan una herramienta criptográfica altamente prometedora que permite demostrar la corrección de un predicado sin revelar información adicional. Estas pruebas son útiles en aplicaciones que requieren tanto integridad computacional como privacidad, como por ejemplo la validación de los resultados de una computación delegada a otra entidad, donde pueden estar involucrados valores de entrada confidenciales. Sin embargo, existe un desafío significativo que obstaculiza su adopción práctica: el proceso potencialmente lento de generación de pruebas. Como resultado, este proyecto explora la viabilidad de acelerar las pruebas de conocimiento cero utilizando hardware, con el objetivo de superar este obstáculo crítico.

Palabras clave: computación verificable, pruebas de conocimiento cero, zk-SNARKs, aprendizaje con errores, criptografía basada en retículos, acelerador

MSC2020: 94A60

First and foremost, I would like to thank Nikola Samardzic and Simon Langowski for their help and patience answering all my doubts and guiding me through the intricacies of this project. Your advice and support has been instrumental and deeply appreciated.

I would also like to express my sincere thanks to Professor Daniel Sanchez for hosting me over these recent months and welcoming me into his group. Being part of it and participating in its meetings has proved to be a truly enriching experience.

I want to thank Professor Antonio González for helping me get this opportunity and offering invaluable feedback on this project.

I am also grateful to the program Centre de Formació Interdisciplinària Superior (CFIS) for all their support during these last years. Moreover, this experience has been possible thanks to the funding provided by Fundació Privada Cellex and Fundació Ciutat de Valls. Thank you all for your contributions.

Lastly, thanks to my family and friends for being there throughout all these years. This journey would not have been possible without you.

Contents

Chapter 1

Introduction

In the ever-changing world of communication technologies, the need for trust and security has never been more paramount. In the past few years, individuals and organizations are increasingly relying on digital platforms for sensitive transactions or data processing. Therefore, the need to ensure the integrity, privacy and efficiency of these operations has become a major concern. This is where the world of verifiable computation and zero-knowledge proofs enters the stage.

Verifiable computation looks to address the challenge of efficiently validating computations that have been outsourced to another party. Furthermore, it also seeks to enable the verification of those computations without the need to reveal sensitive or private input data provided by the party in charge of the computation.

The cryptographic primitives that provide the tools for its implementation are zero-knowledge proofs. Succintly, they allow one party, the prover, to demonstrate the validity of a statement to another party, the verifier, without revealing any specific details about the statement itself. The main drawback for using them in practice is that the proof generation is very time-consuming. Therefore, in this project we want to study if a specific protocol that provides an implementation for verifiable computation can be accelerated through hardware.

In summary, the main goal of this work is to learn what Zero Knowledge Proofs are and to understand the basic concepts on how to implement them practically. Then, we want to investigate if it is possible to accelerate the generation of the proof using hardware design.

In the following chapters we will explore the concepts explained previously. Our journey will be divided into the following sections:

- In Chapter 2, we will dive into the basics of Verifiable Computation and Zero Knowledge Proofs. We will use examples to make these ideas clear and also look at real-life situations where researchers have suggested using them.

- Chapter 3 will introduce zk-SNARKs, a group of protocols that are widely used to implement Zero Knowledge Proofs. There will also be a practical explanation on how to generate a proof for an arithmetic circuit as well as a description of the characteristics of the protocol mainly used in this project.
- In Chapter 4 we will focus on the design of some units needed to accelerate the proof generation in a zk-SNARK.
- In Chapter 5 we will explain the code made with the purpose of executing workloads to be used as baselines. We will also do a back-of-the-envelope calculation of the time and area needed when using the hardware explained in the previous chapter.
- Chapter 6 and 7 will provide the conclusions of the project as well as some proposals on how this work could continue in the future.

Chapter 2

Background and motivation

The purpose of this chapter is to provide an understanding of essential concepts outlined in this thesis. Additionally, it aims to present compelling examples of their applications to the real world.

2.1. Verifiable Computation (VC)

Due to the uneven distribution of computational power, there is an increasing demand to outsource computation from a **weak client** (such as mobile phones) to one or more **powerful workers** (like an external server). This leads to the need to verify that the results obtained have been computed correctly. This verification serves the purpose of protecting the client against malfunctioning workers or malicious ones that are able to return plausible results without performing the computation.

2.1.1. Definition

In a nutshell, a **public verifiable computation scheme** allows a client to outsource the computation of a function F on an input u to one or more workers. The workers, who we shall call **provers**, will generate a proof of the correctness of the computation that the client, also known as **verifier**, can later check. The verification should require less work than would be needed for the evaluation of F.

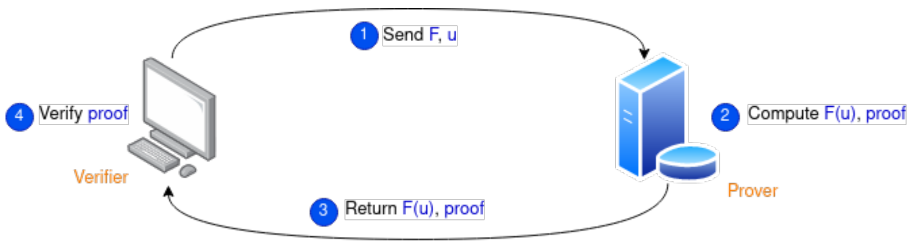


Figure 2.1: Verifiable Computation diagram.

We can find a more formal definition in [1] and [2]:

Definition 2.1.1 (Public Verifiable Computation Scheme)**:** A Public Verifiable Computation Scheme $\mathcal{VC}$ consists on the set of three polynomial-time algorithms defined below:

1. **KeyGen**$(F, 1^\lambda) \rightarrow (EK_F, VK_F)$: Given the security parameter λ (see definition 2.1.2), the randomized *key generation algorithm* generates a public evaluation key EK_F that encodes the target function F and a public verification key VK_F.
2. **Compute**$(EK_F, u) \rightarrow (y, \pi_y)$: In a deterministic algorithm, the worker uses the public evaluation key EK_F and input u to *compute* $F(u) \rightarrow y$ and the proof of y's correctness π_y.
3. **Verify**$(VK_F, u, y, \pi_y) \rightarrow \{0, 1\}$: The deterministic *verification* algorithm uses the verification key VK_F to output 1 if $F(u) = y$ and 0 otherwise.

In the field of cryptography, the security parameter measures the difficulty for adversaries to break a chryptographic scheme. In our definition the security parameter λ sets an upper limit on the probability that an adversary can successfully forge a proof using the EK_F and VK_F. It is usually expressed as a string of λ 1s, conventionally 1^λ. Its formal definition is given in [1]:

Definition 2.1.2 (Security parameter)**:** For any function F and any probabilistic polynomial-time adversary $\mathcal{A}$:

$$\Pr\left[(\hat{u}, \hat{y}, \hat{\pi}_y) \leftarrow \mathcal{A}(EK_F, VK_F) \ : \ \begin{array}{c} F(\hat{u}) \neq \hat{y} \\ \text{and} \\ \textbf{Verify}(VK_F, \hat{u}, \hat{y}, \hat{\pi}_y) = 1 \end{array} \right] \leq \textbf{negl}(\lambda)$$

2.1.2. Public and designated verifier

In the definition above, the entity that checks the correctness of the proof is called a **public verifier**. The main characteristic of this scheme is that the verifier can be any third party that is willing to participate in the verification process, i.e., with a single proof, everyone can be convinced of the correctness of the computation.

In contrast, we could also have a VC scheme that is **designated verifier**, where only this individual will be convinced of the proof. In this case, the verification key VK_F and the output of the verification function (i.e., if the proof provided by the prover is correct or not) need to be kept a secret in order to maintain the scheme secure against attacks.

2.2. Zero Knowledge Proofs

Until now, we have considered all the computation's inputs u to be known by the verifier. However, in some situations, the prover might have a private input w (also known as *witness*) they do not want to share with the verifier.

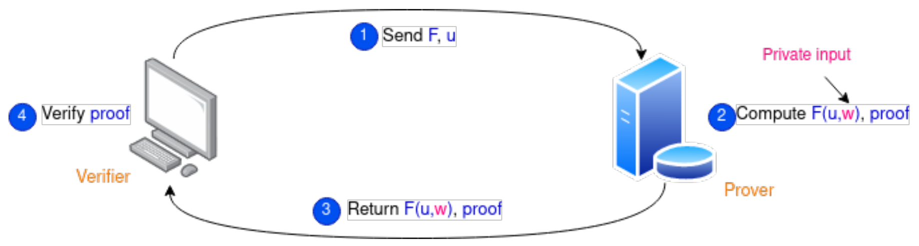


Figure 2.2: VC diagram with private inputs.

This leads to the need to make the $\mathcal{VC}$ scheme **zero-knowledge**, which essentially means that the verifier learns nothing about the prover's private input beyond the output of the computation.

2.2.1. Definition

Zero Knowledge Proofs were first conceived by Shafi Goldwasser, Silvio Micali and Charles Rackoff in 1985 ([3]). Since then, they have become more popular due to its multiple applications in fields such as verifiable computing, authentication or privacy.

Essentially, a **Zero Knowledge Proof (ZKP)** is a proof that yields nothing beyond the validity of the assertion ([4]). In other words, ZKP systems allow one party (the **prover**) to convince the other parties (the **verifiers**) that a **statement** is true without revealing any other information about it. A ZKP system should satisfy three properties ([5],[6]):

- **Completeness**: if a statement is true and both the prover and the verifier follow the protocol, then the prover can convince the verifier to accept the statement.
- **Soundness**: if a statement is false and the verifier follows the protocol, the verifier will not be convinced to accept it.
- **Zero-knowledge**: if a statement is true and the prover follows the protocol, the interaction will not reveal any additional information beyond its truth to the verifier.

2.2.2. Examples

To better understand the concept of Zero Knowledge Proofs, in this section we will provide illustrative examples to facilitate the reader's comprehension of the subject.

Red and green balls

In this example, suppose there are two balls: one red and one green. Alice (the prover) wants to prove to his color-blind friend Bob (the verifier) that these balls have different colors. To do so they follow these steps:

1. Bob takes the two balls, apparently identical to him, and shows Alice the initial position of the balls.
2. Then, in private, Bob tosses a coin and exchanges the balls if it lands on heads.
3. Bob shows the balls to Alice and asks her if he exchanged the balls.

If Alice is indeed able to tell apart the color of the two balls she will always know if Bob switched the balls. Hence, after repeating the steps enough times, Bob will be convinced that the colors of the two balls are different (completeness). It is worth noting that if Alice could not distinguish the balls she would have a 50% chance of guessing correctly each time. After repeating the test 20 times she would have a 1 in 1,048,576 chance of (luckily) answering right every time.

If Alice is also color-blind and the experiment is repeated enough times, at some point she will be "unlucky" and give a wrong answer. Thus, Bob will not be convinced that she is saying the truth (soundness).

Finally, Bob never discovers which is the color of each ball and therefore we achieve the zero-knowledge property.

Graph isomorphism

The following example is a more mathematical one and first we will recall the definition of graph isomorphism:

Definition 2.2.1 (Graph isomorphism)**:** A graph $\mathbf{G_0}$ is isomorphic to $\mathbf{G_1}$ if $\exists$ and isomorphism $\pi : [N] \to [N]$ such that $\forall i, j$:

$$(\pi(i), \pi(j)) \in E_1 \iff (i, j) \in E_0$$

where N is the number of nodes and E_0, E_1 are the sets of edges of $\mathbf{G_0}$, $\mathbf{G_1}$ respectively.

Note that saying that there is an isomorphism between two graphs is equivalent to the concept of being able to obtain one graph pattern from the other just by "moving the nodes". In Figure 2.3 we can see an example of two isomorphic graphs. In this case an isomorphism would be:

$$(1,2,3,4,5,6,7,8,9,10) \xrightarrow{\pi} (5,4,9,7,2,8,1,6,10,3)$$

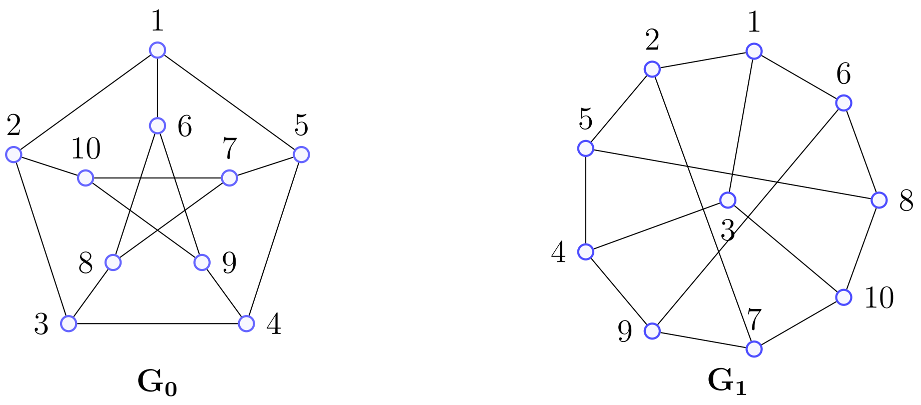


Figure 2.3: Isomorphic graphs.

In this example, Alice (our prover) wants to convince Bob (the verifier) that she knows an isomorphism for two graphs $\mathbf{G_0}$ and $\mathbf{G_1}$.

