UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación y Facultad de Ciencias

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Zero-Knowledge Proofs and their Blockchain Applications

Presentado por:
Javier Antxon Garrues Apecechea

Curso académico 2023-2024

Zero-Knowledge Proofs and their Blockchain Applications

Javier Antxon Garrues Apecechea

Javier Antxon Garrues Apecechea *Zero-Knowledge Proofs and their Blockchain Applications*.
Trabajo de fin de Grado. Curso académico 2023-2024.

Responsable de tutorización

Pedro A. García-Sánchez
Departamento de Álgebra

Doble Grado en Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación y Facultad de Ciencias

Universidad de Granada

Declaración de originalidad

D. Javier Antxon Garrues Apecechea

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2023-2024, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 14 de Julio de 2024

Fdo: Javier Antxon Garrues Apecechea

This dissertation is dedicated to my parents and sister.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to Prof. Dr. Pedro A. García-Sánchez for his insightful suggestions and guidance throughout the writing of this dissertation. His assistance has been instrumental in helping me comprehend numerous mathematical complexities and in critically analyzing my work. He provided invaluable feedback on various aspects of the dissertation, mathematical formulation, formal writing, overall coherence and choice of topics. Furthermore, he assisted me in choosing the topic of this dissertation and was readily available for discussions even before officially becoming my advisor.

I would also like to thank Prof. Dr. Xu Wang from the University of Technology in Sydney. The project on Attribute-Based Encryption I made for his Cryptography course during my exchange year made me rekindle zero-knowledge proofs, which I had initially encountered a few months earlier. His insights on Blockchain helped me grow a great interest in the technology and ultimately led me to choose the topic of this dissertation. He also explained to me several concepts that have been proven valuable when developing this dissertation.

I must also acknowledge the guidance provided by my father Prof. Dr. Josean Garrues. His ample experience in academic writing and Master's dissertation tutoring has been very valuable in helping me clearly describe the research question and craft insightful introductions and conclusions.

Finally, I would like to recognize the effort made by Prof. Dr. Francisco Torralbo and my tutor, Prof. Dr. Pedro A. García-Sánchez, in developing a comprehensive LaTeX template and making it available to students. Their contribution greatly facilitated the presentation and formatting of this dissertation.

Summary

This dissertation's objective is to provide the necessary mathematical tools to understand how zero-knowledge proofs work and how they are applied in Blockchain. The gap between theory and application is bridged with the use of several classic and original examples, concluding with the implementation of a state-of-the-art protocol. The integration of zero-knowledge proofs in Blockchain is studied. Emphasis is placed on their particular efficacy in addressing the critical issues of privacy and scalability.

The dissertation first establishes the foundational mathematical principles essential for understanding zero-knowledge proofs. Topics covered range from modular arithmetic to elliptic curves and bilinear mappings, accompanied by introductions to complexity theory, language theory, graph theory and hash functions. Zero-knowledge proofs are presented from a historical point of view, tracing their evolution through critical literature that has shaped the field, allowing it to reach its current state. New concepts are introduced using classic examples, stating their properties and providing full proofs for some of them using the aforementioned mathematical fundamentals.

The concept of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) is briefly introduced and practically implemented following a state-of-the-art protocol called *Groth16*. The implementation consists of solving an original example knowledge problem defined on $\mathbb{Z}_{19}$ using a Python program. The program makes use of some mathematical functions extracted from the *SageMath* Python library. SageMath is an open-source mathematics software system that provides a comprehensive set of tools for Algebra, Calculus, and Number theory, which are essential for the implementation. Intuition on the Groth16 protocol parameters is provided, and a step-by-step explanation of the proof definition process is also given. The program responsible for generating the proof and simulating the verification process is included. It demonstrates the correctness of the proof and contains insightful comments regarding the properties and subgroups of the chosen elliptic curve. In addition, the particular values that the protocol generates for the problem and an alternative verification of the proof's correctness are also included.

Subsequently, the dissertation explores Blockchain, explaining its foundational principles such as public-key cryptography, Merkle trees, and key generation. The operational mechanisms of Bitcoin and Ethereum are outlined, highlighting their transaction and validation processes and identifying the primary challenges of privacy and scalability. Solution proposals, extracted from the most relevant literature and references on the topics, are presented and analyzed. The necessity of zero-knowledge proofs is inductively derived from these descriptions, and it is clearly seen as a potential solution. Specific Blockchain protocols are described to conclude the dissertation. The application of zk-SNARKs in Blockchain becomes evident through case studies of *ZCash*, *Tornado Cash*, and *zk-Rollups*. Detailed diagrams are provided to illustrate the protocols of these technologies, and a strong emphasis is placed on outlining the benefits zero-knowledge proofs offer and highlighting their contemporary relevance in Blockchain.

Keywords— Zero-knowledge proofs, Blockchain, zk-SNARKs, Groth16, Elliptic curves, Weil pairing, Bitcoin, Ethereum, scalability, privacy, ZCash, Tornado Cash, zk-Rollups

Summary (Spanish)

El objetivo de este trabajo de fin de grado es proporcionar las herramientas matemáticas necesarias para comprender cómo funcionan las pruebas de conocimiento cero, así como estudiar cómo se aplican en la cadena de bloques. La brecha entre la teoría y las aplicaciones se cierra mediante el uso de varios ejemplos clásicos y originales, concluyendo con la implementación de un protocolo del estado del arte. Se estudia la integración de pruebas de conocimiento cero en la cadena de bloques, haciendo hincapié en su particular eficacia para abordar las cuestiones críticas de privacidad y escalabilidad.

El trabajo primero establece los principios matemáticos fundamentales esenciales para comprender las pruebas de conocimiento cero. Los temas cubiertos incluyen desde aritmética modular hasta curvas elípticas y aplicaciones bilineales, complementados con introducciones a la teoría de la complejidad, teoría de lenguajes formales, teoría de grafos y funciones hash. Las pruebas de conocimiento cero se presentan desde un punto de vista histórico, siguiendo su evolución a través de la literatura fundamental que ha dado forma al campo, permitiéndole alcanzar su estado actual. Se introducen nuevos conceptos utilizando ejemplos clásicos de los que se exponen sus propiedades, para algunas de las cuales se proporcionan demostraciones completas utilizando los fundamentos matemáticos antes mencionados.

El concepto de argumentos de conocimiento sucintos no interactivos de conocimiento cero (zk-SNARKs) se presenta brevemente y se implementa de manera práctica siguiendo un protocolo del estado del arte llamado *Groth16*. La implementación consiste en resolver un problema original de conocimiento definido en $\mathbb{Z}_{19}$ utilizando un programa en Python. El programa hace uso de algunas funciones matemáticas extraídas de la biblioteca de Python *SageMath*. SageMath es un sistema de software matemático de código abierto que proporciona un conjunto completo de herramientas para álgebra, cálculo y teoría de números, que son esenciales para la implementación. Se proporcionan indicaciones perspicaces sobre los parámetros del protocolo Groth16 y se brinda una explicación paso a paso del proceso de definición de la prueba. Se incluye el programa encargado de generar la prueba y simular el proceso de verificación. El programa demuestra la correción de la prueba y contiene comentarios esclarecedores sobre las propiedades y subgrupos de la curva elíptica elegida. Además, también se incluyen los valores particulares que genera el protocolo para el problema y una verificación alternativa de la corrección de la prueba.

Posteriormente, el trabajo explora la cadena de bloques y explica sus principios fundamentales, como la criptografía de clave pública, los árboles Merkle y la generación de claves. Se describen los mecanismos operativos de Bitcoin y Ethereum, destacando sus procesos de transacción y validación e identificando los principales desafíos de privacidad y escalabilidad. Se presentan y analizan soluciones propuestas, extraídas de la literatura y referencias más relevantes sobre los temas. La necesidad de las pruebas de conocimiento cero en dicho área se deriva inductivamente de estas descripciones y se ve claramente como una potencial solución. Para concluir el trabajo, se describen protocolos específicos de la cadena de bloques. La aplicación de los zk-SNARKs en la cadena de bloques se hace evidente a través de los estudios de caso de *ZCash*, *Tornado Cash* y *zk-Rollups*. Se proporcionan diagramas detallados para ilustrar los protocolos de estas tecnologías, y se hace un fuerte énfasis en describir los be-

neficios que ofrecen las pruebas de conocimiento cero y resaltar su relevancia contemporánea en la cadena de bloques.

Keywords— Pruebas de conocimiento cero, Cadena de bloques, Blockchain, zk-SNARKs, Groth16, Curvas elípticas, Emparejamiento de Weil, Bitcoin, Ethereum, escalabilidad, privacidad, ZCash, Tornado Cash, zk-Rollups

Introduction

Zero-knowledge proofs are protocols that demonstrate a statement's truth without disclosing any other information about the statement being proven. This strong feature makes this technology highly valuable in various areas of cryptography. This dissertation will focus on the applications of zero-knowledge proofs in Blockchain.

A blockchain is a digitally distributed ledger that records transactions between parties and is maintained by a network of collaborative nodes. While Blockchain technology has found applications in diverse areas such as insurance, healthcare, and copyright protection [CXS+18], our focus will be on cryptocurrencies. Cryptocurrencies are digital currencies secured by cryptography and implemented in Blockchain. Among these, Bitcoin and Ethereum stand out due to their significant user bases and daily trading volumes (see Figure 1).

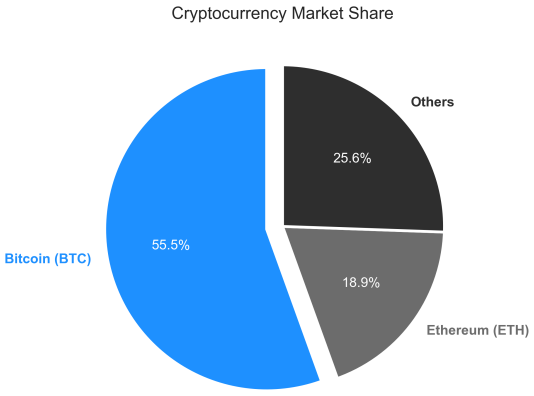


Figure 1.: Pie chart that demonstrates Bitcoin's dominance over the rest of cryptocurrencies based on its market share as of 21 June 2024. Note that the denominated group *Others* does not include any single cryptocurrency that has a greater share than 5 %. Source: Elaborated by the author using the data provided in [Sta24].

The popularity of many cryptocurrencies surged during the COVID-19 pandemic (see Figure 2). However, this rise highlighted two intrinsic problems with Blockchain technology: privacy and scalability.

Privacy is the assurance that the confidentiality of certain information is protected. Despite users assuming cryptocurrencies enable fully anonymous transactions, this could not be further from the truth. On the other hand, scalability is the ability of the network to maintain its performance when the number of transactions increases considerably. The surge in cryptocurrency popularity exposed many cryptocurrencies' infrastructural weaknesses in supporting higher transaction volumes. Additionally, the influx of less knowledgeable

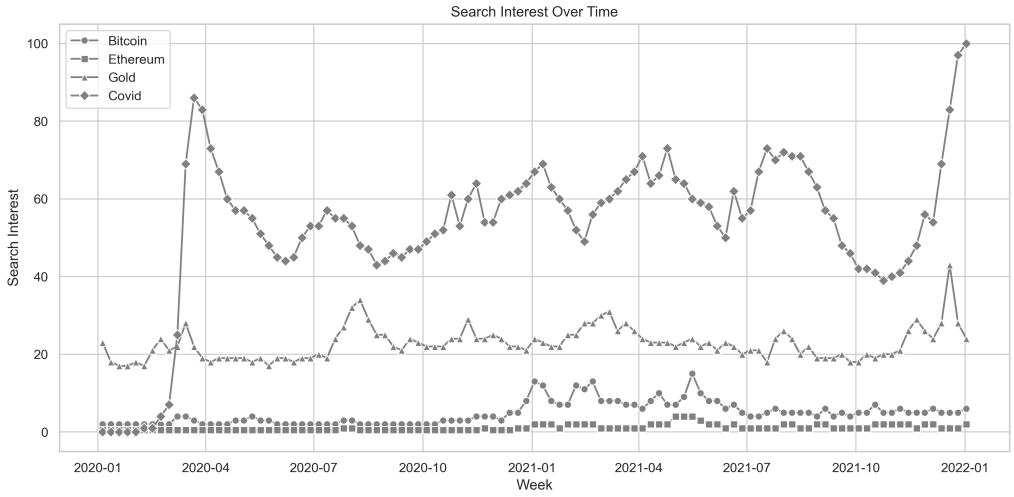


Figure 2.: Line chart comparing the number of Google searches for *Bitcoin*, *Ethereum*, *Gold* and *Covid* between the 1 June 2020 and 1 June 2022. Source: Elaborated by the author using the data provided by Google Trends [Tre24].

users led to frequent information leaks on social media and similar platforms, compromising Blockchain's privacy capabilities.

Zero-knowledge proofs have proven effective in addressing these Blockchain issues. The core idea behind zero-knowledge proofs is simple: proving that a statement is true without revealing why it is true. However, the implementation of this concept is significantly more complex, often leading to zero-knowledge proofs being treated as black boxes with unknown internal workings.

