Zero Knowledge Proofs

Facultad de Ciencias
Universidad Zaragoza

Universidad Zaragoza
1542

Roberto García Alvira

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: Miguel Ángel Marco Buzunárriz
14 de junio de 2023

Resumen

Este trabajo de fin de grado se va a centrar en el análisis del algoritmo de demostración en conocimiento cero, **Groth16**. Este algoritmo fue creado por el británico Jens Groth en 2016, de donde proviene el nombre del protocolo. En sentido informal, estas pruebas se basan en la idea de que un probador consiga convencer a un verificador de que posee un cierto secreto, sin desvelar ese mismo secreto.

Como introducción, se verá como surgió la idea de las pruebas de conocimiento cero, basadas en la idea de comunicaciones seguras en las que se enviara la menor cantidad de información posible, para mantener dicha información en secreto. La primera semilla de esta idea surgió en 1985, cuando Goldwasser, Micali y Rackoff publicaron un primer paper [6] en el que se asentaban las bases de la criptografía de conocimiento cero. A partir de entonces, la comunidad fue trabajando en el desarrollo de algoritmos funcionales que se pudieran basar en estas propiedades. Finalmente, en 2013, el algoritmo Pinocchio [8] fue creado por Parno, Howell, Gentry y Raykova, y asentó las bases de la criptografía de conocimiento cero moderna. Este fue el primer algoritmo funcional y completo, ya que permite resolver cualquier tipo de computación, algo que los anteriores algoritmos no podían hacer al presentar ciertas restricciones en sus esquemas de resolución. Tras esto, la comunidad buscó refinar y mejorar este tipo de protocolos, dando lugar al algoritmo Groth16 [5], en el que se centra este estudio.

Posteriormente, comenzará el desarrollo matemático del esquema. El primer paso será la explicación del problema del logaritmo discreto, muy frecuente en la criptografía, y sobre el que esta basado Groth16. Más concretamente, está basado en el problema del logaritmo discreto sobre curvas elípticas, sobre las que se introducirá brevemente su definición y la operación interna que tienen, la cual proporciona a la curva elíptica una estructura de grupo muy útil en la construccion del algoritmo. También, se procederá a introducir diversas herramientas más, como las aplicaciones de hiding, que son aplicaciones que nos ayudarán a esconder información dentro de la curva elíptica, y pairing, que servirá para relacionar los grupos de diferentes curvas y hacer la verificación final del algoritmo. Además, se explicará el esquema de relaciones entre grupos que tiene el algoritmo, usando los grupos de cada curva elíptica y las aplicaciones nombradas anteriormente.

A continuación, se realizará la definición formal del algoritmo, empezando por la construcción de los llamados ZK-SNARKS (Zero Knowledge Succint Argument of Knowledge), los cuales son el modelo de protocolo que sigue el algoritmo Groth16. Antes de comenzar con dicha construcción, se definirán los conceptos de enunciado y testigo, que forman una pareja en la que se basan estas pruebas. El enunciado es básicamente el probema o computación que se quiere demostrar, mientras que el testigo es el secreto o clave que permite demostrar dicho enunciado y que solo conoce el probador. Los ZK-SNARKS tienen una estructura fija, que se basa en la cuaterna de algoritmos (Setup,Prove,Vfy,Sim), aplicada a una única pareja de enunciado y testigo. Informalmente, la función de cada uno de estos algoritmos es la siguiente: Setup generará elementos específicos a partir del enunciado y testigo; Prove usará los elementos generados por el Setup para generar una prueba y Vfy comprobará si la prueba generada es correcta o no. Finalmente, Sim (o simulador), es un algoritmo que no es usado en la práctica, sin embargo, su existencia es necesaria para argumentar sobre ciertas propiedades de seguridad del protocolo.

Tras la explicación de las distintas partes del algoritmo, se definirán formalmente los conceptos que todo algoritmo de conocimiento cero debe cumplir. Estos son:

1. Completitud: dada una pareja correcta de enunciado y testigo, el probador simepre debe ser capaz de convencer al verficiador.
2. Solidez: un probador no puede convencer al verificador de un enunciado falso.
3. Conocimiento Cero: al realizar la prueba, la única información revelada es la veracidad del enunciado. Por tanto, el testigo del probador no es revelado.

Con estas definiciones, realizadas formalmente, queda construida la noción de ZK-SNARK en general.

Tras todas las definiciones del apartado anterior, se procede a la explicación de los QAP's (Programa Cuadrático Aritmético). Los QAP's son una forma muy específica de escribir una computación, la cual es clave para la forma en la que el algoritmo encripta la información. Esta consiste en expresar la computación que se desea resolver como un sistema de ecuaciones polinómicas. En dichas ecuaciones aparecerán el enunciado y el testigo como incógnitas del sistema. Como lo que obtenemos son ecuaciones lineales, el sistema se puede reescribir en forma matricial. A partir de las matrices anteriores, se calcularán los polinomios interpoladores de Lagrange de sus columnas, los cuales serán usados en el proceso de encriptación del algoritmo, más concretamente en el Setup y en el Prove. Estos polinomios interpoladores, junto con el polinomio que anula a todos los nodos de los polinomios interpolantes, serán los elementos que proporciona el QAP y que son necesarios para comenzar finalmente el algoritmo de encriptación.

El último apartado teórico explica las computaciones para implementar el algoritmo Groth16. Cuando se tiene la computación escrita en forma de QAP, este apartado indica paso a paso las operaciones que se tienen que realizar en cada uno de los subalgoritmos (Setup,Prove,Vfy,Sim). Con esto, se finaliza la contrucción teórica del protocolo.

Por último, para ilustrar la descripción de este algoritmo, se ha realizado una sencilla implementación en Sagemath. Esta implementación resuelve el siguiente problema: dada la factorización en primos, $n = p \cdot q$, el probador quiere demostrar que sabe los dos factores primos p, q del número n, sin desvelar cuales son dichos dos números. Por tanto, el enunciado será la ecuación de la factorización, y el testigo, los valores de p, q. En este caso, la ecuación que usaremos será $143 = 11 \cdot 13$, una expresión simple ya que lo que se busca con este ejemplo es ilustrar de forma sencilla el algoritmo, y de esta forma, obtendremos computaciones menos complejas, que ayudarán a seguir mejor el proceso.

Comenzaremos definiendo las curvas elípticas sobre las que se trabajará, junto con sus cuerpos finitos, generadores y aplicaciones de hiding y pairing. Una vez tengamos la estructura de estos grupos formada, realizaremos el QAP. La ecuación de factorización se reescribirá usando los factores en binario, para que se pueda representar como sistema de ecuaciones. Posteriormente, seguiremos los pasos explicados anteriormente en el QAP para obtener los polinomios interpolantes. A continuación, comenzaremos con la parte principal del algoritmo, donde se verán detalladamente los procesos que se siguen en cada uno de los subalgoritmos (Setup,Prove,Vfy,Sim). Finalmente, el ejemplo acaba con el Vfy mostrando una correcta implementación del algoritmo y efectivamente comprobando que el probador ha conseguido demostrar que sabe los factores p, q, sin desvelar información sobre ellos.

Contents

Chapter 1

Introduction

The main focus of this thesis is going to be the analysis of the Zero-Knowledge Proof protocol, **Groth16**. This protocol was created by the British Jens Groth in 2016, where its name comes from. Informally explained, this kind of proofs follow the next idea: A prover tries to convince a verifier that he knows a certain secret, without revealing any information about that secret.

The roots of Zero Knowledge methods date back to 1985, with the publication of the paper [6] by Goldwasser, Micali and Rackoff. In this paper, we can already find definitions and examples of use of concepts, such as interactive proofs and Zero Knowledge, be defined and used. During the 90's and 2000's, there were theoretical advances in the field of zero knowledge, but it looked like no real application was found. This changed when, in 2013, Parno, Howell, Gentry and Raykova, published the paper [8] creating an algorithm named Pinocchio. **Pinocchio** is defined as a built system for efficiently verifying general computations while relying only in cryptographic assumptions. In this paper, there are all the steps detailed to build the specific algorithm, that its authors refer to as "nearly practical", because they were unsure of it being efficient enough to surpass other algorithms. Finally, at the end of 2013, Ben-Sasson, Chiesa, Tromer and Virza published [1], where they showed that an implementation of Pinocchio was actually useful and complete, presenting the full protocol on its appendix.