Before starting the interaction, Alice defines a new graph $\mathbf{H}$ that is isomorphic to $\mathbf{G_0}$ (and $\mathbf{G_1}$). She can do this by applying some random permutation $\gamma : [N] \to [N]$ to the nodes of $\mathbf{G_0}$ (it would be analogous for $\mathbf{G_1}$): $\mathbf{H} = \gamma(\mathbf{G_0})$. Then, the steps for the proof would be:

1. Alice creates and sends graph $\mathbf{H}$ to Bob.
2. Bob flips a coin and sends the result to Alice.
3. If the result was heads, Alice needs to return an isomorphism to $\mathbf{G_0}$. Otherwise, she has to send an isomorphism to $\mathbf{G_1}$.

If Alice knows an isomorphism π such that $\mathbf{G_0}$ and $\mathbf{G_1}$ are isomorphic, she will always be able to return an isomorphism to $\mathbf{H}$ to the prover. If it landed on heads, she will just return γ. In the other case, she can return $\gamma \circ \pi^{-1}$ because $\mathbf{H} = \gamma(\mathbf{G_0}) = \gamma(\pi^{-1}(\mathbf{G_1}))$. Thus, we have completeness.

We also have soundness after repeating the experience multiple times. If the prover does not truly know the isomorphism between both graphs, they can try to fake a proof by guessing which isomorphism will be required by the verifier and constructing $\mathbf{H}$ from $\mathbf{G_0}$ or $\mathbf{G_1}$. The probability of having a lucky guess once is $1/2$ so after k repetitions $\Pr[\textit{Accepting a false proof}] = 1/2^k$.

The zero-knowledge property is also achieved because the verifier never learns what the isomorphism between both graphs is.

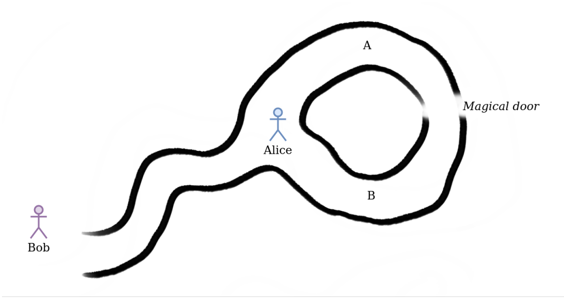
Ali-Baba Cave

Another well-known example of a ZKP is the one proposed by Jean-Jacques Quisquater and others in [7].

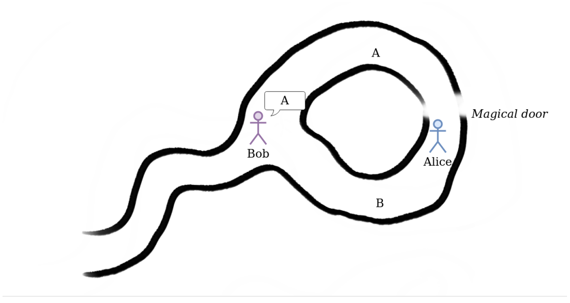
Suppose there is a cave with exactly one entrance from the outside. Inside, the cave forks into two passages (which we will call A and B) in an intersection that cannot be seen from outside the cave (see Figure 2.4). In this case Alice (prover) wants to show Bob (verifier) that she knows the password to a secret door that connects both passages.

The procedure to prove this statement is the following:

1. Alice goes inside the cave and chooses one of the two passages while Bob stands outside the cave. This means that Bob is unable to see which of both passages Alice has chosen.
2. Bob goes into the cave and stops at the intersection. Then he shouts which of the two passages (A or B) he wants Alice to come out from.
3. Alice comes back through the passage Bob asked her, using the secret door if needed.



(a) Alice chooses a passage to go in.



(b) Bob shouts through which passage Alice should come back.

Figure 2.4: Ali-Baba cave example.

After repeating this procedure enough times (similarly to the previous examples), Bob will be convinced that Alice's statement was true.

One important remark that needs to be made is the fact that a third party would not be convinced by this proof because Bob and Alice could have colluded to fake the results by agreeing beforehand which passage would be chosen each time. This means that Alice is just disclosing this knowledge to Bob and not to the rest of the world. One way to convince a third party would be to flip a coin (or some other random algorithm) to decide which passage should Alice come back from. Then, the observer would be reassured that Alice and Bob have not colluded.

2.2.3. Interactivity of a proof

All the examples we have seen until now in 2.2.2 have been **interactive proofs**. This means that the prover and the verifier can interact back and forth in order to complete the proof. In the following paragraphs we will explain a way to change them into **non-interactive proofs**. This will be useful because it will allow us to directly generate a proof that will convince the verifier without needing to exchange so many messages.

This procedure was first explained in [8] and it is commonly known as the **Fiat-Shamir heuristic**. Essentially, it takes a **public coin** interactive protocol[1] and turns it into a non-interactive one. The practical way to do this is making the prover generate the "coins" given by the verifier with a hash function.

In a public coin protocol, the interaction could be something like what is shown on the left of Figure 2.5. Firstly, the prover sends a message m_1, then the verifier sends back the result of one or more coin tosses and finally the prover replies back with m_2. Afterwards, the verifier checks that the result is correct using the public input x (if there is one), the result of the coin toss and the messages m_1 and m_2. In the graph isomorphism example 2.2.2, m_1 would be the graph $\mathbf{H}$ and m_2 would be the isomorphism to $\mathbf{G_0}$ or $\mathbf{G_1}$ depending on the result of the coin toss.

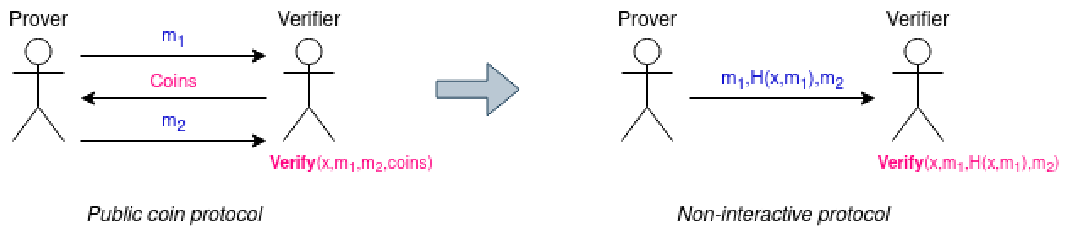


Figure 2.5: Fiat-Shamir heuristic. On the left we can see the public coin protocol and on the right we have the modified one so that it is non-interactive.

Let $H : \{0,1\}^* \rightarrow \{0,1\}^k$ be a cryptographic hash function[2]. The way to remove the interactivity is to apply the hash function H to the first message m_1 and the public input x. As the results of the hash function are supposed to be random, we

[1] In this setting, the random choices made by the verifier are made public.

[2] Informally, they are functions that map a string of arbitrary length to one with a fixed length k. They have many useful properties that can be applied to cryptography, for instance, it is unfeasible to retrieve an input from its hash under certain conditions or to find other inputs with the same hash value.

will replace the random value that comes from the verifier's coin toss by $H(x, m_1)$ and generate m_2 using that value. The verification is made using x, m_1, $H(x, m_1)$ and m_2 (right side of Figure 2.5).

Note that this procedure can be used for our previous examples in 2.2.2 because they all could use a coin to simulate the behaviour of the verifier. Therefore, we can transform them into non-interactive proofs using the Fiat-Shamir heuristic. Beware that this does not mean that every interactive zero-knowledge proof can be transformed so that they are not interactive. Firstly, they need to be converted into public-coin protocols. Also, for Fiat-Shamir heuristic to work, we need the prover to be computationally bounded so that they are not able to invert H. Nonetheless, it is used in many implementations of zero-knowledge proofs ([9]).

There is one last comment on this heuristic. Suppose that we consider H to be a random-oracle i.e. a theoretical "black box" that responds to every unique query with a truly random string and that gives the same response if the query is repeated. Then, completeness and soundness hold ([9]). However, there are still some questions about the security of the new protocol due, in part, to the fact that a random-oracle is often used as an idealised version of a hash function. More work on this can be found in [10] and [11].

2.2.4. Applications

Having explored some "toy examples" in section 2.2.2, let's now delve into some real-world applications.

Preventing identity theft

One of the first application ideas that were devised right after the conception of zero-knowledge proofs was a way to prevent identity theft ([8]). Essentially, if a user wants to prove their identity to some server, the user should create a zero-knowledge proof stating that they know the answer to a hard theorem such as factoring or graph isomorphism (as seen in 2.2.2). This method provides a way to avoid the use of passwords, which can be stolen from the server and even if they are encrypted, said encryption could be broken if it is not secure enough.

Nuclear disarmament

Another idea that was proposed and is rather interesting is using it in the context of nuclear disarmament, which has the aim of reducing or completely eliminating nuclear weapons from the world in order to avoid a potential nuclear catastrophe, both accidental or caused by wars. One of the reasons it is not being done faster is that countries need reassurance that the others are doing it as well, an that is when zero-knowledge proofs come in. Their privacy-preserving characteristics would allow to prove that certain operations regarding the elimination of nuclear weapons have been performed without needing to disclose sensitive information. Work on this was done by Boaz Barak among others and is shown in papers such as [12].

Forensics

Zero-knowledge proofs could also be used in forensics. This field consists in applying scientific methods to investigate and analyze evidence from crime scenes and other incidents, and can be used to aid in legal investigations and proceedings. In this case, we could prove that and individual's DNA is different from the one found in a crime scene without disclosing the DNA of the person. This application was explored in [13] by Ben Fisch, Daniel Freund and Moni Naor.

Cryptocurrencies with privacy and anonymity

One of the fields where zero-knowledge that has had the most success are cryptocurrencies. In this area, the motivation of zero-knowledge proofs was maintaining the privacy of the transactions the users make. There was work on this in [14] and there are some digital currencies such as Zcash that already use it.

Verification dilemmas in the law

There are other cases where we want to avoid exposing personal or proprietary information. For instance, when applying for a loan it is necessary to disclose personal data such as the salary to the bank, when for example it should only be necessary to prove that it is above a certain quantity. Also, in deal-making negotiations between firms, they might need to disclose some proprietary information that could be used by the competitor if the deal falls through. Additionally, litigants in court proceedings might have to reveal trade secrets in order to prove their claim. In [15], they explore these and other cases and present zero-knowledge proofs as a way to keep the sensitive data private.

Electronic voting

The possibility of voting electronically opens many possibilities like the option of having completely transparent vote counting. Then, zero-knowledge proofs can be used here to ensure voter's privacy. Some work on this is explained by Jens Groth in [16].

Fight disinformation

Lastly, a very peculiar example is the one presented in [17] on how to fight disinformation. The main idea is to be able to generate proofs that the images in some articles come from the place and time the article is talking about. To do so, the camera has an embedded certified signing key that cannot be extracted from it. There is a standard for it called C2PA but there is a problem with post-processing because these images might be cropped or manipulated. The solution for it is generating a proof that the new image comes from the original. That is where we can use zk-SNARKs, an implementation of all we have been talking about in this chapter and that we will discuss in the following chapter (3).

Chapter 3

Implementing Zero Knowledge with zk-SNARKs

Among the most promising and widely used candidates for a practical implementation of non-interactive zero-knowledge proofs we can find **zk-SNARKs**[1]. They encompass a group of protocols that enable the prover to generate a proof for a computation without revealing the values used in such computation.

In this chapter we will find a brief description of zk-SNARKs as well as an intuition on how to create a proof for a specific arithmetic circuit. Later we will also describe the specific protocol we will be working with and discuss some of its advantages and drawbacks with respect to other state-of-the-art protocols. Finally, we will provide a brief analysis of the algorithm we will want to accelerate in later chapters.

3.1. Overview

zk-SNARKs are a type of VC (2.1) schemes that allow the prover to generate ZKP. The acronym zk-SNARK stands for *zero knowledge succint non-interactive argument of knowledge*:

- **Zero-Kownledge**: the verifier of the statement cannot acquire any new information through the process, they only learn whether the statement is true or false.
- **Succint**: the proof of the statement is short and can be efficiently verified without the need to perform the full computation. In practice, this means that the length of the proof and the verification complexity should be sublinear in the size of the circuit computing F we are generating the proof for ([18]).

[1]There are also other type of protocols such as zk-STARKs (Zero-knowledge Scalable Transparent Argument of Knowledge) or Bulletproofs which we will not see in this project.

- **Non-interactive**: it does not require any back-and-forth interaction between the prover and verifier, the prover just needs to send a single message with the proof to the verifier.
- **ARgument of Knowledge**: the proof system provides evidence that the prover knows certain information required to complete the proof. It is worth noting that this is an argument and not a proof, though they are often used interchangeably. If a proof system only satisfies the soundness condition with respect to polynomial-time provers, then it is called an argument system.

3.2. Intuition of a proof's structure

At this point we have seen the definitions of Zero Knowledge Proofs and zk-SNARKS but we have not yet learned how to create a proof for a given function. In this section we are going to try to give an intuition on that. It is meant to be an introductory explanation based on the ones given in [19] and the lectures found in [9]. For a more complete one it can be useful to refer to some articles explaining their respective protocol in more detail ([1], [6] and [18]).

3.2.1. Arithmetic circuit setup

Suppose that we are given an arithmetic circuit C over a finite field $\mathbb{F}$ that computes the function we are generating the proof for[2]. The prover claims to know a secret witness w such that $C(x, w) = y$, where x is the public input of the circuit and y is its output.