The objective of this dissertation is to provide the necessary mathematical tools to understand how zero-knowledge proofs work and how they are applied in Blockchain.

We aim to present zero-knowledge proofs, starting from their theoretical foundations to explain a state-of-the-art cryptographic protocol that implements them: Groth16 [Gro09]. Most papers about this technology fall short in one of the two areas. They either do not explain why proofs are built the way they are, or they theorize about the zero-knowledge property without reaching practical conclusions. Conversely, we bridge the gap between theory and application.

The topic of linking zero-knowledge proofs and Blockchain has been previously studied, but without taking a similar approach to ours. For instance, the paper 'Overview of Zero-Knowledge Proof and Its Applications in Blockchain' [ZWMT22] approaches the topic from a non-mathematical viewpoint without focusing on the technology's evolution. Similarly, 'An Introduction to Zero-Knowledge Proofs in Blockchains and Economics' [BLN23] provides a naive approach to the theoretical fundamentals but effectively presents a practical use-case scenario. Lastly, 'Leveraging Zero-Knowledge Proofs for Blockchain-Based Identity Sharing: A Survey of Advancements, Challenges, and Opportunities' [ZDS+24] offers a comprehensive overview of zero-knowledge proofs for achieving identity privacy in Blockchain but lacks detailed background on the origin and rationale behind the protocols. Unlike these articles, our dissertation includes practical implementations and detailed theoretical explanations.

1. Structure

The dissertation topics are divided into five chapters.

1. **Mathematical fundamentals.**
2. **Zero-Knowledge Proofs.**
3. **zk-SNARKs.**
4. **Blockchain.**
5. **Zero Knowledge Proofs in Blockchain.**

Let us succinctly describe the topics covered by each of these chapters.

In Chapter 1, we present the mathematical foundations that support zero-knowledge proofs. These go from modular arithmetic basics to elliptic curves and bilinear mappings. We will also introduce and explain topics on complexity theory, language theory, hash functions and graph theory, as a basic understanding of these topics will be needed to understand some results. We provide proofs for some of the most important theorems and propositions and give some insight into how these mathematical tools will be used in the following chapters.

A formal definition of zero-knowledge proofs, as well as their types and properties, is given in Chapter 2. We guide the reader through the early history of the proofs, sequentially introducing more complex ideas that appeared in the literature in the following years. We use many classic examples, such as the 'Quadratic Residuosity Problem' or the 'Discrete Logarithm Problem.' We slowly increase the complexity of the proofs until we introduce non-interactive zero-knowledge proofs or NIZKs.

In Chapter 3, zero-knowledge succinct arguments of knowledge are defined. We motivate their existence and thoroughly explain which formal tools are commonly used to model problems that can be proven using zk-SNARKs. Finally, we define an example problem and explain how one of the state-of-the-art protocols, Groth16, can be used to define a proof. We provide intuition on the parameters defined during the protocol. In addition to this, we include in Appendix B and Appendix C a software that simulates the protocol and generates the proof.

We present Blockchain, its basic functioning and the concepts necessary to understand how Bitcoin and Ethereum are constructed in Chapter 4. Among those concepts, more mathematical topics such as public-key cryptography, Merkle trees and key generation are treated. We explain Bitcoin and Ethereum's transactions and validation processes and point out their differences. The chapter concludes with a description of the main Blockchain problems, privacy and scalability, and various proposals used to solve them are studied.

In the last chapter of the dissertation (Chapter 5), all the previously explained concepts come into play to allow us to provide specific examples of how zero-knowledge proofs are used in Blockchain. We introduce *ZCash*, *Tornado Cash* and *zk-Rollups*, how they are implemented, and the benefits zero-knowledge proofs provide in these technologies. We provide a descriptive diagram for each of them to help us create an overall picture of these technologies' protocols.

We conclude the dissertation by summarising the topics covered and recognizing the dissertation's limitations in Chapter 6. At last, Chapter 7 outlines the conclusions obtained and research questions that could be studied during future works, such as a Master's thesis or PhD thesis on the topic.

2. Project plan and Timeline - Research and Coding

A dissertation in our double degree in Computer Science and Mathematics comprises 18 ECTS. Considering each ECTS is equivalent to 25 hours of work, we theoretically require 450 hours for this project. Because of the degree of connection between computer science and mathematics in this dissertation, we have combined an analysis of both, taking into account all the hours spent on this project.

The project's start date was February 2024, and the weekly dedication to the project was decided to be 30 hours per week. These made 120 hours a month and 660 hours in the span of five months and a half (February-July).

The initial finishing date of this project was intended to be 19 June 2024. However, the difficulties in accommodating enough time to finalize the formatting details and the tutor recommendations made us decide to move the finish date forward a month. The extra month was used to include the final tutor recommendations, improve the formatting, improve the overall wording and clarity of the text, and generalize the Python code provided as much as possible.

The dissertation project was divided into the following phases.

1. **Planning.** Analyze the project's objectives and trace a clear path for the coming months. Decide on the dissertation chapters and their contents, taking the dissertation proposal as a basis.

2. **Mathematical research and literature review.** Find and study appropriate books, manuscripts and lecture notes for the mathematical part. Do research on zero-knowledge proofs and their history. Explore their applications in state-of-the-art technological solutions. Browse for information on Blockchain and their projects by reviewing their original documentation.

3. **Software implementation.** Create Python code using the functions provided in the SageMath library. Follow an iterative waterfall model as described below.

4. **Writing.** Formalize all the notes, proofs, calculus and research done for each chapter, paying special attention to the vocabulary used. Present the final results, obtained using Python, formally.

5. **Diagram design.** Create diagrams for the last chapter following conventions similar to the ones used in related articles in the field.

6. **Formatting and review.** Finalize formatting details, improve sections that were not clearly articulated, and include all final recommendations.

Table 1 provides a table adding up the number of weeks used up in each part of the process of this dissertation.

Table 2 provides an overall timeline, including the time spent in each part of the dissertation.

Regarding the program's implementation (the one used to solve the problem given in Chapter 3), we have followed an iterative waterfall model (as described by Winston W. Royce) with the following parts.

1. **Feasibility study.** During the literature research part of the project, we analyzed the task's difficulty and searched for previous 'manual' implementations of the Groth16 protocol.

2. **Requirement study.** We set up the software's objectives: being didactic, being as general as sensibly possible and using the least amount of predefined functions. Those functions could hinder the visualization of the protocol's internal workings.

3. **Design.** We design the program's structure following the implementation description given by the protocol authors and modifying some snippets found in [Aut24] used to obtain the elliptic curves.

4. **Coding.** We defined the data structure and created the necessary functions to work with the data (polynomials) obtained when formalizing the problem.

5. **Testing.** We tested the code using different base fields (such as $\mathbb{Z}_{43}$) and changed the simulation trapdoor, witness, instance and random values (see Section 3.3 to understand what each of these terms means).

6. **Results reporting and maintenance.** To conclude, we reported the results and ensured any potential problem sources were diminished.

This methodology, being an iterative waterfall model, implied that all phases of this software implementation project used feedback to fix errors and improve during the later stages of the project.

Phase	**Time span**
1. Planning	1 week
2. Mathematical research and literature review	8 weeks
3. Software implementation	4 weeks
5. Writing	4 weeks
4. Diagram design	2 weeks
6. Formatting and review	4 weeks

Table 1.: Project phases duration in weeks.

Gantt chart	February				March				April					May				June				July	
	5	12	19	26	4	11	18	25	1	11	15	22	29	6	13	20	27	3	10	17	24	1	8
1.Planning																							
2.Literature																							
3.Software																							
4.Writing																							
5.Diagram																							
6.Review																							

Table 2.: Gantt Chart of the project timeline.

3. Literature and Methodology - Research

We continue this introduction by providing an overview of the methodology followed for the literature acquisition, reference contrast, and source verification.

We extensively used the Web of Science [oS24] to find relevant articles on the topics at hand. We must also highlight that most of the information related to Blockchain has been obtained directly from the web, as nearly all cryptocurrencies in the market have been presented using white papers published on their web pages. This is the case for Bitcoin and Ethereum.

We have cited all the sources used during this project, although we must differentiate between the different types of citations.

1. **Papers read in full.** These include the papers that have been instrumental in understanding how zero-knowledge proofs, Bitcoin and Ethereum are defined.

2. **Papers cited for historical reference.** These include the articles in which the different improvements zero-knowledge proofs have experienced were introduced. They also include the journals or books in which the fundamental mathematical results were first presented.

3. **Articles cited for definitions or explaining complex Blockchain concepts**. These were found in the form of web articles made by experts or articles published in top-tier journals.

4. **Web pages containing up-to-date data.** These include the ones used to obtain information on Google trends, cryptocurrency market share, exchange rates between currencies or the number of users of the different Blockchain solutions.

5. **Lectures notes from prestigious universities.** These universities include Princeton [Bar07a, Bar07b, Bar07c] and Stanford [Wu19a, Wu19b]. Because lecture notes may lack rigor, they have only been used as references for understanding complex mathematical concepts. However, the professors teaching these courses are associated with the research groups of the leading researchers in zero-knowledge proofs. Some of these researchers are Dan Boneh (Stanford University), Amit Sahai (University of California Los Angeles), and Boaz Barak (Harvard).

4. Assessment of the dissertation's initial proposal objectives

We conclude this introduction by reviewing the topics from the dissertation's initial proposal, which we have successfully covered.

1. **Study of the mathematical fundamentals of zero-knowledge proofs.** This objective was satisfactorily covered in the first chapter (Chapter 1) of the dissertation. The discrete logarithm, modular arithmetic roots, hash functions and elliptic curves were extensively studied, providing the primordial theorems, propositions, lemmas and most important proofs. Furthermore, theoretical fundamentals in language theory, complexity theory and graph theory were also included to form a very broad and complete description of the mathematical tools needed to understand zero-knowledge proofs.

2. **Study of Blockchain's mathematical fundamentals.** The second objective was sufficiently covered with the topics studied in the first (Chapter 1) and fourth chapter (Chapter 4) of the dissertation. Hash functions, elliptic curves, Merkle trees and key generation are extensively explained. Moreover, their particular implementations in the cryptocurrencies' transaction and validation processes are also described.

3. **Study of the mathematical structures of Blockchain's cryptographic processes (zero-knowledge proofs).** The ring of integers modulo n $\mathbb{Z}_n$, the field $\mathbb{Z}_p$ where p is prime (including the elliptic curves built on top of it, their subgroups and r-torsion groups), the group of units $\mathbb{Z}_n^*$, and the cyclic prime-order groups are all presented in Chapter 1. These are the fundamental structures needed to implement the zero-knowledge protocols used in Blockchain. Furthermore, they are extensively used when generating keys, encrypting information and defining hash functions.

4. **Study of the zero-knowledge proofs variations.** Chapter 2 and Chapter 3 extensively covered how the different variations of zero-knowledge were subsequently introduced in the literature upon reaching the state-of-the-art zk-SNARKs. Each variation is studied in depth, providing examples, insightful comments about its strengths and weaknesses, and historical references to provide context. Furthermore, a particular protocol (Groth16) is used to fully develop how a zk-SNARKs example is built for a real problem.

5. **Study of the zero-knowledge proofs applications in Blockchain.** After introducing the two main cryptocurrencies (Bitcoin and Ethereum) and their main processes, scalability and privacy are introduced as major Blockchain problems. This introduction naturally leads us to describe the Blockchain solutions that implement zero-knowledge proofs in Chapter 5. This description is accompanied by contemporary market data and explanatory diagrams to better understand the benefits zero-knowledge proofs offer to ZCash, Tornado Cash and zk-Rollups.

This section finalizes the introduction to my Bachelor's in Mathematics and Bachelor's in Computer Science dissertation and gives way to the primordial mathematical fundamentals.

1. Mathematical fundamentals

Before delving into the intricacies of Zero-knowledge proofs, we will present a few concepts on number theory, language theory, complexity theory and graph theory, amongst others, that will allow us to understand the presented examples of zero-knowledge proof systems. We will also present hash functions and elliptic curves here so the reader can quickly and easily find all the mathematical fundamentals needed to understand this dissertation.

Regarding the references used during this chapter, we must highlight 'An Introduction to the Theory of Numbers' [HW59] by G. H. Hardy and E. M. Wright, 'A Classical Introduction to the Theory of Numbers' [KI90] by Kenneth Ireland and Michael Rosen, and 'A Course in Number Theory and Cryptography' [Kob94] by Neal Koblitz. These books contain some important proofs, helped us understand the more complex concepts and verify the correctness of some other proofs found online. We also used [Lyn24, Wik24a, Onl24, Bis14] for clarifying some concepts related to modular arithmetic roots, the Legendre symbol and the Jacobi symbol. The MoonMath manual found in [Aut24], was our primary source of information when describing hash functions and elliptic curves. Wikipedia and Mathematics Stack Exchange were also visited to get a general idea of the mathematical tools being studied and clarify our doubts about some of the proofs, respectively.