Therefore, Pinocchio was the first implementation of a Zero-Knowledge proving system that was complete and universal for all kinds of computations. However, since the Pinocchio protocol appeared, cryptographers have tried to improve and refine zero knowledge algorithms. One of those more efficient protocols is **Groth16**. This protocol significantly improved the performance given by Pinocchio, as its proof consists of three elliptic curve points, contrary to the eight needed by Pinochio, and the verification is faster with just three pairings operations instead of twelve.

Mathematically, Groth16 and other cryptographic algorithms are based on the discrete logarithm problem, which will be explained later. Most specifically, Groth16 is based on the discrete logarithm problem over elliptic curves, whose inner operation and group structure will hold an important place in the construction of the algorithm. Other tools that will be used are hiding maps that will help to hide the information we want inside the elliptic curve groups, and pairing maps, that will relate different elliptic curve groups. The relation between these concepts will be explained, giving a whole group diagram of this construction.

In general, Zero-Knowledge algorithms are based on the pair statement-witness. The statement is the problem or computation we want to solve, whereas the witness is the secret solution the prover knows to that specific problem.

When building these type of algorithms, there are three properties every zero knowledge proof must satisfy:

1. **Completeness**: given a pair of statement and witness, the prover can always convince the verifier.
2. **Soundness**: a prover can not convince the verifier of a false statement.
3. **Zero-knowledge**: when proving, the only information revealed is the truth of the given statement. Thus, the prover's witness is not revealed.

Zero knowledge systems are implemented in practice by what is know as **ZK-SNARK**: Zero Knowledge Succint Non-interactive Argument of Knowledge. This type of protocols satisfy the three properties mentioned as well as generating proofs that short and easy to compute. This property is regarded as being a **succint** proof.

Groth16 follows a ZK-SNARK schema satisfying all these properties mentioned above. This schema is composed by the quadruple of algorithms (Setup, Prove, Vfy, Sim), that take as inputs the statement and witness, and generate a valid proof. Informally, the role of every algorithm is:
For the Setup, generate some specific elements given the statement and witness. The Prove algorithms, uses these elements generated by the Setup to construct a proof. Vfy will check if the generated proof is valid or not. Finally, the Sim (or simulator) is not used in practice, but its existence is important to argue about the properties we just mentioned.

An important procedure to introduce are QAP's (Quadratic Arithmetic Programs). QAP's are an specific form of writing computation, that is key in the construction of the protocol. It consists on expressing the computation we want to solve as a system of polynomial equations. These equations will contain the statement and the witness as its variables. We can express this system as matrices, and we would want to compute the Lagrange interpolating polynomials of the columns of this matrices, in order to use them in the protocol. These Lagrange interpolators, as well as, the polynomial that vanishes in all of the nodes of the interpolators, will be the elements the QAP generates.

After showing how the QAP works, we will explain the specific computations for each of the (Setup, Prove, Vfy, Sim) algorithms, in order to implement the Groth16 protocol. We will also proof that this specific quadruple satisfies Completeness, Zero-knowledge and Soundness.

Finally, we will illustrate the implementation of the protocol with a simple example in Sagemath. In this case we will focus on the next problem:
Suppose that you know two numbers $p,q \in \mathbb{N}$, that yield a factorization $n = p \cdot q$. Then, you want to prove that you know such a factorization, without revealing any information about p,q.
The equation $m = p \cdot q$ will be the statement and the witness will be the values of p,q. We have chosen the simple factorization, $143 = 11 \cdot 13$, in order to clarify the process and not make computations over complicated.
We will start by defining the elliptic curve group structure, with the hiding and pairing maps. Then, we will show how to rewrite this computation as a QAP, using binary. We will obtain the previously mentioned Lagrange interpolators for the QAP. Then we will start with the main part of the protocol, where we will show step by step, how each of the (Setup, Prove, Vfy, Sim) algorithms work. Finally, in the Vfy algorithm, we will prove to the verifier that we know the value of p,q without revealing any information about them, making the algorithm work as intended.

From now on, this thesis will focus on explaining how the Groth16 algorithm can be built and implemented.

Chapter 2

Theoretical introduction

To correctly explain how the Groth16 algorithm works, we need to introduce some mathematical matters, such as elliptic curves and pairing maps. These are only tools we will use and not the main focus of this thesis, we will only give a brief discussion of what is needed from them in the algorithm.

2.1 Discrete logarithm problem and elliptic curves

Public key cryptography algorithms are built over the resolution of a specific type of problem, which is easy to solve knowing a certain secret, but impossible in practice, if this secret is not known. One of these problems is called the discrete logarithm problem, which is the foundation of our algorithm. This problem consists in solving for x in the following equation:

$$a^x = b$$

where a and b are known values in a fixed multiplicative group. At first glance, solving this equation might look easy, but it is not in certain situations. Groth16 is based on this problem applied to finite fields and elliptic curves.

We are now going to define the concept of elliptic curve and some results needed for this construction. Let us consider the following equation, in a field $\mathbb{F}$:

$$y^2 = x^3 + ax + b$$

Where the field $\mathbb{F}$ has characteristic different from 2 and 3, and $a, b \in \mathbb{F}$. a, b are fixed values that satisfy $4a^3 + 27b^2 \neq 0$.
An **elliptic curve** is formed by the set of solutions of the previous equation, and an extra point $\tilde{0}$, that will be called the point of infinity. We will denote this set as E.

Now we will show how an elliptic curve has a group structure, first defining its inner operation, where the point $\tilde{0}$ will be the identity element:

Let $P, Q \in E$ be points of the curve and take the line L that passes trough them. This line will intersect the curve in another point that we will call $-R \in E$. Now, take the symmetric point of $-R$ respect to the x-axis, that will be denoted as its opposite R. Finally, the point $R := P + Q$, which gives the inner sum operation of two points we wanted.
We have the following special cases:
If we have $P = Q$ then the line L will be the tangent to the curve in that point and $-R = P$ will be the second point of intersection. Then, the final operation will be, $P + P = R$, or equivalently, $2P = R$.
We must also consider the case of a line being tangent to a point P and not intersecting into any other point. In this case $P = Q = -R$, thus, we have $P + P = -P$, or equivalently, $3P = \tilde{0}$.

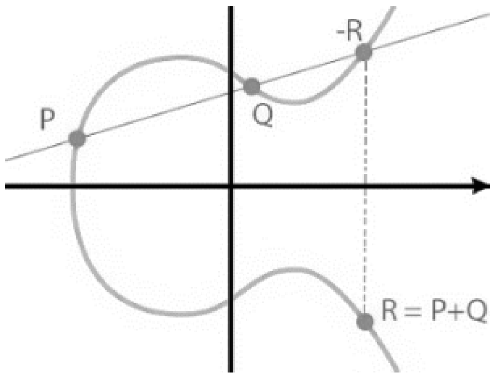

Figure 2.1: Sum of two points of a generic elliptic curve

With this definition of sum operation, it can be proven that the points of the curve form an abelian group. There also exist a Theorem that gives a boundary to the number of elements this group has. This is **Hasse's Theorem**:

Theorem 1. *Let $K = \mathbb{F}_p$ a finite field of p elements and E the set of points of an elliptic curve defined over $\mathbb{F}_p$. Then:*

$$|\#E - p - 1| \leq 2\sqrt{p}$$

Based on this result, we can deduce that the number of points in the elliptic curve, is similar to the size p of the field it is defined in.

Now, we can proceed to explain how the discrete algorithm problem works in elliptic curves:

Take a large prime number p and define an elliptic curve C that lies in the field $\mathbb{F}_p$. Now, let $A, B \in E$ be two points of the curve, such that they satisfy the equation $B = kA$ for some $k \in \mathbb{F}_p$.