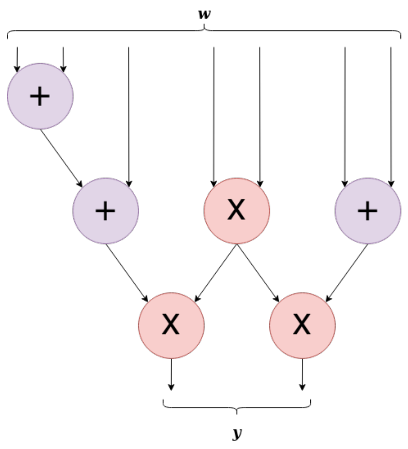


Figure 3.1: Circuit description. It is composed by 3 addition gates (purple) and 3 multiplication gates (red). The arrows at the top of the picture represent the secret input w and the ones at the bottom are part of the output y.

[2]In the example shown in Figure 3.1 the function would be $F(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = ((w_1 + w_2) + w_3)(w_4, w_5), (w_4 w_5)(w_6 + w_7)$.

The circuit we will use for this example is shown in Figure 3.1 and for the sake of simplicity, we will take the public input x to be empty.

In our example, the output (known by the verifier) will be the vector

$$y = (132, 108)$$

which we can get with the secret input (which the prover wants to prove they know)

$$w = (2, 5, 4, 3, 4, 7, 2)$$

To generate the proof, we will also numerate the multiplication gates (as can be seen in Figure 3.2) and define a vector with the inputs/outputs of the multiplication gates. If there is an addition gate before a multiplication one, we will add the inputs of the addition gate to the vector instead of the output value of the addition gate. For clarity, these values are shown in red in Figure 3.2.

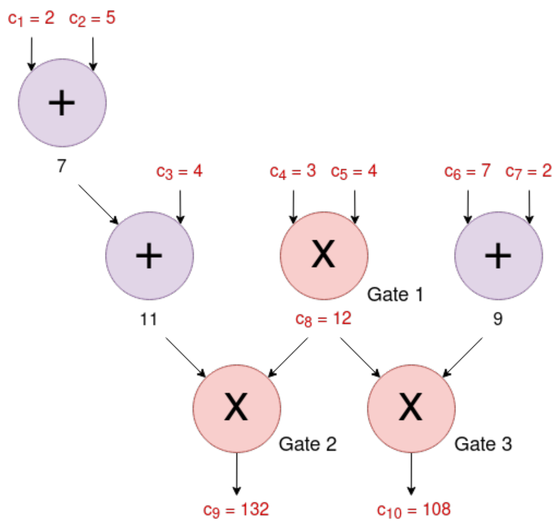


Figure 3.2: Circuit with values specified. The inputs/outputs of the multiplication gates are shown in red and each multiplication gate is numbered.

This will give us the following vector

$$c = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}) = (2, 5, 4, 3, 4, 7, 2, 12, 132, 108)$$

3.2.2. Selector polynomials

We are now able to start constructing the proof. For each value in vector c we will first define three **selector polynomials** which will encode if they are the left input, right input or output of a multiplication gate [3]. Each gate will have a

[3]As we will see, the addition gates will be implicit in the procedure.

distinct evaluation point assigned, i.e. when evaluating each selector polynomial at this point, we will get if it is the left input, right input or output of that gate. In this example we will choose these points as roots of the unity $\{\omega, \omega^2, \omega^3\}$ in $\mathbb{F}$. This is done to make polynomial manipulation faster as we will use a variation of a Fast Fourier Transform for finite fields[4].

For example, the selector polynomials assigned to the 1st element of c: $l_1(x)$, $r_1(x)$ and $o_1(x)$. They would be defined as:

- $l_1(x) : l_1(\omega) = 0$, $l_1(\omega^2) = 1$, $l_1(\omega^3) = 0$ because it is a left input to the second multiplication gate (through the additions in between) but that is not the case for the first and third multiplication gates.
- $r_1(x) : r_1(\omega) = 0$, $r_1(\omega^2) = 0$, $r_1(\omega^3) = 0$ as it is not a right input to any of the gates.
- $o_1(x) : o_1(\omega) = 0$, $o_1(\omega^2) = 0$, $o_1(\omega^3) = 0$ as it is not an output to any of the gates.

As it can be seen in Figure 3.2, these polynomials will be identical for the 2nd and 3rd element in c because even though they are inputs to different addition gates, they all end up being left inputs to the second multiplication gate. The 4th and 5th element are just a left and right input to the first gate respectively. The 6th and 7th one are both right inputs to the third gate.

In the case of the 8th element the polynomials would be defined as:

- $l_8(x) : l_8(\omega) = 0$, $l_8(\omega^2) = 0$, $l_8(\omega^3) = 1$ as it is a left input to the third multiplication gate.
- $r_8(x) : r_8(\omega) = 0$, $r_8(\omega^2) = 1$, $r_8(\omega^3) = 0$ because it is also a right input to the second gate.
- $o_8(x) : o_8(\omega) = 1$, $o_8(\omega^2) = 0$, $o_8(\omega^3) = 0$ as it is the output of the first gate.

And finally the 9th and 10th elements are outputs of the second and third gate respectively.

The value of the three selector polynomials assigned to every element of vector c at ω, ω^2 and ω^3 can be found on Table 3.1. Once we have all the selector polynomials, we will use them to define three polynomials for the whole circuit that will encode all the operations in it:

- $L(x) = \sum_{i=1}^{10} c_i \times l_i(x)$
- $R(x) = \sum_{i=1}^{10} c_i \times r_i(x)$
- $O(x) = \sum_{i=1}^{10} c_i \times o_i(x)$

[4]More information on that can be found in section 4.2.

	ω	ω^2	ω^3		ω	ω^2	ω^3		ω	ω^2	ω^3
$l_1(x)$	0	1	0	$r_1(x)$	0	0	0	$o_1(x)$	0	0	0
$l_2(x)$	0	1	0	$r_2(x)$	0	0	0	$o_2(x)$	0	0	0
$l_3(x)$	0	1	0	$r_3(x)$	0	0	0	$o_3(x)$	0	0	0
$l_4(x)$	1	0	0	$r_4(x)$	0	0	0	$o_4(x)$	0	0	0
$l_5(x)$	0	0	0	$r_5(x)$	1	0	0	$o_5(x)$	0	0	0
$l_6(x)$	0	0	0	$r_6(x)$	0	0	1	$o_6(x)$	0	0	0
$l_7(x)$	0	0	0	$r_7(x)$	0	0	1	$o_7(x)$	0	0	0
$l_8(x)$	0	0	1	$r_8(x)$	0	1	0	$o_8(x)$	1	0	0
$l_9(x)$	0	0	0	$r_9(x)$	0	0	0	$o_9(x)$	0	1	0
$l_{10}(x)$	0	0	0	$r_{10}(x)$	0	0	0	$o_{10}(x)$	0	0	1

Table 3.1: Selector polynomials.

It can be easily seen that when evaluating any of the polynomials at the point corresponding to any of the multiplication gates, we will get the expected value for it. For instance, if we evaluate the polynomials at ω^2:

$$L(\omega^2) = \sum_{i=1}^{10} c_i \times l_i(\omega^2) = c_1 + c_2 + c_3 = 2 + 5 + 4 = 11$$

$$R(\omega^2) = \sum_{i=1}^{10} c_i \times r_i(\omega^2) = c_8 = 12$$

$$O(\omega^2) = \sum_{i=1}^{10} c_i \times o_i(\omega^2) = c_9 = 132$$

which are the values expected for the 2nd gate.

3.2.3. Master and vanishing polynomial

Once we have all the selector polynomials and their coefficients, we define the **master polynomial** as

$$p(x) := L(x)R(x) - O(x) = (\sum_{i=1}^{m} c_i \times l_i(x)) \times (\sum_{i=1}^{m} c_i \times r_i(x)) - (\sum_{i=1}^{m} c_i \times o_i(x))$$

where m is the number of inputs and outputs of all the multiplication gates (the size of vector c), in our case $m = 10$. The key fact of defining this polynomial is that

$$p(\omega^i) = 0, \quad \forall i \in \{1, \dots, n\}$$

where n is the number of multiplication gates, in the example $n = 3$. This is true owing to the fact that in these points $p(x)$ is essentially multiplying the inputs and subtracting the gate output. Therefore, the n powers of ω will be roots of $p(x)$ which will lead us to define a **target** or **vanishing polynomial**:

$$t(x) = (x - \omega) \cdots (x - \omega^n)$$

that exactly divides $p(x)$ i.e. $p(x) = t(x)h(x)$ for some $h(x)$.

3.2.4. Quadratic Arithmetic Program

The last piece of the puzzle is the following claim which roughly states that:

The prover claims to know a witness w such that $C(x, w) = y$.	$\Longleftrightarrow$	The prover claims to know a vector c such that $p(x) = t(x)h(x)$.

The simple idea of the proof would be to do something like this:

1. The verifier samples a random point γ, calculates $t = t(\gamma)$ and gives γ to the verifier.
2. The prover calculates $h(x) = \frac{p(x)}{t(x)}$, computes $p = p(\gamma)$, $h = h(\gamma)$ and sends p and h to the verifier.
3. The verifier checks that $p = t \cdot h$ and accepts if the equality holds.

This approach is based on the **Schwartz-Zippel lemma** which for our problem states that given a value γ chosen at random uniformly from $\mathbb{F}$, the probability that $t(\gamma) = \alpha$ is $\frac{d}{|\mathbb{F}|}$ (where d is the degree of the polynomial).

Remark. Obviously, this construction has multiple issues, among them:

1. The prover could easily fake a proof because they can compute t. Hence they can take a random value h and compute $p = t \cdot h$, which will be accepted by the verifier because the equality holds.
2. Additionally, even if the prover had a $p(x)$ that truly satisfied the equality, it could be obtained from sampling a random $h(x)$ and computing $p(x) = t(x)h(x)$. This means that there is no enforcement that $p(x)$ represents the function we are computing.

In practice, these issues are solved by encrypting the polynomials and other values used in the protocol. For the sake of brevity, we will refrain from delving into it. Nonetheless, the construction used in [1] is thoroughly explained in [19] and even though it is not the protocol we will use in this project, its reading can be very useful to further understand the different issues that can emerge from this kind of protocols.

This construction we have explained is a Quadratic Arithmetic Program (QAP), whose definition was first implicitly found in [20] and is expressed in [1]:

Definition 3.2.1 (Quadratic Arithmetic Program): A Quadratic Arithmetic Program (QAP) $\mathcal{Q}$ over field $\mathbb{F}$ contains three sets of $m+1$ polynomials $\mathcal{L} = \{l_k(x)\}$, $\mathcal{R} = \{r_k(x)\}$, $\mathcal{O} = \{o_k(x)\}$ for $k \in \{0 \dots m\}$, and a target polynomial $t(x)$. Suppose f is a function that takes n elements of $\mathbb{F}$ as input and outputs n' elements of the same field, where $N = n + n'$ is the total of I/O elements. Then, we say that $\mathcal{Q}$ computes f if:
$(c_1, \cdots, c_N) \in \mathbb{F}^N$ is a valid assignment of F's inputs and outputs.

$$\Updownarrow$$

There exist coefficients $(c_{N+1}, \cdots, c_m)$ such that $t(x)$ divides $p(x)$, where

$$p(x) = \left(l_0(x) + \sum_{k=1}^{m} c_k \cdot l_k(x)\right) \cdot \left(r_0(x) + \sum_{k=1}^{m} c_k \cdot r_k(x)\right) - \left(o_0(x) + \sum_{k=1}^{m} c_k \cdot o_k(x)\right)$$

In other words, there must exist some polynomial $h(x)$ such that $p(x) = h(x) \cdot t(x)$. The size of $\mathcal{Q}$ is m. The degree of $\mathcal{Q}$ is $\deg(t(x))$.

Observe that in the definition there are $l_0(x), r_0(x), o_0(x)$ polynomials which allow us to have additive constants in the problem. See also that if we allow the selector polynomials to evaluate to constants different from $\{0, 1\}$ in the evaluation points chosen for each gate, the system can also have multiplicative constants.

3.2.5. Rank-1 Constraint Satisfiability

There is another construction that is more suited for expressing the different constraints for the function and that can be easily transformed into a QAP. It is called **Rank-1 Constrain Satisfiability (R1CS) system** and its definition is the following (found in [18]):

Definition 3.2.2 (Rank-1 Constraint Satisfiability): An R1CS system over a finite field $\mathbb{F}$ is specified by a tuple $\mathcal{CS} = (n, N_g, N_w, \{\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i\}_{i \in [N_g]})$ where $n, N_g, N_w \in \mathbb{N}$, $n \leq N_w$ and $a_i, b_i, c_i \in \mathbb{F}^{N_w+1}$. The system $\mathcal{CS}$ *is satisfiable for a statement* $\mathbf{x} \in \mathbb{F}^n$ if there exists a witness $\mathbf{w} \in \mathbb{F}^{N_w}$ such that

- $\mathbf{x} = (w_1, \dots, w_n)$ and
- $[1 \mid \mathbf{w^T}] \cdot \mathbf{a}_i * [1 \mid \mathbf{w^T}] \cdot \mathbf{b}_i = [1 \mid \mathbf{w^T}] \cdot \mathbf{c}_i$ for all $i \in [N_g]$

We express this concept as $\mathcal{CS}(\mathbf{x}, \mathbf{w}) = 1$ and we refer to n as the statement size, N_w as the number of variables and N_g as the number of constraints.

Observe that there is a 1 appended to the witness ($[1 \mid \mathbf{w^T}]$) in the last equation in the definition. It has the purpose of allowing additive constants in the system, just as the polynomials $l_0(x), r_0(x), o_0(x)$ did for the QAP.