1.1. Modular arithmetic basics

Similar to what occurs in many other cryptographic tools, modular operations with big prime numbers play an important role. In this chapter, we will define the basic concepts that will be used in some of the examples we will present later. In particular, modular arithmetic will be primordial in the quadratic residuosity problem, the first example used when zero-knowledge proofs were initially presented in 1895. Let us begin by presenting the Division Algorithm.

Lemma 1.1 (Division Algorithm)**.** *Given any two integers a and b, where b is non-zero, there exist unique integers q and r such that $a = b \cdot q + r$ and $0 \leq r \leq |b|$.*

For instance, let $a = 3475$ and $b = 152$. Then $q = 22$, $r = 131$ and $3475 = 152 \cdot 22 + 131$. After this simple example, we can define what we understand by *a modulo b*.

Definition 1.1. Let a and b be two integers. We define $a \bmod b$ (*a modulo b*) as the remainder of dividing a by b.

While this definition will be useful when presenting the Extended-Euclidean Algorithm (see Section 1.2), another enables us to define a congruence relation. This relation will be used to define a set on top of which several constructions, such as the discrete logarithm and the discrete roots, are built.

Definition 1.2. Let a and b be integers and n an integer such that $n \geq 2$. We write $a \equiv b \pmod{n}$ (*a is congruent with b modulo n*) if $b - a$ is a multiple of n.

This relation defines an equivalence relation in $\mathbb{Z}$. Given a, b, c and n integers, it is easy to see that $a \equiv a \pmod{n}$, $a \equiv b \pmod{n}$ is equivalent to $b \equiv a \pmod{n}$, and if $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$. Furthermore, addition and multiplication behave well with respect to this binary relation. Then, this equivalence relation is a congruence.

Proposition 1.1. *Let a, b, c and d be integers and n a natural number such that $n \geq 2$. If $a \equiv b \pmod{n}$, then $a \cdot c \equiv b \cdot c \pmod{n}$ and $a + c \equiv b + c \pmod{n}$. In addition, if $c \equiv d \pmod{n}$, then $a \cdot c \equiv b \cdot d \pmod{n}$ and $a + c \equiv b + d \pmod{n}$.*

If $a \equiv b \pmod{n}$, then $n \mid (b - a)$. We can write $b - a = (b + c) - (a + c)$. Then $a + c \equiv b + c \pmod{n}$.

Similarly, if $n \mid (b - a)$, then $n \mid c \cdot (b - a)$. Using that $c \cdot (b - a) = c \cdot b - c \cdot a$, then $c \cdot a \equiv c \cdot b \pmod{n}$.

The second part of the proposition is directly proven using transitivity.

1.2. Bézout's Identity and Euclidean Algorithm

Bézout's identity was introduced by Étienne Bézout in 1799. Bézout was a renowned French mathematician known for his work "Théorie générale des équations algébriques". It will be used in numerous proofs throughout this dissertation when working with two coprime integers. Although we will not explicitly need to find the identity coefficients manually, we will present the Extended Euclidean Algorithm to allow readers to understand the computational simplicity of finding them.

Lemma 1.2 (Bézout's identity)**.** *Let a and b be integers with $\gcd(a, b) = d$. Then, there exist integers s and t such that $as + bt = d$.*

There exists a tool that allows us to find those coefficients efficiently. The main mathematical construction on which this tool is built is the Division Algorithm we presented in Lemma 1.1. Assuming that the algorithm is understood, we are prepared to present the aforementioned tool named after the ancient Greek mathematician Euclid: the Euclidean Algorithm. It was introduced in his work "Euclid's Elements" around 300 BC.

The Euclidean Algorithm is a continual repetition of the Division Algorithm for integers that allows us to find the greatest common divisor of two integers. Let a, b, q, and r be integers, then the algorithm works as follows.

1. Find q and r such that $a = b \cdot q + r$ and $0 \leq r \leq |b|$.
2. Let $a = b$, $b = r$.
3. Go to step 1 while $r \neq 0$.
4. Exit the algorithm.

Note that the last step will be reached when the remainder r equals zero. Let us see an example of how the algorithm works.

Example 1.1. Let $a = 3475$ and $b = 152$ and let us find their greatest common divisor using the Euclidean Algorithm.

$$\begin{aligned} 3475 &= 152 \cdot 22 + 131, \\ 152 &= 131 \cdot 1 + 21, \\ 131 &= 21 \cdot 6 + 5, \\ 21 &= 5 \cdot 4 + 1, \\ 5 &= 1 \cdot 5 + 0. \end{aligned}$$

We can also consider the Extended Euclidean Algorithm, a modified version of the previous algorithm in which we extract and save data in a dynamic table. This data will be useful in recovering the coefficients from Bézout's identity. The Extended Euclidean Algorithm is a set of ordered steps that lead to obtaining the coefficients s and t from Bézout's identity. It works by reversing the steps of the Euclidean Algorithm after the $\gcd(a, b)$ is found. Provided two integers a and b, the steps are as follows.

1. Set up a table of values with the following columns.
   a) Column $index_i$. It contains the step index.
   b) Column $remainder_i$. It contains the quotient obtained at each step of the Euclidean Algorithm.
   c) Column $remainder_i$. It contains the remainder obtained at each step of the Euclidean Algorithm.
   d) Column s_i. It contains coefficients for a in Bézout's identity such that $\gcd(\mathrm{a}, \mathrm{b}) = remainder_i = as_i + bt_i$.
   e) Column t_i. It contains coefficients for b in Bézout's identity such that $\gcd(\mathrm{a}, \mathrm{b}) = remainder_i = as_i + bt_i$.
2. Set $remainder_0 = a$ and $remainder_1 = b$.
3. Execute the Euclidean Algorithm, saving the successive values of quotients and remainders in the first two non-index columns. Add as many columns as necessary until $remainder_i = 0$.
4. Define the variables $s_0 = 1$, $s_1 = 0$, $s_0 = 0$ and $t_0 = 1$.
5. Set $s_i = s_{i-2} - s_{i-1} \cdot remainder_i$ and set $t_i = t_{i-2} - t_{i-1} \cdot remainder_i$.
6. Go to step 5 while $i \in index_i$ column.
7. Exit the algorithm.

Note that the last step will be reached when $i \notin index_i$ column. Let us see an example of how the algorithm works.

Example 1.2. In Table 1.1 we find the integers z and x such that $3475z + 152x = 1$ after finding that $\gcd(3475, 152) = 1$ in the previous example.

$index_i$	$quotient_i$	$remainder_i$	s_i	t_i
0		**3475**	1	0
1		**152**	0	1
2	$3475 \div 152 = 22$	$3475 - 152 \cdot 22 = 131$	$1 - 22 \cdot 0 = 1$	$0 - 22 \cdot 1 = -22$
3	$152 \div 131 = 1$	$152 - 131 \cdot 1 = 21$	$0 - 1 \cdot 1 = -1$	$1 - (-22) \cdot 1 = 23$
4	$131 \div 21 = 6$	$131 - 21 \cdot 6 = 5$	$1 - 6 \cdot (-1) = 7$	$-22 - 6 \cdot 23 = -160$
5	$21 \div 5 = 4$	$21 - 5 \cdot 4 = 1$	$-1 - 4 \cdot 7 =$ **-29**	$23 - 4 \cdot (-160) =$ **663**
6	$5 \div 1 = 5$	$5 - 5 \cdot 1 = 0$	$7 - 5 \cdot (-29) = 152$	$-160 - 5 \cdot 663 = -3475$

Table 1.1.: Extended Euclidean Algorithm used to find the Bézout coefficients for the identity $3475z + 152x = 1$. We obtain $z = -29$ and $x = 663$.

1.3. Ring of integers modulo n

In the quadratic residuosity problem and others, such as the discrete logarithm problem, the set in which the problem is defined is $\mathbb{Z}_n^*$ where n is a large prime number. This provides the necessary robustness to ensure the security of the zero-knowledge proof being used. Let us first define a congruence class modulo n.

Definition 1.3. Given an integer n and $a \in \mathbb{Z}$, a congruence class modulo n is $[a]_n = \{c \in \mathbb{Z} \text{ such that } c \equiv a \pmod{n}\}$.

Remark 1.1. For simplicity reasons, we will represent $[a]_n$ as the representant of the class which is in $\{0, 1, 2, ..., n-1\}$. For instance, $\{-13, -5, 3, 11, 19, 27\} \in [11]_8$, but we represent this class by the representant 3.

We are now prepared to define $\mathbb{Z}_n$.

Definition 1.4. Given an integer n, $\mathbb{Z}_n$ denotes the set of congruence classes modulo n. There are two operations defined on this set. The addition of two classes is defined as $[a]_n + [b]_n = [a+b]_n$ and the multiplication is defined as $[a]_n \cdot [b]_n = [ab]_n$. Note that these relations are well-defined by Proposition 1.1.

Note that using Remark 1.1, we have that $\mathbb{Z}_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$ or $\mathbb{Z}_4 = \{0, 1, 2, 3\}$. Given the congruence relation previously defined, we can give $\mathbb{Z}_n$ a structure.

Proposition 1.2. *For any positive integer n, $\mathbb{Z}_n$, equipped with the congruence relation previously defined, is a ring.*

A property that can be studied of each of the elements in $\mathbb{Z}_n$ is the possession or not of an inverse modulo n. Ensuring all elements have an inverse will play a crucial role in Section 2.6 when proving certain knowledge can be extracted from a zero-knowledge proof. Let us define what an inverse modulo n is.

Definition 1.5. Let $a \in \mathbb{Z}$. We say a has an inverse modulo n, if there exists $b \in \mathbb{Z}$ such that $ab \equiv 1 \pmod{n}$.

Note that not all elements in $\mathbb{Z}_n$ have a multiplicative inverse. For instance, for 2 in $\mathbb{Z}_4$ there is no $b \in \mathbb{Z}_4$ such that $2b \equiv 1 \pmod{n}$. We define the group of units of $\mathbb{Z}_n$.

Definition 1.6. The set of units of $\mathbb{Z}_n$, which is denoted as $\mathbb{Z}_n^*$, is the set of elements of $\mathbb{Z}_n$ that have a multiplicative inverse modulo n.

We can return to one of the examples given before and see that $\mathbb{Z}_8^* = \{1,3,5,7\}$. It is important to note the set $\mathbb{Z}_n^*$ is a group with multiplication modulo n as the operation. This group will be cyclic if and only if n is equal to 2, 4, p^k or $2p^k$ where p^k is a power of an odd prime number (a necessary proof used to demonstrate this statement can be found on page 44 of [KI90]). We can consider a generator α of this cyclic group ($\mathbb{Z}_n^* = \langle\alpha\rangle$), which is called a *primitive root modulo n*, and verifies that for all $\beta \in \mathbb{Z}_n^*$ there exists $x \in \mathbb{N}_0$ such that $\alpha^x \equiv y \pmod{n}$.

When n is any integer, we must also provide an important characterization of the elements in $\mathbb{Z}_n$ that belong to its units set. This characterization is given by the following proposition.

Proposition 1.3. *An element a of $\mathbb{Z}_n$ is a unit, if and only if, a and n are coprimes.*

Let us first suppose that a is a unit in $\mathbb{Z}_n$ and prove that a and n are coprimes. Let us reason by reductio ad absurdum. Suppose a and n are not coprimes. Therefore, there exists $b \in \mathbb{Z}_n$ such that $b \mid a$ and $b \mid n$ and $b > 1$. In other words, there exists $b, c, d \in \mathbb{Z}_n$ such that $n = bc$, $a = bd$ and $b > 1$. We can now multiply n by d obtaining $nd = (cb)d = c(bd) = ca$ and consequently $ca \equiv 0 \pmod{n}$. Using that a has a multiplicative inverse, let us call it $e \in \mathbb{Z}_n$. Then $(ca)e \equiv c(ae) \equiv c \pmod{n}$ and $(ca)e \equiv 0 \cdot e \equiv 0 \pmod{n}$. We conclude that $c \equiv 0 \pmod{n}$ (and cannot be 0 because $n \neq 0$), which is a contradiction if $b > 1$ because b cannot be greater than n.

Another interesting way of proving this is using Bézout's identity (see Lemma 1.2). Suppose again a is a unit, then there exists $b \in \mathbb{Z}_n$ such that $ab \equiv 1 \pmod{n}$. Therefore $1 - ab = kn$ for some $k \in \mathbb{Z}$. Reorganizing the equality we obtain that $1 = ab + kn$. Now suppose there exists some $c \in \mathbb{Z}_n$ such that $c \mid a$ and $c \mid n$, then $c \mid ab + kn = 1$. We conclude that $c = \pm 1$.

Let us now prove the other implication of the proposition. Suppose now that a and n are coprime, and so $\gcd(a, n) = 1$. Thus, using Bézout's identity (see Section 1.2 for details), there exist integers y and z such that $ay + zn = 1$. We can simply express $ay = 1 - zn$ and we note that $ay \equiv 1 - zn \equiv 1 \pmod{n}$. Therefore we have proven there exists an element in $\mathbb{Z}_n$ (which is y) such that $ay \equiv 1$.