In this case, the discrete logarithm problem consists on finding k. The point A is called the generator and it is fixed. B is known as the hiding of k, as we will define later, and k is the private key. This problem gets harder as the size p of the finite field grows.

2.2 Hidings, Pairings and Bilinear Groups

Let $\mathbb{G}$ be an additive group and g a non-trivial element from this group. **Hidings** are maps that send elements from a finite field, which will be $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ in our case, to elements of the group $\mathbb{G}$, which will be points of an elliptic curve:

$$\begin{array}{rcl} \mathbb{Z}/p\mathbb{Z} & \xrightarrow{h} & \mathbb{G} \\ n & \longmapsto & ng \end{array}$$

This map is used to hide our information safely into the elliptic curve thanks to the discrete logarithm problem, as explained before. In particular, inverting this map is the same as trying to solve the discrete logarithm problem. We use this map because finding its image is efficient computationally, as it takes the order of $2\log_2(n)$ operations to find the hiding of n. This is thanks to the elliptic curve sum operation and the successive duplicates method, where we can take a point a find all its powers of two efficiently, following the first special case noted before in the definition of the sum operation. With this powers of two, it is possible to find every multiple of a point making the correct additions. However, to find the inverse of this map, we might need to check the operation for all the elements in the group, so for a large number of elements in this group is not efficient.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two additive groups and $\mathbb{G}_T$ a multiplicative group. **Pairings** are maps that send a pair of elements, one from each of the first two groups, into an element of the third group:

$$\begin{array}{rcl} \mathbb{G}_1 \times \mathbb{G}_2 & \xrightarrow{e} & \mathbb{G}_T \\ (a,b) & \longmapsto & e(a,b) \end{array}$$

A pairing is bilinear if it satisfies:

$$e(P+Q,R) = e(P,R)e(Q,R) \quad P,Q \in \mathbb{G}_1, R \in \mathbb{G}_2 \quad and \quad e(P,Q+R) = e(P,Q)e(P,R) \quad P \in \mathbb{G}_1, Q,R \in \mathbb{G}_2$$

Also we say a pairing is non-degenerate when:

$$\text{If } \tilde{0} \neq P \in \mathbb{G}_1 \text{ then, there exists } Q \in \mathbb{G}_2 \text{ such that } e(P,Q) \neq 1$$

Note that, in the case of elliptic curves, there exist various kinds of pairings such as Weil, Tate or Ate pairing, that have different properties. In this case, we will use **Weil's Pairing**. For this pairing, the group $\mathbb{G}_T$ is the group of q-th roots of unity in $\overline{\mathbb{F}}_p$.

Finally, we can describe the full schema of operations in our algorithm:

- We will work with three abelian groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order q, whose generators are g, h, and $e(g,h)$ respectively.
- Pairing map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is bilinear.
- We will want to work with groups $\mathbb{G}_1, \mathbb{G}_2$ such that an efficient computable homomorphism does not exist between the two groups.
- There are efficient algorithms for computing group operations, evaluating the pairing map, deciding membership of the groups, deciding equality of group elements and computing group generators. These will be called generic group operations.

Thus we will have the following diagram:

$$\begin{tikzcd} \mathbb{Z}/p\mathbb{Z} \arrow[d, "h_1"] & & \mathbb{Z}/p\mathbb{Z} \arrow[d, "h_2"] & \mathbb{Z}/p\mathbb{Z} \arrow[d, "h_T"] \\ \mathbb{G}_1 & \times & \mathbb{G}_2 \arrow[r, "e"] & \mathbb{G}_T \end{tikzcd}$$

Figure 2.2: Diagram of the group pairing structure

If we choose generators $g, h, e(g,h)$ of each respective group, the hidings and the pairing map satisfy the following relation:

$$e(h_1(a), h_2(b)) = h_T(ab) \qquad a,b \in \mathbb{Z}/p\mathbb{Z}$$

Most notably, this relation between maps allows us to transform sums in the first two groups, into multiplications in the third one, as this one is a multiplicative group. For example, one type of computation we would want to solve later on is given by the statement:

$$(a_1 + \cdots + a_n), (b_1 + \cdots + b_m), (c_1 + \cdots + c_l) \in \mathbb{Z}/p\mathbb{Z}$$

$$s.t. \qquad (a_1 + \cdots + a_n)(b_1 + \cdots + b_m) = (c_1 + \cdots + c_l)$$

Where if we use the relation given above, the hiding structure is the following one:

$$e(h_1(a_1) + \cdots + h_1(a_n), h_2(b_1) + \cdots + h_2(b_m)) = e(h_1(c_1) + \cdots + h_1(c_l), h_2(1))$$

We will denote each group element by its discrete logarithm, given by its respective hiding map. Thus we will write, $[a]_1$ for $h_1(a)$, $[b]_2$ for $h_2(b)$ and $[c]_T$ for $h_T(c)$. The generators are $g = [1]_1$, $h = [1]_2$, $e(g,h) = [1]_T$ and the neutral elements are $[0]_1, [0]_2, [0]_T$.

2.3 Non-interactive zero-knowledge arguments of knowledge

Before beginning this section we must define some notation. Let A be a random algorithm, then when A produces the output y given the input x, we denote $y \leftarrow A(x)$. We also write $(y;z) \leftarrow (A||B)(x)$ when algorithm A outputs y and algorithm B outputs z, both on input x, where A, B use the same source of randomness. Finally, we denote $y \leftarrow S$ when we are sampling y uniformly from a set S.

Let S, W be two sets and M a deterministic polynomial algorithm that takes as input values from $S \times W$ and outputs values in the set $\{0,1\}$. Then we define a relation R as:

$$R := \{(\phi, \omega) \in S \times W | M(\phi, \omega) = 1\}$$

Elements from S will be called statements and elements from W will be witnesses. We are interested in true statements, which are elements from the set:

$$S_T := \{\phi \in S | \exists \omega \in W, (\phi, \omega) \in R\}$$

We want to work with elements from this set since the objective of this protocol is proving $\phi \in S_T$, without revealing any information of its ω pair. Let's show some examples of what a pair of statement and witness can be:

- Let $S \subseteq \mathbb{N}$ and $W \subseteq \mathbb{N}^2$. The statement will be $n \in S$ and the witness will be $(p,q) \in W$. This pair will define the relation:
$$M(n,(p,q)) = \begin{cases} 1 & \text{if} \quad n = p \cdot q \\ 0 & \text{otherwise} \end{cases}$$

- Let $S \subseteq \mathbb{F}^n$ and $W \subseteq \mathbb{F}^m$. The statement will be $(a_1, \dots, a_n) \in S$ and the witness will be $(a_{n+1}, \dots, a_{n+m}) \in W$. For matrices $U, V, W \in \mathbb{M}_{l \times (n+m)}(\mathbb{F})$, the pair defines the relation:
$$M((a_1,\dots,a_n),(a_{n+1},\dots,a_{n+m})) = \begin{cases} 1 & \text{if} \quad U(a_1,\dots,a_{n+m})^\intercal \circ V(a_1,\dots,a_{n+m})^\intercal = W(a_1,\dots,a_{n+m})^\intercal \\ 0 & \text{otherwise} \end{cases}$$
Where the operation $\circ$ is defined as, $(x_1,\dots,x_n) \circ (y_1,\dots,y_n) = (x_1y_1,\dots,x_ny_n)$
This specific relation is called QAP and it will be used later in the formalization of the protocol.

After these definitions, we are going to show the general schema our protocol is going to follow. The standard structure of an efficient publicly verifiable non-interactive argument for R is a quadruple of probabilistic polynomial algorithms:

- $(\sigma, \tau) \leftarrow$ **Setup**(R): the setup algorithm produces a common reference string (CRS) σ and a simulation trapdoor τ for the relation R. τ will be only used to reason about Zero Knowledge property later on.

- $\pi \leftarrow$ **Prove**$(R, \sigma, \phi, \omega)$: the prover algorithm takes as input the CRS σ and the pair $(\phi, \omega) \in R$ and gives the argument π. This π will be the zero knowledge proof presented to the verifier.