Before creating the R1CS for the example, see that in our circuit 3.2 the computation can be summarized with:

$$\begin{aligned} c_4 \times c_5 &= c_8 \\ (c_1 + c_2 + c_3) \times c_8 &= c_9 \\ (c_6 + c_7) \times c_8 &= c_{10} \end{aligned}$$

In this case $n = N_w = 10$ and $N_g = 3$ and the vectors from the R1CS system will be:

$$A = \begin{array}{c|ccccccccccc|} & 1 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} \\ \hline a_1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_3 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}$$

$$B = \begin{array}{c|ccccccccccc|} & 1 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} \\ \hline b_1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ b_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ b_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$$C = \begin{array}{c|ccccccccccc|} & 1 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} \\ \hline c_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ c_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ c_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}$$

$$\begin{bmatrix} 1 \\ w \end{bmatrix} = \begin{array}{c|c|} 1 & 1 \\ c_1 & 2 \\ c_2 & 5 \\ c_3 & 4 \\ c_4 & 3 \\ c_5 & 4 \\ c_6 & 7 \\ c_7 & 2 \\ c_8 & 12 \\ c_9 & 132 \\ c_{10} & 108 \end{array}$$

The reason we have expressed the vectors in matrices is because checking that the system is satisfiable will be equivalent to cheking that the following equality holds:

$$A \cdot \begin{bmatrix} 1 \\ w \end{bmatrix} * B \cdot \begin{bmatrix} 1 \\ w \end{bmatrix} - C \cdot \begin{bmatrix} 1 \\ w \end{bmatrix} = \mathbf{0}$$

3.2.6. Linear PCPs

The kind of SNARKs that use the proof that we have described during this section are based on **Linear Probabilistic Checkable Proofs (PCPs)**. Informally, this is the kind of proof that can be verified by checking just a few random bits of it. The way to do this is by granting the verifier access to a linear oracle[5] $\pi : \mathbb{F}^l \to \mathbb{F}$ so that they can submit several queries and decide to accept or reject a statement on an input x ([18]).

[5]An oracle can be thought of a "black box" capable of providing a solution given an instance of a computational problem.

For instance, it is possible to transform the QAP into a 4-query linear PCP. Recall that in the original problem we have:

$$p(x) = (\sum_{i=1}^{m} c_i \times l_i(x)) \times (\sum_{i=1}^{m} c_i \times r_i(x)) - (\sum_{i=1}^{m} c_i \times o_i(x)) = t(x)h(x)$$

To perform the verification of the proof, we choose a random value γ and define the following vectors:

$$\begin{aligned} l_\gamma &= [l_1(\gamma), \ldots, l_m(\gamma)]^T \\ r_\gamma &= [r_1(\gamma), \ldots, r_m(\gamma)]^T \\ o_\gamma &= [o_1(\gamma), \ldots, o_m(\gamma)]^T \\ \Gamma &= [1, \gamma, \ldots, \gamma^{m-1}]^T \end{aligned}$$

Then, we can check the proof by making the following queries (shown in blue in the equation) to the oracle:

$$(c \cdot l_\gamma) \times (c \cdot r_\gamma) - (c \cdot o_\gamma) = t(\gamma)(h \cdot \Gamma)$$

Note that the dot product between two vectors is a linear function and that the verifier can easily compute $t(\gamma)$.

Besides that, as it was said before, it is a bit more difficult to implement it so that the prover can not fake a proof. To understand each implementation it is better to check the original papers ([1], [6],[18]).

3.3. The protocol

Now that we have covered the fundamentals of zk-SNARKs, our attention will be directed towards the protocol featured in the paper "*Shorter and Faster Post-Quantum Designated-Verifier zkSNARKs from Lattices*" by Yuval Ishai, Hang Su adn David J. Wu ([18]). Firstly, we will explain some of the protocol's advantages and drawbacks in comparison to other state-of-the-art protocols. Subsequently, we will provide an overview of the three primary functions within the protocol.

3.3.1. Benefits and drawbacks

In recent years, there has been a substantial amount of research dedicated to exploring constructions based on various assumptions and optimizing both the asymptotic and practical efficiency of zkSNARKs. A potential concern for the future for a significant portion of these new zk-SNARKs is that they rely on group-based or pairing-based assumptions. Thus, they are insecure against quantum adversaries[6].

[6]For example, Shor's algorithm can be used to break some of these cryptographic schemes.

The protocol we are exploring is based on the problem of Learning With Errors (LWE) which essentially encrypts some secret values by adding some noise to them. This problem comes from lattice-based cryptography and it is thought to be quantum secure ([18]).

Another advantageous side of the protocol is the fact that it uses fields that use modular arithmetic. This is an improvement with respect to other kinds of protocols because for instance, multiplications on elliptic curves (used in [6] for example) are more expensive.

This protocol is designated verifier, therefore, a proof can only convince the intended party and not the general public. This can be seen as both a benefit and a drawback depending on the applications. For blockchain applications, it would not work because not everyone could verify the proof. But for e-mail signatures, the prover might only want to convince the person they are sending the e-mail to, so that if anyone else finds it, they will not be able to know that it was truly written by them.

Lattice-based protocols are relatively new and have not been as explored as others. Thus, they are still slower than the ones based on pairing-based assumptions for example. We think that they can be accelerated to become equal or better. In the next section we will explain the algorithm and focus on the part we want to accelerate.

3.3.2. Implementation

In the implementation of this SNARK there are three main parts: key generation, proof generation and verification of the proof. As we will see, these will follow the scheme explained in section 2.1.

Key generation

This is the part of the proof that is in charge of the setup. Note that it only needs to happen once and then we can just execute the other two algorithms without needing to run this one as well. Therefore, it will not be important to accelerate it. We can see that the algorithm:

$\texttt{Setup}(1^\lambda, 1^\kappa) \rightarrow (\texttt{crs},\texttt{st})$

takes a security parameter λ and the system index κ as inputs. It outputs the common reference string `crs` and the verification state `st`. The `crs` is generated using a trusted setup and it is later used in the proof generation algorithm. The `st` is used to verify the proof and it should be secret to the designated verifier.

Proof generation

The proof generation algorithm:

`Proof generation`$(\mathrm{crs}, x, w) \rightarrow \pi$

is the one that takes as inputs the public input x, the witness w and the common reference string and then generates a proof. It is the costly step that we want to accelerate, as it has some characteristics that make it ideal for it. We will see more of this in the next chapter.

Verification

Finally, the verification step needs to output zero or one depending on the correctness of the proof:

`Verify`$(\mathrm{st}, x, \pi) \rightarrow \{0, 1\}$

The function takes the verification step, the public input and the proof as parameters and outputs a bit that is 0 or 1.

Chapter 4

Hardware design

The purpose of this chapter will be to outline the problem we aim to address and to present potential hardware solutions to tackle it.

4.1. Description of the problem

As it has been hinted throughout previous chapters, the problem we will focus on is *improving the proof generation time for zk-SNARKs* and more concretely, doing it for the protocol shown in [18]. To do so, in this chapter we will describe some strategies to design the hardware for the critical parts of the proof generation algorithm which collectively contribute to around 80% of the computation time for the proof. The parts that we will study are the computation of the H polynomial and a matrix vector multiplication.

4.1.1. Computing H polynomial

As we have seen in previous sections, computing the polynomial $H(x)$ is one of the fundamental parts of the proof. This is because the essence of the proof relies on a simple divisibility check (see Section 3.2). To compute this polynomial and comply with the protocol requirements it is needed to perform some polynomial operations such as interpolation or evaluation.

In Figure 4.1 we can see how H is computed based on the representation coming from R1CS. The most costly parts of this computation are the 7 NTTs (Number Theoretic Transform) and INTTs (Inverse Number Theoretic Transform) which are in charge of evaluating or interpolating a polynomial in an adequate set of points. In Section 4.2 we will see how to implement them in hardware to make them efficient. The other operations shown in the picture are scalar multiplications (M orange blocks), an element-wise product[1] (x) and element-wise subtraction (-).

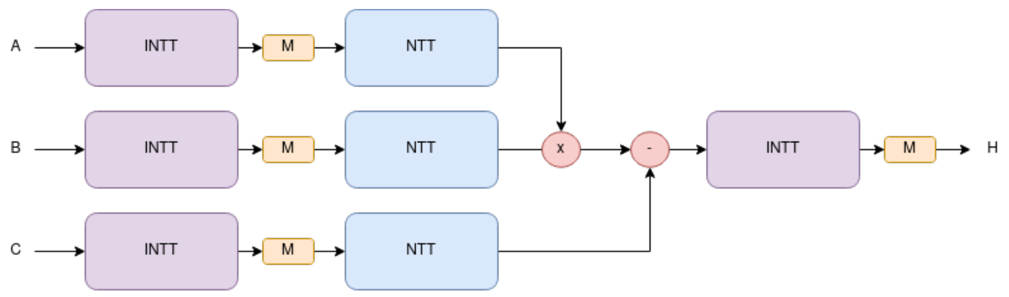


Figure 4.1: Diagram for the computation of the polynomial H during the proof generation algorithm.

4.1.2. Matrix vector multiplication

Later on, in the code there is also another time-consuming step which mainly consists on multiplying a matrix that is generated from a small seed with a vector that comes from previous computations. This part is the one that might take longer to compute in the algorithm with about 70% of running time.

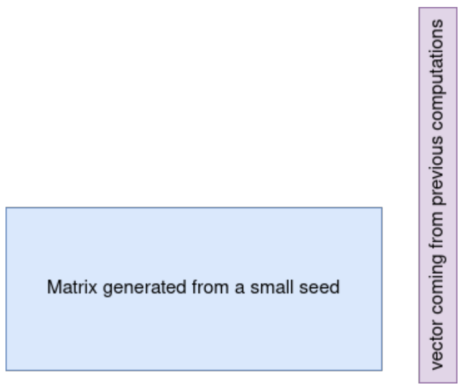

Matrix generated from a small seed
vector coming from previous computations

Figure 4.2: Matrix-vector multiplication used in the proof generation algorithm.

4.2. Number Theoretic Transform

As we have seen in previous sections, the evaluation and interpolation of polynomials are crucial steps in the algorithm. In this section we will introduce the **Number Theoretic Transform (NTT)**, which is a variant of the Fast Fourier Transform in finite fields. It will allow us to compute costly polynomial operations in a more efficient way. Later we will also introduce a way of accelerating it through hardware.

[1]Also called Hadamard product.

4.2.1. Improving polynomial multiplication

Let $\mathbb{F} = \mathbb{Z}/p\mathbb{Z}$ be the field of the integers modulo a prime p, $a = (a_0, \dots, a_{n-1}) \in \mathbb{F}^n$ and $b = (b_0, \dots, b_{n-1}) \in \mathbb{F}^n$. Then, we define the convolution of these two vectors as[2]:

Definition 4.2.1 (Vector convolution)**:** The convolution of two vectors a and b is given by:

$$a \otimes b \coloneqq (c_0, \dots, c_{2n-2}) \text{ where } c_k = \sum_{i=0}^{k} a_i b_{k-i}, \quad a_j, b_j = 0 \text{ for } j < 0 \text{ or } j \geq n.$$

Note that one of the most direct applications of the convolution is polynomial multiplication. Let $A(x) \coloneqq \sum_{i=0}^{n-1} a_i x^i$ and $B(x) \coloneqq \sum_{i=0}^{n-1} b_i x^i$. Then

$$\begin{aligned} A(x)B(x) &= \left(\sum_{i=0}^{n-1} a_i x^i\right)\left(\sum_{i=0}^{n-1} b_i x^i\right) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{i+j} = \\ &= \sum_{k=0}^{2n-2}\left(\sum_{i=0}^{k} a_i b_{k-i}\right) x^k = \sum_{k=0}^{2n-2} c_k x^k \eqqcolon C(x) \end{aligned}$$

As shown above, the coefficients of the resulting polynomial are the elements of $a \otimes b$. The problem of using the convolution for polynomial multiplication is that the algorithm is $O(n^2)$ which is too slow.

Before seeing how we can improve it, we will introduce some notation regarding polynomial representation:

Coefficient representation It is the one we have been using until now, where the polynomial is defined by the values that multiply the powers of x. To simplify matters later, we will extend the coefficients with 0s until we have $N = 2^t$ coefficients such that $n \leq N$:

$$A(x) = \sum_{i=0}^{N-1} a_i x^i, \; B(x) = \sum_{i=0}^{N-1} b_i x^i \qquad a_i, b_i = 0 \; \forall i \geq n$$

Point-value representation We can also use the evaluation of the polynomials at N points to represent them. Given $\alpha = (\alpha_0, \dots, \alpha_{N-1}) \in \mathbb{F}^N$, the values $A(\alpha_0), \dots, A(\alpha_{N-1})$ represent the polynomial $A(x)$.

In Figure 4.3 we can see a diagram for polynomial multiplication. As we have said before, the convolution has a complexity of $O(N^2)$ and it is trivial that the point-wise multiplication is $O(N)$. The interesting part comes with the evaluation and

[2]It comes from the definition of convolution for discrete functions.

interpolation operations. They both can be computed in $O(N \log N)$ when choosing adequate values for the point-value representation.

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Coefficient representation}\\ \text{of } a \text{ and } b\end{array}} \arrow[r, "\textit{Evaluation}"'] \arrow[d, "\textit{Convolution}"'] & \boxed{\begin{array}{c}\text{Point-value representation}\\ \text{of } a \text{ and } b\end{array}} \arrow[d, "\textit{Point-wise multiplication}"] \\
\boxed{\begin{array}{c}\text{Coefficient representation}\\ \text{of } c\end{array}} & \boxed{\begin{array}{c}\text{Point-value representation}\\ \text{of } c\end{array}} \arrow[l, "\textit{Interpolation}"']
\end{tikzcd}$$

Figure 4.3: Polynomial multiplication diagram.