A direct consequence of this previous proposition is that any non-zero element of $\mathbb{Z}_n$, where n is a prime number, has a multiplicative inverse modulo n. Therefore, $\mathbb{Z}_n$, where n is a prime number, is a field.

Coming back to the set of units of $\mathbb{Z}_n$, where n is a prime number, we must note that $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$. In general (for any integer n), the size of this set is given by a function introduced by Leonhard Euler in the 18th century. However, its connection to the size of the group of units was not discovered until significant developments in algebraic structures occurred in the 19th century.

Definition 1.7 (Totient function)**.** Given a positive integer m, we define $\varphi(m)$ as the number of positive integers smaller than m and coprime with m. Formally,

$$\varphi(m) = \#\{n \in \mathbb{N} : n \leq m \text{ and } \gcd(m, n) = 1\}.$$

The totient function is used to establish the relation between two coprime integers in the Euler-Fermat Theorem.

Theorem 1.1 (Euler-Fermat Theorem)**.** *Let a and n be two coprime integers, then $a^{\phi(n)} \equiv 1 \pmod{n}$.*

As we mentioned, $\varphi(n) = |\mathbb{Z}_n^*|$ where $|G|$ denotes the order of a group G. We also know that because of Lagrange's theorem, the order of any of $\mathbb{Z}_n^*$'s subgroups divides the order of

$\mathbb{Z}_n^*$. Using that a is coprime to n, we have that $a \in \mathbb{Z}_n^*$. Therefore, the finite cyclic subgroup generated by a is contained in $\mathbb{Z}_n^*$. This means that $a, a^2, \ldots, a^k \in \mathbb{Z}_n^*$ for $k \in \mathbb{N}_0$ the order of a. Since the order of the finite cyclic subgroup generated by a is k, we have that $k \mid \varphi(n)$. This means that there exists $b \in \mathbb{N}_0$ such that $\varphi(n) = kb$. Consequently $a^{\varphi(n)} = a^{kb} = a^{k^b}$, and we already had that $a^k \equiv 1 \pmod{n}$. Therefore, $a^{\varphi(n)} \equiv 1 \pmod{n}$ and the proof is complete.

A direct consequence of this theorem is that $a^{p-1} \equiv 1 \pmod{p}$ for all integer a and prime number p. This consequence is exactly the one that will be used when proving Euler's criterion. This version is also used in many cryptographic tools because it is very common to work with prime numbers.

We will also need to know what relation exists between two integers a and b such that $a^2 \equiv b^2 \pmod{n}$ where n is prime for a proof in Section 2.6. To find that relation, we need to note that because $\mathbb{Z}_n$ is a field when n is prime, it is also a domain. This is true for any field.

Another result that is true for any field is that if $a^2 - b^2 = 0$, then $a = b$ or $a = -b$. This is because $a^2 - b^2 = (a - b) \cdot (a + b) = 0$ and any field is a domain, therefore either $a - b = 0$ or $a + b = 0$ or both. Then, as a direct consequence, we have that given a, b and n be three integers such that n is prime, if $a^2 \equiv b^2 \pmod{n}$, then $a \equiv \pm b \pmod{n}$.

Remark 1.2. The previous implication would be false if we stated the conclusion as $a \equiv b \pmod{n}$. For instance, let $a = 3$, $b = 2$ and $n = 5$. Then, $a^2 \equiv 2^2 \equiv 4 \pmod{5}$ and $b^2 \equiv 3^2 \equiv 9 \equiv 4 \pmod{5}$. So $2^2 \equiv 3^2 \pmod{5}$. However, $a \not\equiv b \pmod{n}$, since $2 \not\equiv 3 \pmod{n}$.

Once the fundamental algebraic structures and their properties that will be used throughout this dissertation are understood, we can proceed to present the operations used to construct cryptographic protocols on them.

1.4. Modular arithmetic roots

Let p be a prime number and a an integer. The equation $x^2 - a$ can have 0, 1 or 2 roots in $\mathbb{Z}_p$ because $\mathbb{Z}_p$ is a field. It is known that if r is a root of $x^2 - a$, then $-r$ is another root of the equation. Therefore, the equation will have only one root if and only if $r = -r$. This occurs when $r = 0$. It also occurs when $r = 1$ and $p = 2$.

Definition 1.8. Let p be a prime number and a an integer. We will say that a is a quadratic residue modulo p if $x^2 - a$ has a root in $\mathbb{Z}_p$.

Similarly, a is a quadratic nonresidue modulo p if $x^2 - a$ does not have a root in $\mathbb{Z}_p$.

The main objective of this section is to allow the reader to understand how relatively simple it is to verify whether an integer is a quadratic residue modulo any other integer with the use of some tools we will present. On the other side, it is computationally challenging to find all the quadratic residues of a large prime number. This discrepancy will be generalized in Section 1.6, and it is robust enough to be the basis on which the first formal example of zero-knowledge proofs was given by Goldwasser, Micali and Rackoff in 1985 (see Subsection 2.2.1).

We will use a tool to help us operate with quadratic residues modulo p called the *Legendre Symbol*. It was introduced by A. M. Legendre in [Leg98]. Later in this section, we will also present the *Jacobi Symbol*, an extension of Legendre's Symbol used to easily calculate it. Let us first start with the Legendre Symbol.

Definition 1.9. Let a be an integer and p a prime number. The *Legendre Symbol* of a and p is defined as:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \mid a, \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p, \\ -1 & \text{otherwise.} \end{cases}$$

Let us now provide some useful properties that will ease the calculus when working with the Legendre Symbol.

Proposition 1.4 (Euler's criterion). *Let a be an integer and p an odd prime number, then $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \mod p$.*

The proof of this proposition illustrates the importance of considering the set of congruence classes $\mathbb{Z}_p$, where p is prime, and how the Euler-Fermat theorem gives a powerful tool to determine the number of quadratic residues.

As we mentioned before, because $\mathbb{Z}_p$ is a field, $x^2 \equiv a \pmod p$ has, at most, 2 solutions for each $a \in \mathbb{Z}_p$. Therefore, there are at least $\frac{p-1}{2}$ different quadratic residues modulo n (besides 0). This is because, at most, each quadratic residue can be accompanied by another value (if r is one, -r is the other). More formally, $(p-x)^2 \equiv x^2 \pmod p$ because $p^2 - 2px + x^2 \equiv x^2 \pmod p$. Therefore, we have a lower bound to the number of quadratic residues.

We now obtain an upper bound to the number of quadratic residues. Since p is prime, a is coprime with p and, using Fermat's Theorem (see Theorem 1.1), $a^{p-1} \equiv 1 \pmod p$. We can subtract 1 from both sides of the congruence, obtaining $a^{p-1} - 1 \equiv 0 \pmod p$, and using that $b^2 - c^2 = (b-c)(b+c)$, then $(a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \equiv 0 \pmod p$. Now, using that $\mathbb{Z}_p$ is a field, one of both terms in the previous product must be congruent with 0 and therefore, either of

$$\begin{cases} a^{\frac{p-1}{2}} \equiv 1 \pmod p \text{ or} \\ a^{\frac{p-1}{2}} \equiv -1 \pmod p, \end{cases} \tag{1.1}$$

holds. When a is a quadratic residue modulo p ($a \equiv x^2 \bmod p$), $(x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \pmod p$. Therefore, the first factor of the product shown before is congruent with 0. Considering the first equation of (1.1) is built on the field $\mathbb{Z}_p$, we are sure that the maximum number of roots of $a^{\frac{p-1}{2}} - 1 \equiv 0 \pmod p$ is at most $\frac{p-1}{2}$ (because it is a polynomial of order $\frac{p-1}{2}$). This, added to the fact that we proved there were at least $\frac{p-1}{2}$ distinct quadratic residues, leads to the conclusion that there are precisely $\frac{p-1}{2}$ distinct quadratic residues modulo p (excluding 0). Furthermore, when a is not a quadratic residue, the second equation in (1.1) must hold because we are working in a domain and we had $(a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \equiv 0 \pmod p$. Because any non-zero element in $\mathbb{Z}_p$ is either a quadratic residue or a quadratic non-residue, and we noted that the first equation in (1.1) holds when a is a quadratic residue, then the second equation must hold when a is a quadratic non-residue. We are sure both can not be true simultaneously because p is odd. If $a^{\frac{p-1}{2}} + 1 \equiv 0 \pmod p$ holds then, $a^{\frac{p-1}{2}} \equiv -1 \pmod p$.

In conclusion,

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod p & \text{if there is an integer x such that } x^2 \equiv a \pmod p, \\ -1 \pmod p & \text{if there is no such integer.} \end{cases}$$

This is equivalent to writing that, $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \mod p$, using the Legendre symbol notation

introduced before. This concludes the proof.

An interesting alternative proof for this theorem can also be done using some previously presented properties. We can assume $a \neq 0$ (when $a = 0$ the proof becomes trivial) and observe that because of Fermat's Theorem (see Theorem 1.1) $a^{p-1} \equiv 1 \pmod{p}$, and consequently $(a^{\frac{p-1}{2}})^2 \equiv 1 \pmod{p}$. Therefore, $a^{\frac{p-1}{2}}$ is a square root of 1 and for that reason $a^{\frac{p-1}{2}} \equiv \pm 1 \pmod{p}$. This implies that proving that $\left(\frac{a}{p}\right) = 1$ if and only if $a^{\frac{p-1}{2}} \equiv 1$ we will prove the proposition.

Suppose now that $\left(\frac{a}{p}\right) = 1$. Then there exists $b \in \mathbb{Z}_p$ such that $b^2 \equiv a \pmod{p}$. It is direct to see that $a^{\frac{p-1}{2}} \equiv b^{2\frac{p-1}{2}}$ and simplifying the exponent we obtain $b^{2\frac{p-1}{2}} = b^{p-1}$. Using Fermat's Theorem, we conclude that $b^{p-1} \equiv 1$ and therefore $a^{\frac{p-1}{2}} \equiv 1$.

Let us prove the other implication. Assume $a^{\frac{p-1}{2}} \equiv 1$ and let α be a primitive root modulo p ($\mathbb{Z}_p^* = \langle \alpha \rangle$). Because $a \in \mathbb{Z}_p^*$ there exists $i \in \mathbb{N}_0$ such that $a = \alpha^i$. Substituting we obtain $(\alpha^i)^{\frac{p-1}{2}} \equiv 1$. This implies that $\text{ord}(\alpha) \mid i\frac{p-1}{2}$ where $\text{ord}(\alpha)$ is used denote α's order. This comes from the fact that for all $x \in \mathbb{Z}_p^*$ we have that $\text{ord}(x)$ is the smallest exponent such that $x^{\text{ord}(x)} \equiv 1 \pmod{p}$. We now use that $\text{ord}(\alpha) = p - 1$ because α is a generator of a cyclic group of size $p - 1$. We have obtained that there exists $k \in \mathbb{Z}$ such that $(p-1)k = i\frac{p-1}{2}$. It is direct to see that $i = 2k$ obtaining that $\alpha^{2k} = a$. This means $(\alpha^k)^2 = a$. We conclude that a is a quadratic residue modulo p and therefore $\left(\frac{a}{p}\right) = 1$, which completes the proof.

A direct consequence of Euler's criterion is that when n is a prime number, half of the elements in $\mathbb{Z}_n^*$ are quadratic residues, while the other half are not. This will be used when determining the probability of randomly creating a zero-knowledge proof without possessing the secret being proven. To grasp this idea practically, consider the example in Subsection 2.1.2. The probability of guessing the correct exit path of the cave is $\frac{1}{2}$. Here, the probability of stating a member of $\mathbb{Z}_n^*$ is a quadratic residue is also $\frac{1}{2}$.

This fact strengthens the computational complexity idea mentioned previously regarding the brute force method of finding whether an integer is a quadratic residue or not. The property proven also eases the calculus needed to verify if an integer is indeed a quadratic residue. Let us present now some basic properties that will help us do this verification.

Proposition 1.5. *Let a and b be integers and p a prime number. Then, the following properties hold true.*

1. $\left(\frac{1}{p}\right) = 1$.
2. $\left(\frac{a}{p}\right) = \left(\frac{a \bmod p}{p}\right)$.
3. $\left(\frac{a \cdot b}{p}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right)$.
4. $\left(\frac{a^2}{p}\right) = 1$ *if* $\gcd(a, p) = 1$.

The first property given in Proposition 1.5 is demonstrated by seeing that $x^2 \equiv 1 \pmod{p}$ taking $x = 1$. The second property can be proven by seeing that the equation $x^2 \equiv a \pmod{p}$ has exactly the same roots as $x^2 \equiv (a \bmod p) \pmod{p}$. We can directly prove the third property by using Euler's criterion. We have that $(\frac{a \cdot b}{p}) = (a \cdot b)^{\frac{p-1}{2}} \pmod{p}$. Therefore, $(\frac{a \cdot b}{p}) = a^{\frac{p-1}{2}} \cdot b^{\frac{p-1}{2}} \pmod{p}$. Using Proposition 1.1, $(\frac{a \cdot b}{p}) = (a^{\frac{p-1}{2}} \pmod{p}) \cdot (b^{\frac{p-1}{2}} \pmod{p})$.

