Smart registration in Blockchain using zk-SNARKs

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Sílvia Margarit Jaile

In partial fulfillment
of the requirements for the master in
Advanced Telecommunication Technologies

Advisors: Jose Luis Muñoz Tapia
Héctor Masip Ardevol

Barcelona, January 2022

Contents

List of Figures 4

List of Tables 4

1 Introduction 7
- 1.1 Project development planning . . . 8

2 State of the art 10
- 2.1 Zero-Knowledge . . . 10
- 2.2 Cryptography Based on the Discrete Logarithm Problem . . . 13
  - 2.2.1 The Discrete Logarithm Problem . . . 13
  - 2.2.2 Elliptic Curve Cryptography . . . 15
  - 2.2.3 Elliptic Curves over Cyclic Groups . . . 16
  - 2.2.4 Diffie-Hellman over Elliptic Curves . . . 18
  - 2.2.5 Elliptic Curve Cryptography Applications . . . 20
- 2.3 zk-SNARK . . . 22
  - 2.3.1 zk-SANRK computation . . . 33
  - 2.3.2 zero-Knowledge Protocol . . . 36
  - 2.3.3 zk-SNARK Protocol . . . 38
  - 2.3.4 zk-SNARK Comparison . . . 40
- 2.4 Pinocchio protocol . . . 41
  - 2.4.1 Circuit design . . . 41
  - 2.4.2 Trusted Setup . . . 45
  - 2.4.3 Prover . . . 46
  - 2.4.4 Verifier . . . 46
- 2.5 The `circom` language . . . 51
  - 2.5.1 The `circomlib` library . . . 51
  - 2.5.2 Combination with `snarkJS` . . . 52

3 Methodology 53
- 3.1 Setup of our voting system . . . 59
- 3.2 Circuit . . . 60

4 Evaluation 63
- 4.1 Testing the voting system . . . 63

5 Budget 67

6 Conclusions and future development 68

References 69

7 Appendices 71
- 7.1 Code for the `merkletree2.js` file . . . 71
- 7.2 Code for the verification circuit (`circuits/mkt2.circom` file) . . . 72

7.3 Testing files 73
7.3.1 merkletree2_tester.circom file 73
7.3.2 mkt2cir.js file 73

List of Figures

1 Project's Gantt diagram . . . 9
2 "The cave of Ali Baba" example. . . . 10
3 Common time complexity comparison. . . . 13
4 Diffie-Hellman operation scheme. . . . 15
5 Different Shapes of Elliptic Curves. . . . 15
6 Adding Points on an Elliptic Curve. . . . 16
7 Diffie-Hellman operation scheme over Elliptic Curves. . . . 18
8 Mapping two encrypted inputs to a different output set of numbers. . . . 19
9 Graphical representation of the polynomial $f(x) = x^3 - 6x^2 + 11x - 6$. . . . 22
10 Graphical comparison of two polynomials of the same degree. . . . 23
11 Graphical representation of the computation. . . . 34
12 Graphical representation of the polynomial multiplication. . . . 35
13 NAND gate Truth table . . . 42
14 NAND gate . . . 42
15 Intermediate interpolation of the NAND. . . . 42
16 Circuit with two NAND gates. . . . 42
17 Decimal to binary circuit. . . . 43
18 Polynomial representation. . . . 45
19 Visual summary of the combination of circom and SNARKJS[15]. . . . 52
20 Block diagram of a verification voting system. . . . 58
21 Representation of the Merkle tree used in for the program. . . . 58
22 Scheme of the circuit system. . . . 60
23 Representation of the Merkle with the values of our example. . . . 64

List of Tables

1 Background study task. . . . 8
2 Process task. . . . 8
3 Documentation task. . . . 8
4 Special cases of the implementation of the Elliptic Curve. . . . 17
5 Comparison between the different systems of Zero Knowledge. . . . 41
6 Estimated budget for the project. . . . 67

Revision history and approval record

Revision	Date	Purpose
0	12/09/2021	Document creation
1	17/01/2022	Document revision
2	23/01/2022	Document delivery

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Sílvia Margarit Jaile	silvia.margarit@estudiantat.upc.edu
Jose Luis Muñoz Tapia	jose.luis.munoz@upc.edu
Héctor Masip Ardevol	hector.masip@upc.edu

Written by:		Reviewed and approved by:	
Date	13/01/2022	Date	23/01/2022
Name	Sílvia Margarit Jaile	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Abstract

Ensuring privacy in public blockchains is a challenge and a necessity for the success of distributed applications also known as Web3 applications. The cryptographic protocols that allow the implementation of privacy are the Zero Knowledge Proofs (ZKPs) and this is the scope of this work. In particular, this Thesis analyzes in detail the Pinocchio/-Groth16 protocol, which is a type of zero knowledge Succinct Argument of Knowledge (zk-SNARK). Then, we use an implementation of this protocol that uses a new programming language called `circom` which, together with JavaScript libraries, allows the user to validate circuit-based computations while keeping private some of the inputs. Different circuits are described and tested to prove that the privacy requirements of a distributed application can be met using the Pinocchio/Groth16 protocol.

Keywords: Zero Knowledge Proof, zk-SNARK, Pinocchio, Groth16, circom.

1 Introduction

Guaranteeing users' anonymity has been a big challenge for years. Several methods and protocols have been applied to achieve it, such as Zero Knowledge Proof protocol, which will be described in the state of the art section of this Thesis. The aim is to let a certain user (called prover) prove that he/she knows some information without having to reveal unnecessary data, and then another person or entity (called verifier) will check through a proving system that the prover's statement is correct.

For this work, the prover will be an external user of the system, the verifier will be the entity that owns the proving system and the proving system will be a combination of one or more circuits programmed with the circom language which will guarantee that a certain statement is accomplished. Only if this statement is true means that the prover knows the information that he/she is stating to know.

In this Thesis it will be analyzed the Pinocchio protocol, also named as zk-SNARK, and then a voting system will be presented in order to prove that a system based in zero-knowledge can detect if a person that wants to vote belongs to the voting census and if he/she has already voted or not.

Therefore, the objective of this Thesis is to demonstrate that using circom circuits and following the Pinocchio protocol is possible to prove a statement without having information about the prover (the person that wants to vote in this case). This way, the prover's privacy will be guaranteed.

This Master Thesis is organised in six sections. The first one, the introduction, presents the problem to be solved, the project structure, the scopes and the requirements.

In the second section, the background of the project is described, taking into account the previous work done in this field and the previous steps that are needed for this thesis, such as the protocols Zero Knowledge Proof, zk-SNARK, Pinocchio and the mathematical basis needed for them. Moreover, the programming language that implements the Pinocchio protocol will also be presented.

In the third section, after a brew practical introduction explaining how the circom language works, the system that this work wants to evaluate is explained in detail, as well as its setup and the functions and circuits that will be used.

In the forth section, the results are presented, and the data set used for this thesis is described, too.

Then, in the fifth section it is presented the estimated budget that would be needed for this project in case it was intended for business purposes.

In the last section, a complete analysis of the topics introduced in the Thesis is carried out, and the conclusions and future work are also explained. A summary of the most important ideas seen in this thesis will be made, as well as some branches of study that have been opened during the project.

1.1 Project development planning

In order to develop properly this Thesis, some tasks were defined. These tasks have been changing while the project was evolving, but the definitive ones, which have finally been achieved, are those described in tables 1, 2 and 3.

Background Study	
Description Before starting to work in the project, it is important to analyze and make conclusions from the previous work done around this topic. The protocols Zero Knowledge Proof and zk-SNARK were analyzed, together with the mathematical expressions used on them. Following, the Pinocchio protocol was analyzed in detail, as well as its implementation in circom.	Start event: 13/09/2021 End event: 29/10/2021

Table 1: Background study task.

Process	
Description During the execution of this task, a performance using circom circuits will be done and tested in order to achieve the aim of this Thesis.	Start event: 01/11/2021 End event: 10/01/2022

Table 2: Process task.

Documentation	
Description As it is needed to report all the work that has been made, the information about this project will be reported in the final delivery.	Start event: 20/09/2021 End event: 23/01/2022

Table 3: Documentation task.

Finally, the timeline followed for doing the tasks described above is the one showed in figure 1.

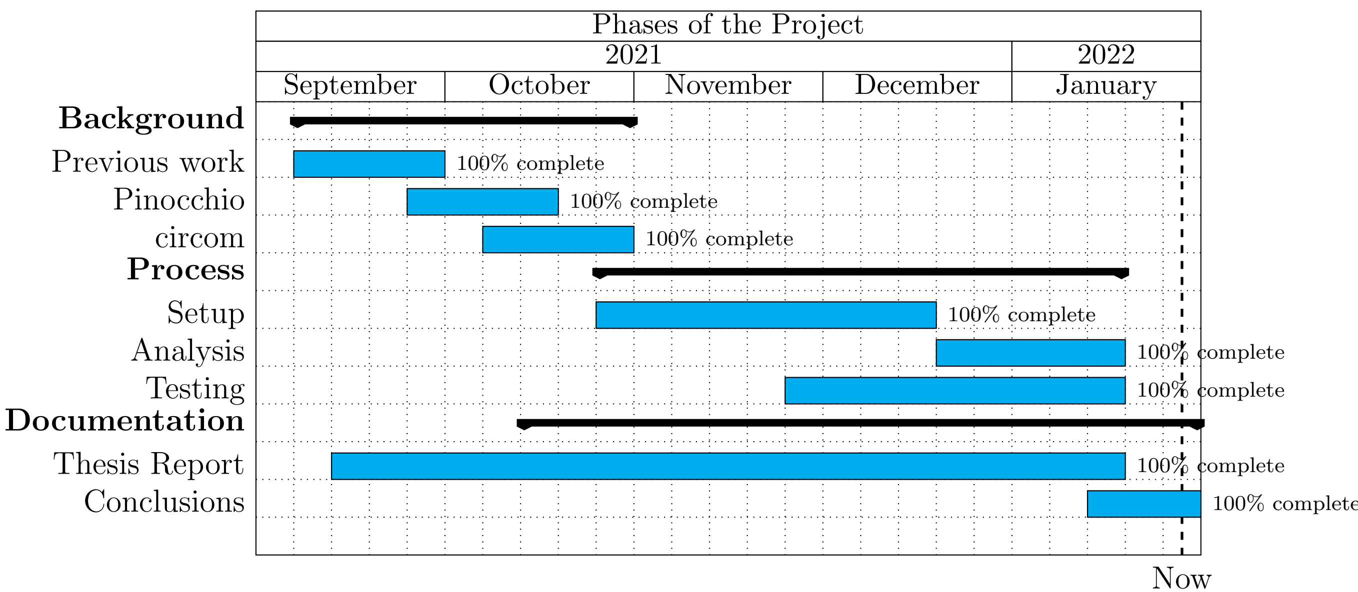


Figure 1: Gantt diagram of the project.

2 State of the art

2.1 Zero-Knowledge

The Zero Knowledge Proof (ZKP) protocol will be the basis of this work. It is an advanced cryptography protocol used to create highly secure and anonymous distributed systems. ZKP protocols allow information to be shared and verified without revealing unnecessary data, thus maintaining a very high level of security. It was first proposed in the 1980s by MIT researchers Shafi Goldwasser, Silvio Micali and Charles Rackoff [3], where they were working on problems related to **interactive proof systems** between a Prover and a Verifier.

A **zero-knowledge proof** is a protocol between two parties, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, in which $\mathcal{P}$ makes some statement, and tries to convince $\mathcal{V}$ that the statement is true, whilst revealing nothing more than the fact that the statement is true.

Therefore, a ZKP serves as an authentication method where it is not necessary to reveal secrets to achieve the goal of proving that you have a certain secret information. This is important because the fact of not sharing secrets means that they cannot be stolen.

Accordingly, the aim of this protocol is to prove that one or more secrets are known to someone, without actually revealing these secrets. The "zero knowledge" term originates from the fact that no information is disclosed.

The basic idea behind this protocol is to unequivocally prove that the Prover knows the secret without revealing it. The better for it, is that to verify this information you do not need to consult a third party, you only have to take the information from the Prover and apply the protocol. In this way, the Verifier can know if the information is true. Finally, the secret information can be statistically or deterministically verifiable.

Another way to explain this protocol is the one explained in [4], where the authors managed to explain in a completely simple way the operation of this protocol. To achieve it, they devised a simple example called "**The cave of Ali Baba**". This example explains that a person called Mick Ali wants to show to a reporter that he knows the magic word that opens the magic door of Ali Baba's cave, but he does not want to reveal the secret to the reporter. For this, Mick and the reporter go to the cave. The paths that they will take once they are inside the cave is illustrated in image 2.

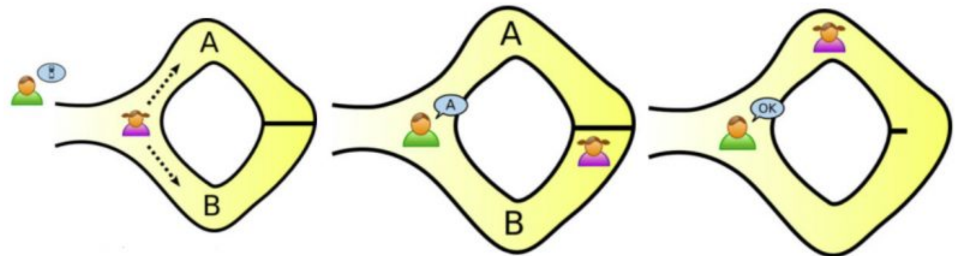


Figure 2: "The cave of Ali Baba" example.

Mick (the girl in image 2) pledges to go either way A or B. Both paths are only communicated through the magic door. At this point, the reporter waits for Mick to go to A or B while he waits for him at the entrance of the cave.

At some point, the reporter asks Mick to go out via A or B. He decides one of the two options randomly. If Mick did not know the magic words, he would have the 50% of probability of going out through the path chosen randomly by the reporter. The repetition of this scheme on several occasions serves to determine that Mick really knows the magic words to open the door, but he has never revealed the secret words to the reporter.

A protocol is a Zero Knowledge Proof if it accomplishes these three requisites:

1. **Completeness.** If the Prover is honest, then she will eventually convince a honest Verifier.

   Both parties involved (the prover and the verifier) are assumed to be honest and will follow the protocol. This means that if a prover gives a statement, the verifier will be effectively convinced by it.

2. **Soundness.** The Prover can only convince the Verifier if the statement is true.

   The protocol must assume that honesty is little or no. So in order to prove that the prover does indeed have a secret, the verifier must be convinced. All this while minimizing the chances of successfully tricking the verifier.

   A dishonest prover can not generate a valid proof.

   This property is satisfied except of some small probability, and it is usually the hardest property to prove, but a way to do it is prove the existence of an Extractor algorithm, which will be explained later.

3. **Zero knowledge.** The Verifier learns no information beyond the fact that the statement is true.

   The aim of this property is to protect the Prover. This property can be verified using the following formal definition to build the proof: *Any verifier $\mathcal{V}$ might as well have generated (or simulated) the interaction on his own.* This means that, to prove ZK, we must prove that there exists a **Simulation algorithm** $\mathcal{V}$ for the interaction.

Compliance of these three requirements is essential for a protocol to be able to comply with “zero knowledge”. In case you cannot do it, the protocol cannot be called that way as it does not guarantee leakage of knowledge.

In zero-knowledge we will want to prove, at a high level, two different kind of statements:

1. Statements about “facts”. Each of these is a statement about some intrinsic property of the universe.
2. Statements about my personal knowledge. These go beyond merely proving that a fact is true, and actually rely on what the Prover knows.

Apart from this, the protocol must guarantee a safe source of randomness. The justification is given because the generation of random numbers is another necessary condition for its correct operation.

The main use cases of this protocol are secure communication systems, such as the military or spy organizations, authentication systems, secure voting systems or for cryptocurrencies such as zcash and Monero. ZKP are also advantageous in a myriad of application such as:

1. Proving statement on private data
    - Person A has more than X in his/her bank account.
    - In the last year, a bank did not transact with an entity Y.
    - Matching DNA without revealing full DNA.
    - One has a credit score higher than Z.
2. Anonymous authorization
    - Proving that requester R has right to access web-site's restricted area without revealing its identity (e.g., login, password).
    - Prove that one is from the list of allowed countries/states without revealing from which one exactly.
    - Prove that one owns a monthly pass to a subway/metro without revealing card's id.
3. Anonymous payments
    - Payment with full detachment from any kind of identity [8].
    - Paying taxes without revealing one's earnings.
4. Outsourcing computation.
    - Outsource an expensive computation and validate that the result is correct without redoing the execution; it opens up a category of trustless computing.
    - Changing a blockchain model from everyone computes the same to one party computes and everyone verifies.

A system that uses ZKP provides **security**, **privacy** and **anonymity**, as it does not require the revelation of any secret. However, this protocol has also some disadvantages: it is limited to using numerical values, it is also computationally expensive compared to other cryptographic primitives, and it does not solve the problem of secure transmission of information because it is vulnerable to a third party that can intercept the transmission, modifying or destroying the message. Finally, the implementation and algorithmic review of these systems is complex, in addition to being an area dominated by very few people around the world. This has as its main problem a low capacity to improve the system and debug it.

2.2 Cryptography Based on the Discrete Logarithm Problem

In this section we will describe some mathematical elements that will be used to define the Pinocchio (or zk-SNARK) protocol.

2.2.1 The Discrete Logarithm Problem

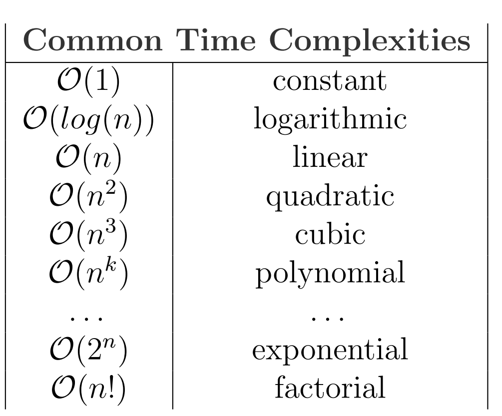
Given {$\mathbb{G}$, p, g, y}, the discrete logarithm problem asks to find an $x \in \mathbb{Z}_p^*$ such that $y = g^x$. If we assume that the discrete logarithm problem is hard in the group $\mathbb{G}$, then it is hard to obtain x from y.

The elements {$\mathbb{G}$, p, g} are public, where $\mathbb{G}$ is a cyclic group of prime order p and a generator g:

- The group $\mathbb{G} = (\mathbb{Z}_3, +)$, with prime $p = 3$ and generator $g = 2$.
- The group $\mathbb{G} = (\mathbb{Z}_7^*, \times)$, with prime $p = 7$ and generator $g = 3$.

There is a secret random element $x \in \mathbb{Z}_p = \{0, 1, .., p-1\}$ and a public element $y = gx$.

The complexity of the Discrete Logarithm Problem depends on the $\mathbb{G}$ group. In $\mathbb{Z}_p = \{0, 1, .., p-1\}$, the discrete algorithm is NOT hard, since given $y = xg(modp)$, the extended Euclidean algorithm can be used to find x in $\mathcal{O}(log(p))$ time. In picture 3 we can see the common time complexity comparison.