4.2.2. Evaluation and interpolation using the Vandermonde matrix

As we have hinted in previous sections, the evaluation points we will choose[3] will be powers of the primitive Nth root of unity in the corresponding field. In this case, it is an $\omega \in \mathbb{F}$ such that $\omega^N \equiv 1 \mod p$. Therefore, we will take the evaluation points defined as $\alpha_i := \omega^i \quad \forall i \in 0, \dots, N-1$.

Now we can define the **Vandermonde matrix for** ω:

$$M_{N,\omega} = \begin{bmatrix} 1 & \alpha_0 & \cdots & \alpha_0^{N-1} \\ 1 & \alpha_1 & \cdots & \alpha_1^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_{N-1} & \cdots & \alpha_{N-1}^{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

We can evaluate the polynomial at N points by applying the matrix to the vector of coefficients and similarly, we can also obtain the coefficients from applying the inverse of the matrix to the vector of evaluated points:

Evaluation: $(M_{N,\omega} \cdot a)_i = A(\alpha_i) = A(\omega^i)$

Interpolation: $(M_{N,\omega}^{-1} \cdot C(\alpha))_i = c_i$

An interesting observation regarding the matrix is the following:

Proposition 1. $M_{N,\omega}^{-1} = \frac{1}{N} M_{N,\omega^{N-1}}$

[3]Note that when doing multiplication through the alternative path instead of doing the convolution means that we can choose whichever evaluation points we want because the final coefficient representation will be independent of this choice.

Proof. We want to prove that the equality $\frac{1}{N}M_{N,\omega^{N-1}} \cdot M_{N,\omega} = \mathbf{I}_N$ holds.

$$\left(\frac{1}{N}M_{N,\omega^{N-1}} \cdot M_{N,\omega}\right)_{i,j} = \frac{1}{N}\sum_{k=0}^{N-1}(\omega^{N-1})^{ik}\omega^{jk} = \frac{1}{N}\sum_{k=0}^{N-1}(\omega^{ikN})\omega^{k(j-i)} =$$
$$= \frac{1}{N}\sum_{k=0}^{N-1}\omega^{k(j-i)} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

where in the case $i \neq j$ we are using the geometric sum property:

$$\frac{1}{N}\sum_{k=0}^{N-1}\omega^{k(j-i)} = \frac{1}{N} \cdot \frac{\omega^{N(j-i)} - 1}{\omega^{j-i} - 1} = \frac{1}{N} \cdot \frac{1-1}{\omega^{j-i} - 1} = 0$$

$\square$

This property will make the inverse matrix easy to compute and will allow the algorithms for interpolation and evaluation to be very similar.

4.2.3. The radix-2 DIT algorithm

Let's now think about the algorithm for the evaluation. See that we can decompose the step of evaluating a polynomial into:

$$\left.\begin{aligned} A^{[0]}(x) &:= a_0 + a_2x + a_4x^2 + \cdots + a_{N-2}x^{\frac{N}{2}-1} \\ A^{[1]}(x) &:= a_1 + a_3x + a_5x^2 + \cdots + a_{N-1}x^{\frac{N}{2}-1} \end{aligned}\right\} \implies A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

which basically gives us a recursive case we can use in our program.

The algorithm given by this recursion is expressed in pseudo-code in Listing 4.1. The function `NTT` takes as inputs the vector of N coefficients a and a value ω and computes the evaluation of a at the powers of ω. As we have said, in this case ω will be the Nth root of unity in $\mathbb{F}$.

The steps of the algorithm are:

1. **Base case**: if N (which is equal to the size of a) is 1, the algorithm returns a because the polynomial is just the constant term.
2. Otherwise we store the even coefficients of a into vector $a^{[0]}$ and the odd ones into $a^{[1]}$.
3. Then, we call the `NTT` with vectors $a^{[0]}$ and $a^{[1]}$ with a value of ω^2 and we store it into $c^{[0]}$ and $c^{[1]}$ respectively. This means that $c^{[i]} = A^i(\omega^2)$ $i \in \{0, 1\}$.
4. We create a new vector c of N elements which will hold the result of the function i.e. $c_i = A(\omega^i)$ $i \in \{0, \dots, N-1\}$. Each element is obtained with

linear combinations of $c^{[0]}$ and $c^{[1]}$ and some powers of ω which we will call **twiddle factors**. More concretely, the code is using the following equalities:

$$A(\omega^k) = A^{[0]}(\omega^{2k}) + \omega^k A^{[1]}(\omega^{2k})$$

$$A(\omega^{k+\frac{N}{2}}) = A^{[0]}(\omega^{2k}\omega^N) + \omega^{k+\frac{N}{2}} A^{[1]}(\omega^{2k}\omega^N) = A^{[0]}(\omega^{2k}) + \omega^{k+\frac{N}{2}} A^{[1]}(\omega^{2k})$$

This last step is performed inside the `for` shown in the pseudocode.

```
NTT(a,$\omega$) {
    if (N==1) return a;
    $a^{[0]}$ = ($a_0$,$a_2$,...,$a_{N-2}$);
    $a^{[1]}$ = ($a_1$,$a_3$,...,$a_{N-1}$);
    $c^{[0]}$ = NTT($a^{[0]}$, $\omega^2$);
    $c^{[1]}$ = NTT($a^{[1]}$, $\omega^2$);
    for (int k = 0; k $\leq$ N/2 - 1; ++k) {
        $c_k$ = $c^{[0]}_k$ + $\omega^k c^{[1]}_k$;
        $c_{k+N/2}$ = $c^{[0]}_k$ + $\omega^{k+N/2} c^{[1]}_k$;
    }
    return c;
}
```

Listing 4.1: Pseudo-code for evaluating a at N powers of ω.

Therefore, we have an algorithm that allows us to evaluate a polynomial in adequate points with a complexity of $O(N \log N)$[4]. The algorithm for interpolation is very similar to the one we have seen here (see Proposition 1). Thus, by looking at diagram 4.3 we now can do evaluation, interpolation and multiplication in $O(N \log N)$.

4.2.4. Generic algorithm

The algorithm described above corresponds to the simplest case of the **Cooley-Tukey algorithm**[21], where in each recursive call we divide the NTT in two smaller NTTs with half the size of the original one[5]. The more general case considers splitting $N = N_1 N_2$ and doing the following steps (see Figure 4.4):

1. Reorganize the elements of the coefficient vector into a $N_2 \times N_1$ matrix.
2. Perform N_1 simultaneous NTTs with a vector size of N_2 (input data considered as a $N_2 \times N_1$ matrix).
3. Multiply the resulting data, considered as a $N_2 \times N_1$ matrix, by the corresponding twiddle factors and transpose the resulting matrix into a $N_2 \times N_1$ matrix.

[4]We will not see it here but it can be easily obtained by applying the master theorem for analysis of algorithms.

[5]This is the reason we took N to be a power of 2.

4. Perform N_2 simultaneous N_1-point NTTs on the resulting $N_1 \times N_2$ matrix.

Usually, N_1 or N_2 are a small value which we will call the **radix**. If N_1 is the radix we are performing a **Decimation In Time (DIT)** and otherwise, it is a **Decimation In Frequency (DIF)**. For example, in the previous code 4.1 we were doing a **radix-2 DIT**.

$$a = [a_0, a_1, \dots, a_{N_1N_2-1}] \xrightarrow{\text{Reorganize elements}} \begin{array}{c} \begin{matrix} \downarrow & \qquad & \downarrow \end{matrix} \\ \begin{bmatrix} a_0 & \cdots & a_{N_1-1} \\ a_{N_1} & \cdots & a_{2N_1-1} \\ \vdots & \ddots & \vdots \\ a_{(N_2-1)N_1} & \cdots & a_{N_2N_1-1} \end{bmatrix} \end{array}$$

$$\xrightarrow{N_1 \ \texttt{NTTs with } \omega^{N_1}} \begin{bmatrix} c_0^{[0]} & \cdots & c_0^{[N_1-1]} \\ c_1^{[0]} & \cdots & c_1^{[N_1-1]} \\ \vdots & \ddots & \vdots \\ c_{N_2-1}^{[0]} & \cdots & c_{N_2-1}^{[N_1-1]} \end{bmatrix}$$

$$\xrightarrow[\text{Twiddle factors}]{\text{Transpose +}} \begin{array}{c} \begin{matrix} \downarrow & \qquad \downarrow & \qquad\qquad & \qquad\qquad \downarrow \end{matrix} \\ \begin{bmatrix} c_0^{[0]} & c_1^{[0]} & \cdots & c_{N_2-1}^{[0]} \\ c_0^{[1]} & \omega c_1^{[1]} & \cdots & \omega^{N_2-1} c_{N_2-1}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ c_0^{[N_1-1]} & \omega^{N_1-1} c_1^{[N_1-1]} & \cdots & \omega^{(N_1-1)(N_2-1)} c_{N_2-1}^{[N_1-1]} \end{bmatrix} \end{array}$$

$$\xrightarrow{N_2 \ \texttt{NTTs with } \omega^{N_2}} \begin{bmatrix} c_0 & \cdots & c_{N_2-1} \\ c_{N_2} & \cdots & c_{2N_2-1} \\ \vdots & \ddots & \vdots \\ c_{(N_1-1)N_2} & \cdots & c_{N_1N_2-1} \end{bmatrix}$$

Figure 4.4: Diagram for the `NTT` with an input vector of $N = N_1N_2$ elements.

For completeness, we are going to show the correctness of the twiddle factors found in Figure 4.4 for the NTT:

Proposition 2. *The twiddle factor that multiplies $c_j^{[k]}$ after performing the first NTT is ω^{jk}.*

Proof. We want to see that after performing the second NTT (which evaluates vectors on powers of ω^{N_2}) on $\omega^{jk}c_j^{[k]}$, coefficient c_{iN_2+j} will have the evaluation of

A in ω^{iN_2+j}. We will also use the fact that $c_j^{[k]}$ is the result of performing an NTT with ω^{N_1} on the vector of coefficients a:

$$c_{iN_2+j} \overset{?}{=} \sum_{k=0}^{N_1-1} \omega^{iN_2k}(\omega^{jk}c_j^{[k]}) = \sum_{k=0}^{N_1-1} \omega^{(iN_2+j)k} \sum_{l=0}^{N_2-1} \omega^{jlN_1} a_{k+lN1}$$

Now, we use the fact that in $\mathbb{F}$ we have the identity $\omega^N = \omega^{N_1N_2} \equiv 1 \mod p$ to add some powers of $\omega^{N_1N_2}$ that do not alter the value of the previous equation:

$$\begin{aligned}\sum_{k=0}^{N_1-1} \omega^{(iN_2+j)k} \sum_{l=0}^{N_2-1} \omega^{jlN_1+liN_1N_2} a_{k+lN1} &= \sum_{k=0}^{N_1-1}\sum_{l=0}^{N_2-1} \omega^{(iN_2+j)(k+lN_1)} a_{k+lN1} \\ &= \sum_{l=0}^{N_2-1}\sum_{k=0}^{N_1-1} \omega^{(iN_2+j)(k+lN_1)} a_{k+lN1}\end{aligned}$$

And we finally do the following change of variable $s = k + lN_1$ that gives us the result we were looking for:

$$\sum_{s=0}^{N_1N_2-1} \omega^{(iN_2+j)s} a_s = A(\omega^{iN_2+j})$$

Therefore, we have seen that:

$$A(\omega^{iN_2+j}) =: c_{iN_2+j} = \sum_{k=0}^{N_1-1} \omega^{(iN_2+j)k} c_j^{[k]}$$

where $i = 0, \dots, N_1 - 1$ and $j = 0, \dots, N_2 - 1$. $\square$

4.2.5. Hardware design

After giving all the mathematical background, we are now ready to design the hardware implementation. The initial idea would be to just translate the code given in Listing 4.1 into its hardware equivalent. This is the idea that is used in the butterfly diagram[6] shown in Figure 4.5.

In the picture we can see an 8-input NTT that uses the 2-radix recursive algorithm by decomposing the problem into two $\frac{N}{2} = 4$ NTTs. In hardware, the lines in the figure would be translated into wires and on the ones that have a power of ω next to them, we would add a multiplier to perform the operation. Then, we would follow the exact same strategy to do the 4-point NTT, then the one with 2 elements and finally the one with just 1 (the base case of the recursion).

[6]It is called butterfly diagram because its shape is thought to resemble a butterfly.

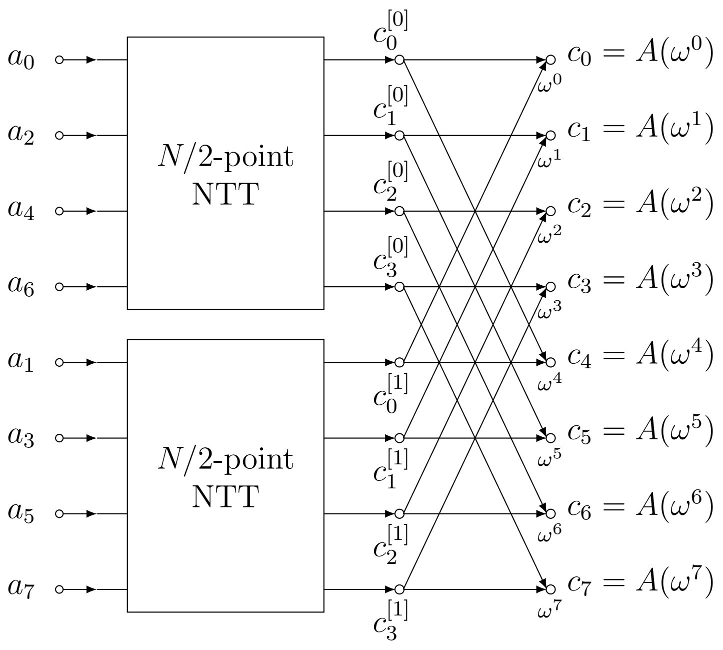


Figure 4.5: Butterfly diagram for an 8-point NTT. Adapted from [22].