In conclusion, and using Euler's criterion once more, $(\frac{a\cdot b}{p}) = (\frac{a}{p})\cdot(\frac{b}{p})$. Last, we suppose $\gcd(a,p)=1$ to prove the fourth property. Therefore, $a^{2\frac{p-1}{2}} \pmod p = a^{p-1} \pmod p$ and because of Fermat's Theorem (see Theorem 1.1), $a^{p-1} \equiv 1 \pmod p$. Consequently $\left(\frac{a^2}{p}\right) = 1$.

Another useful tool when finding the Legendre symbol of a number is the Law of quadratic reciprocity.

Proposition 1.6 (Law of quadratic reciprocity)**.** *Let p and q be two odd prime numbers. Then,* $\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)\cdot(-1)^{\frac{(p-1)(q-1)}{4}}$.

Einstein provided an elegant geometrical proof for this proposition, and a simplified version can be found in [Lyn23]. Another less geometrical proof can be found on page 77 of [HW59].

Let us see how all these properties can be put to use to find if an integer is a quadratic residue modulo another prime number.

Example 1.3. Let us find $(\frac{31}{103})$. First, we use the Law of quadratic reciprocity to obtain that $(\frac{31}{103}) = -(\frac{103}{31})$. Then, using property 2 of Proposition 1.5, we see that $-(\frac{103}{31}) = -(\frac{10}{31})$. Using now the product property from the same proposition $-(\frac{10}{31}) = -(\frac{2}{31})\cdot(\frac{5}{31})$ is obtained. Using that $2^5 \equiv 1 \pmod{31}$ and Eueler's criterion leads to $-(\frac{2}{31})\cdot(\frac{5}{31}) = -(\frac{5}{31})$. We can conclude by applying the Law of quadratic reciprocity and properties 2 and 1, such that $-(\frac{5}{31}) = -(\frac{31}{5}) = -(\frac{1}{5}) = -1$.

However, and looking at this example, some readers may point out that this method is flawed as we need to rely on factoring, so we may as well find all the squares modulo 103 and check whether 31 is found amongst them. Therefore, finding and verifying if an integer is a quadratic residue would be as computationally burdensome. To avoid this kind of problem, the Legendre symbol is extended by the Jacobi Symbol, which takes an odd number (instead of a prime number) as the second argument. This symbol significantly simplifies the calculus needed to compute the Legendre Symbol, verifies all its properties and adds some more.

Definition 1.10. The *Jacobi Symbol* of an integer a and an odd number n is defined as

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1}\cdot\left(\frac{a}{p_2}\right)^{e_2}\cdot\left(\frac{a}{p_3}\right)^{e_3}\cdot\left(\frac{a}{p_4}\right)^{e_4}\cdots\left(\frac{a}{p_k}\right)^{e_k},$$

where $n = {p_1}^{e_1}\cdot{p_2}^{e_2}\cdot{p_3}^{e_3}\cdot{p_4}^{e_4}\cdots{p_k}^{e_k}$.

Besides the Legendre Symbol's properties, there are some specific ones that are associated to the Jacobi Symbol. These properties will play a crucial role when avoiding factoring when finding whether an integer is a quadratic residue or not.

Proposition 1.7. *Let a and b be odd integers. Then, the following properties hold true.*

1. $\left(\frac{-1}{b}\right) = (-1)^{\frac{b-1}{2}}$.
2. $\left(\frac{2}{b}\right) = (-1)^{\frac{b^2-1}{8}}$.
3. $\left(\frac{a}{b}\right)\cdot\left(\frac{b}{a}\right) = (-1)^{\frac{a-1}{2}\frac{b-1}{2}}$.

The proofs for these properties can be found in [Bis14]. However, we are more interested in seeing how the properties are used. To see that, we will find the value of $(\frac{31}{103})$ using them. Similarly to what we did in Example 1.3, $(\frac{31}{103}) = -(\frac{103}{31})$. Considering that $103 \equiv -21 \pmod{31}$, we have that $-(\frac{103}{31}) = -(\frac{-21}{31})$, and using the product property inherited from the Legendre Symbol, we have that $-(\frac{-21}{31}) = -(\frac{-1}{31}) \cdot (\frac{21}{31})$.

Using property 1 of Proposition 1.7 and the property 3 of Proposition 1.7 we obtain $-(\frac{-1}{31}) \cdot (\frac{21}{31}) = (\frac{31}{21})$. Now, using $31 \equiv -11 \pmod{21}$ we obtain that $(\frac{31}{21}) = (\frac{-11}{21})$. Applying the product property again $(\frac{-11}{21}) = (\frac{-1}{21}) \cdot (\frac{11}{21})$ and using property 1 of the Jacobi Symbol $(\frac{-1}{21}) \cdot (\frac{11}{21}) = (\frac{11}{21})$.

Last, we use property 3 of Proposition 1.7 concluding that $(\frac{11}{21}) = (\frac{21}{11})$ and using that $21 \equiv -1 \pmod{11}$ implies $(\frac{21}{11}) = (\frac{-1}{11})$ it is clear that it is 21 is not a quadratic residue mod 103 for property 1 of the Jacobi Symbol.

Remark 1.3. If q is a product of odd prime numbers, a is any integer and $\left(\frac{a}{q}\right) = 1$, then a may not always be a quadratic residue modulo q. Note that the converse is true because it derives from the definition of the Legendre Symbol.

Therefore, we derive that if the Jacobi symbol is different to 1 we are sure that a is a quadratic non-residue modulo q. However, if it is equal to 1 a could be a quadratic non-residue. For instance, see Example 1.4.

Example 1.4. Let us see that the Jacobi symbol $\left(\frac{2}{15}\right) = 1$, but 2 is not a quadratic residue modulo 15. See that

$$\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right)\left(\frac{2}{5}\right) = (-1) \cdot (-1) = 1,$$

but all the squares modulo 15 are given in Table 1.2.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x^2 \pmod{15}$	0	1	4	9	1	10	6	4	4	6	10	1	9	4	1

Table 1.2.: Squares modulo 15.

As we can see, the only square roots modulo 15 are 4, 11 and 14. However, the Jacobi symbol $\left(\frac{2}{15}\right) = 1$.

Some of these definitions and lemmas regarding the Legendre and Jacobi Symbols will not be explicitly used when presenting the quadratic residuosity example. Still, they will be helpful if the reader wants to program or verify the example empirically and are definitely primordial to show that verifying the quadratic residuosity property is indeed simpler than finding quadratic residues.

We can continue by studying a problem that could be considered inverse to the one presented in this section: the discrete logarithm problem.

1.5. Discrete logarithms

Given three integers a, c and m, we wonder if there is some b, such that $a^b \equiv c \pmod{m}$. This is equivalent to finding b, such that $a^b = c$ in $\mathbb{Z}_m$. The value b is called the discrete logarithm of c to the base a modulo m.

Once again, it is simple to verify whether an integer is the discrete logarithm of another, but it is computationally complex to solve the problem. The zero-knowledge proof example given in Subsection 2.6.3 is built on this discrepancy and it is one of the most illustrative examples when demonstrating the characteristics of a zero-knowledge proof are satisfied.

Verifying if an integer is the discrete logarithm of another is computationally simple because modular exponentiation is computationally efficient. One of the most commonly used algorithms that improve the direct method's efficiency is the *Right-to-left binary method* [Em23], whose efficiency is $O(\log_2(\text{exponent}))$ (see Section 1.6 for the big O notation). In both cases, the verification complexity is independent of the sample space size (how large m is).

We can see the conceptual simplicity of the *Right-to-left binary method* following the explanation given in [Em23]. Assume we aim to raise a to the power of n. This can be expressed as $n-1$ products such as $a^n = a \cdot a \cdot \cdots \cdot a$. The number of multiplications needed can be reduced by expressing the exponent n in binary and using one basic exponentiation property: $a^{n+m} = a^n \cdot a^m$. From this property, it is direct to see that $a^{2n} = a^{n^2}$. Let us see how the number of multiplications is reduced by studying the process that must be followed for finding 5^{29}.

We first see that 29 is 11101_2 in binary (w_2 is used to note the binary representation of a number). We can use the property given above to express $5^{29} = 5^{11101_2} = 5^{16} \cdot 5^8 \cdot 5^4 \cdot 5^1$. We now just need to know the previous product factors' values, which are easily computable as we can recursively compute the sequence $\{5^1, 5^2, 5^4, 5^8, 5^{16}\}$. This can be done by starting with 5, and subsequently squaring the result of squaring that value. Because of that, in the previous example, we must also compute 5^2 even though we will not use it in the product. It is important to note that squaring only involves a single multiplication. Because the length of n in base two is $\lfloor \log_2(n) \rfloor + 1$ (where $\lfloor b \rfloor$ denotes the floor function given a real number b), we know that we must compute $O(\log_2(n))$ multiplications to obtain the product, and $O(\log_2(n))$ multiplications to obtain the powers. This is a lower number of multiplications than in the direct method (multiplying a by itself $n-1$ times).

The above-presented exponentiation method could be used in the brute force approach to solving this problem (finding a discrete logarithm if a solution exists). This approach consists of inserting values for b and checking if $a^b \bmod m = c$. However, the number of exponents to try is bounded above by $\varphi(m)$, with φ being the φ Euler's function or totient function (see Definition 1.7). This implies that the number of possible values that need to be checked is considerably high, even for small values of m (see Table 1.3).

m	12	1234	1234	1234567	123456789
$\varphi(m)$	4	616	616	1224720	82260072

Table 1.3.: Totient function evaluated in some integers.

In most of the examples we will present in future sections, we will be working with $\mathbb{Z}_n^*$ where n is a big prime number. As it was explained in Section 1.3, this set is a cyclic group with the multiplication modulo n. The size of the group makes solving the discrete logarithm problem unfeasible. However, there exists several efficient algorithms that can be used when the group's size is not too big. The Baby-step giant-step algorithm, whose time complexity is $O(\sqrt{|\mathbb{Z}_n^*|\}})$ (see Section 1.6 for the big O notation) where $|G|$ denotes the order of the group G, is an example of those algorithms . The algorithm was introduced by Dan Shanks in 1969 [Sha71].

Let G be a finite cyclic group of positive order k and a generator $\alpha \in G$. Therefore, we have that $G = \langle \alpha \rangle = \{\alpha^i \text{ such that } i \in \{0, 1, \ldots, k-1\}\}$. Let $\beta \in G$ and suppose we want to find $x \in \mathbb{N}_0$ such that $\alpha^x = \beta$.

We express $x = im + j$ where $i \in \{0, 1, 2, \ldots, m-1\}$ and $j \in \{0, 1, 2, \ldots, m-1\}$ and $m = \lceil \sqrt{k} \rceil$. Note that given a real value y, $\lceil y \rceil$ denotes the ceiling function. This function maps the value y to the smallest integer greater or equal to y. Using $x = im + j$ we have that $\alpha^{im+j} = \beta$. Then $\alpha^{im}\alpha^j = \beta$ and we conclude $\alpha^j = \beta(\alpha^{-m})^i$. Once this is done, the algorithm precomputes the values of the equality's left side and tests different values of i to see if the equality (which is really a congruence) holds.

Let us see how the actual algorithm works.

1. Compute $m = \lceil \sqrt{k} \rceil$.
2. Construct a table of baby steps.

$$\{(j, \alpha^j) \mid j \in \{0, 1, 2, \ldots, m-1\}\}.$$

3. Compute α^{-m} using the inverse of α in G.
4. Define $\gamma = \beta$.
   For each $i \in \{0, 1, 2, \ldots, m-1\}$.
5. Check if γ matches any α^j from the baby step table.
   a) If a match is found at $\alpha^{im} = \alpha^j$, then return $x = im + j$ and end the algorithm.
   b) If no match is found, $\gamma = \gamma \cdot \alpha^{-m}$.

Note the discrete logarithm always exists because G is a finite cyclic group, and therefore, the algorithm will always finish in step 5.a.

Example 1.5. Let G be the multiplicative group of integers modulo 29. We want to find x such that $2^x \equiv 11 \pmod{29}$. We will use the Baby-step giant-step algorithm.

1. First, we find $m = \lceil \sqrt{29} \rceil = 6$.
2. Second, we compute the table of baby steps.

$$\begin{aligned}
\alpha^0 &= 2^0 \equiv 1 \pmod{29}, \\
\alpha^1 &= 2^1 \equiv 2 \pmod{29}, \\
\alpha^2 &= 2^2 \equiv 4 \pmod{29}, \\
\alpha^3 &= 2^3 \equiv 8 \pmod{29}, \\
\alpha^4 &= 2^4 \equiv 16 \pmod{29}, \\
\alpha^5 &= 2^5 \equiv 3 \pmod{29}.
\end{aligned}$$

   Obtaining $T = \{(0,1), (1,2), (2,4), (3,8), (4,16), (5,3)\}$.
3. Third, we compute $2^{-6} \pmod{29} \equiv (2^{-1})^6 \pmod{29} \equiv 15^6 \pmod{29} \equiv 5$.
4. Fourth, we define $\gamma = 11$.