Common Time Complexities	
$\mathcal{O}(1)$	constant
$\mathcal{O}(log(n))$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^3)$	cubic
$\mathcal{O}(n^k)$	polynomial
...	...
$\mathcal{O}(2^n)$	exponential
$\mathcal{O}(n!)$	factorial

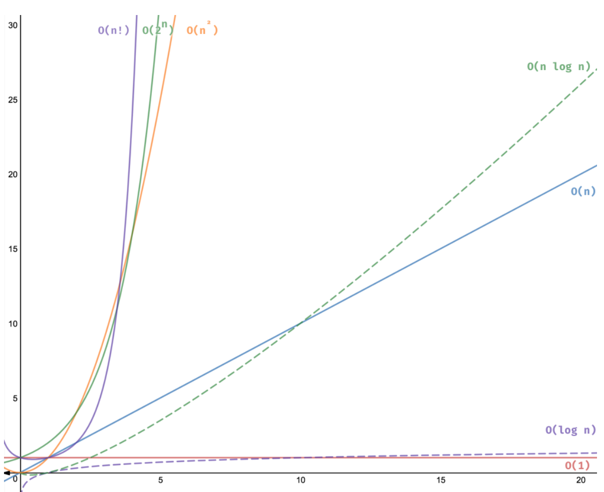


Figure 3: Common time complexity comparison.

For solving the Discrete Logarithm Problem in $\mathbb{Z}_p$, we have the following conditions:

- Let r and s be two integers such that $r < s$.
- The extended Euclidean algorithm is used to find $d = gcd(r, s)$ and two integers a, b such that: $ar + bs = d$, in $\mathcal{O}(log(s))$ time.
- Given the generator g and $y = xg(modp)$, we can use the previous algorithm to find $gcd(g, p) = 1$ and two integers a, b such that: $ag + bp = 1$.

- Multiplying this equation by y we obtain: $(ay)g + (by)p = y \longrightarrow (ay)g = y(modp)$.
- Then $x = ay(modp)$ is the solution of the discrete logarithm problem in $\mathbb{Z}_p$.

However, for the case of the $\mathbb{Z}_p^*$ group, we have the following conditions:

- $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ is a cyclic group of $p-1$ elements with multiplication modulo p: $gx = g \cdot g \cdot \ldots \cdot g(modp)$.
- The order of any subgroup of $\mathbb{Z}_p^*$ divides $p-1$ (Lagrange's Theorem).
- Let q be a prime number dividing $p-1$.
- Let $g \in \mathbb{Z}_p^*$ be a generator of a subgroup $\mathbb{G} = \langle g \rangle \subseteq \mathbb{Z}_p^*$ of order q.
- $\mathbb{G}$ is a cyclic subgroup with multiplication modulo p.
- For example, let $p = 23$, $q = 11$ and $g = 2$:

  $g^0 = 1, g^1 = 2, g^2 = 4, g^3 = 8, g^4 = 16, g^5 = 32 \equiv 9,$

  $g^6 = 18, g^7 = 13, g^8 = 3, g^9 = 6, g^{10} = 12, g^{11} = 1.$

  Hence, $\mathbb{G} = \{1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18\} \subseteq \mathbb{Z}_{23}^*$.
- The discrete logarithm problem is **hard** in this kind of groups.

The reason why the discrete algorithm is hard in the $\mathbb{Z}_p^*$ group is because it is not known any algorithm that solves the problem in $\mathcal{O}(n)$ time, that is, in **polynomial** time. The best known algorithm that solves the discrete logarithm problem in groups $\mathbb{G} \subseteq \mathbb{Z}_p^*$ is the **General Number Field Sieve (GNFS)**. The complexity of this algorithm is of the form of equation 1, which is subexponential but not polynomial.

$$e^{\mathcal{O}(1) \cdot log(q)^{\frac{1}{3}} \cdot (log(log(q)))^{\frac{2}{3}}} \tag{1}$$

For $p \geq 2^{3072}$, the algorithm takes at least 2^{128} steps to output the solution.

There are several cryptography systems or protocols based on the Discrete Logarithm Problem that try to solve it, such as Diffie-Hellman, which is a Key Exchange Protocol, ElGamal and Cramer-Shoup (DDH), that are Public Key Cryptosystems and Schnorr, DSA and EdDSA which are Digital Signature Schemes.

Moreover, other problems were generated from the Discrete Logarithm one. It has to be taken into account that for the computation of Diffie-Hellman (CDH) and having the values of image 4, given $\{\mathbb{G}, g, p, g^a, g^b\}$ we find g^{ab}, and for the decisional Diffie-Hellman Problem (DDH), given $\{\mathbb{G}, g, p, g^a, g^b\}$ we have to decide if $w = g^{ab}$.

This ends to a complexity reduction, because if we can solve the Discrete Logarithm Problem (DL) then we will be able to solve the CDH and the DDH. A way to make the method more robust is using **Elliptic Curves**.

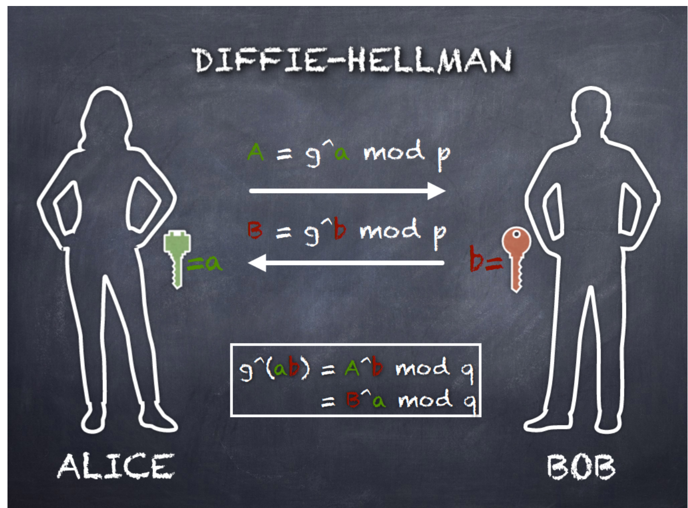


Figure 4: Diffie-Hellman operation scheme.

2.2.2 Elliptic Curve Cryptography

An elliptic curve is typically defined as the set of solutions to an equation of the form of equation 2, in some field $\mathbb{F}$, where A and B are constants such that $4A^3 + 27B^2 \neq 0$.

$$y^2 = x^3 + Ax + B \tag{2}$$

This is referred to as the **Weierstrass form** for an elliptic curve. Some examples of elliptical curves over $\mathbb{R}$ are shown in image 5.

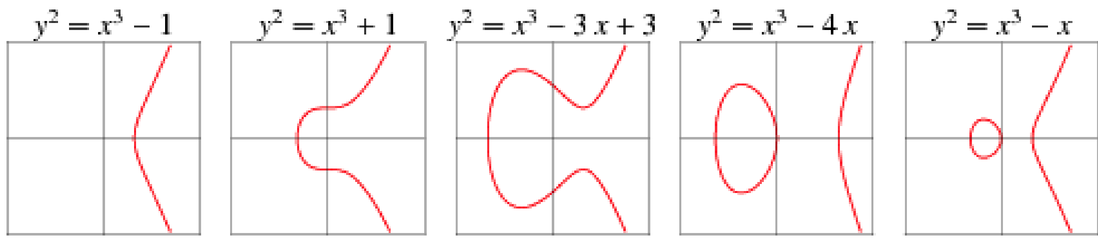


Figure 5: Different Shapes of Elliptic Curves.

Methodology of the Elliptic Curve

It first adds two points $P = (x_1, y_1)$, $Q = (x_2, y_2)$, located on an elliptic curve E given by the equation $y^2 = x^3 + Ax + B$. Then, it is drawn the line $L(x)$ through P and Q. This line L intersects E in a third point R. Reflecting R across the x-axis (i.e. changing the sign of the y-coordinate) we obtain another point R'. A graphical representation of this implementation is shown in image 6. Finally, we can also define equation 3.

$$P + Q = R' \tag{3}$$

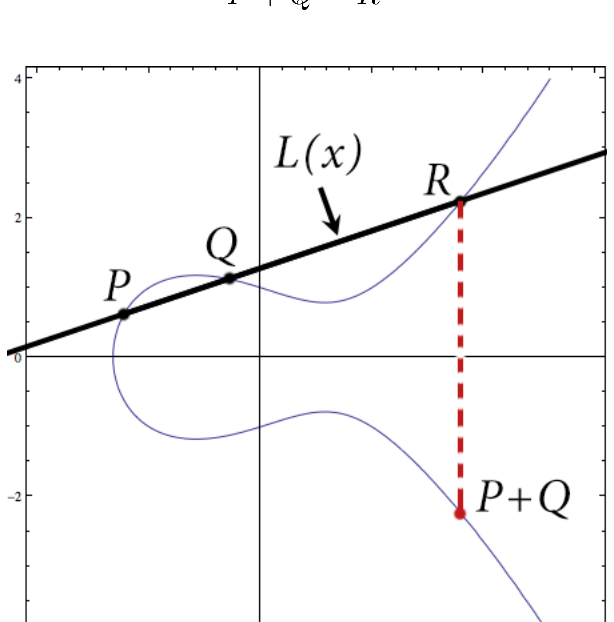


Figure 6: Adding Points on an Elliptic Curve.

There are some special cases that we can find when we add the points on an elliptic curve. The most common cases are exposed in table 4.

The addition of points on an elliptic curve E satisfies the following properties:

1. Closure: $P_1 + P_2 \in E$ for all $P_1, P_2 \in E$.
2. Existence of identity: $P + \mathcal{O} = P$ for all points P on E.
3. Existence of inverses: Given P on E, there exists P' on E such that $P + P' = \mathcal{O}$. This point P' is usually denoted as $-P$.
4. Associativity: $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ for all P_1, P_2, P_3 on E.
5. Commutativity: $P_1 + P_2 = P_2 + P_1$ for all P_1, P_2 on E.

In other words, the points on E form an **additive abelian group** with $\mathcal{O}$ as the identity element.

2.2.3 Elliptic Curves over Cyclic Groups

In this work we are interested in elliptic curves defined over finite fields $\mathbb{Z}_p$ (with $p > 3$): $E(\mathbb{Z}_p) = \{(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p | y^2 = x^3 + Ax + B\} \cup \mathcal{O}$.

We can estimate the number of points that these elliptic curves have with the Hasse-Weil Theorem [5], which provides an estimate of the number of points on an elliptic curve over a finite field, bounding the value both above and below.

	Case $P \neq Q$ and $x_1 = x_2$	Case $P = Q$	Case $P = \mathcal{O}$ or $Q = \mathcal{O}$
Conditions	If $x_1 = x_2$, but $y_1 \neq y_2$ the line trough P and Q is vertical, which therefore intersect E in $\mathcal{O}$. This is why we put $\mathcal{O}$ at the top and the bottom of the y-axis. In this case, $P + Q = \mathcal{O}$.	When two points on a curve are very close to each other, the line through them approximates a tangent line. Therefore, when the two points coincide, we take the line L through them to be the tangent line.	If $Q = \mathcal{O}$, the line through P and $\mathcal{O}$ is a vertical line that intersects E in the reflection of P across the x-axis. If we reflect the reflection, we are back at P ($P + \mathcal{O} = P$ for all points P on E). We also extend this to include the case $\mathcal{O} + \mathcal{O} = \mathcal{O}$.
Representation			

Table 4: Special cases of the implementation of the Elliptic Curve.

If N is the number of points on the elliptic curve E over a finite field with q elements, then Hasse's result states Eq. 4. The reason is that N differs from $q + 1$, the number of points of the projective line over the same field, by an 'error term' that is the sum of two complex numbers, each of absolute value $\sqrt{q}$.

$$|N - (q + 1)| \leq 2\sqrt{q} \tag{4}$$

A generalization of the Hasse bound to higher genus algebraic curves is the Hasse–Weil bound [6]. This provides a bound on the number of points on a curve over a finite field, defined in Eq. 5.

$$|E(\mathbb{Z}_p)| \in [p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}] \tag{5}$$

Now, Let P be a point on an elliptic curve E of prime order q such that $\underbrace{P + P + \ldots + P}_{q} = \mathcal{O}$, the subgroup $G = \langle P \rangle$ is a **cyclic group** of primer order q.

With the Discrete Logarithm, given $\mathbb{G}$ and a point Q, we will want to find an integer m such that $Q = m \cdot P$.

With $p \geq 2^{256}$ we have a complexity of 2^{128}, and in a classic scenario we have $p \geq 2^{3072}$.

2.2.4 Diffie-Hellman over Elliptic Curves

Using Diffie-Hellman over Elliptic Curves changes the scheme shown in image 4, now we have the one of image 7, where we can define a Computational Diffie-Hellman (CDH) because given $\{\mathbb{G}, G, aG, bG\}$ we have to find abG and a Decisional Diffie-Hellman Problem (DDH), because given $\{\mathbb{G}, G, aG, bG, P\}$ we have to decide if $P = abG$.

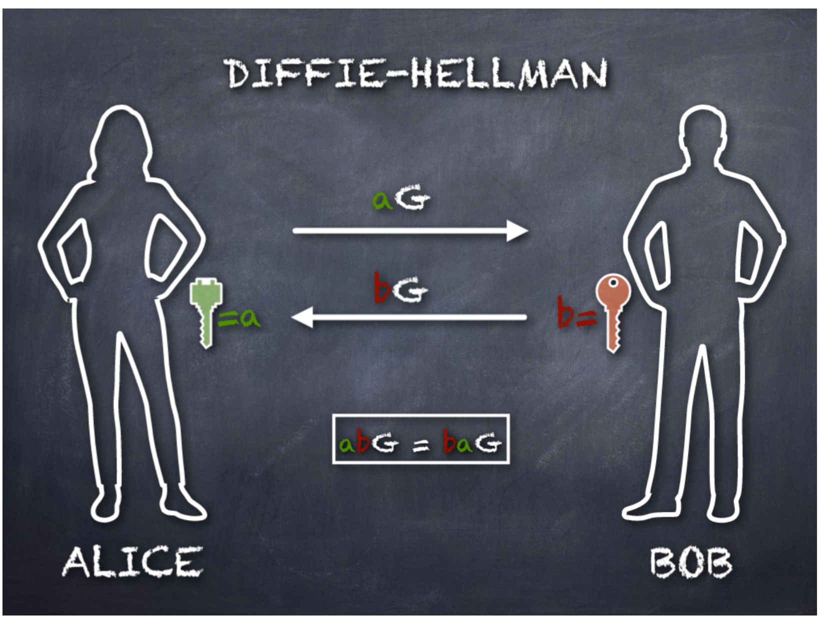


Figure 7: Diffie-Hellman operation scheme over Elliptic Curves.

We define $\mathbb{G}_1 = \langle G_1 \rangle$ and $\mathbb{G}_2 = \langle G_2 \rangle$ that are two cyclic groups of prime order p and $\mathbb{G}_T$ another cyclic group of prime order q.

We have a **symmetric pairing** if we have a map $e \colon \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ that satisfies the following properties:

1. **Computability**: There exists an efficient algorithm to compute $e(\cdot, \cdot)$.
2. **Non-Degeneracy:** $\mathbb{G}_T = \langle e(G_1, G_1) \rangle$.
3. **Bilinearity:** For all $P, Q, R \in \mathbb{G}_1$, it has to be satisfied that:

$$e(P + R, Q) = e(P, Q) \cdot e(R, Q),$$
$$e(P, Q + R) = e(P, Q) \cdot e(P, R).$$

   In particular, this implies that $e(aP, bQ) = e(P, Q)^{ab}$, for any $a, b \in \mathbb{Z}_p$.

On the other hand, we would have an **asymmetric pairing** if a map $e \colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is satisfying the following properties:

1. **Computability**: There exists an efficient algorithm to compute $e(\cdot,\cdot)$.
2. **Non-Degeneracy:** $\mathbb{G}_T = \langle e(G_1, G_2)\rangle$.
3. **Bilinearity:** For all $P, R \in \mathbb{G}_1$, and $Q, S \in \mathbb{G}_2$, it has to be satisfied that:
$$e(P + R, Q) = e(P, Q) \cdot e(R, Q),$$
$$e(P, Q + S) = e(P, Q) \cdot e(P, S).$$
In particular, this implies that $e(aP, bQ) = e(P, Q)^{ab}$, for any $a, b \in \mathbb{Z}_p$. Given two encrypted inputs (e.g., aP, bQ) from one set of numbers, this mathematical construction allows to map them deterministically to their multiplied representation in a different output set of numbers, i.e., $e(P, Q)^{ab}$:

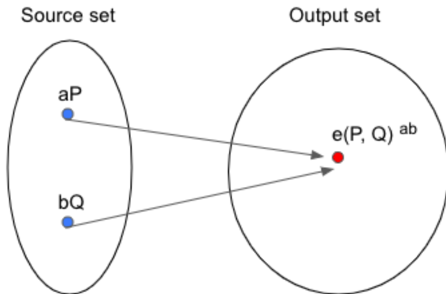


Figure 8: Mapping two encrypted inputs to a different output set of numbers.

The most known and widely-used pairings (Weil, Tate, Ate,...) are build over groups relative to elliptic curve groups.

The signature of a pairing is of the form:

$$e\colon E(\mathbb{Z}_p)[r] \times E(\mathbb{Z}_p)[r] \to \mu_r \tag{6}$$

where:

1. The integers p, r are primes and $p \approx r$.
2. $E(\mathbb{Z}_p)[r]$ is the group of points of order r in the elliptic curve $E(\mathbb{Z}_p)$.
3. $\mu_r \subset (\mathbb{Z}_p)^\alpha$ is a subgroup with p^α elements.

Using pairings, **the discrete logarithm problem over** $E(\mathbb{Z}_p)[r]$ **is much easier** and can be described with the Menezes-van Oorschot-Vanstone Reduction.

Menezes-van Oorschot-Vanstone Reduction

Let (P, Q) be an instance of the discrete logarithm problem over $E(\mathbb{Z}_p)[r] = \langle P\rangle$.

1. First, compute $Y = e(P, Q)$.
2. Then, compute the discrete logarithm of Y in the group $\mu_r = \langle e(P, P)\rangle$.
3. Since $\mu_r \subset (\mathbb{Z}_p)^\alpha$, if α small (for example $\alpha = 1$), we can use the GNFS (subexponential in p), to compute the discrete logarithm of Y.

4. The output of this algorithm is a $x \in \mathbb{Z}_p^*$ such that:

$$Y = e(P,P)^x \tag{7}$$

5. Using the bilinearity property of pairings we obtain Eq. 8

$$Y = e(P,P)^x = e(P,xP) = e(P,Q) = Y \tag{8}$$

   implying that x is also the discrete logarithm of Q with respect to P over $E(\mathbb{Z}_p)[r]$.

The conclusion is that using **larger keys** or choosing elliptic curves with **larger** α will make the discrete algorithm problem more robust.