This is a very common implementation of the NTT. However, if the input size gets too large, it is no longer feasible to implement it like this. One answer to this problem was proposed in the paper *FFTs in External or Hierarchical Memory* by David H. Bailey ([23]). It essentially consists on using the generic case of the algorithm we showed in the previous section where instead of directly computing the NTT it is split into 4 steps: perform N_1 N_2-point NTTs, multiply the results by the corresponding twiddle factors, transpose it and compute N_2 N_1-point NTTs.

This implementation of the NTT was done in *F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption* [24]. In the paper, the idea is to implement an N-point NTT as a composition of smaller E element NTTs, where E is the number of vector lanes in the accelerator. In Figure 4.6 there is an example on how to implement a 16-point NTT by using 4-point NTTs ($E = 4$ and it is analogous to using $N_1 = N_2 = 4$ in the previous section).

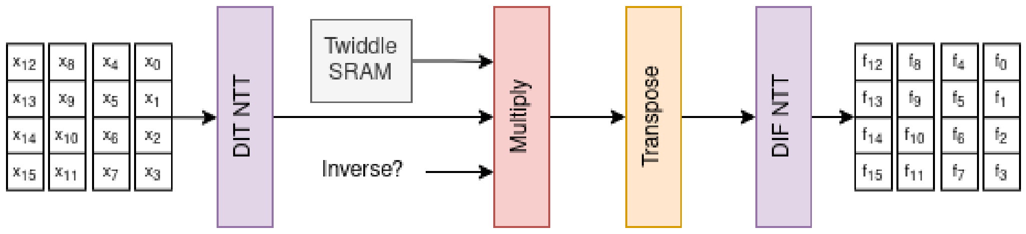

x_{12} x_8 x_4 x_0
x_{13} x_9 x_5 x_1
x_{14} x_{10} x_6 x_2
x_{15} x_{11} x_7 x_3
DIT NTT
Twiddle SRAM
Inverse?
Multiply
Transpose
DIF NTT
f_{12} f_8 f_4 f_0
f_{13} f_9 f_5 f_1
f_{14} f_{10} f_6 f_2
f_{15} f_{11} f_7 f_3

Figure 4.6: Four-step NTT datapath that uses 4-point NTTs to implement 16-point NTTs. Adapted from [24].

Even though this implementation of the NTT adds a multiplication (in red in Figure 4.6) to the algorithm, it has other advantages besides allowing bigger NTT sizes. For one, it produces an ordered NTT but it also grants the option of performing an INTT using the same pipeline by modifying the contents in the Twiddle SRAM as explained in [24].

Another important unit in this pipeline is the one that is used for transposing a matrix of size $E \times E$. This unit can be implemented by using the identity:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^T = \left[\begin{array}{c|c} A^T & C^T \\ \hline B^T & D^T \end{array}\right]$$

Then the idea is to have a *quadrant-swap* unit (left of Figure 4.7) of size $K \times K$ that follows three steps, where each of them takes $K/2$ cycles. In cycle `i`:

1. First step: read `A[i]` and `C[i]` and store them in `top[i]` and `bottom[i]`, respectively.
2. Second step: read `B[i]` and `D[i]`. Activate the first swap MUX so that `D[i]` is stored in `top[i]` and `A[i]` is outputed. Activate also the bypass line to output `B[i]` as well.
3. Third step: output `D[i]` from `top[i]` and `C[i]` and activate the second swap MUX so that they are correctly ordered (`C[i] on top`).

The benefits of using this design is that it is fully pipelined because steps 1 and 3 can be done in parallel and it can be used for all values of N by bypassing some of the initial quadrant swaps (where $N = G \times E$ and with power-of-2 $G < E$). The structure of the full unit for $E = 8$ can be found in Figure 4.7.

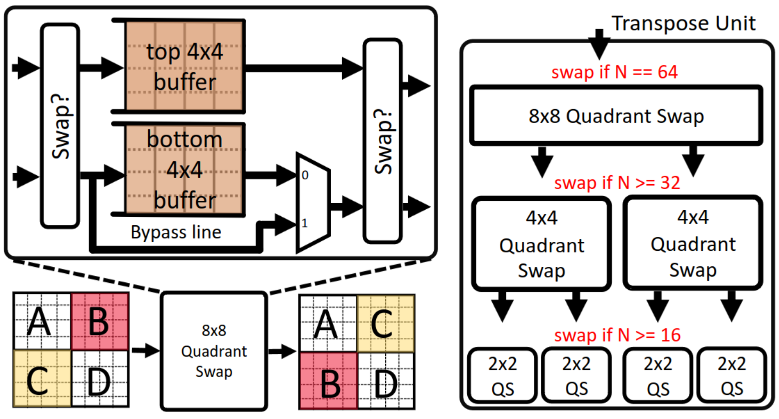


Figure 4.7: Transpose unit for an 8×8 matrix (right) and its component quadrant-swap unit (left). From [24].

4.3. Matrix vector multiplication

The other costly step in the computation is a dense matrix vector multiplication. One of the key facts is that the matrix is generated from a seed and, therefore, we do not need to load it from memory. First we will explain the naïve way to do a matrix-vector multiplication and then we are going to explain how to make it more memory-friendly.

4.3.1. *Naïve* algorithm

Suppose we want to multiply a matrix A with size $n \times m$ by a vector b with length m to obtain a new vector c of length n:

$$A \cdot b = \begin{bmatrix} a_{0,0} & \cdots & a_{0,m-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,m-1} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ \vdots \\ b_{m-1} \end{bmatrix} = \begin{bmatrix} a_{0,0}b_0 + \cdots + a_{0,m-1}b_{m-1} \\ \vdots \\ a_{n-1,0}b_0 + \cdots + a_{n-1,m-1}b_{m-1} \end{bmatrix} =: c$$

The coefficients of c can be computed as

$$c_i = \sum_{j=0}^{m-1} a_{i,k} b_j$$

and that means that the intuitive algorithm to implement it would be the one expressed in Listing 4.2.

```
MVM(A,b) {
    n = nrows(A);
    m = nrows(b);
    vector c(n,0);
    for (i = 0; i < n; ++i) {
        for (j = 0; j < m; ++j) {
            c[i] += A[i][j]*b[j];
        }
    }
    return c;
}
```

Listing 4.2: Intuitive pseudo-code for computing a matrix vector multiplication.

Though this algorithm might be fine if n and m are small, it becomes a problem with bigger computations. The reason for that is that the contents of the vector and the matrix loaded from memory can be spilled if they occupy more space than the one supplied by the registers or the cache used to hold these values. It also does not benefit from the possibilities of having more than one core in the processor. Therefore, the computation can be improved when taking the underlying hardware into account.

4.3.2. Tiling

To improve the performance of the pseudo-code seen before, we will rely on these concepts:

- **Data reuse**: it consists on efficiently using data that is already in a fast-access level of the memory hierarchy (for example, the registers or the cache). The issue with this kind of memories is that often they are not big enough to hold all date necessary for the computation. Thus, the objective is to be able to use the data as much as possible (following the code guidelines) before returning it to memory to make space for other elements.
- **Data locality**: data is typically retrieved from memory in groups or blocks of elements. Hence, it is important to consider how is the data stored in memory and strive to access consecutively the elements that are in the same block (which is similar to element-wise data reuse but on a block level). This approach minimizes the need to repeatedly retrieve the entire block from memory.
- **Exploiting parallelism**: in a processor there are often different cores or functional units that can be used in parallel. To improve performance it is key to have an adequate data distribution and load balancing among them.

In our case, the computation we are performing is a matrix vector where the *matrix is randomly generated from a seed*. Therefore, the data we will need for the computation are the elements from the vector and the seed to generate the matrix. We will also need space to store the temporary multiplication results before returning them to memory.

Observe that each element of the matrix only needs to be used once. Thus it is not necessary to store the randomly generated value after its initial use. This is not the case for the vector elements, as each one multiplies each value in a matrix column. Consequently, the idea is to compute partial results of the resulting vector c by computing the multiplication of a block or tile of k elements of vector b and its corresponding A columns. This idea is conveyed in Figure 4.8, where the program should first compute the result of multiplying the first $k = 2$ elements of vector b by the first two columns of matrix A. Then, it should load the next k elements of b (overwriting the previous ones if necessary, as they are no longer needed) and accumulate the results of the new multiplication. By executing the program this way, we would avoid unnecessarily reading the vector more than once.

Note also that by doing this it is also possible to distribute the computation between several different cores/units. Each one would receive the computation of one or more tiles and the only thing that it is needed to do at the end is to sum the results coming from the different cores.

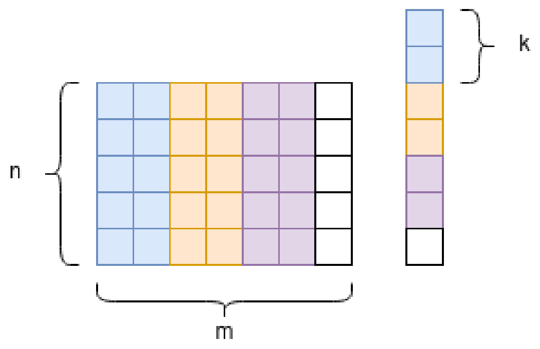
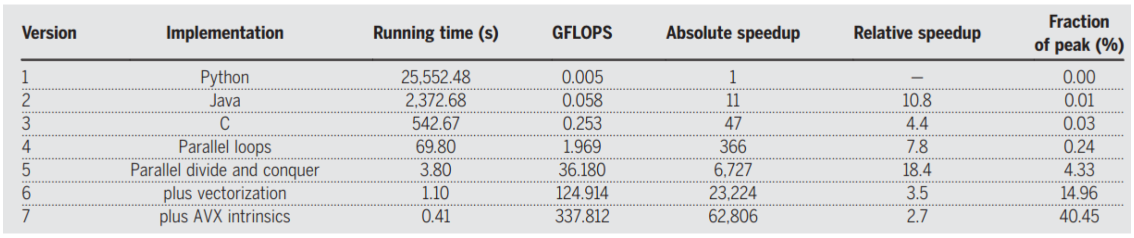


Figure 4.8: Example of tiling in a matrix vector multiplication.

By doing these steps we would achieve a much better use of the resources available because we would avoid unnecessary loads from memory and use all the cores/units available.

As a curiosity, an example on how a program is coded influences the performance of it is brought up in the article *There's plenty of room at the Top: What will drive computer performance after Moore's law?* ([25]).In the article they compare the performances obtained from running a code that computes a the multiplication of two 4096-by-4096 matrices. First they show that by changing the programming language used (Python, Java or C) we can get an execution time that is 47 times faster. And not only that, an even more impressive gain can be obtained from tailoring the code to the specific features of the hardware by using its 18 cores (version 4), exploiting the memory hierarchy (version 5), vectorizing the code (version 6) and using the Intel's special intructions (version 7). As it can be seen the absolute speedup is incredible!



Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Figure 4.9: Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. *Absolute speedup* is time relative to Python, and *relative speedup*, which is shown with an additional digit of precision, is time relative to the preceding line. *Fraction of peak* is GFLOPS relative to the computer's peak 835 GFLOPS. From [25].

What we have seen so far consists on correctly using the hardware resources by programming the code correctly, the design of the hardware comes from the same ideas explained for software. Essentially, the only things needed for it would be (1) Some circuitry to compute the matrix random elements[7] and (2) Circuitry that is able to compute matrix vector product. For the latter, the ideas explained before translate to having units that can compute several MADD each cycle (multiplications from different rows of the matrix can be done in parallel) and after k elements they have a reduction operation where they add the results from the other units.

As there are not many dependencies in this computation, it is possible to achieve a high utilization of the units and to fully parallelize it. The size of each block would depend on the size of the circuits that would be used for the proof and on the area allowed for the whole chip. There are some estimations on the time needed to perform the proof generation at the end of the next section.

[7]This is computationally cheap as it uses a Pseudorandom function (PRF).

Chapter 5

Evaluation

An essential part of designing hardware (or making performance improvements in a program) is having baselines to test it. Their purpose is allowing the comparison with both the original performance and against work from other research.

In [18] they do have an proof-of-concept implementation of their protocol, but it does not allow to input arithmetic circuits. Instead they just create a random number of constraints in some field $\mathbb{F}$ and run the program. Because of that, in this chapter we will talk about how to be able to have common arithmetic circuits as an input to the protocol. There were two ideas for that:

- Design a language to describe the circuit inputs and create the code to create the constraints in the program.
- Integrate `jsnark` ([26]), a library for building circuits for this type of SNARKs, in the program.

Another thing that is also important for the evaluation is knowing the time and area that would be needed for using an accelerator instead of just a CPU. This is the reason why in Section 5.3 we do a back-of-the-envelope calculation of these two parameters.