- $\nu \leftarrow$ **Vfy**(R, σ, ϕ, π): the verification algorithm takes as input the CRS σ, a statement ϕ and an argument π and returns a veredict ν. This ν will be 0 if the proof is rejected or 1 if it is accepted.

- $\pi \leftarrow \textbf{Sim}(R, \tau, \phi)$: the simulator takes as input the trapdoor τ and a statement ϕ and returns an argument π. This algorithm gives a proof without knowing the witness. The simulator is not used in practice, but its existence is important to argue about some properties the protocol must satisfy.

Now we are going to define some properties that the quadruple of algorithm $(\textbf{Setup}, \textbf{Prove}, \textbf{Vfy}, \textbf{Sim})$ could have. After its formalization, we will explain what each definition means in a more informal way, so that the concepts can be better understood.
Before beginning with the definitions, let's introduce the conditional probability notation we are going to use: $Pr[A_1; A_2; \ldots; A_n : B]$ is the probability of B occurring, given that $A_1, \ldots, A_n$ have already happened.

Definition. A quadruple of algorithms (Setup, Prove, Vfy, Sim) satisfies **Perfect Completeness**, if for every pair $(\phi, \omega) \in R$, then:

$$Pr[(\sigma, \tau) \leftarrow \textbf{Setup}(R); \pi \leftarrow \textbf{Prove}(R, \sigma, \phi, \omega) : \textbf{Vfy}(R, \sigma, \phi, \pi) = 1] = 1$$

In this case we have that, the probability of the Vfy algorithm being accepted, given that the setup algorithm has been run and a proof has been generated correctly, is one. Informally, this represents that an honest prover that generated a proof correctly, should always be able to convince a verifier.

Definition. A quadruple of algorithms (Setup, Prove, Vfy, Sim) satisfies **Perfect Zero-Knowledge**, if for every pair $(\phi, \omega) \in R$ and all adversary algorithms A:

$$\begin{gathered} Pr[(\sigma, \tau) \leftarrow \textbf{Setup}(R); \pi \leftarrow \textbf{Prove}(R, \sigma, \phi, \omega) : A(R, \sigma, \tau, \pi) = 1] = \\ Pr[(\sigma, \tau) \leftarrow \textbf{Setup}(R); \pi \leftarrow \textbf{Sim}(R, \tau, \phi) : A(R, \sigma, \tau, \pi) = 1] \end{gathered}$$

The adversary algorithm's objective is trying to find if a proof has been created with a witness, while only knowing the public inputs (CRS, trapdoor and proof). When the adversary detects the proof uses the witness, it outputs 1, while if it does not detect its usage, it outputs 0.
Informally, an argument is zero-knowledge if it does not leak any information apart from the truth of the statement. This is modelled by stating that the adversary has the same output when the proof is created with the witness, by the Prove algorithm, and when the proof is created without knowing the witness, by the simulator. Since the adversary can not distinguish between a proof created with or without the use of the witness, the protocol does not reveal information about it, and consequently the quadruple is Zero-Knowledge.

Definition. A quadruple of algorithms (Setup, Prove, Vfy, Sim) satisfies **Computational Knowledge Soundness for security parameter** ε, if for all polynomial time adversaries A there exists a polynomial time extractor χ_A such that:

$$Pr[(\sigma, \tau) \leftarrow \textbf{Setup}(R); ((\phi, \pi); \omega) \leftarrow (A || \chi_A)(R, \sigma) : (\phi, \omega) \notin R \quad and \quad \textbf{Vfy}(R, \sigma, \phi, \pi) = 1] < \varepsilon$$

The boundary parameter ε[1] depends on the relation and can be as small as it is wanted.
The extractor is a polynomial algorithm that has full access to the adversary's computations, and from them while performing some simple operations, it is able to obtain the witness. Soundness states that the verifier can not be convinced of a false statement. That is, the prover can convince the verifier only if he knows the witness to that statement.

[1]The size of ε is given by the size of the elliptic curve chosen. The bigger the curve, the smaller the ε.

In this case, the adversary has chosen a random false statement and has produced a proof for it, using public inputs only. If this proof convinces the verifier, the adversary is proving a statement without knowing the witness. Here is where the extractor comes to play. The extractor, looking at all the computations the adversary has done and performing basic operations, has obtained a witness for this statement. This means that in reality, the adversary does know the witness because it can be easily obtained from its computations. So essentially, this tells us that if the verifier is convinced only if the prover knows the witness.

Taking into account this previous algorithms and properties, we can define the following concepts:

Definition. The tuple $(\mathbf{Setup}, \mathbf{Prove}, \mathbf{Vfy}, \mathbf{Sim})$ is a **perfect non-interactive argument of knowledge** for R if has perfect completeness, perfect zero-knowledge and computational knowledge soundness as defined above.

We can extend this last definition to the notion of **SNARK** (succinct non-interactive argument of knowledge) if the arguments that satisfy the previous definition are succint, i.e, arguments whose proofs are short and can be easily verified by computers. This type of arguments will be the focus of our study from now on, as Groth16 is an protocol with this type of construction.

Chapter 3

Construction of the Protocol

3.1 Quadratic Arithmetic Programs

In order to encode computations, Groth16 uses Quadratic Arithmetic Programs (or QAP's). We have already seen a QAP as an example of relation, defining one between two vectors of values. Now, we are going to explain its use in the construction of the protocol. A QAP is a system of equations that is written like:

$$\begin{array}{ccc}(a_1u_{11}+\cdots+a_mu_{1m})(a_1v_{11}+\cdots+a_mv_{1m}) & = & (a_1w_{11}+\cdots+a_mw_{1m}) \\ \vdots & & \vdots \\ (a_1u_{n1}+\cdots+a_mu_{nm})(a_1v_{n1}+\cdots+a_mv_{nm}) & = & (a_1w_{n1}+\cdots+a_mw_{nm})\end{array}$$

Where $u_{ij}, v_{ij}, w_{ij}, a_i \in \mathbb{F}$. The values, u_{ij}, v_{ij}, w_{ij} are fixed coefficients and a_i are tha variables of the system.

We will now show how a relation is built with the use of a QAP. For a given QAP structure, take a fixed length l, then we get:

$$((a_1,\ldots,a_l),(a_{l+1},\ldots,a_m)) \in R \longleftrightarrow a_1,\ldots,a_m \quad \text{is solution of the QAP}$$

In practice, we have that the values $(a_1,\ldots,a_l)$ are public and $(a_{l+1},\ldots,a_m)$ are private. Then, the objective is to solve for the private values given the public values.
It is a known result, that every NP solvable problem can be rewritten as a QAP. That is, the structure of QAP is enough to build any relation in NP, so using it makes this protocol universal.

After writing the computation as this specific system of equation, we can start with the QAP procedure:

Given n equations we pick arbitrary values $r_1,\ldots,r_n \in \mathbb{F}$ and we define the polynomial $T \in \mathbb{F}[X]$, such that,

$$T(X) = \prod_{j=1}^{n}(X - r_j)$$

Now, we define $U_i(x), V_i(x), W_i(x) \in \mathbb{F}[X]$ as the degree $n-1$ Lagrange interpolating polynomials of the i-th column, that is, polynomials that satisfy,

$$U_i(r_j) = u_{i,j} \quad V_i(r_j) = v_{i,j} \quad W_i(r_j) = w_{i,j} \qquad for \quad i = 0,\ldots,m, \quad q = 1,\ldots,n$$

These polynomials will give us the coefficients of the system of equations, when evaluating them in the nodes r_j.