5. Fifth, we check if $11 \equiv \alpha^j \pmod{29}$ for some $j \in \{0, 1, 2, 3, 4, 5\}$.

   We look into T and verify this does not occur, and we need to redefine $\gamma = 11 \cdot 5 \pmod{29}$. Now we have $\gamma = 26 \pmod{29}$, which is not found in any of the pairs in T.

   We need to redefine $\gamma = 26 \cdot 5 \pmod{29}$. Now we have $\gamma = 14 \pmod{29}$, which is not found in any of the pairs in T.

   We need to redefine $\gamma = 14 \cdot 5 \pmod{29}$. Now we have $\gamma = 12 \pmod{29}$, which is not in any of the pairs in T.

   We need to redefine $\gamma = 12 \cdot 5 \pmod{29}$. Now we have $\gamma = 2 \pmod{29}$, which is found in the pair $(1, 2) \in T$.

Therefore, for $i = 4$, we have found an element in the table T. The algorithm concludes that $x = 4 \cdot 6 + 1 = 25$ and we can verify this is a correct solution because $2^{25} \equiv 11 \pmod{29}$.

1.6. Computational complexity theory

Cryptographic protocols often rely on mathematical problems that are easy to verify but computationally difficult to solve efficiently. Examples of these are the quadratic residuosity problem and the discrete logarithm problem, as previously explained in the corresponding sections.

In this section, we will define what we understand as computationally hard and computationally efficient. It is called computational complexity theory the branch of computer science that studies and classifies computational problems based on the resources needed to solve these problems.

Definition 1.11. A computational problem is a defined task usually solved by a computer following an algorithm.

Where an algorithm is an ordered set of mathematical steps. For instance, determining if a number is prime is a computational problem that can be solved following an algorithm such as the Eratosthenes Sieve.

The two most common resources that are measured in computational complexity theory are time and space. Both of them are measured relative to the size of the input fed to the algorithm. The commonly used notation is referred to as *big O notation.*

Definition 1.12. Let f and g be two real-valued functions defined on some unbounded set of the real positive numbers, and g is strictly positive for all large enough values of its domain. Then, we say $f(x) = O(g(x))$ when $x \to \infty$ if there exists $M \geq 0$ and a real value x_0 such that $|f(x)| \leq g(x)$ if $x \geq x_0$.

When analyzing algorithms, the functions used as g are usually simple functions such as n^2 or $n^3 \log(n)$. Constants and lower-order terms are usually omitted because they will become negligible when the size becomes very large.

It is also worth mentioning that different machine models in which the algorithm can be run usually only vary in a constant number of steps. Because these terms will become irrelevant when the size tends to infinity, these technical differences between systems do not affect the big O notation.

Example 1.6. Let us consider the computational problem of finding the square root of a number of size n (the number has n digits). This is known to be computed in time $O(M(n))$ being $M(n)$ the complexity of the number of steps taken to multiply two numbers.

Based on the big function that is used with the big O notation problems are classified in complexity classes (see Table 1.4).

Resource	Determinism	Complexity class	Resource constraint
Time	Deterministic	DTIME($f(n)$)	$O(f(n))$
		P	$O(\text{poly}(n))$
		NEXPTIME	$O(2^{\text{poly}(n)})$
	Non-Deterministic	NTIME($f(n)$)	$O(f(n))$
		NP	$O(\text{poly}(n))$
		NEXPTIME	$O(2^{\text{poly}(n)})$

Table 1.4.: Complexity classes classification extracted from [Wik24b] where poly is a polynomial and f is any real-valued function.

Remark 1.4. Solving algorithms are considered to be efficient if they take polynomial time ($O(p(n))$ where p is a polynomial.

Conversely, problems that are not considered efficient are considered to be computationally hard. These two terms can be used with greater precision using the big O notation and an exact real-valued function. It is worth highlighting the NP class is the one that groups all the problems that we will solve using zero-knowledge proofs

Remark 1.5. NP-problems are those that are easy to verify that a satisfying statement is true (polynomial time) but hard to prove that a given statement satisfies the problem.

As we already mentioned, it is this computational difficulty discrepancy what gives robustness to the cryptographic protocols. If someone in the future finds a method to solve these problems efficiently, the proofs that rely on their computational hardness will lose their soundness property.

Throughout this dissertation, we will commonly use the term *probabilistic polynomial-time* to refer to algorithms or computational processes that, with access to random bits, can solve problems in polynomial time on average or with high probability. These algorithms will usually be modeled using Turing Machines. The following section presents formal languages and how these machines are used to work with them.

1.7. Language theory

Although proving that we know the location of Waldo without revealing his exact location is appealing, it is not so simple to formally demonstrate that it is (almost) zero-knowledge. Consequently, proofs and statements are usually constructed around formal languages because of their theoretical simplicity and computational friendliness.

Definition 1.13. A formal language L is a set of words constructed with letters taken from an alphabet following a set of well-defined rules called formal grammar.

For instance, $L_1 = \{a^n b^n : a, b \in \{0, 1\}\}$ is an example of a formal language.

These languages facilitate the technical expression of problems so they can be easily understood by computers. All the examples we will present have the same structure. A prover wants to prove to a verifier that some element belongs to a formal language. Deciding whether the element belongs to the language is called a decision problem.

Definition 1.14. A decision problem is a yes-no question based on the value of the problem's input.

For instance, whether an integer is a perfect square is an example of a decision problem. Decision problems are usually represented by Turing Machines. This representation normalizes the problem, providing it with a mathematical and computational rigorous structure.

Definition 1.15. A Turing Machine is a theoretical model of computation proposed by Alan Turing. In its most basic configuration, it has an input tape, output tape, set of rules and set of states, which are used to prove if a word belongs to a predefined language.

There are two types of Turing Machines. A Turing Machine is non-deterministic when the state to which it can transition is not completely defined by the previous state. It has more than one option. On the other hand, a Turing Machine is deterministic when the previous state completely defines the state to which it can transition.

Decision problems, as a sub-type of computational problems, can be classified based on their complexity in complexity classes. The time a Turing machine takes on some defined input is the total number of steps (or number of state transitions) taken before the machine halts. When a machine halts, it returns a positive or negative answer to the decision problem. Based on the number of steps taken, a decision problem can be classified following the criteria of Table 1.4.

Turing Machines are commonly used when demonstrating the properties that make a general proof a zero-knowledge proof. Although we will not study all their complexities, we will frequently make assumptions about the data accessible by each machine and their honesty. This will enable us to analyze how much information is leaked during a zero-knowledge proof and how a machine will behave during a normal iteration of the protocol. More details on how these assumptions will be made and the particular structure of some Turing Machines will be given when necessary.

1.8. Graph theory

We present some fundamental graph theory definitions as they will be used in the examples found in Subsection 2.2.2 and Subsection 2.6.4. This last example is particularly important because there is an important result, whose proof escapes the scope of this dissertation, that states that any NP problem accepts a zero-knowledge proof by reducing it to the graph colouring problem.

Definition 1.16. A graph G_1 is a pair (V, E), where V is a non-empty set of vertices (or nodes) and E is an unordered set of edges, where each edge is an unordered pair of vertices, which are called endpoints, of that given edge.

In Subsection 2.2.2, two graphs will be given, and a decision problem will be presented, asking whether they are isomorphic. The graph isomorphism problem is considered to be NP-intermediate.

Definition 1.17. Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are said to be isomorphic if there exists a bijective application $\phi : V_1 \to V_2$ such that for any pair of vertices u and v in V_1, $\{u, v\}$ is an edge in E_1, if and only if, $\{\phi(u), \phi(v)\}$ is an edge in E_2.

Coming back to the example in Subsection 2.6.4, we need to define formally when a graph is 3-colourable and how this colurability varies under permutation. Unlike the previous example, this decision problem is NP-complete.

Definition 1.18. A graph, $G = (V, E)$, is considered 3-colourable if its vertices can be coloured with only three colours, such that no two adjacent vertices are the same colour.

At last, we conclude with two important remarks on the colourability property that will be used during the zero-knowledge proof.

Remark 1.6. Any permutation of the three colours used results in a new valid 3-colouring of the graph.

Remark 1.7. Any not 3-colourable graph has at least one edge that connects two vertices of the same colour.

1.9. Hash functions

A hash function is any function that can map inputs of arbitrary size to an output of fixed size. For instance, the function's input could be the last sentence, and the output would be a string of 128, 256 or 512 bits.

We present these functions as they will be used in Subsection 2.7.2 and especially in Subsection 2.8.1, in the zero-knowledge part of this dissertation, and also when working with the Blockchain in Chapter 4. The impossibility of finding a preimage given a value the function takes is the hash functions' property exploited in the three previously mentioned parts. Besides that property, which we will present formally, there are some basic characteristics that a hash function must satisfy. Let h be a hash function.

- **Ease obtaining the image of an input**. Given a message m, it should be computationally simple and quick to find the value of $h(m)$.
- **Unidirectionality**. Given $y = h(m)$, it should be computationally unfeasible (or very costly) to compute m.
- **Diffusion**. The function output must depend on all the input bits. A change of one bit in the input should cause the output to be completely different.
- **Collision resistance**. A collision occurs when two different preimages lead to the same image after applying the hash function. There are two Proposition 1.1 types of collision resistance.
  - Weak collision resistance. Given a message m, it is computationally unfeasible to find another m' such that $h(m) = h(m')$.
  - Strong collision resistance. It is computationally unfeasible to find arbitrary messages m and m', such that $h(m) = h(m')$.

Some examples of widely used hash functions are SHA256, SHA512 and SHA-3.

Hash functions will be used for obfuscation purposes when aiming to commit to a value, in commitment schemes, or concisely represent information in blockchain technologies such as Bitcoin. This obfuscation will also be desirable when demonstrating knowledge of a statement in a zero-knowledge proof. The proof of a SNARK will be represented by a polynomial and a set of instances and statements. Both these pieces of information will need to behave in a certain way in order to demonstrate knowledge of a statement. However, in order to achieve the zero-knowledge property, they cannot be sent plainly. In summary, we will try to send some kind of obfuscated data that behaves exactly as the plain data, and there is no possibility of recovering the plain data from the obfuscated one. Here is where elliptic curves and bilinear mappings come into play. Let us present the mathematical fundamentals we will need.

1.10. Elliptic curves

A whole dissertation could be solely written about elliptic curves (it has already been done). However, we will present some basic concepts that will be key to understanding how zk-SNARKs will achieve the zero-knowledge property (see Section 3.3). Furthermore, elliptic curve cryptography is widely used in the most up-to-date cybersecurity solutions when defining the private-public key pairs (see Subsection 4.2.2). Let us start with a definition.

Definition 1.19. An elliptic curve is the set of points of $\mathbb{F} \times \mathbb{F}$, where $\mathbb{F}$ is a field, that satisfy an equation in two variables and a special point denoted as ω, which we will later define. Given a and b any two real values, the equation can be written as

$$\mathrm{E}_{a,b}(\mathbb{F}) \equiv y^2 = x^3 + ax + b,$$

when the characteristic of the field on which it is built differs from 2 and 3. We also impose that the discriminant $4a^3 + 27b^2 \neq 0$ to avoid singularities. This curve is also referred to as the Short Weierstrass elliptic curve.

A binary operator is defined over this set of points. Let P and Q be two points belonging to the elliptic curve. Then, we define $P \oplus Q$ as another point on the curve obtained by intersecting the vertical line drawn from the intersection of the line connecting P and Q and the curve (see Figure 1.1 and note how the y and x-axis of the graph in the figure are not on the same scale for visual purposes).

If both P and Q are in the same vertical line, the line connecting both points does not intersect the curve at any other point. How would we define $P \oplus Q$, then? To solve this problem, a point ω in the infinite is defined (as done in the projective space). This point will act as the neutral identity of the group $(\mathrm{E}_{a,b}(\mathbb{F}), \oplus)$.

One way of simplifying the calculus without relying on projective notation is by choosing a specific type of curves with group laws that do not differentiate between cases and use as few field operations as possible. This will be very handful when working with zk-SNARKs as the number of branches of the program will be drastically reduced. Some of these curves are the so-called SNARK-friendly Twisted Edwards Curves.