Going back to the Decisional Diffie-Hellman Problem on Elliptic Curves, given $\{\mathbb{G}, G, p, aG, bG, W\}$, we will have to decide if $W = abG$ or W is a random element from $\mathbb{G}$. If $\mathbb{G}$ admits a symmetric pairing, first we will calculate $e(aG, bG) = e(G, bG)^a = e(G, abG)$, and, on the other hand, we will compute $e(G, W)$. If the result is the same, then $W = abG$. Otherwise, $W \neq abG$.

The Decisional Diffie-Hellman is **easy** in groups $\mathbb{G}$ that admit symmetric pairings. For example, ElGamal cryptosystem is **unsafe** in these kind of groups.

2.2.5 Elliptic Curve Cryptography Applications

There are different applications for elliptic curves, they are listed below.

1. Key Exchange Protocol with 2 Parties (Diffie-Hellman).
   (a) A chooses a secret key $x_A \in \mathbb{Z}_p$ and publish $Y_A = x_A G$.
   (b) B chooses a secret key $x_B \in \mathbb{Z}_p$ and publish $Y_B = x_B G$.
   (c) A and B can now compute the shared key:

$$K_{AB} = x_A x_B G = x_A Y_B = x_B Y_A \tag{9}$$

2. Key Exchange Protocol with 3 Parties (Desmedt-Burmester).
   (a) A chooses a secret key $x_A \in \mathbb{Z}_p$ and publish $Y_A = x_A G$.
   (b) B chooses a secret key $x_B \in \mathbb{Z}_p$ and publish $Y_B = x_B G$.
   (c) C chooses a secret key $x_C \in \mathbb{Z}_p$ and publish $Y_C = x_C G$.
   (d) A computes and publish $K_A = x_A(Y_B - Y_C) = (x_A x_B - x_A x_C)G$.
   (e) B computes and publish $K_B = x_B(Y_C - Y_A) = (x_B x_C - x_B x_A)G$.
   (f) C computes and publish $K_C = x_C(Y_A - Y_B) = (x_C x_A - x_C x_B)G$.

   The shared key is: $\boxed{K_{ABC} = (x_A x_B + x_B x_C + x_A x_C)G}$.

   For example, A can compute the shared key as: $(3x_A)Y_C + 2K_A + K_B = K_{ABC}$.

3. Key Exchange Protocol with 3 Parties & Pairings. If the group $\mathbb{G} = \langle G \rangle$ admits symmetric pairings $e\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, then the key exchange protocol can be realized in less steps than the previous case.

   (a) A chooses a secret key $x_A \in \mathbb{Z}_p$ and publish $Y_A = x_A G$.

   (b) B chooses a secret key $x_B \in \mathbb{Z}_p$ and publish $Y_B = x_B G$.

   (c) C chooses a secret key $x_C \in \mathbb{Z}_p$ and publish $Y_C = x_C G$.

   The shared key is: $K_{ABC} = e(G, G)^{x_A x_B x_C} = e(Y_A, Y_B)^{x_C} = e(Y_A, Y_C)^{x_B} = e(Y_B, Y_C)^{x_A}$.

2.3 zk-SNARK

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK) is the truly ingenious method of ZKP, with which we wanted to prove that something is true without revealing any other information. The zk-SNARK term itself was introduced in [9], built on [10] following the Pinocchio protocol [11] [12] and making it applicable for general computing. It is the zero knowledge system whose proof is faster to compute. This means that the prover allows a result faster than the other systems. Also, the verifier is much faster in this system.

This section intends to clarify how zk-SNARK works, basing the explanation on the work [7]. This article first tries to prove something without worrying about the zero-knowledge, non-interactivity, its form, and applicability. Imagine that we have an array of bits of length 10, and we want to prove to a verifier (e.g., program) that all those bits are set to 1.

$$b = [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]$$

The verifier can only check one element at a time. It can proceed by checking the elements in some arbitrary order and examining if it is truly equal to 1 and if so the confidence in that statement is $\frac{1}{10} = 10\%$ after the first check, or statement is invalidated altogether if the bit equals to 0. The verifier must proceed to the next round until he reaches sufficient confidence. In some cases, one may trust a prover and require only 50% confidence which means that 5 checks must be executed, in other cases where 95% confidence is needed all cells must be checked. It is clear that the downside of such a proving protocol is that one must do the number of checks proportionate to the number of elements, which is non-practical if we deal with arrays of millions of elements.

Now we will consider polynomials, which can be visualized as a curve on a graph shaped by a mathematical equation like we see in the graph 9.

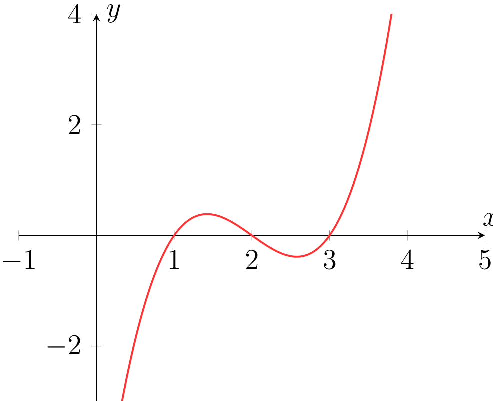


Figure 9: Graphical representation of the polynomial $f(x) = x^3 - 6x^2 + 11x - 6$.

Polynomials have an advantageous property that we will use: if we have two non-equal polynomials of degree at most d, they can intersect at no more than d points. For example, let us modify the original polynomial of graph 9 to be $x^3 - 6x^2 + 10x - 5$ and we will

visualize it in green in the same graph as the polynomial that we had before.

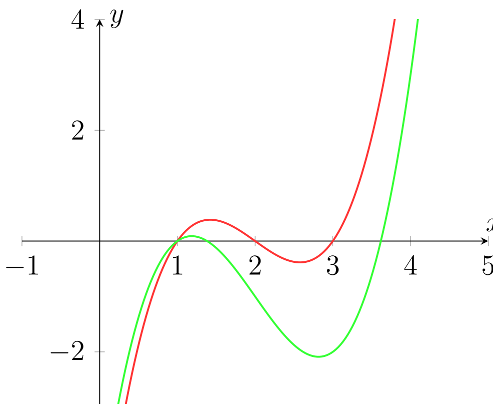


Figure 10: Graphical comparison of two polynomials of the same degree.

Such a tiny change produces a dramatically different result. In fact, it is **impossible** to find two non-equal polynomials, which share a consecutive chunk of a curve (excluding a single point chunk case). In our example, we have only one intersection between the two polynomials, it is when $x = 1$, which can also be mathematically demonstrated by equating the two polynomials and finding the d or less possible solutions to the equation.

Moreover, we can conclude that evaluating any polynomial at an arbitrary point x is akin to the representation of its unique identity. Let us evaluate our example polynomials at $x = 10$.

$$x^3 - 6x^2 + 11x - 6\Big|_{x=10} = 504$$
$$x^3 - 6x^2 + 10x - 5\Big|_{x=10} = 495$$

For these two reasons exposed, if a prover claims to know some polynomial (no matter how large its degree is) that the verifier also knows, they can follow a simple protocol listed below:

1. Verifier chooses a random value for x and evaluates the polynomial locally.
2. Verifier gives x to the prover and asks to evaluate the polynomial in question.
3. Prover evaluates his polynomial at x and gives the result to the verifier.
4. Verifier checks if the local result is equal to the prover's result, and if so then the statement is proven with a high confidence.

If we, for example, consider an integer range of x from 1 to 10^{77}, the number of points where evaluations are different is $10^{77} - d$. Henceforth the probability that x accidentally "hits" any of the d shared points is equal to $\frac{d}{10^{77}}$, which is considered negligible.

This protocol requires only one round and gives overwhelming confidence (almost 100% assuming d is sufficiently smaller than the upper bound of the range) in the statement compared to the inefficient bit check protocol that we exposed at the beginning.

Knowing a polynomial means knowing its degree and its coefficients. A polynomial can be expressed in the form:

$$c_n x^n + \cdots + c_1 x^1 + c_0 x^0$$

The Fundamental Theorem of Algebra states that any polynomial can be factored into linear polynomials, as long it is solvable. Consequently, we can represent any valid polynomial as a product of its factors:

$$(x - a_0)(x - a_1) \ldots (x - a_n) = 0$$

Let us say that the prover claims to know a degree 3 polynomial, such that $x = 1$ and $x = 2$ are two of all possible solutions. One of such valid polynomials is $x^3 - 3x^2 + 2x = 0$. For $x = 1$: $1 - 3 + 2 = 0$. For $x = 2$: $8 - 12 + 4 = 0$. Therefore, our example can be factored into the following polynomial:

$$x^3 - 3x^2 + 2x = (x - 0)(x - 1)(x - 2)$$

If the prover says that he knows a polynomial of degree 3 with the roots 1 and 2, this means that his polynomial has the form $(x - 1)(x - 2) \ldots$ and therefore that $(x - 1)$ and $(x - 2)$ are the cofactors of the polynomial in question. Hence, if the prover wants to prove that indeed his/her polynomial has those roots without disclosing the polynomial itself, he/she needs to prove that his/her polynomial $p(x)$ is the multiplication of those cofactors $t(x) = (x - 1)(x - 2)$, called **target polynomial**, and some arbitrary polynomial $h(x)$, which would be $(x - 0)$ in our example.

At this point we have the evidence of Eq. 10: there exists some polynomial $h(x)$ which makes $t(x)$ equal to $p(x)$, therefore $p(x)$ contains $t(x)$. Consequently, $p(x)$ has all roots of $t(x)$, the very thing to be proven.

$$p(x) = t(x) \cdot h(x) \tag{10}$$

If the prover cannot find such $h(x)$ that means that $p(x)$ does not have the necessary cofactors $t(x)$, in which case the polynomials division will have a remainder. In our example if we divide $p(x) = x^3 – 3x^2 + 2x$ by the $t(x) = (x–1)(x–2) = x^2–3x + 2$ the result will be $h(x) = x$ without remainder.

Using our **polynomial identity check protocol** we can compare polynomials $p(x)$ and $t(x) \cdot h(x)$ following these steps:

1. Verifier samples a random value r, calculates $t = t(r)$ (i.e., evaluates) and gives r to the prover.
2. Prover calculates $h(x) = \frac{p(x)}{t(x)}$ and evaluates $p(r)$ and $h(r)$; the resulting values p, h are provided to the verifier.
3. Verifier then checks that $p = t \cdot h$, if so those polynomials are equal, meaning that $p(x)$ has $t(x)$ as a cofactor.

If we apply this protocol to our example we will have something like the following:

1. Verifier samples a random value 23, calculates $t = t(23) = (23–1)(23–2) = 462$ and gives 23 to the prover.
2. Prover calculates $h(x) = \frac{p(x)}{t(x)} = x$, evaluates $p = p(23) = 10626$ and $h = h(23) = 23$ and provides p, h to the verifier.
3. Verifier then checks that $p = t \cdot h = 10626 = 462 \cdot 23$, which is true, and therefore the statement is proven

However, there are multiple issues with this construction:

- Prover may not know the claimed polynomial $p(x)$ at all. He can calculate evaluation $t = t(r)$, select a random number h and set $p = t \cdot h$, which will be accepted by the verifier as valid, since equation holds.
- Because prover knows the random point $x = r$, he can construct any polynomial which has one shared point at r with $t(r) \cdot h(r)$.
- In the original statement, prover claims to know a polynomial of a particular degree, in the current protocol there is no enforcement of degree. Hence prover can cheat by using a polynomial of higher degree which also satisfies the cofactors check.

The two first issues are possible becuse values are presented at raw, prover knows r and $t(r)$. It would be ideal if those values would be given as a black box, so one cannot temper with the protocol, but still able to compute operations on those obscure values. Something similar to the hash function, such that when computed it is hard to go back to the original input.

Homomorphic Encryption

We could solve the two issues previously mentioned with homomorphic encryption. Namely, it allows to encrypt a value and be able to apply arithmetic operations on such encryption. There are multiple ways to achieve homomorphic properties of encryption, and we will briefly introduce a simple one.

The general idea is that we choose a base natural number g (e.g. 5) and to encrypt a value we exponentiate g to the power of that value. For example, if we want to encrypt the number 3:

$$5^3 = 125$$

Where 125 is the encryption of 3. If we want to multiply this encrypted number by 2, we raise it to the exponent of 2:

$$125^2 = 15625 = (5^3)^2 = 5^{2\times 3} = 5^6$$

We were able to multiply an unknown value by 2 and keep it encrypted. We can also add two encrypted values through multiplication, for example, $3 + 2$:

$$5^3 \cdot 5^2 = 5^{3+2} = 5^5 = 3125$$

Similarly, we can subtract encrypted numbers through division, for example, 5–3:

$$\frac{5^5}{5^3} = 5^5 \cdot 5^{-3} = 5^{5-3} = 5^2 = 25$$

However, since the base 5 is public, it is quite easy to go back to the secret number, dividing encrypted by 5 until the result is 1. **The number of steps is the secret number**.

Modular Arithmetic

The idea of modular arithmetic is to declare that we select only first n natural numbers, i.e., $0, 1, \ldots, n$–1, to work with, and if any given integer falls out of this range, we "wrap" it around. We can achieve this using the modulo operation.

The most important property of the modulo is that the order of operations does not matter, e.g., we can perform all operations first and then apply modulo or apply modulo after every operation. For example $(2 \cdot 4–1) \cdot 3 = 3(\text{mod } 6)$ is equivalent to:

$$\begin{aligned} 2 \cdot 4 &= 2(\text{mod } 6) \\ 2 - 1 &= 1(\text{mod } 6) \\ 1 \cdot 3 &= 3(\text{mod } 6) \end{aligned}$$

This operation is helpful because having a result of operation it is non-trivial to go back to the original numbers because many different combinations will have the same result.

Without the modular arithmetic, the size of the result gives a clue to its solution. This piece of information is hidden otherwise, while common arithmetic properties are preserved.

Strong Homomorphic Encryption

We can combine the homomorphic encryption with the modulo operation. Now, different exponents will have the same result:

$$5^5 = 3(\text{mod } 7)$$
$$5^{11} = 3(\text{mod } 7)$$
$$5^{17} = 3(\text{mod } 7)$$

This is where it gets hard to find the exponent. In fact, if modulo is sufficiently large, it becomes infeasible to do so, and a good portion of the modern-day cryptography is based on the "hardness" of this problem.

All the homomorphic properties of the scheme are preserved in the modular realm:

1. Encryption: $5^3 = 6(\text{mod } 7)$
2. Multiplication: $6^2 = (5^3)^2 = 5^6 = 1(\text{mod } 7)$
3. Addition: $5^3 \cdot 5^2 = 5^5 = 3(\text{mod } 7)$

With all this information, now we can state the encryption function Eq. 11, where v is the value that we want to encrypt.

$$E(v) = g^v(\text{mod n}) \tag{11}$$

However, there are limitations to this homomorphic encryption scheme while we can multiply an encrypted value by an unencrypted value, we cannot multiply (and divide) two encrypted values, or we cannot exponentiate an encrypted value. They will be evaluated later.

Encrypted Polynomial

We can now evaluate a polynomial with an encrypted random value of x and modify the zero-knowledge protocol that we analyzed before accordingly.

As we have established previously to know a polynomial is to know its coefficients, if we want to evaluate polynomial $p(x) = x^3 – 3x^2 + 2x$, its coefficients are: 1, –3, 2. Because homomorphic encryption does not allow to exponentiate an encrypted value, we must give

encrypted values of powers of x from 1 to 3: $E(x), E(x^2), E(x^3)$, so that we can evaluate the encrypted polynomial as follows:

$$\begin{aligned}
E(x^3)^1 \cdot E(x^2)^{-3} \cdot E(x)^2 &= \\
(g^{x^3})^1 \cdot (g^{x^2})^{-3} \cdot (g^x)^2 &= \\
g^{1x^3} \cdot g^{-3x^2} \cdot g^{2x} &= \\
g^{x^3-3x^2+2x}
\end{aligned}$$

As the result of such operations, we have an encrypted evaluation of our polynomial at some unknown to us x. This is quite a powerful mechanism, and because of the homomorphic property, the encrypted evaluations of the same polynomials are always the same in encrypted space.

We can now update the previous version of the protocol, for a polynomial of degree d:

- Verifier
  - Samples a random value s, i.e., secret.
  - Calculates encryptions of s for all powers i in 0, 1, . . . , d, i.e.: $E(s^i) = g^{s^i}$.
  - Evaluates unencrypted target polynomial with s: $t(s)$.
  - Encrypted powers of s are provided to the prover: $E(s^0), E(s^1), \ldots, E(s^d)$.
- Prover
  - Calculates polynomial $h(x) = \frac{p(x)}{t(x)}$.
  - Using encrypted powers $g^{s^0}, g^{s^1}, \ldots, g^{s^d}$ and coefficients $c_0, c_1, \ldots, c_n$ evaluates.
  - The resulting g^p and g^h are provided to the verifier.
- Verifier
  - The last step for the verifier is to check that $p = t(s) \cdot h$: $g^p = (g^h)^{t(s)} \Rightarrow g^p = g^{t(s)\cdot h}$

As the prover does not know anything about s, it makes it hard to come up with non-legitimate but still matching evaluations.

Restricting a Polynomial

The knowledge of a polynomial is the knowledge of its coefficients $c_0, c_1, \ldots, c_i$ and the way we "assign" those coefficients in the protocol is through exponentiation of the corresponding encrypted powers of the secret value s. We do already restrict a prover in the selection of encrypted powers of s, but if such restriction is not enforced one could, for example, use any possible means to find some arbitrary values z_h and z_p which satisfy Eq.

12 and provide them to the verifier instead of g^p and g^h . That is why verifier needs the proof that only encryptions of powers of s were used and nothing else.

$$z_p = (z_h)^{t(s)} \tag{12}$$

Knowledge-of-Exponent Assumption (KEA), introduced in [13], is a method to make sure that only encryption of s, i.e., g^s , was homomorphically "multiplied" by some arbitrary coefficient c and nothing else when needed. An example of how it works is the following:

- Alice has a value a, that she wants Bob to exponentiate to any power (where a is a generator of a finite field group used), the single requirement is that only this a can be exponentiated and nothing else, to ensure this she:
  1. Chooses a random α.
  2. Calculates $a' = a^{\alpha}(\text{mod n})$.
  3. Provides the tuple (a, a') to Bob and asks to perform the same arbitrary exponentiation of each value and reply with the resulting tuple (b, b') where the exponent "α-shift" remains the same, i.e., $b^{\alpha} = b'$ (mod n).
- As Bob cannot extract α from the tuple (a, a') other then through a brute-force which is infeasible, it is conjectured that the only way Bob can produce a valid response is through the procedure:
  1. Chooses some value$c$.
  2. Calculates $b = (a)^c \pmod{n}$ and $b' = (a')^c \pmod{n}$.
  3. Replies with (b, b').
- Having the response and α, Alice checks the equality:

$$\begin{aligned} (b)^{\alpha} &= b' \\ (a^c)^{\alpha} &= (a')^c \\ a^{c \cdot \alpha} &= (a^{\alpha})^c \end{aligned}$$

From this process we can conclude that Bob has applied the same exponent (i.e., c) to both values of the tuple, he could only use the original Alice's tuple to maintain the α relationship, and that he knows the applied exponent c, because the only way to produce valid (b, b') is to use the same exponent. On the other hand, Alice has not learned c for the same reason Bob cannot learn α. However, we will have to be aware that the range of c is the optimum for preserving zero-knowledge property.