We will now have that $(a_1,\ldots,a_m) \in \mathbb{F}^m$ satisfies the n equations if and only if the equality is true for the evaluations in each point $r_1,\ldots,r_n$:

$$\sum_{i=0}^{m} a_i U_i(r_j) \sum_{i=0}^{m} a_i V_i(r_j) = \sum_{i=0}^{m} a_i W_i(r_j)$$

Thus, since $T(X)$ is the generator of the ideal of polynomials of $\mathbb{F}[X]$ that vanish when evaluating in any point $r_1,\ldots,r_n$, it is the smallest polynomial that has all $r_1,\ldots,r_n$ as zeros. Then, we can reformulate the previous expression as:

$$\sum_{i=0}^{m} a_i U_i(X) \sum_{i=0}^{m} a_i V_i(X) \equiv \sum_{i=0}^{m} a_i W_i(X) \quad mod(T(X))$$

Or equivalently, there exist a polynomial $H(X)$ of degree $n+1$ that satisfies the next equality:

$$\sum_{i=0}^{m} a_i U_i(X) \sum_{i=0}^{m} a_i V_i(X) = \sum_{i=0}^{m} a_i W_i(X) + H(X)T(X) \tag{3.1}$$

3.1.1 Random point evaluation

If we know a solution to the system of equations 3.1 defined over a finite field of size p, we have that for every random element $x \in \mathbb{F}_p$ we choose, the system of equations is going to hold when we evaluate the polynomials in x.
Conversely, assume the system of polynomial equations 3.1 does not hold. Then if we choose a random $x \in \mathbb{F}_p$, the probability of x being a solution of 3.1 when evaluating the polynomials in it, is at most $\frac{\text{degree of equation}}{\#\mathbb{F}_p} = \frac{2n-2}{p}$.
Then, if p is large enough, we have that proving we know a solution evaluating in a random $x \in \mathbb{F}_p$ is equivalent, almost every time (taking into account the previous probability), to proving that we know a solution to the system of equations 3.1 in general.

3.2 ZK-SNARKS for QAP's

Finally, we will explain how, the Groth16 algorithm, builds a pairing-based ZK-SNARK for quadratic arithmetic programs.
We consider the relations of the following form:
Take a large prime number p, who will define the size of the finite field $\mathbb{Z}/p\mathbb{Z}$. We define the language of statements of the form $(a_1,\ldots,a_l) \in (\mathbb{Z}/p\mathbb{Z})^l$ and witnesses $(a_{l+1},\ldots,a_m) \in (\mathbb{Z}/p\mathbb{Z})^{m-l}$ such that they satisfy,

$$\sum_{i=0}^{m} a_i U_i(X) \sum_{i=0}^{m} a_i V_i(X) = \sum_{i=0}^{m} a_i W_i(X) + H(X)T(X)$$

with $a_0 = 1$, for some degree $n-2$ quotient polynomial $H(X)$.
This is the general definition of a QAP defined over a field $\mathbb{F} = \mathbb{Z}/p\mathbb{Z}$. Proving that you know a solution to this QAP is equivalent to saying that you know the values $a_1,\ldots,a_m$ and the polynomial $H(X)$.

For the construction of the protocol, let's choose three groups of order q, $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, with hidings $[\cdot]_1, [\cdot]_2, [\cdot]_T$ respectively, and with a bilinear, non-degenerate pairing map between them as explained in the introduction. Since the representation of group elements is smaller in $\mathbb{G}_1$, in the following construction we will choose A and C to belong to this group and B to the second group, in order to maximize efficiency. Also, remember that $(a_1,\ldots,a_l)$ are public values, whereas, $(a_{l+1},\ldots,a_m)$ are private and only known by the prover. Therefore, we get the following schema:

$(\sigma, \tau) \leftarrow$ **Setup**(R):
Some random elements $\alpha, \beta, \gamma, \delta, x$ are picked uniformly from $(\mathbb{Z}/p\mathbb{Z})^*$. Here is where the x named before is chosen. It is really important for the security of the protocol that its value remains private, as it is part of the trapdoor τ. We denote $\tau = (\alpha, \beta, \gamma, \delta, x)$ and then $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$ is computed, where:

$$\sigma_1 = \left(\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta U_i(x) + \alpha V_i(x) + W_i(x)}{\gamma} \right\}_{i=0}^{l}, \left\{ \frac{\beta U_i(x) + \alpha V_i(x) + W_i(x)}{\delta} \right\}_{i=l+1}^{m}, \left\{ \frac{x^i T(x)}{\delta} \right\}_{i=0}^{n-2} \right)$$

$$\sigma_2 = \left(\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}\right)$$

$\pi \leftarrow$ **Prove**$(R, \sigma, a_1, \ldots, a_m)$:
Two random elements r, s are picked from $\mathbb{Z}/p\mathbb{Z}$, and then $\pi = ([A]_1, [C]_1, [B]_2)$ is computed, where:

$$A = \alpha + \sum_{i=0}^{m} a_i U_i(x) + r\delta \qquad B = \beta + \sum_{i=0}^{m} a_i V_i(x) + s\delta$$

$$C = \frac{\sum_{i=l+1}^{m} a_i(\beta U_i(x) + \alpha V_i(x) + W_i(x)) + H(x)T(x)}{\delta} + As + Br - rs\delta$$

$v \leftarrow$ **Vfy**$(R, \sigma, a_1, \ldots, a_l, \pi)$:
Finally the verifier accepts the proof if and only if the next identity is satisfied:

$$[A]_1 \, [B]_2 = [\alpha]_1 \, [\beta]_2 + \sum_{i=0}^{l} a_i \left[\frac{\beta U_i(x) + \alpha V_i(x) + W_i(x)}{\gamma} \right]_1 [\gamma]_2 + [C]_1 \, [\delta]_2$$

$\pi \leftarrow$ **Sim**$(R, \tau, a_1, \ldots, a_l)$:
Now for the simulator, pick random values A, B from $\mathbb{Z}_p$ and compute a simulated proof vector $\pi = ([A]_1, [C]_1, [B]_2)$ where the value of C is given by:

$$C = \frac{AB - \alpha\beta - \sum_{i=0}^{l} a_i(\beta U_i(x) + \alpha V_i(x) + W_i(x))}{\delta}$$

Now, in order to demonstrate this protocol yields a ZK-SNARK, we are going to prove it satisfies the properties of Perfect completeness, Perfect Zero-Knowledge and Computational Knowledge Soundness for security parameter ε:

Theorem 2. *The quadruple of algorithms (**Setup**,**Prove**,**Vfy**,**Sim**) given by the construction above, satisfies **Perfect Completeness**.*

Proof. Perfect completeness is satisfied if an honest prover can always convince an honest verifier. Then let's check if the Prove algorithm generates a proof that holds the Vfy algorithm pairing equation. Remember that the values of A, B, C are:

$$A = \alpha + \sum_{i=0}^{m} a_i U_i(x) + r\delta \qquad B = \beta + \sum_{i=0}^{m} a_i V_i(x) + s\delta$$

$$C = \frac{\sum_{i=l+1}^{m} a_i(\beta U_i(x) + \alpha V_i(x) + W_i(x)) + H(x)T(x)}{\delta} + As + Br - rs\delta$$

And the pairing equation is:

$$[A]_1 \, [B]_2 = [\alpha]_1 \, [\beta]_2 + \sum_{i=0}^{l} a_i \left[\frac{\beta U_i(x) + \alpha V_i(x) + W_i(x)}{\gamma} \right]_1 [\gamma]_2 + [C]_1 \, [\delta]_2$$

We are going to do the computations for each side of the pairing equation, to see if the equality holds. We are not going to use hiding notation as we are only interested in computations, but note that this pairing products will result in elements of the third group $\mathbb{G}_T$. For the left side of the equation we have:

$$(\alpha+\sum_{i=0}^{m} a_iU_i(x)+r\delta)\cdot(\beta+\sum_{i=0}^{m} a_iV_i(x)+s\delta)=$$

$$=\alpha\beta+\alpha\sum_{i=0}^{m} a_iV_i(x)+s\alpha\delta+\beta\sum_{i=0}^{m} a_iU_i(x)+\sum_{i=0}^{m} a_iU_i(x)\sum_{i=0}^{m} a_iV_i(x)+s\delta\sum_{i=0}^{m} a_iU_i(x)+r\beta\delta+r\delta\sum_{i=0}^{m} a_iV_i(x)+rs\delta^2$$

$$=\sum_{i=0}^{m} a_iU_i(x)\sum_{i=0}^{m} a_iV_i(x)+\alpha\beta+\alpha\sum_{i=0}^{m} a_iV_i(x)+\beta\sum_{i=0}^{m} a_iU_i(x)+s\alpha\delta+s\delta\sum_{i=0}^{m} a_iU_i(x)+r\beta\delta+r\delta\sum_{i=0}^{m} a_iV_i(x)+rs\delta^2$$