5.1. Arbitrary input circuits

The goal of this first approach was to be able to have arbitrary arithmetic circuits as an input of the proof generated with the protocol explained in [18]. Here, we will learn how to do it and we will start by describing the language used for the circuit description. Then, we will explain the code needed to read and run the proof using it, along with some problems there were during its implementation. Finally there will be a description of several workloads, how they were obtained and transformed to our language and some time results using a CPU.

5.1.1. Circuit description

The operations we wanted to support for the circuit were addition and multiplication in field $\mathbb{F}$ and the boolean operations `and`, `xor` and `inv`. Also, we wanted to differentiate between having a boolean or just a field element input. Evidently, the language can be extended to incorporate more operations but these were the ones needed for the tests we wanted to run. To implement each operation the syntax was:

- `i input_value idx`
- `b boolean_input_value idx`
- `AND idx1 idx2 idx_res`
- `XOR idx1 idx2 idx_res`
- `INV idx1 idx2 idx_res`
- `+ idx1 idx2 idx_res`
- `* idx1 idx2 idx_res`

The input operations `i` and `b` have two parameters: the value of the input and the index of the wire it will be assigned to. The other operations (`AND` `XOR`, `INV`, `+` and `*`) describe gates and have three parameters: the indices of the two wires entering the gate and the index of the output wire.

5.1.2. Explaining the code

The original code from [18] can be found in the Github project `lattice-zksnark` ([27]). This code is constructed over `libsnark` ([28]), a c++ library for implementing zk-SNARKs. In this library there are implementations of other zk-SNARKs such as Groth16 ([6]) or Pinocchio ([1]).

One of the advantages of using this library is that it has two gadget libraries that can be used for constructing R1CS instances out of modular gadget classes:

- `gadgetlib1`: it is a low-level library that exposes all the features of the zk-SNARKs. Its design is based on templates (which will be key afterwards) and it is used for most of the constraint-building in `libsnark`, both internal and in examples and applications
- `gadgetlib2`: this is an alternative library that also allows constructing R1CS instances. One of its advantages is that it is better documented than `gadgetlib1` and it is also easier to use. On the other hand, it does not use templates and fewer useful gadgets are provided.

The initial idea was to use `gadgetlib2` because, as it was said before, it is much easier to use due to being better documented. However, the problem with it was that the field used in every place of the code was hard-coded to it. Thus, as `libsnark` is a library that is primarily meant to be used with elliptic curves, it was not useful with our protocol, which uses the field of integers modulo a prime p. As a consequence, the library used to program the code was chosen to be `gadgetlib1`.

The first step of the code was to read its description, which is essentially a sequence of the instructions such as the ones described before. After this step, it is possible to create the R1CS constraints. The class that holds all the information regarding the constraints, its values and other similar information is the `protoboard`. To have a good structure for the code, we created different gadgets extending the `gadget` class from `gadgetlib1`. These gadgets facilitate the creation of the `protoboard` by encapsulating the behaviour of each gate in it. For every gadget class it was needed to have two functions defined:

- `generate_r1cs_constraints`: its target is to create the R1CS constraint by providing the variable indexes and coefficients accompanying each of them. Recall that a constraint has the form of $a \times b = c$, where a, b and c are linear combinations of other variables in the system.
- `generate_r1cs_witness`: this function is in charge of creating the witness values from the input (which can be private).

```
template<typename FieldT>
MULT_gadget<FieldT>::MULT_gadget(protoboard<FieldT> &pb,
                        const pb_linear_combination<FieldT> &A,
                        const pb_linear_combination<FieldT> &B,
                        const pb_linear_combination<FieldT> &out,
                        const std::string &annotation_prefix) :
    gadget<FieldT>(pb, annotation_prefix),
    A(A),
    B(B),
    out(out) {}

template<typename FieldT>
void MULT_gadget<FieldT>::generate_r1cs_constraints() {
    /* out = A * B */
    this->pb.add_r1cs_constraint(r1cs_constraint<FieldT>(A, B, out),
                          FMT(this->annotation_prefix, "out = A*B"));
}

template<typename FieldT>
void MULT_gadget<FieldT>::generate_r1cs_witness() {
    this->pb.lc_val(out) = this->pb.lc_val(A) * this->pb.lc_val(B);
}
```

Listing 5.1: Code for creating a multiplication gadget.

In Listing 5.1 we can see the code needed to create a multiplication gadget. In the constructor class we can see the three elements holding the linear combinations `A`, `B` and `C` and the `protoboard` we will associate the gadget with. In the `generate_constraints` function, we call the `protoboard` function that adds a `r1cs_constraint`. The constructor of this last class takes three linear combinations and an annotation (which can be used in debugging). It interprets the three elements just as we said before: the first linear combination multiplies the second one and should be equal to the third. Finally, the value for the witness associated to this constraint gets assigned in `generate_r1cs_witness`. This function calls the evaluation of the variables `A` and `B` and assigns the value of its product to the variable `out`.

For the other gadgets the procedure is quite similar to the one for the multiplication gate. We will use the following constraints to create the other gates:

- To impose that a value is boolean: $A \times (1 - A) = 0$
- AND gate: $A \times B = out$
- XOR gate: $2A \times B = A + B - out$
- INV gate: $1 \times (1 - A) = out$
- Addition gate: $1 \times (A + B) = out$

Even though there are other steps in the code that initialize parameters, check the validity of the constraints and so on and so forth, what we have explained until now are the key ones. After creating the `protoboard`, we can obtain the inputs needed to call the function `run_r1cs_lattice_snark`, which is the one that encapsulates the procedures explained in [18] to generate a key, create a proof and verify it.

5.1.3. Workloads

Once we have the code working and it can be used to run proofs for simple arithmetic circuits, it is time to find more interesting ones. In [29] there are examples of "Bristol Fashion" circuits which are circuits that are given in the following format:

- A line defining the number of gates and wires in the circuit.
- The number of input values n_{iv} and then n_{iv} numbers defining the number of input wires per input value.
- The number of input values n_{ov} and then n_{ov} numbers defining the number of input wires per input value.
- The gates, given in topological order, defined by:
  - Number of input wires.
  - Number of output wires.

- List of input wires
- List of output wires
- The gate operation (XOR, AND, INV, EQ, EQW or MAND)

Therefore, to use them it was necessary to first convert them to our own circuit description language. As ours is more simple, it was only needed to remove the unnecessary information and restructure the order followed in each line.

Even though there are more circuits in [29], the ones we were interested in were SHA-2 and AES.

Secure Hash Algorithm 2

SHA-2 is a set of cryptographic hash functions that were designed by the National Security Agency of the United States. The purpose of an n-bit hash function is to map arbitrary length messages to n-bit hash values. In the context of cryptography, they should also be *one-way* and *collision-resistant*, which means that it is difficult to find a message that hashes to a specific hash value and it is also hard to find two messages that hash to the same hash value. There are different hashes in the SHA-2 family: SHA-256, SHA-384 and SHA-512 which provide 128, 256 and 192 bits of security respectively ([30]).

SHA-256 starts with a preprocessing step where the message is padded so that the result is a multiple of 512 bits and then is split into 512-bit message blocks $M^{(1)} \ldots M^{(N)}$. Then, each message block is processed one at a time using the following equation:

$$H^{(i)} = H^{(i-1)} + C_{M^{(i)}}(H^{(i-1)})$$

where C is the SHA-256 compression function and the addition is word-wise and modulo 2^{32}. There is a fixed initial hash value $H^{(0)}$ that comes from the fractional parts of the square roots of the first eigth primes. Therefore, the hash for the message is $H^{(N)}$.

The circuits provided in [29] are simulating one step of this recursion, its inputs are the 512-bits message and a 256-bits hash. The procedure explained is very similar for SHA-384 and SHA-512 but changing the size of the message block among other things. In Table 5.1 we can see the time spent generating a proof for both functions.

Function	#constraints	Polynomial H	MVM	Total proof time
SHA-256	139,264	0.2958	0.9654	1.5031
SHA-512	393,216	0.3249	2.6574	4.1030

Table 5.1: Computing time for SHA-2 functions expressed in seconds. The 3rd and 4th column refer to the time spent computing H and the matrix-vector multiplication respectively. The results are obtained using 8 cores of an i5-8250U at 1.60GHz.

Advanced Encryption Standard

AES, also known as **Rijndael**, is a specification for the encryption of electronic data ([31]) established in 2001 by the US National Institute of Standards and Technology (NIST). It is a *symmetric cipher*, which means that you need the same secret key to encrypt a plaintext and decrypt its corresponding ciphertext. The key can be 128, 192 or 256 bits long. AES is also a *block cipher* as it operates on fixed-length groups of bits, in this case, 128.

The algorithm consists on a sequence of 10, 12 or 14 rounds for a 128, 192 or 256 bit key. There is a different key for each round that is generated from the initial one using a key schedule algorithm. The input for the algorithm are the 16 bytes of the plaintext structured as a 4×4 matrix. Then, on each round, there are operations that involve byte substitution, row shifting, matrix multiplication and additions of the round key performed on the matrix. After the corresponding number of rounds, the message is considered to be encrypted. Table 5.2 shows the results of generating a proof for AES.

Function	#constraints	Polynomial H	MVM	Total proof time
AES-128	40,960	0.0772	0.2808	0.4530
AES-192	49,152	0.0857	0.3213	0.5103
AES-256	65,536	0.0567	0.6308	0.7966

Table 5.2: Computing time for AES functions expressed in seconds. The 3rd and 4th column refer to the time spent computing H and the matrix-vector multiplication respectively. The results are obtained using 8 cores of an i5-8250U at 1.60GHz.

5.2. Integrating `jsnark`

Another option to obtain workloads to use as benchmarks was using `jsnark`. It is essentially a Java library made for building circuits for zk-SNARKs and is used to create workloads in other papers such as *PipeZK* ([32]). This library is also based on using gadgets and it uses `libsnark` as a backend but it has several advantages with respect to just using `gadgetlib1` and `gadgetlib2`. For starters, it already has many circuits available that are ready to be used (and can be directly compared with previous work). The other benefits come from the fact that it has some options that allow to work at a higher level than `gadgetlib1`. For instance, it is not needed to have two separate functions for the generation of the constraints and witness for primitive operations. Also, there are other optimization techniques applied that also allow canceling unneeded constraints. Therefore, integrating it with `lattice-zksnark` seemed like a good way to get more workloads.

The integration was a little bit tricky because it relied on functions from `gadgetlib2` and as we have said before, it is not useful for us because it uses elliptic curves instead of fields of integers modulo a prime. Thus, it was necessary to find its analogous functions in `gadgetlib1` (which uses templates) and modify the code accordingly. It was also necessary to change some other functions that were implemented having elliptic curves in mind so that it made sense with our fields. Lastly, we also needed to prepare the parameters and inputs of the function running the proof so that everything was correct and functional.

After integrating it, there were some of the circuits that the library came with that could correctly be used with `lattice-zksnark` but it was not possible to run most of them with it because the element sizes were too large for it (elliptic curves can use more bits than the fields used in `lattice-zksnark`). Therefore, to fix it, it would be necessary to modify either `lattice-zksnark` so that it accepts bigger elements or the circuit generation so that it splits the constraints that are too big into smaller ones. However, as for reasons that we will explain later we will not probably continue with this protocol, it has not been modified.

Just for comparison though, one of the gadgets that did work was SHA-256. By using `jsnark`, its size was greatly reduced from the one in the previous section as the circuit is more optimized. The results of running it are in Table 5.3.

Function	#constraints	Polynomial H	MVM	Total proof time
SHA-256	32768	0.0429	0.2013	0.3036

Table 5.3: Computing time for SHA-256 using `jsnark` expressed in seconds. The 3rd and 4th column refer to the time spent computing H and the matrix-vector multiplication respectively. The results are obtained using 8 cores of an i5-8250U at 1.60GHz.

Nonetheless, even though there are some already created circuits that cannot be used, `jsnark` is still a useful tool to create other kinds of circuits.

5.3. Estimated hardware results

Lastly, to decide if it is worth it to design hardware for this task, we will do a back-of-the-envelope calculation of the computation time and area needed for it. To do so, we will estimate the number of multiplies needed for the computation of a function, in this case SHA-256 (the one used in the first code). We will focus on just the two parts we saw before: the H polynomial computation and the matrix-vector multiplication.

General estimation

For the computation of H, as we can see in Figure 5.1, we have the computation of 7 NTTs (there are also other operations but we will consider them to be negligible).

As every NTT costs approximately $\frac{N \log N}{2}$ multiplications, in the case of SHA-256 this will mean that we will have roughly 13.7 million multiplies for this first part.

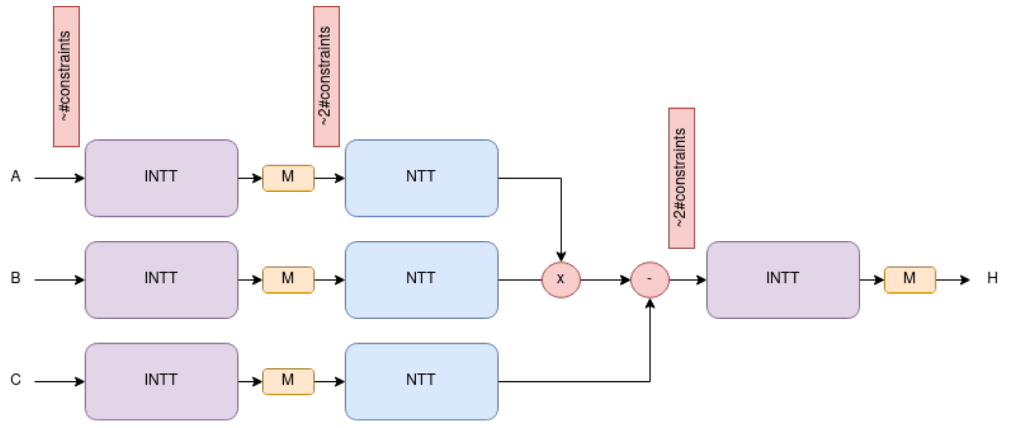


Figure 5.1: Diagram for the computation of the polynomial H during the proof generation algorithm with the size of the inputs to the NTTs.