In the homomorphic encryption context, exponentiation is the multiplication of the encrypted value. We can apply the same construction in the case with the simple one-coefficient polynomial $f(x) = c \cdot x$:

- Verifier chooses random s, and provides evaluation for $x = s$ for power 1 and its "shift":

$$(g^s, g^{a \cdot s})$$

- Prover applies the coefficient c:

$$((g^s)^c, (g^{a \cdot s})^c) = (g^{c \cdot s}, g^{a \cdot c \cdot s})$$

- Verifier checks:

$$(g^{s \cdot c})^\alpha = g^{c \cdot s \cdot \alpha}$$

Such construction restricts the prover to use only the encrypted s provided, therefore prover could have assigned coefficient c only to the polynomial provided by the verifier. We can now scale such one-term polynomial (monomial) approach to a multi-term polynomial because the coefficient assignment of each term is calculated separately and then homomorphically "added" together [10]. So if the prover is given encrypted exponentiations of s alongside with their shifted values, he/she can evaluate original and shifted polynomial, where the same check must hold. In particular, for a degree d polynomial:

- Verifier provides encrypted powers $g^{s^0}, g^{s^1}, \ldots, g^{s^d}$ and their shifts $g^{\alpha s^0}, g^{\alpha s^1}, \ldots, g^{\alpha s^d}$.
- Prover:
  1. Evaluates excrypted polynomial with provided powers of s: $g^{p(s)} = (g^{s^0})^{c_0} \cdot (g^{s^1})^{c_1} \cdot (g^{s^d})^{c_d} = g^{c_0 s^0 + c_1 s^1 + \ldots c_d s^d}$
  2. Evaluates encrypted "shifted" polynomial with the corresponding α-shifts of the powers of s: $g^{\alpha p(s)} = (g^{\alpha s^0})^{c_0} \cdot (g^{\alpha s^1})^{c_1} \cdot (g^{\alpha s^d})^{c_d} = g^{c_0 \alpha s^0 + c_1 \alpha s^1 + \ldots c_d \alpha s^d} = g^{\alpha(c_0 s^0 + c_1 s^1 + \ldots c_d s^d)}$
  3. Provides the result as g^p, $g^{p'}$ to the verifier.
- Verifier checks: $(g^p)^\alpha = g^{p'}$

Now we can be sure that the prover did not use anything else other than the provided by verifier polynomial, since there is no other way to preserve the α-shift.

Verifier could extract knowledge about the unknown polynomial $p(x)$ only from the data sent by the prover (the proof), and we want to avoid this as we want to apply Zero-Knowledge. This proof participate when we want to check that polynomial $p(x)$ has roots of $t(x)$ ($g^p = (g^h)^{t(s)}$) and when we are checking if it is used a polynomial of a correct form ($(g^p)^\alpha = g^{p'}$).

In order to alter the proof such that the checks still hold but no knowledge can be extracted, we can "shift" the values by some random number δ. Prover samples a random δ and exponentiates his proof values with it: $(g^{p(s)})^\delta$, $(g^{h(s)})^\delta$, $(g^{\alpha p(s)})^\delta$ and provides to the verifier for verification the following terms:

$$(g^p)^\delta = ((g^h)^\delta)^{t(s)}$$
$$((g^p)^\delta)^\alpha = (g^{p'})^\delta$$

After consolidation we can observe that the check still holds:

$$g^p \cdot \delta = g^{h\cdot\delta\cdot t(s)}$$
$$g^{\delta\cdot\alpha p} = g^{\delta\cdot p'}$$

We need the secret parameters to be reusable, public, trustworthy and infeasible to abuse. To achieve this statement, **cryptographic pairings** (bilinear map) are used.

Because the source and output number sets are different the result of the pairing is not usable as an input for another pairing operation. Therefore we cannot multiply the result by another encrypted value and we can only multiply two encrypted values at a time. In some sense, it resembles a hash function, which maps all possible input values to an element in the set of possible output values and it is not trivially reversible.

Using cryptographic pairings, we can set up secure public and reusable parameters. Let us assume that we trust a single honest party to generate secrets s and α. As soon as α and all necessary powers of s with corresponding α-shifts are encrypted, the raw values must be deleted (for i in 0, 1, . . . , d): $g^\alpha, g^{s^i}, g^{\alpha s^i}$.

These parameters are usually referred to as common reference string or CRS. After CRS is generated any prover and any verifier can use it in order to conduct non-interactive zero-knowledge proof protocol. While non-crucial, the optimized version of CRS will include encrypted evaluation of the target polynomial $g^{t(s)}$.

Moreover CRS is divided into two groups (for i in $0, 1, \ldots, d$):

- Proving Key (also called evaluation key): $(g^{s^i}, g^{\alpha s^i})$
- Verification Key: $(g^{t(s)}, g^\alpha)$

Being able to multiply encrypted values the verifier can check the polynomials in the last step of the protocol: Having verification key, the verifier processes received encrypted polynomial evaluations $g^p, g^h, g^{p'}$ from the prover:

1. Checks that $p = t \cdot h$ in encrypted space: $e(g^p, g^1) = e(g^t, g^h)$ which is equivalent to $e(g, g)^p = e(g, g)^{t\cdot h}$
2. Checks polynomial restriction: $e(g^p, g^\alpha) = e(g^{p'}, g)$

While the trusted setup is efficient, it is not effective since multiple users of CRS will have to trust that one deleted α and s, since currently there is no way to prove it. Hence it is necessary to minimize or eliminate that trust. One way to achieve that is by generating a composite CRS by multiple parties employing mathematical tools introduced previously in this section, such that neither of those parties knows the secret.

Let us consider three participants Alice, Bob and Carol with corresponding indices A, B and C, for i in $1, 2, \ldots, d$:

1. Alice samples her random s_A and α_A and publishes her CRS: $(g^{s_A^i}, g^{\alpha_A}, g^{\alpha_A s_A^i})$
2. Bob samples his s_B and α_B and arguments Alice's encrypted CRS through homomorphic multiplication and publishes the resulting two-party Alice-Bob CRS: $(g^{s_{AB}^i}, g^{\alpha_{AB}}, g^{\alpha_{AB} s_{AB}^i})$
3. So does Carol with her s_C and α_C and publishes Alice-Bob-Carol CRS: $(g^{s_{ABC}^i}$, $g^{\alpha_{ABC}}$, $g^{\alpha_{ABC} s_{ABC}^i})$

As the result of such protocol, we have composite s^i and α, where

$s^i = s_A^i s_B^i s_C^i$, $\alpha = \alpha_A \alpha_B \alpha_C$

and no participant learns secret parameters of other participants unless they are colluding. In fact, in order to learn s and α, one must collude with every other participant. Therefore even if one out of all is honest, it will be infeasible to produce fake proofs.

However, an adversary could use those randomly for different powers of s or provide random numbers as an augmented common reference string, making CRS invalid and unusable. In order to avoid this, we will perform **consistency check**, starting with the first parameter and ensuring that every next is derived from it. Every published CRS by participants can be checked as follows:

1. We take power 1 of s as canonical value and check every other power for consistency with it: $e(g^{s^i}, g) = e(g^{s^i}, g^{s^{i-1}})\Big|_{i\in\{2,\ldots,d\}}$
2. We now check if the α-shift of values in the previous step is correct: $e(g^{s^i}, g^{\alpha}) = e(g^{\alpha s^i}, g)\Big|_{i\in[d]}$

However, we need another step to make sure that the last participant is not an adversary, because he/she could ignore the previous CRS and construct valid parameters from scratch, as if he was the first in the chain, therefore being the only one who knows secret s and α. This is why we will also require every participant except the first one to encrypt and publish his secret parameters as Alice did in the example.

The more there are unrelated participants in CRS setup (also called ceremony) the faintest the possibility of fake proofs. The probability decreases if competing parties are participating.

Succinct Non-Interactive Argument of Knowledge of Polynomial

We can now consolidate the evolved zk-SNARKOP protocol.

1. Setup
    - Sample random values s, α.
    - Calculate encryptions g^{α} and $\{g^{s^i}\}_{i\in[d]}$, $\{g^{\alpha s^i}\}_{i\in\{0,\ldots,d\}}$

- Proving key: $(\{g^{s^i}\}_{i\in[d]}, \{g^{\alpha s^i}\}_{i\in\{0,\ldots,d\}})$
- Verification key: $(g^{\alpha}, g^{t(s)})$

2. Proving
   - Assign coefficients $\{c_i\}_{i\in\{0,\ldots,d\}}$ (i.e., knowledge), $p(x) = c_d x^d + \cdots + c_1 x^1 + c_0 x^0$
   - Calculate polynomial $h(x) = \frac{p(x)}{t(x)}$
   - Evaluate encrypted polynomials $g^{p(s)}$ and $g^{h(s)}$ using $\{g^{s^i}\}_{i\in[d]}$.
   - Evaluate encrypted shifted polynomial $g^{\alpha p(s)}$ using $\{g^{\alpha s^i}\}_{i\in\{0,\ldots,d\}}$
   - sample random δ.
   - Set the randomized proof $\pi = (g^{\delta p(s)}, g^{\delta h(s)}, g^{\delta \alpha p(s)})$.
3. Verification
   - Parse proof π as $(g^p, g^h, g^{p'})$
   - Check polynomial restriction $e(g^{p'}, g) = e(g^p, g^{\alpha})$
   - Check polynomial cofactors $e(g^p, g) = e(g^{t(s)}, g^h)$

With this protocol, verifier knows that the prover has a valid polynomial but not which particular one.

2.3.1 zk-SANRK computation

It is used Algorithm 1, which can be mathematically expressed as in Eq.13 (assuming w is either 0 or 1).

Algorithm 1 Operation depends on an input

```
function calc(w, a, b)
if w then
    return a × b
else
    return a + b
end if
      end function
```

$$f(w, a, b) = w(axb) + (1 - w)(a + b) \tag{13}$$

For zk-SNARK, the operands will be represented as polynomials, and we will use their arithmetic properties to get the result of an operation imposed by an operand. If the operands and the output are represented correctly for the operation by polynomials, then the evaluation of $l(a)$ **operator** $r(a) = o(a)$ should hold. And moving output polynomial $o(x)$ to the left side of the equation $l(a)$ **operator** $r(a)–o(a) = 0$ is surfacing the fact that the operation polynomial $l(x)$ **operator** $r(x)–o(x) = 0$ has to evaluate to 0 at a, if

the value represented by the output polynomial $o(x)$ is the correct result produced by the operator on the values represented by operand polynomials $l(x)$ and $r(x)$.

Henceforth operation polynomial must have the root a if it is valid, and consequently, it must contain cofactor $(x–a)$ as we have established previously, which is the target polynomial we prove against, i.e., $t(x) = x–a$.

As an example, we define $l(x) = 3x$, $r(x) = 2x$ and $o(x) = 6x$, which can be evaluated to the corresponding values for $a = 1$, i.e., $l(1) = 3$, $r(1) = 2$ and $o(1) = 6$, all them represented in graph 11.

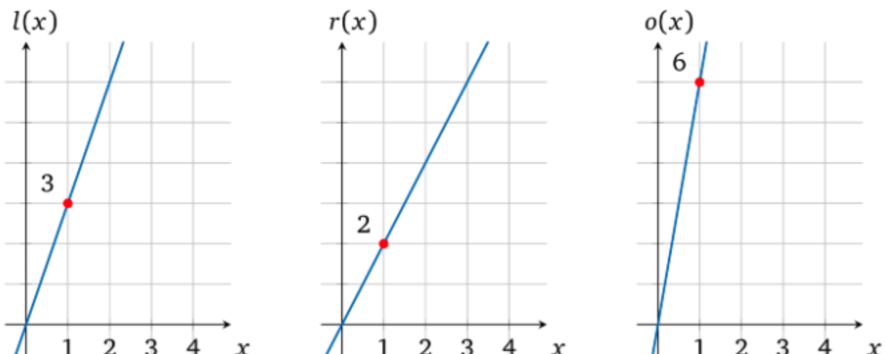


Figure 11: Graphical representation of the computation.

The operation polynomial then will be:

$$
\begin{aligned}
l(x) \times r(x) &= o(x) \\
3x \times 2x &= 6x \\
6x^2 - 6x &= 0
\end{aligned}
$$

which can be visualised as image 12, where is noticeable that the operation polynomial has $(x–1)$ as a co-factor.

Therefore if the prover provides such polynomials $l(x), r(x), o(x)$ instead of former $p(x)$ then the verifier will accept it as valid, since it is divisible by $t(x)$. On the contrary, if the prover tries to cheat and substitutes output value with 4, e.g., $o(x) = 4x$, then the operation polynomial will be $6x^2–4x = 0$, which does not have a solution $x = 1$, and $l(x) \times r(x)–o(x)$ is not divisible by $t(x)$ without remainder. Hence such inconsistent operation will not be accepted by the verifier.

Previously we had one proof of knowledge of polynomial $p(s)$, but now we deal with three $l(s)$, $r(s)$, $o(s)$. We could define $p(s) = l(s) \times r(s)–o(s)$, but the way to compute the operation will change a little bit. In essence what a verifier needs to check in encrypted space is that $l(s) \times r(s) = t(s)h(s) + o(x)$. A verifier can perform multiplication using cryptographic pairings, and we moved the subtraction $(–o(s))$ to the right in order to save computation cost. In encrypted space verifier's check translates to Eq. 14.

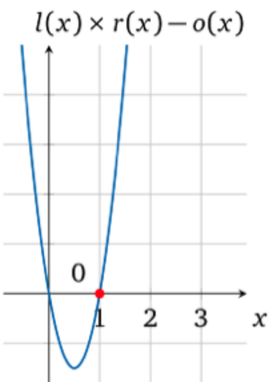


Figure 12: Graphical representation of the polynomial multiplication.

$$\begin{aligned} e(g^{l(s)}, g^{r(s)}) &= e(g^{t(s)}, g^{h(s)}) \cdot e(g^{o(s)}, g) \\ e(g,g)^{l(s)r(s)} &= e(g,g)^{t(s)h(s)} \cdot e(g,g)^{o(s)} \\ e(g,g)^{l(s)r(s)} &= e(g,g)^{t(s)h(s)+o(s)} \end{aligned} \tag{14}$$

For our approach, we will need to prove multiple operations instead of only the one example that we have just analyzed. In order to do so, we will use **polynomial interpolation**, which is a method which given a set of points produces a curved polynomial in such a way that it passes through all those points. This method has different ways it can be performed: Set of equations with unknowns, Newton polynomial, Neville's algorithm, Lagrange polynomials and Fast Fourier transform.

If, as an example, we use the "set of equations with unknowns" method, we will state that there exists a unique polynomial $p(x)$ of degree at most n with yet unknown coefficients which pass through given $n+1$ points such that for each point $\{(x_i, y_i)\}$, $i \in [n+1]$, the polynomial evaluated at x_i should be equal to y_i, i.e. $p(x_i) = y_i$ for all i. For example, using three points it will be polynomial of degree 2 of the form: $ax^2 + bx + c = y$.

Now, if we equalize the evaluated polynomial for each point of the left operand polynomial and solve the system of equations by expressing each coefficient in terms of others:

$$\left\{\begin{array}{lcl} l(1) &=& 2 \\ l(2) &=& 2 \\ l(3) &=& 6 \end{array}\right. \Rightarrow \left\{\begin{array}{lcl} a(1)^2 + b \cdot 1 + c &=& 2 \\ a(2)^2 + b \cdot 2 + c &=& 2 \\ a(3)^2 + b \cdot 3 + c &=& 6 \end{array}\right. \Rightarrow \left\{\begin{array}{rcl} a &=& 2 - b - c \\ 2b &=& 2 - 4(2 - b - c) - c \\ c &=& 6 - 9(2 - b - b) - 3b \end{array}\right. \Rightarrow \left\{\begin{array}{lcl} a &=& 2 \\ b &=& -6 \\ c &=& 6 \end{array}\right.$$

And therefore the left operand polynomial is $l(x) = 2x^2 - 6x + 6$. We will also be able to find $r(x)$ and $o(x)$ in the same way.

The set of all the variable polynomials $\{l_i(x), r_i(x), o_i(x)\}$ for $i \in \{1, \ldots, n\}$ and the target polynomial $t(x)$ is called a **quadratic arithmetic program** (QAP, introduced in [14]).

Our analysis has been primarily focusing on the notion of operation. However, the protocol is not actually "computing" but rather is checking that the output value is the correct result of an operation for the operand's values. That is why it is called a constraint, i.e., a verifier is constraining a prover to provide valid values for the predefined "program" no matter what are they. A multitude of constraints is called a constraint system (in our case it is a rank 1 constraint system (R1CS)). Therefore we can also use constraints to ensure other relationships. For example, if we want to make sure that the value of the variable a can only be 0 or 1 (i.e., binary), we can do it with the simple constraint: $a \times a = a$

Or if we want to constrain a to be 2 we can stablish that $(a - 2) \times 1 = 0$.

Our operation's construction will be of the form of Eq. 15, where c can be ingrained into the proving key, but the v may have any value because the prover supplies it. While we can enforce the $c \cdot v_{one}$ to be 0 by setting $c = 0$, it is hard to find a constraint to enforce v_{one} to be 1 in the construction we are limited by. Therefore the verifier will set the value of v_{one}.

$$\sum_{i=1}^{n} c_{l,i} \cdot v_i \times \sum_{i=1}^{n} c_{r,i} \cdot v_i = \sum_{i=1}^{n} c_{o,i} \cdot v_i \tag{15}$$

2.3.2 zero-Knowledge Protocol

To sum up this section, we will review all the steps we followed to achieve the complete zk-SNARK protocol.

First, we used the random δ shift, expressed in Eq. 16 which makes the proof indistinguishable from random.

$$\delta p(s) = t(s) \cdot \delta h(s) \tag{16}$$

However, with the computation we are proving instead that Eq. 17 is accomplished.

$$L(s) \cdot R(s) - O(s) = t(s)h(s) \tag{17}$$

While we could just adapt this approach to the multiple polynomials using same δ, i.e., supplying randomized values $\delta L(s)$, $\delta R(s)$, $\delta^2 O(s)$, $\delta^2 h(s)$, which would satisfy the valid operations check through pairings as expressed in Eq. 18.

$$e(g,g)^{\delta^2 L(s)R(s)} = e(g,g)^{\delta^2 (t(s)h(s)+O(s))} \tag{18}$$

The issue is that having same δ hinders security, because we provide those values separately in the proof and the following problems could occur:

1. One could easily identify if two different polynomial evaluations have same value, learning some knowledge, e.g.: $g^{\delta L(s)} = g^{\delta R(s)}$.