And now, for the right side:

$$\alpha\beta+\sum_{i=0}^{m} a_i(\beta U_i(x)+\alpha V_i(x)+W_i(x))+H(x)T(x)+As\delta+Br\delta-rs\delta^2=\alpha\beta+\beta\sum_{i=0}^{m} a_iU_i(x)+$$

$$+\alpha\sum_{i=0}^{m} a_iV_i(x)+\sum_{i=0}^{m} a_iW_i(x)+H(x)T(x)+s\alpha\delta+s\delta\sum_{i=0}^{m} a_iU_i(x)+sr\delta^2+r\beta\delta+r\delta\sum_{i=0}^{m} a_iV_i(x)+sr\delta^2-rs\delta^2$$

$$=\sum_{i=0}^{m} a_iW_i(x)+H(x)T(x)+\alpha\beta+\alpha\sum_{i=0}^{m} a_iV_i(x)+\beta\sum_{i=0}^{m} a_iU_i(x)+s\alpha\delta+s\delta\sum_{i=0}^{m} a_iU_i(x)+r\beta\delta+r\delta\sum_{i=0}^{m} a_iV_i(x)+rs\delta^2$$

If we equal both sides of the equation some factors cancel out, and we get,

$$\sum_{i=0}^{m} a_iU_i(x)\sum_{i=0}^{m} a_iV_i(x)=\sum_{i=0}^{m} a_iW_i(x)+H(x)T(x)$$

This is the QAP equation from which we were looking a solution. This implies that if the pairing equation holds, we have obtained a solution for the QAP, which is equivalent to finding the witness for our statement. Then the verifier is convinced by an honest proof, and we have proven Perfect Completeness. □

Theorem 3. *The quadruple of algorithms (**Setup**,**Prove**,**Vfy**,**Sim**) given by the construction above, satisfies **Perfect Zero-Knowledge**.*

Proof. First of all, we know that A,B have a random distribution, given by the values r,s in each case:

$$A=\alpha+\sum_{i=0}^{m} a_iU_i(x)+r\delta \qquad B=\beta+\sum_{i=0}^{m} a_iV_i(x)+s\delta$$

Let's fix A,B to that previous value. If we generate another proof, the algorithm will give us values A',B' such that:

$$A=\alpha+\sum_{i=0}^{m} a_iU_i(x)+r'\delta \qquad B=\beta+\sum_{i=0}^{m} a_iV_i(x)+s'\delta$$

Then, we have the following relation between A,A' and B,B':

$$A'=A+(r'-r)\delta=A+r''\delta \qquad B'=B+(s'-s)\delta=B+s''\delta$$

The values r'',s'' are sampled uniformly from the field $\mathbb{F}_p$ we are working with. This relation implies that we can go across all the possible values for A,B and that all have the same probability of being selected. Now when calculating a real proof, the value C is determined in the verifying expression by A,B. Then, for every pair of points (A,B) there exists a unique C. That is, the triplets (A,B,C) that satisfy the verifying equation create a surface S over $\mathbb{F}_p^3$. This surface is bijective to $\mathbb{F}_p^2$, given by the projection over A and B. A,B are sampled uniformly, so the points of S are also sampled uniformly.

Following the same reasoning, the same could be argued about the simulated proofs, as the value of C is uniquely determined by A, B using the simulator expression.
Then, real proofs and simulated proofs have identical probability distribution in the same set of elements. This means these two proofs are indistinguishable, which implies the protocol satisfies Perfect Zero-Knowledge. □

Theorem 4. *The quadruple of algorithms (**Setup**,**Prove**,**Vfy**,**Sim**) given by the construction above, satisfies **Computational Knowledge Soundness for security parameter** ε, given the generic group assumption.*

The assumption named in the theorem consist on restricting the adversary to only making a polynomial number of generic operations in the group. This theorem has a more technical proof and its beyond the scope of this study, so it will not be demonstrated[1].

[1]The proof is given at page 18 in the original paper [5].

Chapter 4

Practical Example

In this chapter we are going to develop a toy example to illustrate the Groth16 protocol. Suppose that you know two numbers $p,q \in \mathbb{N}$, that for a factorization $n = p \cdot q$. Then, you want to prove that you know such a factorization, without revealing any information about p,q.
In this case we will focus on the simple factorization, $143 = 11 \cdot 13$. In order to better clarify the procedure we have chosen a toy example, so that it is easier to understand and computations do not get over complicated. As numbers in this example are really small, you could break this protocol only by brute force, however, in practice bigger numbers and fields are used in order to improve security.
In this section we will show step by step, how the implementation of Groth16 is built is Sagemath, presenting the code an explaining it with detail.

First of all, we need to define the elliptic curve and finite field we are going to work with. The equation of the chosen elliptic curve is:

$$y^2 = x^3 + x + 1$$

defined over the finite field of $p = 2699$ elements.

```
sage: p = 2699
....: Fp = FiniteField(p)
....: Ep = EllipticCurve(Fp,[0,0,0,1,1])
....: o = FiniteField(Ep.order())(p).multiplicative_order()
....: Ep.order(), o
(2731, 13)
```

Thus, the set of points of this curve over $\mathbb{F}_p$ will be the first group we are going to work with, and it has a total of $q = 2731$ points. We will also define the finite field $\mathbb{F}_q$ for posterior computations:

```
sage: q = Ep.order()
....: Fq = FiniteField(q)
```

The element o is what we call embedding degree of an elliptic curve. The embedding degree is the smallest integer k such that $\mathbb{F}_{p^k}$ contains the subgroup of q-th roots of unity of $\overline{\mathbb{F}_p}$. This degree defines an extension of the curve in a bigger field, which will become the second group of the algorithm. As $o = 13$, the field will be the finite field of 2699^{13} elements. Thus, we define this field, and the extension of the curve:

```
sage: Fp13.<i> = FiniteField(p^13)
....: Ep13 = Ep.change_ring(Fp13)
```

Now we would want to find the generators of our groups and define their respective hiding and pairing maps. We want to obtain a group of order q, which is distinct from the first elliptic curve group. Note that if C_q is the group of the initial curve, the extension group is $(C_q)^i \times (C_{m_0})^{j_0} \times \cdots \times (C_{m_s})^{j_s}$. We compute i:

```
sage: m = Ep13.order()
....: (q^2).divides(m)
True
sage: (q^3).divides(m)
False
```

Therefore, we get $i = 2$ and we want to eliminate this the part of torsion different from q, to get a good generator of the extension group. In order to obtain this generator, we choose a random point of the curve, which will have an order distinct from q. What we want to do is obtain a multiple of this point, multiplying it by a factor, in order to get our generator of order q. This factor will be the product of the orders of the rest of cyclic subgroups that form the extension group:

```
sage: cofactor = Ep13.order()/(Ep.order()^2)
....: p2 = Ep13.random_point()
....: p3 = p2*Integer(cofactor)
....: p3
(896*i^12 + 1940*i^11 + 1871*i^10 + 1619*i^9 + 1058*i^8 + 1255*i^7 + 141*i^6
+ 1946*i^5 + 773*i^4 + 1243*i^3 + 13*i^2 + 248*i + 1767
: 1939*i^12 + 668*i^11 + 257*i^10 + 960*i^9 + 1477*i^8 + 1828*i^7 + 1417*i^6
+ 1551*i^5 + 2006*i^4 + 2118*i^3 + 1988*i^2 + 2397*i + 2574 : 1)
sage: q*p3
(0 : 1 : 0)
```