For the matrix vector multiplication (see Figure 5.2), the size of the matrix is $(n + q) \times (2 \cdot \#constraints)$, where n and q are parameters that depend on the size of the plaintext modulo and the number of constraints among others. The n is the degree of the LWE problem and it is closely related to the security we want to achieve. The q corresponds to the number of queries needed to achieve soundness in the protocol. In this case, they will need to be about $n + q = 4,300$. Therefore, the number of multiplications needed will be about 598.8 million multiplies.

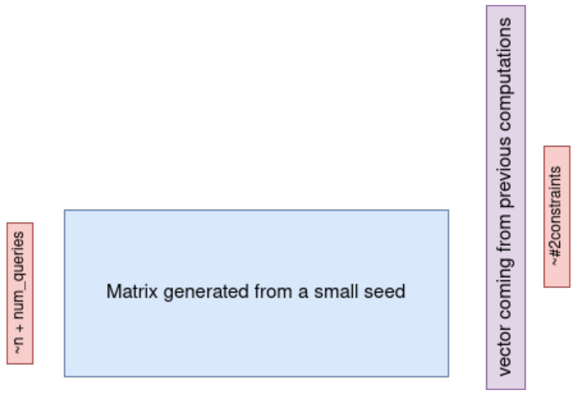

~n + num_queries
Matrix generated from a small seed
vector coming from previous computations
~#2constraints

Figure 5.2: Diagram for the matrix-vector multiplication during the proof generation algorithm with the sizes of the elements.

In total, this will mean that we will have about 612.5 million multiplies in the code. Then, we will assume that we have 100,000 multipliers available that are always computing and that we can work at 1GHz. Therefore, the compute time for this will be of:

$$612.5 \text{ million multiplies} \cdot \frac{1 \text{ cycle}}{10^5 \text{ multiplies}} \cdot \frac{1s}{10^9 \text{ cycle}} = 6.125\mu s$$

Thus, we could ideally accelerate the two critical parts of SHA-256 which originally cost $0.9654+0.2958 = 1.2612s$ to just $6.125\mu s$. Nonetheless, there were also $1.5031-1.2612 = 0.2419s$ that were not from any of the two parts we have talked about and therefore the compute time would be a bit higher, but it still is a significant improvement from what we had at the beginning.

To estimate the area needed, we will use the fact that a multiplier at 12 nanometers occupies $1,250\mu\text{m}^2$ and we will use the approximation of $1\text{MB} \approx 1\text{mm}^2$ for SRAM. Then, we would need $100,000$ multipliers $\cdot\ 1250\mu\text{m}^2 = 125\text{mm}^2$ for the multipliers. We would need to load the inputs of the circuit as well as its description, the matrix seed as well as some other information regarding the number of queries the user makes. To load all of this, and have plenty of space, $50\text{MB} \approx 50\text{mm}^2$ will be enough. Thus, in total it would require at least 175mm^2 for the parts we have described.

5.3.1. Memory bandwidth

Even though the results might seem quite good, there is a problem concerning the memory bandwidth that we found at the end of the project. In the matrix-vector multiplication there are some specific rows of the matrix that need to be provided by the user, which for SHA-256 are of about 140MB[1] This means that we would need at least:

$$\frac{140\text{MB}}{6.125\mu\text{s}} = 2.3 \times 10^{13}\text{B/s}$$

But we are only able to provide a bandwidth of $\approx 10^{12}B/s$. Thus, this would be a bottleneck in the program that would not allow us to make it compute-bound and that would need to be fixed in order to provide a good acceleration of the function.

5.4. Final considerations

Even though it does seem possible to achieve and improve the performance of this protocol by designing specific hardware for it, there are other factors that should be taken into account. The most important one is that by using a designated verifier setting we are losing many use cases where it would be useful to just be able to publish a proof that anyone could verify (in blockchain systems for example). Furthermore, these are precisely the cases where the proof needs to generated fast because there might be users waiting on it so that they can plan their own actions.

[1]log(ciphertext_modulus)$*$ num_queries $*$ 2#constraints = (114 * 35 * 2 * 140k) bits = 139.65MB.

It is for that reason that we consider that it would be better to change the direction of this project in the near future so that we are able to work in a public verifier setting. This does not mean that the ideas proposed were not useful. As there are similarities between the zk-SNARKs protocols, it will probably still be possible to accelerate the proof generation for them using an analogous procedure.

Chapter 6

Conclusions

Zero-knowledge proofs (ZKP) and verifiable computation (VC) are considered to be very powerful tools in the field of privacy. They allow someone to prove that the result of a computation is correct without needing to disclose some input values that might be confidential. Among its multiple applications we can find cryptocurrencies, nuclear disarmament or more curious ones such as fighting disinformation. The latter is implemented by providing a proof that an image that might have been modified by a newspaper comes from an original one that was taken in the place an time specified by the article.

Zero-Knowledge Succint Non-Interactive Argument of Knowledge (zk-SNARKs) is a group of different variants of a protocol that aims provide a practical implementation for ZKP and VC. One of the main obstacles for using zk-SNARKs in real-world applications is its slowness when generating a proof, which makes optimizing their speed and efficiency a focal point of research efforts. Another concern that demands attention in ongoing research is ensuring the post-quantum security of zk-SNARKs, an issue that currently remains a vulnerability in numerous implementations.

Therfore, this thesis explores the protocol explained in *Shorter and Faster Post-Quantum Designated-Verifier zkSNARKs from Lattices* ([18]) that, as the name says, implements zk-SNARKS that use lattice-based cryptography, which is theoretically quantum safe. Furthermore, it operates within the designated verifier setting, characterized by the fact that the proof can only convince the designated party rather than being able to prove the statement to anyone. This might be a drawback because it cannot be used for common applications (such as blockchains) but there are other use cases for it (like electronic voting or digital signatures).

As mentioned earlier, the primary challenge associated with these implementations lies in the slowness of proof generation. This is specially true with post-quantum zk-SNARKs, as there has not been as much research on them. This is why in this project we design hardware targeting the most time-consuming parts of the code:

firstly, a part that evaluates/interpolates polynomials by using Number Theoretic Transforms (NTTs) and another that computes a dense matrix-vector multiplication. In this work we describe the contents of these units and we provide an approximation for time and area in the Evaluation chapter.

Finally, we also needed to have some baselines to be able to compare the new changes. Firstly, we developed a program that enables us to input custom circuits using our own simplified language, making the process more straightforward. Secondly, we integrated the `jsnark` library, originally designed for a different type of zk-SNARKs, into our code. This integration allowed us to access another set of already defined workloads.

After seeing the results of the time estimation using the proposed hardware, it seems feasible to accelerate the proof generation for the chosen protocol. Nonetheless, it might not be necessary to do it for this specific protocol because the cases that require high speed are more likely to be found in procotols that use a public verifier settting. This does not mean that the observation that it is possible to accelerate it is not useful. As many other protocols share the same basic concepts, the ideas explained throughout the project are still helpful, they just need to be applied to a protocol that provides public verifiability.

Chapter 7

Future work

Regarding work to be done in the future, one line of work would be to continue developing the hardware described in Chapter 4 of this project. This could be done by fully designing an accelerator that allowed us to count the cycles it would take depending on the size of the input, the field it is using and the security. The more advanced version of this would be to design the hardware using a Hardware Description Language (HDL) such as Verilog or VHDL. Doing this would give a more precise approach to the resources needed to generate the proof, both time and area-wise.

One of the things that could also be worth studying is another setup that uses public verfiable computation (and is post-quantum safe) instead of designated verifier, as we have seen in this project. This could be interesting because there are many important applications (such as blockchains) that require that everyone is able to verify that a proof is correct. There has been some work on this in the paper *Lattice-Based SNARKs: Publicly Verifiable, Preprocessing, and Recursively Composable* published in 2022 ([33]). The main reason we did not work initially on that paper is that it had more mathematical assumptions than the one we used, which is not inherently negative, but can lead to future problems if one of them is proved to be false. Another (more practical) reason for it is that the paper we worked with had their code already implemented, which is very useful because it is not needed to spend time developing it. Nonetheless, working on publicly verifiable SNARKs can be a very valuable for many applications as it holds the potential to enhance privacy and security in a wide range of fields.

Finally, before I began studying zero-knowledge concepts within this research group, my initial focus was on Fully Homomorphic Encryption (FHE), which is another area of research explored by our group. In a nutshell, FHE is a type of encryption that allows the user to perform computations on encrypted data. The distinction between Fully Homomorphic Encryption and Verifiable Computation lies in their respective trust models. In FHE, you don't need to trust the server with your input data, as computations can be performed on encrypted data. On the other hand, in VC, the concern is not about the input data but rather about ensuring the correctness of the results provided by the server. Considering these differences, it becomes intriguing to explore the potential synergy between FHE and VC, as their combination could offer unique solutions that address both data privacy and result integrity simultaneously.

Bibliography

[1] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[2] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 465–482. Springer, 2010.

[3] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *ACM Symposium on Theory of Computing*, pages 291–304, 1985.

[4] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, Cambridge, United Kingdom, 2001.

[5] Zkproof community reference [online]. https://zkproof.org/. [Acccessed: 2023].

[6] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[7] Jean-Jacques Quisquater et al. How to explain zero-knowledge protocols to your children. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 628–631, New York, NY, 1990. Springer New York.

[8] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[9] Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, and Yupeng Zhang. MOOC, zero knowledge proofs [online]. https://zk-learning.org/, 2023. [Acccessed: 2023].

[10] S. Goldwasser and Y.T. Kalai. On the (in)security of the fiat-shamir paradigm. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 102–113, 2003.

[11] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 387–398, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[12] Alexander Glaser, Boaz Barak, and Robert J Goldston. A zero-knowledge protocol for nuclear warhead verification. *Nature*, 510(7506):497–502, 2014.

[13] Ben Fisch, Daniel Freund, and Moni Naor. Physical zero-knowledge proofs of physical properties. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34*, pages 313–336. Springer, 2014.

[14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[15] Kenneth A Bamberger, Ran Canetti, Shafi Goldwasser, Rebecca Wexler, and Evan J Zimmerman. Verification dilemmas in law and the promise of zero-knowledge proofs. *Berkeley Tech. LJ*, 37:1, 2022.

[16] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005. Proceedings 3*, pages 467–482. Springer, 2005.

[17] Trisha Datta and Dan Boneh. Using zk proofs to fight disinformation [online]. https://medium.com/@boneh/using-zk-proofs-to-fight-disinformation-17e7d57fe52f. [Acccessed: 2023].

[18] Yuval Ishai, Hang Su, and David J. Wu. Shorter and faster post-quantum designated-verifier zksnarks from lattices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 212–234, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Maksym Petkus. Why and how zk-snark works: Definitive explanation. *arXiv preprint arXiv:1906.07221*, 2019.

[20] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on*

the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32, pages 626–645. Springer, 2013.

[21] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[22] Wikipedia, the free encyclopedia. Cooley-tukey fft algorithm [online]. https://commons.wikimedia.org/wiki/File:DIT-FFT-butterfly.svg, 2013. [Acccessed: 2023].

[23] David H Bailey. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242, 1989.

[24] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.

[25] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[26] Ahmed Kosba et al. `jsnark`: a library for building circuits for preprocessing zk-snarks. [online]. https://github.com/akosba/jsnark/tree/master, 2022. [Acccessed: 2023].

[27] Hang Su and David Wu. Lattice-based zksnarks over `libsnark`, 2021. [Acccessed: 2023].

[28] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars Virza, Howard Wu, Michael Backes, Manuel Barbosa, Alexander Chernyakhovsky, Dario Fiore, Jens Groth, Joshua A. Kroll, Shigeo MITSUNARI, Aleksejs Popovs, Raphael Reischuk, and Tadanori TERUYA. `libsnark`: a c++ library for zksnark proofs. https://github.com/scipr-lab/libsnark#libsnark-a-c-library-for-zksnark-proofs, 2012-2020. [Acccessed: 2023].

[29] David Archer, Victor Arribas Abril, Steve Lu, Pieter Maene, Nele Mertens, Danilo Sijacic, and Nigel Smart. "Bristol Fashion" MPC Circuits [online]. https://homes.esat.kuleuven.be/~nsmart/MPC/. [Acccessed: 2023].

[30] Ethereum Improvement Proposals. Descriptions of SHA-256, SHA-384, and SHA-512 [online]. https://eips.ethereum.org/assets/eip-2680/sha256-384-512.pdf. [Acccessed: 2023].

[31] Morris J Dworkin, Elaine B Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, and James F Dray Jr. Advanced encryption standard (AES). 2001.

[32] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428, 2021.

[33] Martin R. Albrecht, Valerio Cini, Russell W. F. Lai, Giulio Malavolta, and Sri AravindaKrishnan Thyagarajan. Lattice-Based SNARKs: Publicly Verifiable, Preprocessing, and Recursively Composable. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 102–132, Cham, 2022. Springer Nature Switzerland.