2. Potential insignificance of differences of values between $L(s)$ and $R(s)$ could allow factoring of those differences through brute-force, for example if $L(s) = 5R(s)$, iterating check $g^{L(s))} = (g^{R(s)})^i$ for $i \in \{1 \dots N\}$ would reveal the $5\times$ difference in just 5 steps. Same brute-force can be performed on encrypted addition operation, e.g. $g^{L(s)} = g^{R(s)+5}$

3. Other correlations between elements of the proof may be discovered, for example, if $e(g^{\delta L(s)}, g^{\delta R(s)}) = e(g^{\delta^2 O(s)}, g)$ then $L(x) \cdot R(x) = O(x)$, etc.

Consequently, we need to have different randomness $(\delta - s)$ for each polynomial evaluation, e.g. Eq.19.

$$\delta_i L(s) \cdot \delta_r R(s) - \delta_0 O(s) = t(s) \cdot (\Delta \textcircled{?} h(s)) \tag{19}$$

To resolve inequality on the right side, we can only modify the proof's value $h(s)$, without alteration of the protocol which would be preferable. Delta (Δ) here represents the difference we need to apply to $h(s)$ in order to counterbalance the randomness on the other side of the equation and $\textcircled{?}$ represents either multiplication or addition operation. If we chose to apply Δ through multiplication $(\textcircled{?} = \times)$ this would mean that it is impossible to find Δ with overwhelming probability, because of randomization showed in Eq.20.

$$\Delta = \frac{\delta_l L(s) \cdot \delta_r R(s) - \delta_0 O(s)}{t(s)h(s)} \tag{20}$$

If we set $\delta_0 = \delta_l \cdot \delta_r$, then, solving Eq.20, we will get that $\Delta = \delta_l \delta_r$.

However, as noted previously this hinders the zero-knowledge property, and even more importantly such construction will not accommodate the verifier's input polynomials since they must be multiples of the corresponding $\delta - s$, which would require an interaction. Therefore we should try applying Δ through addition $(\textcircled{?} = +)$, since it is available for homomorphically encrypted values. We will get Eq.21.

$$\begin{aligned}
(L(s) + \delta_i) \cdot (R(s) + \delta_r) - (O(s) + \delta_0) = t(s) \cdot (\Delta + h(s)) \\
\Delta = \frac{L(s)R(s) - O(s) + \delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_0 - t(s)h(s)}{t(s)} \Rightarrow \\
\Delta = \frac{\delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_0}{t(s)} \\
\Delta = \delta_r L(s) + \delta_l R(s) + \delta_l \delta_r t(s) - \delta_0
\end{aligned} \tag{21}$$

This statement can efficiently compute in the encrypted space and it will result Eq. 22.

$$g^{\Delta} = (g^{L(s)})^{\delta_r} \cdot (g^{R(s)})^{\delta_l} \cdot (g^{t(s)})^{\delta_l \delta_r} g^{-\delta_0} \tag{22}$$

This leads to passing of valid operations check while concealing the encrypted values: $L \cdot R - O + t(\delta_r L + \delta_l R + \delta_l \delta_r t - \delta_0) = t(s)h + t(s)(\delta_r L + \delta_l R + \delta_l \delta_r t - \delta_0)$. The construction

is statistically zero-knowledge due to addition of uniformly random multiples of $\delta_l, \delta_r, \delta_0$ [11].

To make the "variable polynomials restriction" and "variable values consistency" checks coherent with the zero-knowledge alterations, it is necessary to add the parameters $g_l^{t(s)}$, $g_r^{t(s)}$, $g_0^{t(s)}$, $g_l^{\alpha_l t(s)}$, $g_r^{\alpha_r t(s)}$, $g_0^{\alpha_0 t(s)}$, $g_l^{\beta t(s)}$, $g_r^{\beta t(s)}$, $g_0^{\beta t(s)}$ to the proving key.

On the other hand, the original Pinocchio protocol [12] was concerned primarily with the verifiable computation and less with the zero-knowledge property, which is a minor modification and comes almost for free.

2.3.3 zk-SNARK Protocol

The final zk-SNARK Protocol (also called Pinocchio), which includes all the improvements explained in the previous sections, is the following one.

- Setup
  - Select a generator g and a cryptographic pairing e.
  - For a function $f(u) = y$ with n total variables of which m are input/output variables, convert into the polynomial form (quadratic arithmetic program)$(\{l_i(x), r_i(x), o_i(x)\}_{i\in\{0,\ldots,n\}}, t(x))$ of degree d (equal to the number of operations) and size $n+1$.
  - Sample random $s, \rho_l, \rho_r, \alpha_l, \alpha_r, \alpha_0, \beta, \gamma$.
  - Set $\rho_0 = \rho_l \cdot \rho_r$ and the operand generators $g_l = g^{\rho_l}$, $g_r = g^{\rho_r}$, $g_0 = g^{\rho_0}$
  - Set the proving key: $(\{g^{s^k}\}_{k\in[d]}, \{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\}_{i\in\{0,\ldots,n\}}$

    $\{g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\}_{i\in\{m+1,\ldots,n\}}$

    $\{g_l^{t(s)}, g_r^{t(s)}, g_o^{t(s)}, g_l^{\alpha_l t(s)}, g_r^{\alpha_r t(s)}, g_o^{\alpha_o t(s)}, g_l^{\beta t(s)}, g_r^{\beta t(s)}, g_o^{\beta t(s)}\})$
  - Set the verification key:

    $(g^1, g_o^{t(s)}, \{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\}_{i\in\{0,\ldots,m\}}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\gamma}, g^{\beta\gamma})$
- Proving
  - For the input u, execute the computation of $f(u)$obtaining values $\{v_i\}_{i\in\{m+1,\ldots,n\}}$ for all the intermediary variables.
  - Assign all values to the unencrypted variable polynomials $L(x) = l_o(x) + \sum_{i=1}^{n} v_i \cdot l_i(x)$ and similarly $R(x), O(x)$.
  - Sample random δ_l, δ_r and δ_o.
  - Find $h(x) = \frac{L(x)R(x)-O(x)}{t(x)} + \delta_r L(x) + \delta_l R(x) + \delta_l \delta_r t(x) - \delta_o$.
  - Assign the prover's variable values to the encrypted variable polynomials and apply sero-knowledge δ-shift $g_l^{L_p(s)} = (g_l^{t(s)})^{\delta_i} \cdot \Pi_{i=m+1}^{n} (g_l^{l_i(s)})^{v_i}$ and similarly $g_r^{R_p(s)}, g_o^{O_p(s)}$

- Assign its α-shifted pairs $g_l^{L'_p(s)} = (g_l^{\alpha_l t(s)})^{\delta_l} \cdot \Pi_{i=m+1}^{n} (g_l^{\alpha_l l_i(s)})^{v_i}$ and similarly $g_r^{R'_p(s)}, g_o^{O'_p(s)}$
- Assign the variable values consistency polynomials
  $g^{Z(s)} = (g_l^{\beta t(s)})^{\delta_l} (g_r^{\beta t(s)})^{\delta_r} (g_o^{\beta t(s)})^{\delta_o} \cdot \Pi_{i=m+1}^{n} (g_l^{\beta l_i(s)} g_r^{\beta r_i(s)} g_o^{\beta o_i(s)})^{v_i}$
- Compute the proof $(g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g^{h(s)}, g_l^{L'_p(s)}, g_r^{R'_p(s)}, g_o^{O'_p(s)}, g^{Z(s)})$

- Verification
  - Parse a provided proof as $(g_l^{L_p}, g_r^{R_p}, g_o^{O_p}, g^h, g_l^{L'_p}, g_r^{R'_p}, g_o^{O'_p}, g^Z)$
  - Assign input/output values to verifier's encrypted polynomials and add to 1: $g_l^{L_v(s)} = g_l^{l_o(s)} \cdot \Pi_{i=1}^{n} (g_l^{l_i(s)})^{v_i}$ and similarly for $g_r^{R_v(s)}$ and $g_o^{O_v(s)}$.
  - Variable polynomials restriction check: $e(g_l^{L_p}, g^{\alpha_l}) = e(g_l^{L'_p}, g)$ and similarly for $g_r^{R_p}$ and $g_o^{O_p}$.
  - Variable values consistency check: $e(g_l^{L_p} g_r^{R_p} g_o^{O_p}, g^{\beta\gamma}) = e(g^Z, g^\gamma)$
  - Valid operations check: $e(g_l^{L_p} g_l^{L_v(s)}, g_r^{R_p} g_r^{R_v(s)}) = e(g_o^{t(s)}, g^h) \cdot e(g_o^{O_p} g_o^{O_v(s)}, g)$

This effective protocol allows proving computation:

- Succinctly: Independently from the amount of computation the proof is of constant, small size.
- Non-interactively: As soon as the proof is computed it can be used to convince any number of verifiers without direct interaction with the prover.
- With argumented knowledge: The statement is correct with non-negligible probability, i.e., fake proofs are infeasible to construct; moreover prover knows the corresponding values for the true statement (i.e. witness), e.g., if the statement is "B is a result of $sha256(a)$" then the prover knows some a such that $B = sha256(a)$ which is useful since B could only be computed with the knowledge of a as well as it's infeasible to compute a from B (assuming a have enough entropy).
- The statement is correct with non-negligible probability, i.e., fake proofs are infeasible to construct.
- In zero-knowledge — it is "hard" to extract any knowledge from the proof, i.e., it is indistinguishable from random.

It was possible to achieve primary due to unique properties of polynomials, modular arithmetic, homomorphic encryption, elliptic curve cryptography, cryptographic pairings and ingenuity of the inventors. This protocol proves computation of a unique finite execution machine which in one operation can add together almost any number of variables but may only perform one multiplication. Therefore there is an opportunity to both optimize programs to leverage this specificity efficiently as well as use constructions which minimize the number of operations. It is essential that verifier does not have to know any secret data in order to verify a proof so that properly constructed verification key can

be published and used by anyone in a non-interactive manner. Which is contrary to the "designated verifier" schemes where the proof will convince only one party, therefore it is non-transferable. In zk-SNARK context, we can achieve this property if untrustworthy or a single party generates the keypair.

2.3.4 zk-SNARK Comparison

STARK and Bulletproofs are technologies that also apply zero knowledge proof to perform their protocol. zk-SNARK and zk-STARK are the most compelling zero-knowledge technologies in the market today. zk-STARK, in contrast to zk-SNARK, stands for zero-knowledge scalable **transparent** argument of knowledge. Furthermore, both of these zero-knowledge technologies are non-interactive by nature, meaning the code can be deployed and act autonomously.

On the other hand, Bulletproofs protocol is based on two privacy systems: zero-knowledge proofs (zk-SNARKs) and confidential transactions (Confidential Transactions - CT). The Bulletproofs protocol makes a pretty clever combination of both systems. In a more technical concept, Bulletproofs protocols are a much more efficient method than zk-SNARKs. It allows the verification of certain parameters in a confidential and secure way. Bulletproofs can also be used without the need for a trusted configuration by the creator. However, Bulletproof protocol is good for small circuit but for big ones is bad.

Comparing these three zero knowledge systems, we get table 5.

zk System	SNARKs	STARKs	Bulletproofs
Proof size	288 bytes	45 KB-200 KB	$\approx$ 1.3 KB
Prover Time	2.3 s	1.6 s	30 s
Verification Time	10 ms	16 ms	1100 ms
Algoritmic complexity: prover	$O(N \cdot \log(N))$	$O(N \cdot \text{polylog}(N))$	$O(N \cdot \log(N))$
Algoritmic complexity: verifier	$\approx O(1)$	$O(\text{polylog}(N))$	$O(N)$
Communication complexity (proof size))	$\approx O(1)$	$O(\text{polylog}(N))$	$O(\log(N))$
Size estimate for 1 TX	Tx:200 bytes, Key: 50 MB	45 kB	1.5 kb
Size estimate for 10.000 TX	Tx:200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	$\approx$ 600k (Groth16)	$\approx$ 2.5M (estimate, no impl.)	N/A
Trusted setup required?	Yes	No	No
Post-quantum secure	No	Yes	No
Crypto assumptions	Strong	Collision resistant hashes	Discrete log

Table 5: Comparison between the different systems of Zero Knowledge.

2.4 Pinocchio protocol

This section aims to explain in detail the main concepts and operations of the Pinocchio protocol.

The previous sections have described mathematical concepts that will help us to understand in more detail how Pinocchio protocol works. This protocol has four different phases, each one is defined in one of the following subsections.

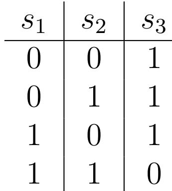
2.4.1 Circuit design

In section 2.3, we have analyzed mathematically how the protocol works. In a practical setting, we will use arithmetic circuits which will be designed to output some of the equations presented in the previous section that are used to verify some proofs.

Imagine we have a simple circuit, which has a NAND gate like the one in figure 14, whose truth table is table 13.

We can find the equation for the NAND doing a Lagrange interpolation using each point of its truth table. For $s_1 = 0$, we interpolate the points $(0, 1), (1, 1)$:

$$1 \frac{(s_2 - 1)}{(0 - 1)} + 1 \frac{(s_2 - 0)}{(1 - 0)} = 1.$$



s_1	s_2	s_3
0	0	1
0	1	1
1	0	1
1	1	0

Figure 13: NAND gate Truth table

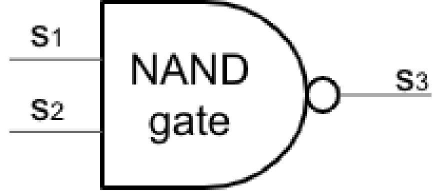


Figure 14: NAND gate

For $s_1 = 1$, we interpolate the points $(0, 1), (1, 0)$:

$$1\,\frac{(s_2 - 1)}{(0 - 1)} + 0\,\frac{(s_2 - 0)}{(1 - 0)} = 1 - s_2.$$

s_1	s_3
0	1
1	$1 - s_2$

Figure 15: Intermediate interpolation of the NAND.

Finally, we interpolate the resulting points $(0, 1), (1, 1 - s_2)$:

$$NAND(s_1, s_2) = 1\,\frac{(s_1 - 1)}{(0 - 1)} + (1 - s_2)\,\frac{(s_1 - 0)}{(1 - 0)} = (1 - s_1) + s_1 - s_1\,s_2 = 1 - s_1\,s_2. \quad (23)$$

As shown, we can express a input/output computation, given as a table, through sums and multiplications and the result is the one of Eq.23.

The circuit operates on values in the finite field $\mathbb{F}_p$, where p is a large primer number.

If we take now a little more complex circuit, like the one of figure 16, the resulting equations will change a little bit too, they are the ones of Eq.24.

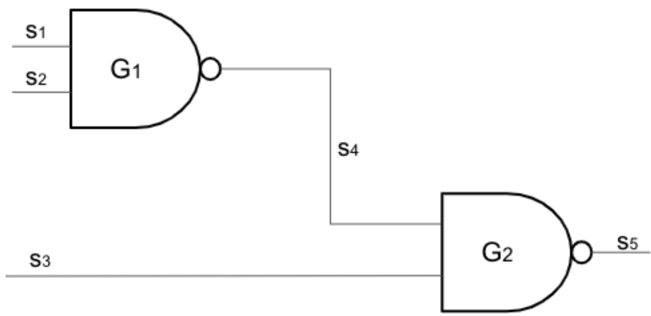


Figure 16: Circuit with two NAND gates.

$$\begin{cases} s_4 = 1 - s_1 s_2 \\ s_5 = 1 - s_3 s_4 \end{cases} \tag{24}$$

If we want to force an input to be a binary number, we can use circuit of figure 17, whose equations are the ones of Eq. 25.

For example, our circuit could have an AND gate, and we could force a value condition for our inputs: we could force one of the inputs to be a binary number, adding the condition of $a(a-1) = 0$. This way we will make sure that a will always be 1 or 0, otherwise this condition would not be fulfilled and therefore this circuit will not run.

We will generate a system of equations in order to meet the circuit conditions. Then, we will normalize this system of equations, which are the constraints that describe the arithmetic circuit, to the R1CS format.

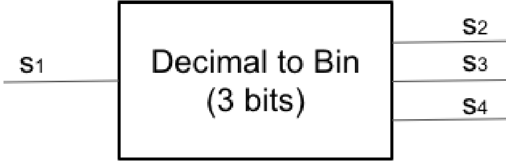


Figure 17: Decimal to binary circuit.

$$\begin{cases} s_2 \cdot 2^0 + s_3 \cdot 2^1 + s_4 \cdot 2^2 = s_1 \\ s_2 \cdot (s_2 - 1) = 0 \\ s_3 \cdot (s_3 - 1) = 0 \\ s_4 \cdot (s_4 - 1) = 0 \end{cases} \tag{25}$$

Let us define a generic input to binary circuit with the system of equations in Eq. 26.

$$\begin{cases} b_0 \cdot 2^0 + b_1 \cdot 2^1 + \cdots + b_{n-1} \cdot 2^{n-1} = f \\ b_0 \cdot (b_0 - 1) = 0 \\ b_1 \cdot (b_1 - 1) = 0 \\ \ldots \\ b_{n-1} \cdot (b_{n-1} - 1) = 0 \end{cases} \tag{26}$$

For the analysis of this section we will also use a generic binary adder, defined with the system of equations of Eq. 27, where the first equation is the equation for the decimal to binary conversion while the last three equations are used to check that each output value is binary.

$$
\begin{cases}
\quad in_{0,0} \cdot 2^0 + in_{0,1} \cdot 2^1 + \quad \dots \quad + \quad in_{0,n-1} \cdot 2^{n-1} + \\
+ \quad in_{1,0} \cdot 2^0 + in_{1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{1,n-1} \cdot 2^{n-1} + \\
\quad \dots \qquad \dots \\
+ \quad in_{m-1,0} \cdot 2^0 + in_{m-1,1} \cdot 2^1 + \quad \dots \quad + \quad in_{m-1,n-1} \cdot 2^{n-1} = \\
= \quad out_0 \cdot 2^0 + out_1 \cdot 2^1 + \quad \dots \quad + \quad out_{n+c-1} \cdot 2^{n+c-1} \\
\\
out_0 \cdot (out_0 - 1) = 0 \\
out_1 \cdot (out_1 - 1) = 0 \\
\dots \\
out_{n+c-1} \cdot (out_{n+c-1} - 1) = 0
\end{cases}
\tag{27}
$$

Any circuit can be represented by a **R1CS system**, as shown in Eq.28, where a_{ij}, b_{ij} and c_{ij} are constants in $\mathbb{F}_r$. The signals of the circuit are represented as s_i.

$$
\begin{cases}
(a_{11}s_1 + a_{12}s_2 + \dots + a_{1n}s_n) \cdot (b_{11}s_1 + b_{12}s_2 + \dots + b_{1n}s_n) - (c_{11}s_1 + (c_{12}s_2 + \dots + c_{1n}s_n) = 0 \\
(a_{21}s_1 + a_{22}s_2 + \dots + a_{2n}s_n) \cdot (b_{21}s_1 + b_{22}s_2 + \dots + b_{2n}s_n) - (c_{21}s_1 + c_{22}s_2 + \dots + c_{2n}s_n) = 0 \\
\dots \\
(a_{m1}s_1 + a_{m2}s_2 + \dots + a_{mn}s_n) \cdot (b_{m1}s_1 + b_{m2}s_2 + \dots + b_{mn}s_n) - (c_{m1}s_1 + c_{m2}s_2 + \dots + c_{mn}s_n) = 0
\end{cases}
\tag{28}
$$

In this system of equations we have some constants and some variables. The constants are what the circuit marks us and the variables are the different signals that the circuit has to meet.