Definition 1.20. Given $a, b \in \mathbb{F}$ a finite field of characteristic q, where $a \neq b$, a SNARK-

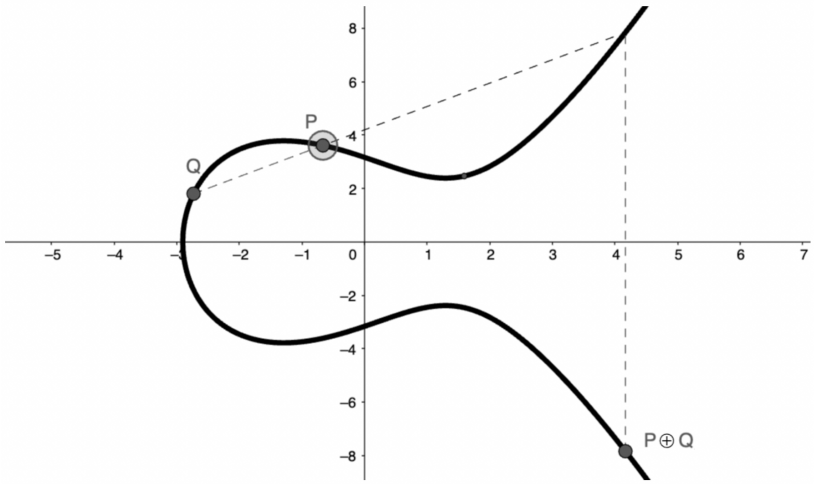


Figure 1.1.: Elliptic curve $y^2 = x^3 - 5x + 10$ generated in [San23] with the points P, Q and $P \oplus Q$ plotted on it.

friendly Twisted Edwards Curve is defined as

$$\mathrm{E}_{a,b}(\mathbb{F}) = \{(x, y) \in \mathbb{F} \times \mathbb{F} \text{ such that } a \cdot x^2 + y^2 = 1 + b \cdot x^2 \cdot y^2\},$$

where $\left(\frac{a}{q}\right) = 1$ and $\left(\frac{b}{q}\right) = -1$.

The point $(0, 1)$ belongs to any such curve, and it is the point at infinity. As mentioned before, the group law of this type of curve is simpler than the mathematical formulation of the visual representation shown in Figure 1.1. Given two elements $(x_1, y_1), (x_2, y_2) \in \mathrm{E}_{a,b}(\mathbb{F})$ it is defined as shown in (1.2).

$$(x_1, y_1) \oplus (x_2, y_2) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + b x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - b x_1 x_2 y_1 y_2}\right). \tag{1.2}$$

As previously mentioned, elliptic curves will be used with bilinear mappings for obtaining that desired obfuscation. Let us succinctly define bilinear mappings and see how they work with elliptic curves.

Definition 1.21. Let $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_3$ be three conmutative groups. A bilinear mapping is a function $\mathrm{e}(\cdot, \cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ that is bilinear for the product operation in groups $\mathbb{G}_1$ and $\mathbb{G}_2$. This means that given $(x_1, x_2), (y_1, y_2) \in \mathbb{G}_1 \times \mathbb{G}_2$, then

$$\mathrm{e}(x_1 \cdot y_1, x_2) = \mathrm{e}(x_1, x_2) \cdot \mathrm{e}(y_1, x_2) \text{ and } \mathrm{e}(x_1, x_2 \cdot y_2) = \mathrm{e}(x_1, x_2) \cdot \mathrm{e}(x_1, y_2).$$

Every elliptic curve has a bilinear mapping associated with it. However, it is important to note that not all curves give rise to efficiently computable mappings. To identify those curves, we must present the embedding degree of a curve.

Definition 1.22. Let $\mathrm{E}_{a,b}(\mathbb{F}_q)$ be an elliptic curve over a finite field $\mathbb{F}_q$ of order $\#\,\mathrm{E}_{a,b}(\mathbb{F}_q) = n$,

and r a prime factor of n. The embedding degree of $E_{a,b}(\mathbb{F}_q)$ with respect to r is the smallest integer k such that $r \mid q^k - 1$.

Assuming r does not divide q, there is always an embedding degree $k(r)$ for each such r. This is because the problem is equivalent to trying to find $k(r)$, such that $1 \equiv q^{k(r)} \pmod{r}$. Taking $k(r) = r - 1$, we have that $k(r) = \varphi(r)$ because r is prime. We also know that, because $\mathbb{F}_q$ is a finite field, q must be the power of a prime p (there exists $c \in \mathbb{N}$ such that $q = p^c$). We conclude the proof by using Euler Fermat's theorem (see Theorem 1.1) and that r and p are coprime (because r does not divide q).

If r divides q, then r must be equal to p (note that r is prime). In that case, we can still always choose $k(r) = 0$. However, r is always different from q to avoid trivial cases.

In the example we will present in Chapter 3, we will work with a curve of embedding degree 6 as it is known this embedding degree allows for efficient computations (see [BLS02]).

We must first extend an elliptic curve to define a structure called *r-torsion group*. Given a finite field $\mathbb{F}_p$, where p is a prime number, we can define the extension of this field by considering a polynomial $q(x) \in \mathbb{F}_p[x]$ and taking the quotient space $\frac{\mathbb{F}_p[x]}{q(x)}$. This quotient will be isomorphic to $\mathbb{F}_{p^d}$ where $\deg(q(x)) = d$ and it will be formed by all the polynomials in $\mathbb{F}_p[x]$ modulo $q(x)$. This is important as it can be proven that the $E_{a,b}(\mathbb{F}_p) \subset E_{a,b}(\mathbb{F}_{p^d})$.

We are now getting closer to the precise mathematical tools used when working with zk-SNARKs. So far, we have presented elliptic curves over finite fields and bilinear mappings. It could be thought that the two groups over which we will define the mapping are two elliptic curves. However, they will be defined over their torsion groups as they have more cryptographically interesting properties.

Definition 1.23. The r-torsion group of a elliptic curve $E_{a,b}(\mathbb{F}_p)$ is defined as

$$E_{a,b}(\mathbb{F}_p)[r] := \{S \in E_{a,b}(\mathbb{F}_p) \text{ such that } [r]S = \omega\},$$

where r is a factor of the group's order and $[r]S$ is the resulting point of adding S r times.

Because of the fundamental theorem of finite cyclic groups, we know that every factor r of the group's order defines a subgroup. We will specifically focus on two subgroups of the full r-torsion group of the elliptic curve, which is defined as

$$E_{a,b}[r] := E_{a,b}(\mathbb{F}_{p^{k(r)}})[r].$$

This group contains r^2 elements and has $r + 1$ subgroups. Furthermore, all the r-torsion groups $E_{a,b}(\mathbb{F}_{p^m})[r]$ such that $m > k(r)$ are all equal to $E_{a,b}[r]$.

We will use the Frobenius isomorphism (see (1.3)) to identify the two particular subgroups we will be interested in. We will work with the Frobenius isomorphism on elliptic curves over extension fields to avoid it being the trivial mapping.

$$\begin{aligned} \pi : E_{a,b}(\mathbb{F}) &\to E_{a,b}(\mathbb{F}), \\ (x, y) &\to (x^p, y^p), \\ \omega &\to \omega. \end{aligned} \tag{1.3}$$

Let us now define in (1.4) the two special subgroups on which we will define the bilinear mapping $\mathbb{G}_1[r]$ and $\mathbb{G}_2[r]$ (also referred to as $\mathbb{G}_1$ and $\mathbb{G}_2$).

$$\begin{aligned} \mathbb{G}_1[r] &:= \{(x,y) \in \mathrm{E}_{\mathrm{a,b}}[r] \text{ such that } \pi(x,y) = (x,y)\}, \\ \mathbb{G}_2[r] &:= \{(x,y) \in \mathrm{E}_{\mathrm{a,b}}[r] \text{ such that } \pi(x,y) = [p](x,y)\}. \end{aligned} \tag{1.4}$$

We can compute these two subgroups from the full r-torsion of an elliptic curve. For instance, examples of these two groups are provided in Section 3.3, and the code used to obtain them is provided in Appendix B.

At last, and to conclude this section, we define in (1.5) a bilinear mapping over these two groups called the Weil pairing. Both elements $f_{r,P}(Q)$ and $f_{r,Q}(P)$ are defined following Miller's algorithm, which we will not study in this dissertation as we will not need the explicit definition of the Weil pairing. The reason for that is that we will work with finite cyclic subgroups, so we will be able to express each element as a multiple of the group generator and use that the Weil pairing is a bilinear mapping.

$$\begin{aligned} \mathrm{e}(\cdot,\cdot) : \mathbb{G}_1[r] \times \mathbb{G}_2[r] &\to \mathbb{F}_{p^k}^*, \\ (P,Q) &\to (-1)^r \cdot \frac{f_{r,P}(Q)}{f_{r,Q}(P)}. \end{aligned} \tag{1.5}$$

All the concepts on elliptic curves presented here will be used when finding a suitable elliptic curve and corresponding r-torsion groups in Section 3.3. These theoretical fundamentals will also be used to code a program that allows readers to construct their own elliptic curves and r-torsion groups given a predefined prime number (see Appendix B).

This concludes the chapter on the mathematical fundamentals that will be needed in this dissertation. We now proceed to formally present zero-knowledge proofs.

2. Zero-Knowledge Proofs

2.1. Introduction

Zero-knowledge proofs (ZKPs) are protocols that demonstrate a statement's truth without conveying any other information about the statement itself. To easily visualize this idea, we will present two abstract examples.

2.1.1. Where's Waldo?

This example was given in [NNR98]. Suppose Alice and Bob are presented with the classic children's book 'Where's Waldo?'. Each of the pages of this book contains (hidden in between many other things) the protagonist of this problem, Waldo. Alice claims that she knows Waldo's exact location on one of the pages and wants to prove it to Bob. The simplest way of doing this would be by pointing at the location where Waldo is found, which would imply revealing this valuable information to Bob. Instead, Alice goes to the stationary shop and buys a big sheet of cardboard (at least twice the size of the book page). Then, Bob closes his eyes, and Alice proceeds to cut out a square the size of Waldo from the cardboard sheet and precisely situates the cardboard sheet on top of the book, making sure Waldo is visible through the recently cut hole. Once this is done, Bob can open his eyes and verify that Waldo is visible through the cardboard hole. However, Bob does not know the relative position of the book with respect to the cardboard sheet, and consequently, he does not know where exactly Waldo is. Alice has demonstrated that she knows where Waldo is without revealing his position. Note that this is not perfect zero-knowledge proof, as Alice has revealed some information about Waldo, such as his body position.

2.1.2. The Ali Baba Cave

This other example was presented in [QG90] and provides a closer idea of how the protocols using zero-knowledge proofs work. This example tells the story of Ali Baba, an old merchant of Baghdad. Ali Baba would go to the bazaar every day to buy and sell things. One of those days, a thief robbed his purse. Ali Baba ran after the thief who entered a cave that had a very particular shape. The cave forked into two dark, winding passages. One of the passages went to the left and the other to the right. As Ali Baba was not quick enough to see which passway the thief took, he had to choose one of the two to find him. After a quick thought, he chose to check the right passway. He did not find the thief, and the passway ended in a dead end. So he thought that the thief may have taken the other passway. He checked the other passway and did not find the thief there either. Weirdly enough, the passway ended in a dead end, too. Ali Baba assumed the thief had been lucky and had chosen the second passway. Then, the thief had left the cave while the merchant was in the first one. Sad for losing his money, Ali Baba returned to his home and went to sleep. The next day, Ali Baba returned to the bazaar, and another thief stole his purse again and headed towards the cave. Again, Ali Baba had to choose between the two passways to find the thief, but the same thing

occurred the day before, and Ali Baba didn not find the thief. The days went by, and Ali Baba was robbed every day by a thief who ran to the cave, and he was never able to catch him. He had convinced himself that the thieves that robbed him were always lucky to choose the passways that he did not choose, but when the fortieth thief robbed him on the fortieth day, he started to think that the cave may be hiding an important secret. Tired of being robbed and decided to discover the secret, the merchant decided to hide in the dark of one of the two passways and wait for a thief to reach the dead end. After long hours of waiting, a thief came running through the passway and, seeing another merchant following him, spoke the magic words "Open sesame". A wall of the dead end of the passway proceeded to open, allowing the thief to pass through and close behind him.

Ali Baba had discovered that the right passway connected with the left one when the cave wall slid open. After many long hours working with the magic words, he was able to modify them (like changing the passcode on a padlock). That way, the thief who robbed him the following day was trapped and could not escape from him. Ali Baba documented this story in an illustrated manuscript but did not write down the new magic words (he did leave some subtle clues so this password could be discovered).

Many years later, the manuscript and the cave were found by a researcher who discovered and learnt the magic words. A crew of reporters arrived at the cave to cover the amazing discovery, and the researcher showed them his discovery without revealing the magic words. How did he do this? Quite simple.

- First, a television crew recorded a detailed tour of the cave with the two dead-ends and left the cave.
- Then, the researcher went back into the cave down one of the passages.
- Later, the reporter, followed by a camera, went inside the cave and stopped at the fork where he flipped a coin.
  - If the coin came up heads, he would tell the researcher to come out through the left.
  - If the coin came up tails, he would tell the researcher to come out through the right.
- The researcher would come out the way the reporter had indicated.

They repeated the test 40 times (in memory of the 40 times Ali Baba was robbed). Each test divided the chances of coming out the right way by two; therefore, the researcher must have known the secret words after coming out the right way each time.