Then, $p3$ is the generator of the second group and has order exactly $q = 2731$. We can easily define the generator of the first group, pe, and using Weil's pairing, the generator of the third one,g:

```
sage: pe = Ep13(Ep.gen(0))
....: g = pe.weil_pairing(p3,Ep.order())
```

The generator pe is defined like above, because we want to see it as a point of the extended curve for the following computations. Now that we have the generators of all the groups, we can define each respectve hiding, and the pairing map between the three groups:

```
sage: def hiding1(n):
....:     return Ep13(Integer(n)*pe)
....:
....: def hiding2(n):
....:     return Integer(n)*p3
....:
....: def hiding3(n):
....:     return g^Integer(n)
....:
....: def pairing(a, b):
....:     return Ep13(a).weil_pairing(b,Ep.order())
```

With this, the definition of the group structure is finished. Thus, now we can proceed to rewrite our problem is such a way that our QAP model can solve it. The most natural way of doing this is expressing 11 and 13 in binary and each bit will be one of the unknowns of the system. The first equation of the system will be the factorization with $11, 13$ rewritten in binary. It will look like this:

$$(1 \cdot x_1 + 2 \cdot x_2 + 4 \cdot x_3 + 8x_4)(1 \cdot y_1 + 2 \cdot y_2 + 4 \cdot y_3 + 8y_4) = 1 \cdot n$$

The following rows will be restrictions for the rest of variables having only value $0, 1$, as they are bits in a binary decomposition. $11, 13$ are 4 bit numbers, thus, there will be 8 of this restriction equations:

$$x_i \cdot x_i = 1 \cdot x_i \qquad i = 1,2,3,4$$

$$y_i \cdot y_i = 1 \cdot y_i \qquad i = 1,2,3,4$$

As, $11 = 1+2+8$ and $13 = 1+4+8$ the solution of the system is a vector in $(\mathbb{F}_q)^9$, which we will call key:

$$key = (n, x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4) = (143, 1, 1, 0, 1, 1, 0, 1, 1)$$

Defined in sage as:

```
sage:key=vector([11*13]+11.bits()+13.bits())
sage:key
(143,1,1,0,1,1,0,1,1)
```

We need to write the system in a matricial way in order to form a QAP. In order to do that, let's define the following matrices first:

$$\mathbb{A} = \begin{bmatrix} 0&1&2&4&8&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{bmatrix} \quad \mathbb{B} = \begin{bmatrix} 0&0&0&0&0&1&2&4&8\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{bmatrix} \quad \mathbb{C} = \mathbb{I}_9$$

Thus, the system of equations we are looking for is:

$$\mathbb{A} \cdot key^{\intercal} \circ \mathbb{B} \cdot key^{\intercal} = \mathbb{C} \cdot key^{\intercal}$$

Where $\cdot$ is the usual matrix product, and the operation $\circ$ is defined as:

$$(x_1, \ldots, x_n) \circ (y_1, \ldots, y_n) = (x_1 y_1, \ldots, x_n y_n)$$

These matrices are constructed stacking vectors to a sub matrix of the identity:

```
sage: nbits=4
....: V=Fq^(2*nbits+1)
....: a=V([2^i for i in range(nbits)]+(nbits+1)*[0])
....: b=V((nbits)*[0]+[2^i for i in range(nbits)]+[0])
....: I=matrix.identity(nbits*2+1)
....: Im=I.submatrix(row=0, col=0, nrows=nbits*2)
....: matA=Im.stack(a)
....: matB=Im.stack(b)
....: matC=I
....: G = SymmetricGroup(9)
....: per1= G("(9,8,7,6,5,4,3,2,1)")
....: per2= G("(1,2,3,4,5,6,7,8,9)")
....: mA=per1.matrix()*matA*per2.matrix()
....: mB=per1.matrix()*matB*per2.matrix()
....: mC=matC
```

Finally, after having the system of equations in matricial form, it is possible to apply the QAP procedure defined in 3.1. The polynomials U_i, V_i, W_i can be obtained by computing the Lagrange interpolators of the columns of $\mathbb{A}, \mathbb{B}, \mathbb{C}$ respectively. To do that we need to establish the set of numbers that will be the nodes of these polynomials. We will simply choose the following set: $base = \{0,1,2,3,4,5,6,7,8\}$. Before computations, we need to create in Sagemath, a ring of polynomials over the finite field $\mathbb{F}_q$, in order to have the Lagrange polynomials well defined. Finally, we can compute the Lagrange interpolating polynomials we want, and also define $T(X) = \prod_{r \in base}^{n} (X - r)$:

```
sage: base= V([i for i in range(nbits*2+1)])
....: R.<t> = Fq[] #Univariate Polynomial Ring in t over Finite Field of size 2731
....: listaA = vector(R.lagrange_polynomial([(base[i],mA[i][j])
....:         for i in range (2*nbits+1)]) for j in range(2*nbits+1))
....: listaB = vector(R.lagrange_polynomial([(base[i],mB[i][j])
....:         for i in range (2*nbits+1)]) for j in range(2*nbits+1))
....: listaC = vector(R.lagrange_polynomial([(base[i],mC[i][j])
....:         for i in range (2*nbits+1)]) for j in range(2*nbits+1))
....: T=prod(t-i for i in range(2*nbits+1))
```

The three lists created are lists of the nine interpolating polynomials U_i, V_i, W_i, one for each column of the matrices $\mathbb{A}, \mathbb{B}, \mathbb{C}$.
To end the QAP creation part, we can check if the computations made are correct if the following expression is satisfied:

$$key \cdot U_i \cdot key \cdot V_i - key \cdot W_i = H \cdot T$$

That is, the division of the left hand side of the equation by the polynomial $T(X)$ shown above, has as quotient a polynomial $H(X)$ and remainder 0.

```
D=key*listaA*key*listaB-key*listaC
H=D.quo_rem(T)[0]
rem=D.quo_rem(T)[1]
sage: H,rem
(570*t^7 + 1328*t^6 + 1530*t^5 + 45*t^4 + 2405*t^3 + 187*t^2 + 2160*t + 1639, 0)
```

Therefore, the QAP is well defined and works as intended. Also we have computed the polynomial H that will be useful in the following parts of the algorithm.

After expressing our problem accordingly and checking the QAP is well defined, we can finally start with the ZK-SNARK protocol. First of all, we begin with the **SETUP**, where elements of the field $\mathbb{F}_q$ are chosen randomly, and then, stored in the vector τ:

```
sage: alpha=Fq.random_element()
....: beta=Fq.random_element()
....: gamma=Fq.random_element()
....: delta=Fq.random_element()
....: x=Fq.random_element()
....: tau=[alpha,beta,gamma,delta,x]
```

Furthermore, the hidings of some elements and linear combinations of them with elements of the QAP are computed to each one of the two groups. For the first group we have the following hidings, stored in the vector σ_1:

```
sage: h1alpha=hiding1(alpha)
....: h1beta=hiding1(beta)
....: h1delta=hiding1(delta)
....: h1x=[hiding1(x^i) for i in range(2*nbits+1)]
```

```
....: h1pub=hiding1((1/gamma)*(beta*listaA[0](x)+alpha*listaB[0](x)+listaC[0](x)))
....: h1priv=[hiding1((1/delta)*(beta*listaA[i](x)+alpha*listaB[i](x)+listaC[i](x)))
....: for i in range(1,2*nbits+1)]
....: h1t=[hiding1((1/delta)*(x^i)*T(x)) for i in range(2*nbits)]
```

And for the second group, we have the next hidings, stored in σ_2:

```
sage: h2beta=hiding2(beta)
....: h2gamma=hiding2(gamma)
....: h2delta=hiding2(delta)
....: h2x=[hiding2(x^i) for i in range(2*nbits+1)]
```

Note that there are more elements hidden in the first group, as it requires less computation capacity because its elements are simpler, and thus, makes the algorithm more efficient.