So in this phase we will be able to compute the constants $a_i(x), b_i(x), c_i(x)$, and we know $Z(x)$. Here we also have the two generator groups $\mathbb{G}_1$ and $\mathbb{G}_2$.

Now, we will apply the **Quadratic Arithmetic Program** (QAP) technique, which describes the circuit through polynomials, interpolating the constants a, b and c at the roots of unity. It is basically a relationship between polynomials. We assign a root of 1 for each constraint. In order to do that, we will find a polynomial $\alpha_1(x)$ as

$$
\begin{cases}
\alpha_1(\omega_1) = a_{11} \\
\alpha_1(\omega_2) = a_{21} \\
\dots \\
\alpha_1(\omega_m) = a_{m1}
\end{cases}
$$

where $w_1, w_2, \dots, w_m$ are fixed points, such as $1, 2, \dots, m$. We also find all the $u_i(x), v_i(x)$ and $w_i(x)$ for all i such that

$$
\begin{cases}
u_i(\omega_1) = a_{1i} \\
u_i(\omega_2) = a_{2i} \\
\dots \\
u_i(\omega_m) = a_{mi}
\end{cases}
\begin{cases}
v_i(\omega_1) = b_{1i} \\
v_i(\omega_2) = b_{2i} \\
\dots \\
v_i(\omega_m) = b_{mi}
\end{cases}
\begin{cases}
w_i(\omega_1) = c_{1i} \\
w_i(\omega_2) = c_{2i} \\
\dots \\
w_i(\omega_m) = c_{mi}
\end{cases}
$$

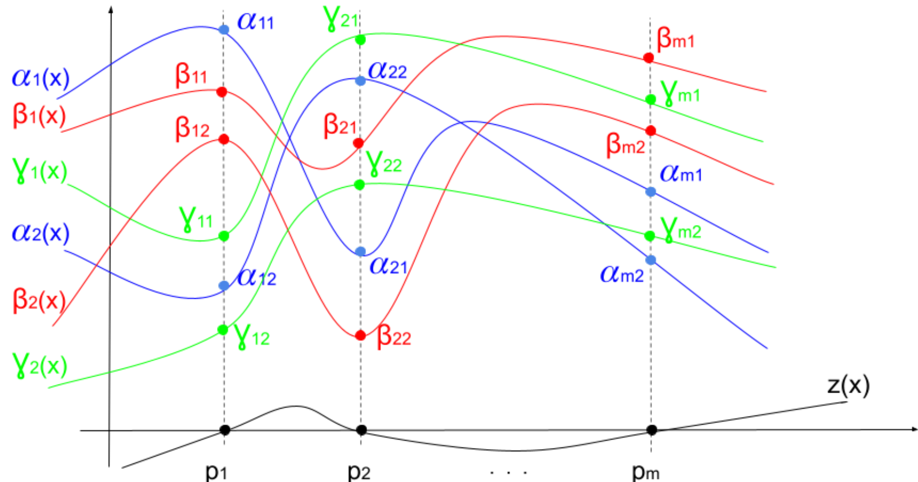


Figure 18: Polynomial representation.

Thus, checking the R1CS system is equivalent to check the following equation of polynomials:

$$\begin{aligned}(u_1(x)s_1 + u_2(x)s_2 + \cdots + u_n(x)s_n) \cdot (v_1(x)s_1 + v_2(x)s_2 + \cdots + v_n(x)s_n) - \\ -(w_1(x)s_1 + w_2(x)s_2 + \cdots + w_n(x)s_n) = Z(x)H(x)\end{aligned} \tag{29}$$

In the QAP system, $Z(x)$ is the polynomial created with the roots in the points selected for the Lagrange interpolation (Eq.30) and $H(x)$ will be the resulting polynomial of dividing by $Z(x)$ (Eq. 31).

$$Z(x) = (x - \omega_0)(x - \omega_1) \ldots (x - \omega_m) = x^m - 1 \tag{30}$$

$$H(x) = \frac{(u_1(x)s_1+u_2(x)s_2+\cdots+u_n(x)s_n)\cdot(v_1(x)s_1+v_2(x)s_2+\cdots+v_n(x)s_n)+(w_1(x)s_1+w_2(x)s_2+\cdots+w_n(x)s_n)}{Z(x)} \tag{31}$$

In section 2.3 it was demonstrated that $P(x) = Z(x) \cdot H(x)$. The system can not compute $P(x)$ once the circuit is developed, as it depends on the s values. Instead, it will be computed by the prover with the fixed values of s. The same happens with the computation of $H(x)$, which is also calculated in the trusted setup by the prover. On the ther hand, $Z(x)$ is fixed, and, for example, for the roots of 1, $Z(x)$ equals $x^n - 1$.

2.4.2 Trusted Setup

This phase could be done in a multi-party ceremony, but we will do the example with only one participant (our prover). This prover chooses a random value for t and computes

$t \cdot \mathbb{G}_1 = H_1(x)$, $t^2 \cdot \mathbb{G}_1 = H_2(x)$, ..., $t^n \cdot \mathbb{G}_1 = H_n(x)$. He/She basically encrypts the powers of t, and $\mathbb{G}$ and H are points. The values of H are public, it is what is known as "Structured Reference String"(SRS). On the other hand, t is the *toxic value*, so it would be like the private key. This way to save the toxic value is a little bit worse than the discrete logarithmic problem but it is still valid and robust.

With the SRS we can take any polynomial, evaluate it on t and encrypt it.

$$
\begin{aligned}
P(x) &= a_0 + xa_1 + x^2 a_2 \\
\mathbb{G}_1 P(t) &= a_0\mathbb{G}_1 + a_1 H_1 + a_2 H_2 = \\
&\qquad \mathbb{G}^{a_0} H_1^{a_1} H_2^{a_2}
\end{aligned}
$$

The part of computing the H is **universal**, which means that could be calculated even before of having designed the definitive circuit and therefore could be used for any circuit.

2.4.3 Prover

He calculates the witness (he executes the circuit for a certain s_n that he knows) and computes $P(x)$, in order to divide it by $Z(x)$ to finally find the value of $H(x)$.

Once the circuit is designed, the prover calculates the witness executing the circuit for a certain s_n that he/she knows: it computes $s_0 \dots s_n$ so that it can then compute the polynomial $P(x)$ and then it divides by $Z(x)$ to know $H(x)$.

This means that if someone knows the witness, he/she is able to calculate the $H(x)$ polynomial.

$$H(x) = h_0 + h_1 x + h_2 x^2 + ... + h_n x^n$$

Knowing the witness, we are sure that this division will have no remainder because the numerator is 0 at all p_i. Therefore, the numerator is divisible by $Z(x)$.

The idea is enabling a prover to being able to prove a verifier that the previous equations hold at a random and unknown point $x = t$. This point t is the toxic value generated randomly in the trusted setup and then discarded.

The prover will also have to compute the variables π_a, π_b, π_c and π_h, whose meaning will be explained in the next section in order to follow in the orderly way the steps of the protocol.

2.4.4 Verifier

In the end, the objective putting all together is that we want to verify the equation $P(x) = Z(x)H(x)$ on t (whose value is unknown for everyone) and encrypt it (meaning that everything will be multilpied by $\mathbb{G}$). The first step is to rewrite the polynomial $P(x)$ (expressed in Eq. 29) evaluating it on t as it is done in Eq. 32.

$$(u_0(t)s_0 + u_1(t)s_1 + \cdots + u_n(t)s_n) \cdot (v_0(t)s_0 + v_1(t)s_1 + \cdots + v_n(t)s_n) - (w_0(t)s_0 + w_1(t)s_1 + \cdots + w_n(t)s_n) = Z(t)H(t) \tag{32}$$

The pairing $\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{(u_0(t)s_0+u_1(t)s_1+\dots+u_n(t)s_n)\cdot(v_0(t)s_0+v_1(t)s_1+\dots+v_n(t)s_n)-(w_0(t)s_0+w_1(t)s_1+\dots+w_n(t)s_n)}$ must be equal to the pairing $\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{Z(t)H(t)}$.

Now, applying pairing properties, we will pass the values of the exponents to one of the two generator groups ($\mathbb{G}_1$, $\mathbb{G}_2$). In this case, u_i and w_i will be in the $\mathbb{G}_1$ group whereas v_i will be in $\mathbb{G}_2$.

$$\hat{e}(\mathbb{G}_1 H(t), \mathbb{G}_2 Z(t)) = \frac{\hat{e}(\mathbb{G}_1[u_1(t)s_1+u_2(t)s_2+\dots+u_n(t)s_n], \mathbb{G}_2[v_1(t)s_1+v_2(t)s_2+\dots+v_n(t)s_n])}{\hat{e}(w\mathbb{G}_1[w_1(t)s_1+w_2(t)s_2+\dots+w_n(t)s_n], \mathbb{G}_2)}$$

Now we define the points A, B, C and H:

$$\begin{cases} A_1 = \mathbb{G}_1 u_1(t) \\ A_2 = \mathbb{G}_1 u_2(t) \\ \dots \\ A_n = \mathbb{G}_1 u_n(t) \end{cases} \begin{cases} B_1 = \mathbb{G}_2 v_1(t) \\ B_2 = \mathbb{G}_2 v_2(t) \\ \dots \\ B_n = \mathbb{G}_2 v_n(t) \end{cases} \begin{cases} C_1 = \mathbb{G}_1 w_1(t) \\ C_2 = \mathbb{G}_1 w_2(t) \\ \dots \\ C_n = \mathbb{G}_1 w_n(t) \end{cases} \begin{cases} H_0 = \mathbb{G}_1 \\ H_1 = \mathbb{G}_1 t \\ H_2 = \mathbb{G}_1 t^2 \\ \dots \\ H_n = \mathbb{G}_1 t^n \end{cases}$$

following the same strategy as before, as $a_i(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$, we can express $\mathbb{G} \cdot a_i(t) = a_0H_0 + a_1H_1 + \dots + a_nH_n$, being $a_0H_0, a_1H_1, \dots, a_nH_n$ the coefficients of the polynomial $\mathbb{G} \cdot a_i(t)$. For each polynomial we have one point (A_i, B_i and C_i). Moreover, this part could also be defined during the trusted setup process.

We will also define the variable vk_z which will be our polynomial $Z(t)$ in the $\mathbb{G}_2$ group.

$$vk_z = \mathbb{G}_2 Z(t)$$

Thus, the equation remains as

$$\hat{e}(\sum_{i=1}^{n} h_i H_i, vk_z) = \frac{\hat{e}(\sum_{i=1}^{n} A_i s_i, \sum_{i=1}^{n} B_i s_i)}{\hat{e}(\sum_{i=1}^{n} C_i s_i, \mathbb{G}_2)}$$

where, as $H(x) = h_0 + h_1x + h_2x^2 + \dots + h_nx^n$, $\mathbb{G}_1 H(t) = h_0\mathbb{G}_1 + h_1H_1 + h_2H_2 + \dots + h_nH_n$, which is computed by the prover and we also defined previously.

Moreover, as we know that $Z(x) = x^m - 1$, $\mathbb{G}_2 Z(t)$ would be equal to $H_m\Big|_{\mathbb{G}_2} - \mathbb{G}_2$, taking into account that $H_m\Big|_{\mathbb{G}_2} = t^m\mathbb{G}_2$, which is establish during the trusted setup.

Now we will define another pairing equation that the verifier will have to check:

$$\hat{e}(\pi_a + vk_{\vec{x}}, \pi_b) = \hat{e}(\pi_h, vk_z)\hat{e}(\pi_c, \mathbb{G}_2) \tag{33}$$

Where

$$vk_{\vec{x}} = A_0 + \sum_{i=1}^{nPublic} A_i s_i$$

$$\pi_a = \sum_{i=nPublic+1}^{n} A_i s_i, \qquad \pi_b = \sum_{i=1}^{n} B_i s_i, \qquad \pi_c = \sum_{i=1}^{n} C_i s_i, \qquad \pi_h = \sum_{i=1}^{n} H_i h_i$$

With these definitions, the prover computes the prove calculating π_a, π_b and π_c. The points B_i will be in $\mathbb{G}_2$, while points A_i and C_i will be in $\mathbb{G}_1$. So, computing the prove will only consist of taking the points A_i and multiply each one for each corresponding s signal ($A_1 \cdot s_1, A_2 \cdot s_2, \ldots, A_n \cdot s_n$, etc.). On the other hand, π_h is the polynomial $H(x)$ evaluated on t and encrypted.

In the case of π_a, we divide the summation in two parts: the first one goes from $i = 1$ to $nPublic$ and it will be computed by the verifier, because it is the part of a that owns to the public signals, the ones that will be forced by the verifier. And the other part of the summatory is one of the proofs that the prover will compute. Forcing the prover to include a certain public signals is the way to check if he/she is a trusted party, and we will check it with the pairing equation, if it satisfied then the verifier will trust the prover.

So this is the first step that does the verifier: he/she computes $vk_{\vec{x}}$, then it takes π_a and sums it, takes π_b, π_c and π_h and sends them to the prover in order that he/she can compute their value with the witness. $\mathbb{G}_2$ is known, $\mathbb{G}_2 Z(t)$ had been previously calculated, which is the variable vk_z. And the verifier basically checks if the pairing equation (Eq.33) is satisfied, where π_a, π_b, π_c and π_h are provided by the prover.

However, until this point, the protocol has some vulnerabilities that could be exploited by an untrusted partner. We apply more restrictions in the protocol in order to be sure that the values of π_a, π_b, π_c and π_h were honestly computed: we will prove that they are a linear combination of the coefficients. And we will also check that the same s_i has been used.

We will first generate another Trusted Setup, which, unlike the first part of this setup, will be specific for this circuit. We will generate an a' for each point of A. It will also be included a k value, which is another "toxic value", meaning that the verifier will compute it once but it will remain secret. So we will ask the prover to compute π'_a, which is defined below.

For testing that A is made of a linear combination we will use this equation:

$$\sum_{i=1}^{n} A_i s_i k_a = \sum_{i=1}^{n} A_i s_i k_a$$

which can be expressed as a pairing equation:

$$\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\sum_{i=1}^{n} A_i s_i k_a} = \hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\sum_{i=1}^{n} A_i s_i k_a}$$

$$\hat{e}(vk_{\vec{x}} + \pi_a, \mathbb{G}_2 k_a) = \hat{e}(\sum_{i=1}^{n} k_a A_i s_i, \mathbb{G}_2)$$

Knowing that

$$A'_i = k_a u_i(t)_1, \qquad vk_a = k_a \mathbb{G}_2, \qquad \pi'_a = \sum_{i=1}^{n} A'_i s_i$$

Then:

$$\hat{e}(vk_{\vec{x}} + \pi_a, vk_a) = \hat{e}(\pi'_a, \mathbb{G}_2) \tag{34}$$

And we have been able to check that we can trust the prover, as, in the end, the verifier has to check that the pairing equalization is accomplished. We will do the same for B and C. Note that we define π'_b in the $\mathbb{G}_1$ group instead of the second because it will take less time to compute it, as it is the complex group and an exponentiation in $\mathbb{G}_1$ is 3.5 times faster than one in $\mathbb{G}_2$.

$$B'_i = k_b v_i(t) \mathbb{G}_1, \qquad vk_b = k_b \mathbb{G}_1, \qquad \pi'_b = \sum_{i=1}^{n} B'_i s_i$$

$$\hat{e}(vk_b, \pi_b) = \hat{e}(\pi'_b, \mathbb{G}_2) \tag{35}$$

$$C'_i = k_c w_i(t) \mathbb{G}_1, \qquad vk_c = k_c \mathbb{G}_2, \qquad \pi'_c = \sum_{i=1}^{n} C'_i s_i$$

$$\hat{e}(\pi_c, vk_c) = \hat{e}(\pi'_c, \mathbb{G}_2) \tag{36}$$

So the verifier will compute these three pairings in order to guarantee that π_a, π_b and π_h were honestly computed by the prover, since the prover does not know the value of k so he/she will not be able to compute two different points that satisfy these pairing equations if they are not a linear combination of $A'/B'/C'$.

Finally, the second check that we will do in order to make robuster our protocol is to check that we use the same coefficients (the same values of $s_1, s_2, \ldots, s_n$) for A, B and C. To do that, we will define k_γ and k_β that are two more toxic values.

$$\sum_{i=1}^{n} [u_i(t)_i + v_i(t) + w_i(t)] s_i k_\gamma k_\beta = \sum_{i=1}^{n} [u_i(t)_i + v_i(t) + w_i(t)] s_i k_\gamma k_\beta$$

We will do the same verification as always, comparing the pairing equations.

$$\hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\sum_{i=1}^{n}[u_i(t)_i+v_i(t)+w_i(t)]s_i k_\gamma k_\beta} = \hat{e}(\mathbb{G}_1, \mathbb{G}_2)^{\sum_{i=1}^{n}[u_i(t)_i+v_i(t)+w_i(t)]s_i k_\gamma k_\beta}$$

Moreover, in the trusted setup we will also have to calculate the following constants:

$$vk_\gamma = k_\gamma \mathbb{G}_2$$
$$vk_{\gamma\beta}^2 = k_\gamma k_\beta \mathbb{G}_2$$
$$vk_{\gamma\beta}^1 = k_\gamma k_\beta \mathbb{G}_1$$

And we will also define another set of points with the variable K_i:

$$K_i = k_\beta(u(t) + v(t) + w(t))\mathbb{G}_1$$
$$\pi_k = \sum_{i=1}^{n} K_i s_i$$

π_k has to be computed by the prover also in the trusted setup, when A', B', C' are computed. Finally, the last equation that the verifier will have to check is the following one:

$$\hat{e}(vk_{\vec{x}} + \pi_a + \pi_c, vk_{\gamma\beta}^2) \cdot \hat{e}(vk_{\gamma\beta}^1, \pi_b) = \hat{e}(\pi_k, vk_\gamma) \tag{37}$$

To sum up, the proof consists on 8 points: π_a in $\mathbb{G}_1$, π_b in $\mathbb{G}_2$, π_c in $\mathbb{G}_1$, π_a', π_b' and π_c' in $\mathbb{G}_1$, π_k in $\mathbb{G}_1$ and π_h in $\mathbb{G}_1$. And the verifier has to check 5 pairing equations (Eq. 33, Eq. 34, Eq. 35, Eq. 36 and Eq. 37). Finally, if all these pairing equations are satisfied, then the proof is accepted.

2.5 The circom language

`circom` is a programming language and a compiler written in Rust for compiling circuits written in the circom language. The compiler outputs the representation of the circuit as constraints and everything needed to compute different ZK proofs.

The aim of this program is to provide a symbolic description of the circuit, which will be a list of R1CS constraints placed in a binary file, and to provide an efficient way to compute the witness from the inputs.

Circom allows the programmer to design arithmetic circtuis at low level. Moreover, all the constraints have to be explicitly added by the programmer, and, at compilation time, constraints can be simplified and signals can be removed at compilation time but it will not be possible to add new signals. A circom program has two representations: the R1CS constraints and the executable code (written in wasm or C++).

A key feature of circom is that it provides different instructions to work only at the symbolic level (defining new constraints):

```
1 out === in1*in2 //symbolic level
```

or to work at the computational level, defining how to compute a signal:

```
1 in1*in2 --> out; // computational level
```

Finally, we could also do both operations together:

```
1 out <== in1*in2; // symbolic and computational level
```

2.5.1 The circomlib library

With circom, it is possible to create large circuits by combining smaller generic circuits called templates. The circomlib is a library of circom templates that contains hundreds of circuits such as comparatives, hash functions, digital signatures, binary and decimal converters and many more. New circuits can always be created with circom.

The packages `circomlibjs`, `circomtester` and `ffjavascript` are the dependencies for circom circuits. circomlibjs is a Javascript library that provides programs to compute the witness of several circuits of circomlib. This library is used to check that the witness computed using the wasm or C++ code generated by circom for many circuits in the circomlib match the ones generated by the corresponding Javascript program in circomlibjs. Moreover, the circomtester is a npm package that provides tools for testing circom circtuits, and the ffjavascript is a npm package with Javascript code to perform finite field operations in Javascript.

2.5.2 Combination with snarkJS

SNARKJS is a npm package that contains code to generate and validate ZK proofs from the artifacts produced by circom.

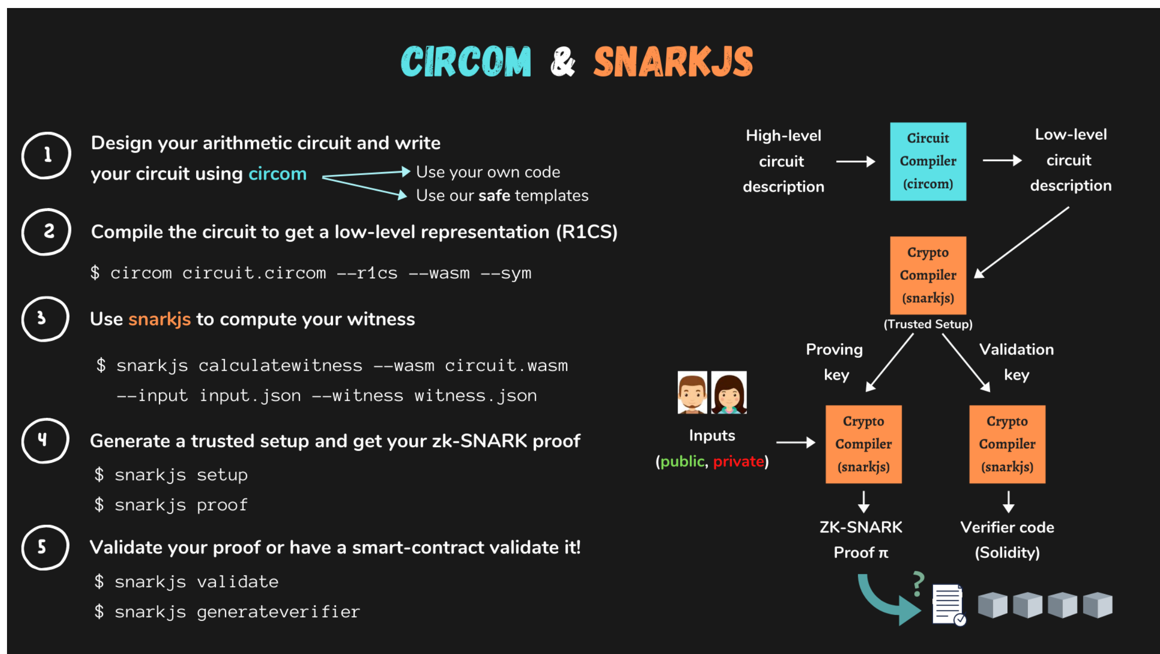


Figure 19: Visual summary of the combination of circom and SNARKJS[15].

3 Methodology

This section describes the circuits proposed to be evaluated using the `circom` language.

In order to write and compile circuits with circom, it is necessary to follow the guide of "installing the circom ecosystem" explained in [16], where it is described the list of dependences that we will need to compute and execute the circuits. The circom compiler is written in Rust, that is why we will install Rust on our system:

```
$curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

And now we also install circom cloning the repository:

```
git clone https://github.com/iden3/circom.git
```

Now we enter the circom directory and use cargo build to compile:

```
cargo build --release
```

However, for macOS we get the following error:

```
error: linking with `cc` failed: exit status: 1
```

In order to solve this error, we will execute:

```
$ sudo xcode-select --reset
$ cargo build --release
```

This will output another error:

```
failed to run custom build command for `wabt-sys v0.8.0`
```

To solve this error we will have to download the CMake application from its website [17] and place it on our Applications folder. Then, we will execute the following:

```
$ sudo mkdir -p /usr/local/bin
$ sudo /Applications/CMake.app/Contents/bin/cmake-gui --install=/usr/
    local/bin
$ cargo build --release
Finished release [optimized] target(s) in 4m 48s
```

Now we have finally achieved to compile it. This command generates the circom binary in the directory target/release. We can install this binary as follows:

```
$ cargo install --path circom
```

Now, we should be able to see all the options of the executable by using the help flag:

```
circom compiler 2.0.3
IDEN3
Compiler for the circom programming language

USAGE:
    circom [FLAGS] [OPTIONS] [input]

FLAGS:
        --verbose    Shows logs during compilation
    -h, --help       Prints help information
        --inspect    Does an additional check over the constraints
   produced
        --O0         No simplification is applied
    -c, --c          Compiles the circuit to c
        --json       outputs the constraints in json format
        --r1cs       outputs the constraints in r1cs format
        --sym        outputs witness in sym format
        --wasm       Compiles the circuit to wasm
        --wat        Compiles the circuit to wat
        --O1         Only applies var to var and var to constant
   simplification
    -V, --version    Prints version information

OPTIONS:
        --O2 <full_simplification>    Full constraint simplification [
   default: full]
    -o, --output <output>              Path to the directory where the
   output will be written [default: .]

ARGS:
    <input>    Path to a circuit with a main component [default: ./
   circuit.circom]
```

The tool that goes after circom is snarkJS, it is a npm package that contains code to generate and validate ZK proofs from the artifacts produced by circom. It can be installed with the following command:

```
$ npm install -g snarkjs
```

Now we will create a template of a basic circuit, a Multiplier, which will simply multiply two values. The code for this circuit is the following:

```
pragma circom 2.0.0;

template Multiplier2(){
    signal input a;
    signal input b;
    signal output c;
    c <== a*b;
}

component main = Multiplier2();
```

With the last line we will initiate of the circuit.

Now we will have to compile the circuit, generating all the equations related with it using the command `circom multiplier2.circom --r1cs --wasm --c --sym`, where `--r1cs` is to generate values for all the wires of the circuits, given some input values that the owner will introduce. Moreover, some files will be generated in the same directory: we will have `multiplier.r1cs`, which will contain all the equations of the circuit, `multiplier.sym`, that it will have some information about the circuit and two folders: `multiplier_cpp` and `multiplier_js`, whose files are written in C++ and wasm respectively and they are two different ways to compute the wires of our circuit.

Once we have compiled the circuit, we will locate in the folder `multiplier_js` and create a json file where we will write the values of the inputs for the circuit:

```
{"a":3, "b":11}
```

Finally, we will execute in terminal `node generate_witness.js multiplier2.wasm input.json witness.wtns` in order to generate the `witness.wtns`, which, together with the r1cs file, is what we need to create the proofs.

The next steps are the practical phases that the Pinocchio (or zk-SNARK) Protocol follows, which we have deeply analyzed in the previous sections 2.4 and 2.3.

We will create proofs, and we will do it with a proving system called groth16. In Groth16, there is a parsed trusted setup, meaning that each sequence has a different trusted setup. In order to create the phase one of the trusted setup, we will execute the following command that will give us randomness to initiate the circuit:

```
$ snarkjs powersoftau new bn128 12 pot12_0000.ptau -v
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: Blank Contribution Hash:
                786a02f7 42015903 c6c6fd85 2552d272
                912f4740 e1584761 8a86e217 f71f5419
```

```
10                 d25e1031 afee5853 13896444 934eb04b
11                 903a685b 1448b755 d56f701a fe9be2ce
12 [INFO]  snarkJS: First Contribution Hash:
13                 9e63a5f6 2b96538d aaed2372 481920d1
14                 a40b9195 9ea38ef9 f5f6a303 3b886516
15                 0710d067 c09d0961 5f928ea5 17bcdf49
16                 ad75abd2 c8340b40 0e3b18e9 68b4ffef
```

With the previous command you can build until 2^{12} different constraints. Now we need to do our trusted setup:

```
1  $ snarkjs powersoftau contribute pot12_0000.ptau pot12_0001.ptau --name=
     "First contribution" -v
2  $ Enter a random text. (Entropy): hagnfbsuwjel
3  [DEBUG] snarkJS: Calculating First Challenge Hash
4  [DEBUG] snarkJS: Calculate Initial Hash: tauG1
5  [DEBUG] snarkJS: Calculate Initial Hash: tauG2
6  [DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
7  [DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
8  [DEBUG] snarkJS: processing: tauG1: 0/8191
9  [DEBUG] snarkJS: processing: tauG2: 0/4096
10 [DEBUG] snarkJS: processing: alphaTauG1: 0/4096
11 [DEBUG] snarkJS: processing: betaTauG1: 0/4096
12 [DEBUG] snarkJS: processing: betaTauG2: 0/1
13 [INFO]  snarkJS: Contribution Response Hash imported:
14                 c6d34ceb f6c3e18e 261f7cb2 30e014f5
15                 d22d7ae9 6714698d ed19b25b 0c60f76d
16                 69089ff8 ec21e976 ce0bcd6e 6a1e5d5c
17                 c93599fd bfd38b2d 161e0724 afbb8e9c
18 [INFO]  snarkJS: Next Challenge Hash:
19                 c67d53ea c403c465 fc7a5678 d4cbc93c
20                 2d9ebf1e 56f32a4c 9467b502 a6687115
21                 660c2d14 a6509680 498ec3fe b15d77f2
22                 a92e6dc0 f1f54061 392d2a23 33db6398
```

Usually, the trusted setup is formed by a multi-party ceremony, with several people's contribution, which provides reliability to the process since if one of the contributors is honest, then we are sure that the process is honest. However, as this is a simply example, we will use only one participant. Now we can compile the circuit in order to compute all the constraints.

```
1  $ snarkjs powersoftau prepare phase2 pot12_0001.ptau pot12_final.ptau -v
```

This phase takes a little bit of time because it is computing the 12 equations of the circuit that we previously established. The next phase is specific for the circuit. In this case we will use the multiplier.r1cs file, which includes all the constraints that are forcing inputs and outputs to be related correctly.

```
1 $ snarkjs groth16 setup multiplier2.r1cs pot12_final.ptau
    multiplier_0000.zkey
```

This generates all the cryptography material for the prover and the verifier. Following, we need to generate the randomness of this second phase, which is specific for the circuit.

```
1 $ snarkjs zkey contribute multiplier_0000.zkey multiplier_0001.zkey --
    name="1st Contributor Name" -v
```

```
1 $ snarkjs zkey export verificationkey multiplier_0001.zkey
    verification_key.json
```

With this last command, we are storing the verification key, extracting the randomness that the verifier needs from this key. Now, we have to copy the `witness.wtns` file to the general folder (as it is only in the multiplier2_js folder) and then we can proceed generating the proof, and, in order to do so, the prover will do:

```
1 $ snarkjs groth16 prove multiplier_0001.zkey witness.wtns proof.json
    public.json
```

Finally, we only have to verify the proof.

```
1 $ snarkjs groth16 verify verification_key.json public.json proof.json
2 [INFO] snarkJS: OK!
```

Now we will go through a more complex example, and we will try to define the part of a voting protocol when it is checked that someone belongs to the voting census, also applying zero knowledge.

Following the scheme of image 20, we want to prove that someone belongs to a census to vote without revealing who is the person, and the Nullifier value determines that he/she cannot vote more than once. For this system, we will build a Merkle tree where the whole census will be, whose representation can be seen in image 21. The prover (the voter in this case) wants to prove to the system that he/she knows a secret key from one of the public keys of the voting list. The input of the merkle tree is a private key, with which we will compute the public key, and we will check that this public key belongs to the census. We also have to take into account that the public key is an intern variable of the system, so none will know who owns this key but we will be able to verify that someone owns the private key of a public key that belongs to the census.

Moreover, in the center of the circuit we will have to verify a merkle proof inside the merkle tree. To this end, prover will also provide the sibling hashes of his/her secret key.

These siblings will be defined later with the practical example. If the circuit can compute the root combining the public key of the voter and the input siblings in the merkle tree, then it will be verified that the voter belongs to the census.

Furthermore, if we do the hash of the private key with a string, we will have a concrete nullifier, which will always be the same for the same private key. However, this nullifier will not be related to the person who owns the private key.

Finally, when publishing the vote we have to generate the constant voting ID and say the census (list of public keys included in the census) and indicate the voting system that will be used.

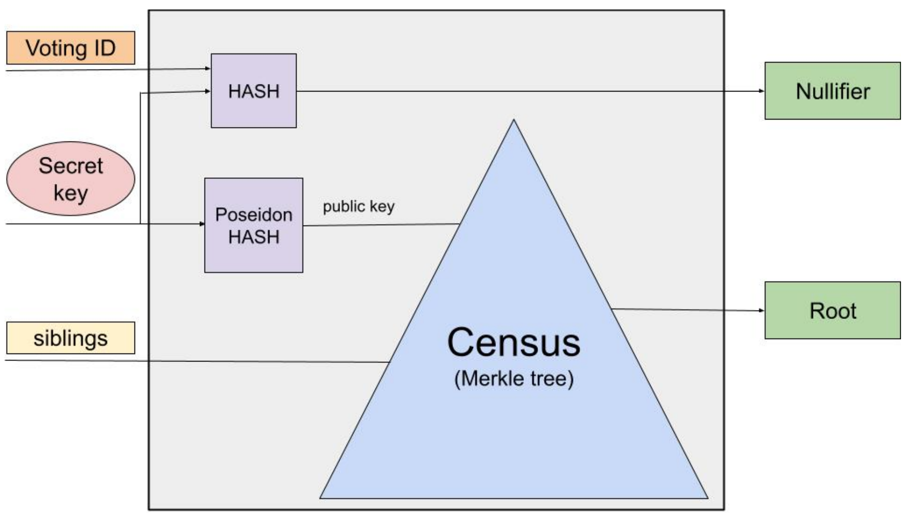


Figure 20: Block diagram of a verification voting system.

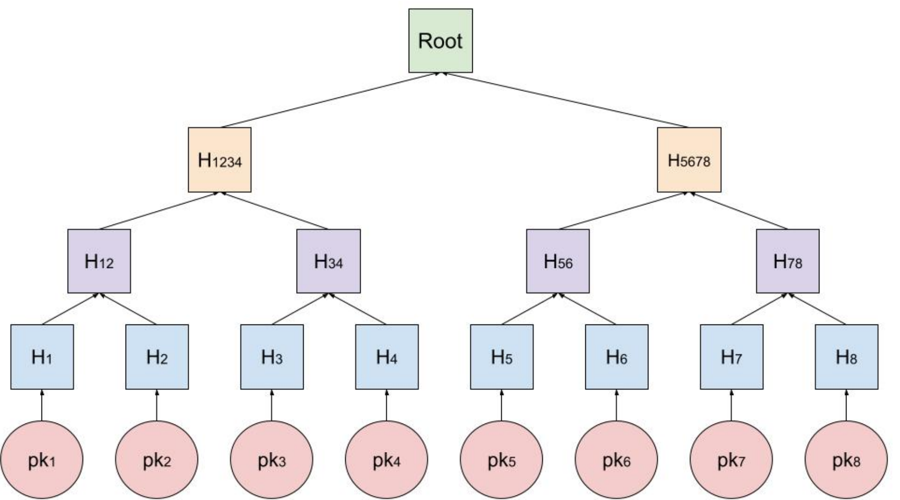


Figure 21: Representation of the Merkle tree used in for the program.

The first step to start the setup of a voting system is to program a Merkle tree in a circuit. Then, the whole circuit can be implemented. So now we will focus on building the

verification of a merkle proof.

3.1 Setup of our voting system

It has been created a directory named 'voting', where we will allocate our program. Then, we also initiate javascript as it will be used.

```
1 $ mkdir voting
2 $ cd voting
3 $ npm init
4 This utility will walk you through creating a package.json file.
5 It only covers the most common items, and tries to guess sensible
    defaults.
```

For the Merkle tree we will need JavaScript to generate the Merkle proofs and the circuit.

The structure of our folders will be the following one. In the JavaScript folder we will have our programs written in JavaScript, such as the Merkle tree. In the circuits folder we will place the circuits that we will use to prove the Zero Knowledge condition, and in the test one we will check our voting system.