We now can continue with the **PROVE** part of the algorithm. In this section of the protocol, we need to find the hidings, $\pi = ([A]_1, [C]_1, [B]_2)$, with A,B,C defined as:

$$A = \alpha + \sum_{i=0}^{m} a_i U_i(x) + r\delta \qquad B = \beta + \sum_{i=0}^{m} a_i V_i(x) + s\delta$$

$$C = \frac{\sum_{i=l+1}^{m} a_i(\beta U_i(x) + \alpha V_i(x) + W_i(x)) + H(x)T(x)}{\delta} + As + Br - rs\delta$$

First of all, two elements $r,s \in \mathbb{F}_q$ are chosen randomly.

```
sage: r=Fq.random_element()
....: s=Fq.random_element()
```

As the hiding map is an homomorphism, it is linear, thus, the hiding of a sum is the sum of hidings, and we can decompose the calculations of A,B,C in separate parts.
Therefore, focusing on the computation of A; we already know the hiding of α, and it is really easy to figure out the hiding of $r\delta$, because we already know the hiding of δ and the hiding is an homomorphism. The difficult operation to do is $\sum_{i=0}^{m} a_i U_i(x)$, as we do not know the value of x. However what we do know, are the hidings of x and its powers until the degree of the Lagrange interpolating polynomials U_i, V_i, W_i. The way to compute this sum is, first of all, compute evaluation of $U_i(x)$, multiplying the coefficients of the interpolating polynomial U_i with the hiding of the corresponding degree power of x. Then, once we got the evaluations of all the polynomials stored in a vector, take the dot product of the key vector and the evaluation of polynomials vector. That will give us the sum we were looking for.

Note that Sagemath gives problem with the multiplication of elements of the field $\mathbb{F}_q$ and hidings, so we have defined the function intcoef that takes the coefficients of the interpolating polynomials and treats them like integers, so that the multiplication for the evaluation procedure can be done.

```
sage: def intcoef(pol):
....:     res = [Integer(i) for i in pol]
....:     return res + (2*nbits+1-len(res))*[0]
```

Therefore, the calculation of the hiding of A in the fis given by:

```
sage: evala=[sum(intcoef(listaA[j])[i]*h1x[i] for i in range(2*nbits+1))
....: for j in range(2*nbits+1)]
....:
....: hA=h1alpha+Integer(r)*h1delta+sum(key[i]*evala[i] for i in range(2*nbits+1))
....:hA
(1854 : 2001 : 1)
```

Similarly, the computation of the hiding of B in the second group is:

```
sage: evalb2=[sum(intcoef(listaB[j])[i]*h2x[i] for i in range(2*nbits+1))
....: for j in range(2*nbits+1)]
....:
....: h2B=h2beta+Integer(s)*h2delta+sum(key[i]*evalb2[i] for i in range(2*nbits+1))
....: h2B
(2006*i^12 + 1027*i^11 + 2347*i^10 + 2564*i^9 + 286*i^8 + 2053*i^7 + 214*i^6
+ 545*i^5 + 655*i^4 + 597*i^3 + 1142*i^2 + 1726*i + 922 :
776*i^12 + 2382*i^11 + 2549*i^10 + 151*i^9 + 136*i^8 + 2530*i^7 + 1635*i^6
+ 1700*i^5 + 1714*i^4 + 531*i^3 + 1227*i^2 + 1466*i + 553 : 1)
```

As the hiding of B in the first group is needed in the hiding of C, we can compute it analogously:

```
sage: evalb1=[sum(intcoef(listaB[j])[i]*h1x[i] for i in range(2*nbits+1))
....: for j in range(2*nbits+1)]
....:
....: h1B=h1beta+Integer(s)*h1delta+sum(key[i]*evalb1[i] for i in range(2*nbits+1))
....: h1B
(2425 : 1352 : 1)
```

Finally, in order to compute the hiding of C, we are going to use the same technique explained before for the evaluation of the polynomials in x. In this case, instead of using the powers of x, we will use the hidings of $\frac{x^i T(x)}{\delta}$ defined in the setup, and we will multiply them by the coefficients of the polynomial H, that we previously obtained while forming the QAP. The rest of components of the C computation were defined on the setup and can be used directly. Thus, the hiding of C in the first group is calculated as:

```
sage: hC=Integer(s)*hA+Integer(r)*h1B-Integer(r)*Integer(s)*h1delta
....: +sum(intcoef(H)[i]*h1t[i] for i in range(2*nbits))
....: +sum(key[i+1]*h1priv[i] for i in range(2*nbits))
....: hC
(2474 : 161 : 1)
```

With this hiding, we have finished the prove section and we can finally start the **VERIFY** process. This section of the algorithm is really straightforward; the verifier needs to check if a pairing product equation holds true or false. The equation will be checked using the public part of the key, in this case, the number $n = 143$ stored in the first component of the vector key:

```
sage: pairing(hA,h2B)==pairing(h1alpha,h2beta)*pairing(Integer(key[0])*h1pub,h2gamma)
....: *pairing(hC,h2delta)
True
```

If the equation holds true, the prover has convinced the verifier that he knows the numbers p, q such that $p \cdot q = 143$ and the protocol has finally ended.

Chapter 5

Conclusion

As we have seen, Zero Knowledge proofs and ZK-SNARKS, are concepts that have been studied theoretically for many years and its implementation has changed this community in the past decade. This has been achieved by a great combination of mathematics and computer science algorithms. Referring to the Groth16 protocol, we could argue it is an excellent algorithm which improves efficiency from its predecessors. This is true, as it only needs three elliptic curve points hidings in its proof, a much smaller number than Pinocchio for example, who needed eight points. Furthermore, its verification is also faster, with a total of only three pairing operations. These characteristics make Groth16 a nice functional algorithm with universal capability to prove any computation.

The implementation of Zero-Knowledge proofs has brought new possibilities and applications in the real world. There are many times you give extra information when trying to prove something. For example, let's say you rent a bike and you use it to move around the city. When your credentials are introduced in order to use that bike you are giving the renting company the full path you are making while riding the bike, as they have a geolocalization system incorporated. This information should be private and you may not want the company to know where you have been while riding the bike. However, the company needs to have the bike located for maintenance purposes. In this conflict of interests, Zero-Knowledge proofs could help solve the problem. We know that Zero-Knowledge proofs make a prover convince a verifier of a secret, without giving any information of that secret. Well, if we choose the path made while riding the bike to be the secret, we could prove we have used the bike and we would give the company the final localization of the bike to do their maintenance work.

This is an example of many from the use of zero-knowledge proofs, as they have already been applied in many areas such as electronic voting, currency transaction, etc. We can only wonder what Zero-Knowledge proofs could bring to our society in the near future, as they have for sure potential to be applied in many fields.

Bibliography

[1] E. Ben-Sasson, A. Chiessa, E. Tromer, M. Virza, *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*, 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, (2014), 781-796.

[2] M. Binello, *The Zero Knowledge Blog*, `https://www.zeroknowledgeblog.com/`.

[3] V. Buterin, *Vitalik Buterin's website, STARK's, Part I: Proofs with Polynomials*, `https://vitalik.ca/general/2017/11/09/starks_part_1.html`.

[4] D. Freeman, *Constructing Pairing-Friendly Elliptic Curves with Embedding Degree 10*, Algorithmic number theory, Springer, Berlin, (2006), 452-465.

[5] J. Groth, *On the Size of Pairing-based Non-interactive Arguments*, Advances in cryptology EUROCRYPT 2016. Part II, Springer, Berlin,(2016), 305-326.

[6] S. Goldwasser, S. Micali, C. Rackoff, *The knowledge complexity of interactive proof-systems*, SIAM Journal on Computing. 18, (1989), 186-208.

[7] V. S. Miller, *The Weil Pairing and its efficient calculation*, Journal of Cryptology. The Journal of the International Association for Cryptologic Research, Vol 17, (2004), 235-261.

[8] B. Parno, J. Howell, C. Gentry, M. Raykova, *Pinocchio: Nearly Practical Verifiable Computation*, 2013 IEEE Symposium on Security and Privacy, (2013), 238-252.

[9] *SageMath, the Sage Mathematics Software System (Version 9.3)*, The Sage Developers, 2023, `https://www.sagemath.org`.