```
1 > voting
2     > Javascript
3         > merkletree2.js
4     > circuits
5     > test
```

For the `merkletree2.js` program we will need a function that creates a Merkle tree using an array of private keys as the input and a function that computes a proof from a concrete point of the Merkle tree.

It can be used a library that automatically generates a Merkle tree, but for this work it has been created from the beginning.

First, we will create a function to merkleize, to which we will pass the field (the wires of our circuit which are elements in $\mathbf{G}_p$, they are provided in the `circomlibjs`), the hash function, an array of values and an optional number of levels, in our case we will include it. We will also give a hash to this function. In our example we will use the Poseidon hash, very used in Zero Knowledge as it creates less constraints than SHA-256 hash on the circuit. This function will return the tree.

The first element of the Merkle tree will always be the root (the level 0 of the tree). It will be followed by two elements of the first level, and then we will have 4 elements of the second level, 8 of the third one, etc. We will work with a binary tree of three levels. The representation of this tree can be seen in image 21, where the eight red values labeled as pk_x are the public keys that will be the input of the function. In case of having 6 public keys instead of 8, we will set an entry to 0, this way its hash will also be equal to 0.

The number of elements that we have is equal to $2^{nlevels}$. The function to generate the merkle tree and the one to generate the proof are recursive.

Notice that to run properly the code we will have to install the module chai and circomlibjs.

In the `merkletree2.js` file we will have the four functions explained above, whose code can be seen in 7.1.

We will use `mocha` to check the program, as we will also have a JavaScript file for testing. So we will first install mocha and then only typing mocha from our voting folder in terminal we will be able to check the system.

Now we can proceed to generate the circuit to prove the Merkle tree.

3.2 Circuit

The circuit will do a verification of the proof computed in the `merkletree2.js` file. It will check that this proof is included in the Merkle tree.

The validation of the circuit is oriented in such a way that for each level we will have a sibling and a value that comes from the level below, and according to a selector, either the two values remain the same or it switches it. Then we will hash the result. A scheme of this system is shown in image 22.

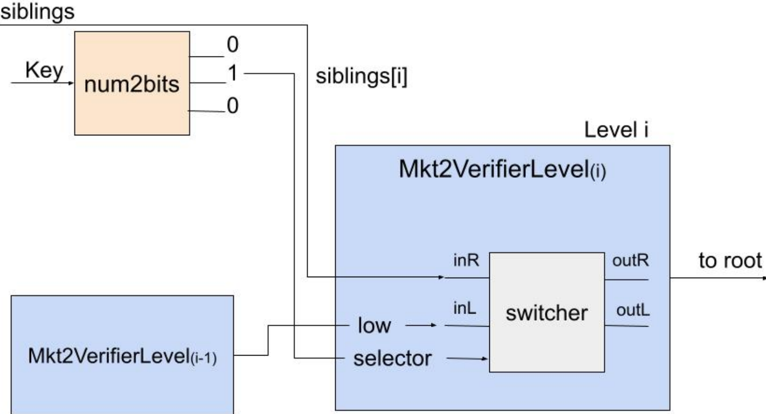


Figure 22: Scheme of the circuit system.

One of the inputs to the system will be the key, which will indicate the location in the merkle tree of the public key. We will use a number to bits circuit that includes a constraint that forces the output to be binary. Each number that will be converted to a binary number will have `nLevels` bits, in our case 3 bits. The `low` variable is the result of the hash of the previous level. Moreover, we will also take the siblings that correspond to this concrete level from the array that the prover inputs. Finally, we will have a circuit called switcher, whose inputs are the variable `low` (inL) and `siblings[2]` (inR), and

depending on the value of the selector, it will cross or not the values of the input and output leaves.

To verify a merkle tree, we will simply build a circuit and if we have three levels of the tree we will have 3 circuits. So this system applied at each level of the merkle tree will finally compute the hash of the two values that it receives and in the order that the selector determines and the circuit finally outputs this hash that we could call a "to root" value. If the final output of the tree is the `root` one we will be able to affirm that the prover is honest and, in our example, it will belong to the voting census.

In order to run the code of the circuit we will need the `circomlib` presented in section 2.5.1. To do so, we could directly copy the circuits in our directory or download the library as a node module (npm packet). Note that previously we also installed the `circomlibjs` library, but it is different from this one. The `circomlibjs` includes JavaScript functions, whereas `circomlib` are the circuits that compute a determinate value.

```
1 npm install -D circomlib
```

At this point, our `package.json` file should look like something like the following.

```
1  {
2    "name": "voting",
3    "version": "0.0.1",
4    "description": "Voting example",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [
10     "voting",
11     "zksnark"
12   ],
13   "author": "Silvia",
14   "license": "GPL-3.0",
15   "dependencies": {},
16   "devDependencies": {
17     "chai": "^4.3.4",
18     "circom_tester": "^0.0.7",
19     "circomlib": "^2.0.2",
20     "circomlibjs": "^0.1.1",
21     "mocha": "^9.1.4"
22   }
23 }
```

The resulting program of the circuit we are creating is the one described in 7.2. First, we define our level with the template `Mkt2VerifierLevel()` which must include a switcher, a hash function (Poseidon in this case), the Bitify circuit that we need in order to do the hash in the correct order and the inputs sibling, low and the selector. This circuit will validate one level. In order to extend it for all the levels that we have, we will instantiate

this circuit as many times as levels we have, this is described in the second part of the code of section 7.2, where we have the inputs `key`, `value` which is the secret key, the `root` of the last level and the array of siblings.

In the next section the entire system will be evaluated.

4 Evaluation

In this section we will test that the circuits that we built in the previous section are actually working properly. In order to check this, we will execute them firstly as a honest prover and later manipulating the public key of the prover. Only if the circuit fails at all the tests, we will be able to affirm that the verification of the circuit is working properly.

On the one hand, we have built a simple multiplier circuit, which we checked that was working properly if the prover was honest. If we modify the public.json file writing another value and execute again the verification of the circuit we will get a negative message.

```
1 $ snarkjs groth16 verify verification_key.json public.json proof.json
2 [ERROR] snarkJS: Invalid proof
```

And for other similar proofs the verification also fails, so we can affirm that the verification of the circuit is performing well.

On the other hand, we will evaluate the complex circuit that we have also built in the previous section, the voting system.

4.1 Testing the voting system

In the previous section we built a circuit that verifies a Merkle tree. In order to use the circuit that we have just created to test our Merkle tree, we will create an implementation of this circuit in our `test` folder. We will create a folder called `circuits` inside the test one and there we will have the file `merkletree2_tester.circom`, whose code is in 7.3.1. Then, we will also create the file `mkt2cir.js` in the folder test to check the Merkle tree using our `circom` circuit described in 7.3.2.

For testing circuits it exists the `circom_tester` library. It creates the file to test it and computes the witness. In this case we are testing with `wasm`, but we could only use C. With lines 12 and 13 of 7.3.2, we compile our circuit and generate its wires before doing the test. Then, we build an object which contains the inputs for the circuit, this is the reason why it includes the key, the value (in our case we will prove the value 33 as we did previously), the root and the siblings, and we check if the Merkle tree is valid computing the Witness.

In order to run the program we will first have to install the `circom_tester` library.

```
1 npm install -D circom_tester
```

Once we run the program and it has been executed properly, we can debug the program to check if it is actually running as expected. To verify it, we look at the constants `m`, `root` and `mp` (Merkle proof): The `m` values are the hashes of the tree arranged from the root to the leaves, and the value of the `root` must be the first value of the array `m`. Finally, `mp` will have a length of 3 in our case and checking the `isMerkleProofValid` function we

will be able to see if these 3 hashes are the siblings that end up at the hash value of our input. In our case we had the following values:

input (public key) = 33

$m = [1867\ldots; 9631\ldots; 8240\ldots; 5069\ldots; 5903\ldots; 1589\ldots; 1802\ldots; 1979\ldots; 1493\ldots;$

$6089\ldots; 9442\ldots; 1546\ldots; 5060\ldots; 1615\ldots, 1463\ldots]$

root = 1867...

$mp = [8240\ldots; 5069\ldots; 9442\ldots]$

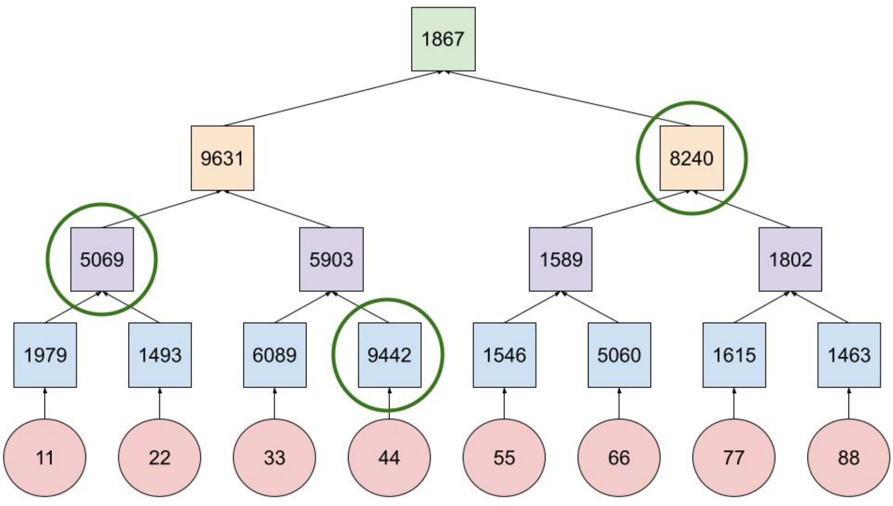


Figure 23: Representation of the Merkle with the values of our example.

However, for checking the code of our circom circuits we can not debug the program the same way we did before because for the circom circuits works different. Now we will add a `log` in the `Mkt2VerifierLevel()` function to print the values of the left and right leaf of the output of the switch for each level.

```
template Mkt2VerifierLevel(){
    ...
    sw.sel <== selector;
    sw.L <== low;
    sw.R <== sibling;

    log(sw.outL);
    log(sw.outR);

    root <== hash.out;
}
```

In this case, values that we got are the following ones:

output= $[6089\ldots; 9442\ldots; 5069\ldots; 5903\ldots; 9631\ldots; 8240\ldots]$

We can check from our particular example scheme of image 23 that these outputs are the ones that correspond to the siblings of the branch that follows our input (33) up to the root. So we could conclude that the system is working, although it will be checked with other tests in the next section.

We will now check that if we introduce a combination of a key and public key that do not correspond to anyone of the merkle key or an array of siblings that are not well computed, the circuit will fail.

Changing one of the inputs, i.e. the `value` variable,

```
const input={
            key: F.e(2),
            value: F.e(44),
            root: root,
            siblings: mp
        };
```

the circuit fails, as the key and the merkle proof do not correspond to the values for the public key 44, so the resulting root will not be the one of the circuit and the verification will fail. We will get the following output:

```
Check Merkle tree Circuit
9442105919245249606993672523850253578n
9442105919245249606993672523850253578n
5069520895576606111431696133718904457n
2145073276306676930687321696419937691n
1729408233379404703901703525806586523n
8240055649093052352355674146556731727n
    1) Should check inclussion in MT

  Merkle tree test
        It should create a 3 level merkle tree, generate a mp and
    validate it


  1 passing (2s)
  1 failing

  1) Check Merkle tree Circuit
        Should check inclussion in MT:
      Error: Error: Assert Failed. Error in template Mkt2Verifier_137
    line: 56
      at /Users/silvia/Downloads/proves_voting2/node_modules/
    circom_tester/wasm/witness_calculator.js:117:27
      at Array.forEach (<anonymous>)
      at WitnessCalculator._doCalculateWitness (node_modules/
    circom_tester/wasm/witness_calculator.js:103:14)
      at WitnessCalculator.calculateWitness (node_modules/circom_tester/
    wasm/witness_calculator.js:128:20)
```

```
24        at WasmTester.calculateWitness (node_modules/circom_tester/wasm/
      tester.js:75:45)
25        at Context.<anonymous> (test/mkt2cir.js:28:23)
26        at processImmediate (internal/timers.js:464:21)
```

Looking at the merkle tree of our example in image 23, we can easily check that the values printed in terminal of the branch of the third leave (whose key value is 2) do not have its correct value. So if the prover does not know one of the variables needed for the circuit, the merkle proof will fail and the system in this case will not let the prover vote.

5 Budget

This section presents the estimated budget needed in order to carry out the project.

First, it will be needed the hard work of a computer engineer who would spend around 300 working hours in the project, and he/she would get paid. Moreover, we have to take into account the cost of the material resources, which in this case is a computer working in a macOS platform. As it is not needed a high computer processing unit, it has been chosen the minimum price in the Apple market for a computer. Finally, office expeses such as electricity and water. On the other hand, as all the software used is open source, we would not have software expenses. With all this information, table 6 is defined showing the general cost of the project.

Concept	**Hours**	**€/h**	**Cost (€)**
Computer Engineer	360	15	5400
Computer macOS			1129
Office expenses			350
Total (€)			**6879**

Table 6: Estimated budget for the project.

6 Conclusions and future development

In this thesis we have studied and tested how we can achieve user's privacy and anonymity in a blockchain using protocols based on the Zero Knowledge Proof. To conclude the work, on the one hand we have described in detail the Pinocchio protocol, and, on the other hand, we have introduced the `circom` language which allows to program circuits that will be used to verify some information from an anonymous source (the prover).

At the beginning of the project we have explained the zero knowledge proof protocol, as the aim of the project was to be able to implement a system that applies this technique. Then, the discrete logarithm problem and the zk-SNARK method have been described in detail, in order to understand properly all the concepts that appear later in the Pinocchio protocol explanation. Finally, we have introduced the `circom` language and its library and how it can be combined with snarkJS.

In the third section we have explained deeply how to use `circom` and we have built a simple circuit that verified a correct proof applying zero knowledge. Then, the structure of a more complex system that can also be built using `circom` circuits has been presented. This system, which pretends to check a census of a voting system, has been evaluated in section four, where we could check that the circuit verifies the proof only if the prover inputs to the circuit the correct values, otherwise it would mean that he/she does not own a secrete key of the merkle tree.

The system presented in the third section and tested in the fourth is a practical example of applying the Pinocchio protocol. One of the objectives of this thesis was to describe cautiously this protocol, as well as the `circom` implementation. An example of the combination of them is the resulting circuit built in section 3.

Nowadays the `circom` language is continuously changing and improving. The next change will be related to the compilation of `circom`, the intention is to extend the compilation not only for the Rust compiler. To this end, the field of study now is to use the compiler plong, which uses a different system of equations when building the elements and wires of the circuits.

References

[1] Wolfgang Skala. Drawing gantt charts in latex with tikz.

[2] Donald Knuth. Knuth: Computers and typesetting.

[3] Lance Fortnow. Shafi goldwasser, silvio micali, and charles rackoff. the knowledge complexity of interactive proof systems. siam journal on computing, vol. 18 (1989), pp. 186–208. - oded goldreich, silvio micali, and avi wigderson. proofs that release minimum knowledge. mathematical foundations of computer science 1986, proceedings of the 12th symposium, bratislava, czechoslovakia, august 25–29, 1986, edited by j. gruska, b. rovan, and j. wiedermann, lecture notes in computer science, vol. 233, springer-verlag, berlin, heidelberg, new york, etc., 1986, pp. 639–650. - oded goldreich. randomness, interactive proofs, and zero-knowledge—a survey. the universal turing machine, a half-century survey, edited by rolf herken, kammerer unverzagt, hamburg and berlin, and oxford university press, oxford and new york, 1988, pp. 377–405. *Journal of Symbolic Logic*, 56(3):1092–1094, 1991.

[4] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to explain zero-knowledge protocols to your children. In *Conference on the Theory and Application of Cryptology*, pages 628–631. Springer, 1989.

[5] Helmut Hasse. Zur theorie der abstrakten elliptischen funktionenkörper iii. die struktur des meromorphismenrings. die riemannsche vermutung. 1936.

[6] André Weil. Numbers of solutions of equations in finite fields. *Bulletin of the American Mathematical Society*, 55(5):497–508, 1949.

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

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. Cryptology ePrint Archive, Report 2014/349, 2014. `https://ia.cr/2014/349`.

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive, Report 2011/443, 2011. `https://ia.cr/2011/443`.

[10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 321–340. Springer, 2010.

[11] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. Cryptology ePrint Archive, Report 2012/215, 2012. `https://ia.cr/2012/215`.

[12] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Report 2013/279, 2013. https://ia.cr/2013/279.

[13] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *Annual International Cryptology Conference*, pages 445–456. Springer, 1991.

[14] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645. Springer, 2013.

[15] circom circuit compiler. https://docs.circom.io/.

[16] Installing the circom ecosystem. https://docs.circom.io/getting-started/installation/#important-deprecation-note.

[17] Installing the cmake application. https://cmake.org/download/.

7 Appendices

7.1 Code for the merkletree2.js file

```
function merkleize(F, hash, arr, nlevels) {
    const extendedLen = 1 << nlevels;

    const nArr = [];
    for (let i=0; i<extendedLen; i++){ //I go throughout all the levels and if there is an element I compute its hash and I push it into the array of hashes and if there is not I just put a zero on it.
        if(i<arr.length){
            nArr.push(hash([F.e(arr[i])]));
        } else {
            nArr.push(F.zero);
        }
    }

    return __merkleize(hash, nArr);
}
```

```
function __merkleize(hash, arr){
    if(arr.length == 1) return arr;

    const nArr = [];
    for(i=0; i<arr.length/2; i++){
        nArr.push(hash([arr[2*i], arr[2*i+1]]));
    }

    const n = __merkleize(hash, nArr);

    return[...n, ...arr];
}
```

```
function getMerkleProof(n, key, nlevels){ //we pass the generated Merkle tree and we generate the proof
    if(nlevels == 0) return [];
    const extendedLen = 1 << nlevels;

    topSiblings = getMerkleProof(n, key >> 1, nlevels -1);
    curSibling = n[extendedLen - 1 + (key^1)];
    return[...topSiblings, curSibling];
}
```

```
function isMerkleProofValid(F, hash, key, value, root, np){
    let h =hash([value]);
    for(let i = np.length-1; i>=0; i--){
        if((1 << (np.length -1 - i)) & key) {
            h = hash([np[i], h]);
```

```
        }else {
            h = hash([h, np[i]]);
        }
    }
    return F.eq(root, h);
}
```

In order to be able to use these functions for running other files we include these lines in the file:

```
module.exports.isMerkleProofValid = isMerkleProofValid;
module.exports.getMerkleProof = getMerkleProof;
module.exports.merkleize = merkleize;
```

7.2 Code for the verification circuit (circuits/mkt2.circom file)

```
pragma circom 2.0.0;

include "../node_modules/circomlib/circuits/switcher.circom";
include "../node_modules/circomlib/circuits/poseidon.circom";
include "../node_modules/circomlib/circuits/bitify.circom";

template Mkt2VerifierLevel(){
    signal input sibling;
    signal input low;
    signal input selector;
    signal output root;

    component sw = Switcher();
    component hash = Poseidon(2);

    sw.sel <== selector;
    sw.L <== low;
    sw.R <== sibling;

    root <== hash.out;
}
```

```
template Mkt2Verifier(nLevels){
    signal input key;
    signal input value;
    signal input root;
    signal input siblings[nLevels];

    component n2b = Num2Bits(nLevels); //imported with bitify.circom
    component levels[nLevels] = Poseidon(2);

    component hashV = Poseidon(1);
```

```
11
12      hashV.inputs[0] <== value;
13
14      n2b.in <== key;
15
16      for (var i=nLevels-1; i>=0; i--){
17          levels[i] = Mkt2VerifierLevel();
18          levels[i].sibling <== siblings[i];
19          levels[i].selector <== n2b.out[i];
20          if (i==nLevels-1){
21              levels[i].low <== hashV.out;
22          } else {
23              levels[i].low <== levels[i+1].root;
24          }
25      }
26
27      root === levels[0].root;
28  }
```

7.3 Testing files

7.3.1 merkletree2_tester.circom file

```
1 pragma circom 2.0.0;
2 include "../../circuits/mkt2.circom;
3
4 component main {public [root]}= Mkt2Verifier(3);
```

7.3.2 mkt2cir.js file

```
1  const path = require("path");
2  const wasm_tester = require("circom_tester").wasm;
3  const hash = require("circomlibjs").poseidon;
4  const {merjleize, getMerkleProof, merkleize} = require("../Javascript/
       merkletree2.js");
5  const F = require("circomlibjs").babyjub.F;
6
7  describe("Check Merkle tree Circuit", function () {
8      let circuit;
9
10     this.timeout(1000000);
11
12     before( async() => {
13         circuit = await wasm_tester(path.join(__dirname, "circuits", "
       merkletree2_tester.circom"));
14     });
15
16     it("Should check inclussion in MT", async () => {
17         const m = merkleize(F, hash, [11, 22, 33, 44, 55, 66, 77, 88],
       3);
```

```
18          const root = m[0];
19          const mp = getMerkleProof(m, 2, 3);
20
21          const input = {
22              key: F.e(2),
23              value: F.e(33),
24              root: root,
25              siblings: mp
26          };
27
28          await circuit.calculateWitness(input, true);
29      });
30  });
```