Now, we have gotten to the interesting part of the story. After recording the original tape, another reporter wanted to tell this extraordinary story, but the researcher was unwilling to collaborate with him. Conversely, he mischievously told the new reporter the location of the cave. The jealous reporter hired an actor who looked like the researcher and went to the same cave where they had recorded the previous tape. Then, they repeated the experiment explained above, but the actor did not know the magic words this time. So every time the actor came out the wrong passage, the reporter would cut out that part of the tape (editing it) and repeat it until the coin tossed would match the choice of the actor. After enough cuts, they created a tape containing 40 correct test repetitions (although it took them way more than that).

The two competing networks broadcast the two stories simultaneously, and the matter was taken to court. The judges and experts could not differentiate the tapes; thus, it was impossible for them to know which was genuine and which was not. The simulation did not convey any knowledge of the secret (not even the actor knew the secret words). But the simulation and the original tape were indistinguishable. Therefore, the genuine tape did not convey knowledge of the secret either.

The reporter who had seen the real researcher (and not the actor) coming out the right passage each time was convinced that the researcher knew the secret words. However, he could not transmit this knowledge to the judge or the experts. The researcher had achieved his real goal. He had shown that it was possible to convince without revealing, and so without disclosing the secret words.

2.2. The birth of ZKPs

Zero-knowledge proofs were introduced by Shafi Goldwasser, Silvio Micali and Charles Rackoff in [GMR85] in 1985 (the full paper can be found in [GMR89]). They defined them as proofs that only conveyed the knowledge of the propositions' correctness.

They also defined interactive proof systems, which consisted of two interactive Turing Machines that would communicate, allowing one to prove a statement to the other in a finite amount of time. Interactive proof systems had the following properties.

- **Completeness**. All true statements can be proven using the proof system.
- **Soundness**. A false statement can never be derived using the proof system.

The authors also generalized the idea and defined it as a probabilistic proof system. Those interactive proof systems were defined as functions that mapped elements in $\{0,1\}^*$ to the probability of the element being accepted (by the proof system) as a member of a predefined formal language $L \subseteq \{0,1\}^*$. They were noted as $(A, B) : \{0,1\}^* \to [0,1]$ where A was the *prover* and B the *verifier* and were presented as Turing machines. Note that $\{0,1\}^*$ could be interchanged with any other space on which a formal language is defined, for instance, $\mathbb{Z}_n$.

Once a probabilistic proof system was executed and completed, the verifier was (and still is) never 100 % sure of the statement's truth using these proof systems. However, very high levels of certainty can be achieved.

The authors' main purpose was to use interactive proof systems to research a simple question: how much knowledge is given to the verifier in an interactive proof system for any given language L? Let us imagine we want to check if $2 \in \{x = z^4 \mod 79 : z \in \mathbb{N}\}$. The prover could simply provide the value z (in this case $z = 3$), giving more information than the statement we want to prove.

A proof system was defined to be **zero-knowledge** when, after the proving protocol has been completed, the verifier could not prove anything about the statement to anyone. In other words, the proof is only valid to the participating verifier. How this property of the proof is technically proven is a little more complicated. A simulator of the verifier, which consists of another Turing Machine with rewinding capabilities, is used to impersonate the genuine verifier. This simulated verifier would equally interact with the real prover. In the example given in Subsection 2.1.2, the simulator would be the reporter that repeated the experiment with the actor playing the real researcher. The simulator simulates the whole protocol, including the interactions with the prover (the actor in the example), who sends

and receives the appropriate messages. At the end of the protocol, a complete transcript has been generated, which, if the property is satisfied (and including some details that we will see in Section 2.3), will be indistinguishable from the real transcript. If the property holds, the verifier did not obtain any information from participating in the protocol that he could not have obtained all by himself. That is equivalent to the definition we gave of zero-knowledge at the beginning of our work.

It is also worth mentioning the existence of other more relaxed versions of this property, such as zero-knowledge arguments (see Section 3.1) or honest-verifier zero-knowledge (see Section 2.7).

Continuing with the results presented in [GMR89], several examples of zero-knowledge proof systems were given for the languages of quadratic residuosity and graph isomorphisms. For simplicity reasons, we will present some slightly modified versions here.

For the following examples, the parties participating in the interactive proof system are A (the prover) and B (the verifier). A is assumed to have some knowledge, for instance, knowing that a statement is true. This knowledge could have been obtained because A has a significantly bigger computational power than B or simply because the decision problem was created by A.

2.2.1. Example: The quadratic residuosity problem

In section Section 1.4, we explained the basics of discrete square roots because they will be necessary to comprehend this example fully. Although the graph colouring problem is the one to which most zero-knowledge proofs are reduced to verify their properties, the quadratic residuosity example is the one that is generally used in the literature to present zero-knowledge proofs.

Let x be an integer. Any integer y such that $1 \leq y < x$ and $\gcd(x, y) = 1$ belongs to ${\mathbb{Z}_x}^*$. Let $QR = \{(x, y) \mid y \text{ is a quadratic residue mod } x\}$.

Both A and B are given the pair (x, y), with $y \in {\mathbb{Z}_x}^*$ and $m = |x|$ being the length of x (number of bits of its binary representation).

A wants to prove to B that y is a quadratic residue mod x. The simplest way to do that would be by telling B a z such that $y \equiv z^2 \bmod x$, but that would not be a zero-knowledge proof as we would be disclosing more information (the value of z) than the truth of the statement.

The following is done m times:

- A sends B a random quadratic residue mod x, u.
- B sends A a random bit, b.
- Then, depending on the value of b:
  - If $b = 0$, then A sends B a random square root of u mod x, w.
  - If $b = 1$, then A sends B a random square root of uy mod x, w.
- B checks that exactly one of the following occurs:
  - $b = 0$ and $w^2 \bmod x = u$.
  - $b = 1$ and $w^2 \bmod x = uy \bmod x$.

It is zero-knowledge because the prover does not provide any information that the verifier could not have obtained by himself. Furthermore, suppose y is not a quadratic residue modulo x, and suppose u and uy are both quadratic residues modulo x. Then, using the notation for the Legendre Symbol given in Section 1.4, $\left(\frac{y}{x}\right) = -1$, $\left(\frac{u}{x}\right) = 1$ and $\left(\frac{uy}{x}\right) = 1$. This leads to $1 = 1 \cdot -1$, using the product property given in Proposition 1.7, which is a contradiction. Furthermore, and considering that A does not see any bit in advance, the chances of B being convinced at the end of the m steps are $\frac{1}{2^m}$. The reason for this is the fact that, as a direct consequence of the Euler's criterion given in Proposition 1.4, the number of quadratic residues in ${\mathbb{Z}_x}^*$ is exactly $\frac{p-1}{2}$. This means one in every two elements of ${\mathbb{Z}_x}^*$ is a quadratic residue modulo x.

We will return in Subsection 2.6.2 to this same example to study how the the knowledge of z such that $y \equiv z^2 \bmod x$ can be extracted under the assumption of total control over the prover.

2.2.2. Example: The graph isomorphism problem

Let $G_1 = (V, E_1)$ and $G_0 = (V, E_0)$ be two graphs. Let ϕ denote an isomorphism between G_1 and G_0 such that $G_1 = \phi(G_0)$.

The following is done m times, each time using independent coin tosses:

- A generates a graph random isomorphic copy of G_1, H. This can be easily done by taking a random permutation $\pi \in \mathrm{Sym}(V)$. A sends this H to B.
- B sends A a random bit, b.
- Then, depending on the value of b:
  - If $b = 0$, then A sends $\pi \cdot \phi$ to B (see (2.1)).
  - If $b = 1$, then A sends π to B.
- If the permutation received by B is not an isomorphism between G_b and H, then B stops and rejects; else, he continues.

To check that the completeness property holds, we see that if G_0 is isomorphic to G_1(which is what the prover wants to demonstrate), then H can always be sent isomorphic to any of them.

To check that the soundness property holds, we suppose that the graphs are not isomorphic. Then H cannot be isomorphic to G_1 and G_0 simultaneously. Then, any prover could send isomorphisms between H and G_b only if H would have been originally created as $H = \pi(G_b)$. Because H is fixed to be a random permutation of G_1 from the beginning, the probability that a malicious prover could cheat the verifier is $\frac{1}{2^m}$ where m is the number of independent coin tosses.

It is simple to see that the system is zero-knowledge when the verifier follows the protocol. The information the verifier receives is 'useless' because he can also generate isomorphic copies of the original graphs.

$$\begin{array}{ccc} G_0 & \xrightarrow{\phi} & G_1 \\ & \searrow^{\chi} & \downarrow{\scriptstyle \pi} \\ & & H \end{array} \tag{2.1}$$

When the verifier deviates from the protocol, proving the zero-knowledge property is significantly more complex.

2.3. Types of Zero-Knowledge based on the information disclosed

As we have seen in the Ali-Baba story, the idea behind demonstrating that a proof is zero-knowledge is based on the fact that the recording of the fake proving process is indistinguishable from the original proving process (the one in which the prover does indeed know something that the verifier does not). This idea is used when proving the zero-knowledge property of actual proof systems. However, the term indistinguishable can be understood in various ways.

As mentioned in Section 2.2, two interactive probabilistic Turing machines carry out the proof. After the interactive proof system has been finalized, a probability distribution has been generated. It is said that an interactive proof system (defined in Section 2.2 too) is *computational zero-knowledge* if the probability of any given member of the generated probability distribution being accepted in polynomial time by either the original (prover, verifier) pair or the simulator is inversely-proportional to the size of the given member,

$$|p_1(x) - p_2(x)| \leq |x|^{-c} \quad \text{for all } c > 0.$$

With x a member of the probability ensemble and $p_i(x)$, $i \in \{1,2\}$, the probability of passing an expected-polynomial time test (in this case being accepted by the legitimate or the simulator pair of interactive Turing Machines).

As specified by [Pop04], a proof is said to be *perfect zero-knowledge* if the transcripts yielded by a pair of interactive Turning machines are identical to the distribution of the genuine protocol. The author also describes the existence of zero-knowledge proofs, which sit between computational and perfect proofs. These are called *statistical zero-knowledge proofs*, and the transcripts generated by the simulator differ in a constant number of instances from the ones generated by the real protocol.

This type of zero-knowledge is required from proofs across most practical solutions. Let us provide a rigorous definition.

Definition 2.1 (Statistical Zero-Knowledge)**.** An interactive proof system (A, B), where A is the prover and B is the verifier, for a language L is said to have *statistical zero-knowledge* if it satisfies the following conditions.

1. **Completeness.** For all $x \in L$, the verifier A accepts with high probability when interacting with the prover B.

$$\Pr[(A, B)(x) = 1] \geq 1 - \epsilon \quad \text{for some negligible } \epsilon.$$

2. **Soundness.** For all $x \notin L$ and for any prover A^*, the verifier B rejects with high probability.

$$\Pr[(A^*, B)(x) = 1] \leq \epsilon \quad \text{for some negligible } \epsilon.$$

3. **Zero-Knowledge.** For every probabilistic polynomial-time (see Section 1.6) verifier B^*, there exists a probabilistic polynomial-time simulator S such that for all $x \in L$, the distributions $\{B^*(x)$ interacting with $A(x)\}$ and $\{S(x)\}$ are statistically indistinguishable.

In opposition to *computational zero-knowledge*, indistinguishability does not depend on the element's size, and we simply state that they are statistically indistinguishable. We say two distributions are statistically indistinguishable if, for all probabilistic polynomial-time distinguishers D, there exists a negligible function ν such that

$$|\Pr[D(\{B^*(x) \text{ interacting with } A(x)\}) = 1] - \Pr[D(\{S(x)\}) = 1]| \leq \nu(|x|).$$

In this previous definition, we used that a probabilistic polynomial-time distinguisher $D : \Omega \to \{0,1\}$ (where Ω is the distribution space) assigns 0 to a distribution when it determines it is $S(x)$ or 1 otherwise. This choice is arbitrary, and we could interchange the values 0 and 1.

This formal definition is presented in the literature but rarely used because, most times, the probabilistic zero-knowledge property can be proven by a direct study of the structure of the generated transcripts.

We conclude the types of zero-knowledge by looking at the weakest of all the zero-knowledge proofs: *no-use zero-knowledge proofs*. These proofs do not necessarily need to be simulatable but can be established to reveal tiny amounts of data to the verifier. There is not much research done about proofs that satisfy this property, and we will not study them in this dissertation.

This proof classification is based on the zero-knowledge property stringency and how much information is leaked. However, we can also differentiate proofs by how much knowledge the prover provides to the verifier. The differences in the following classification are more subtle than in the one we have just discussed.

2.4. Proofs of knowledge

Many outreach articles on the net consider *zero-knowledge proofs* and *zero-knowledge proofs of knowledge* to be identical. However, this is a common misconception. Despite being similar and the term being used indistinguishably in most literature, there are some differences. Specifically, proofs of knowledge are a particular type of zero-knowledge proof.

Definition 2.2. A proof of knowledge is a protocol where a prover aims to prove to a verifier that he knows something.

If we added the zero-knowledge property to this definition, it may seem to match the exact description given for a zero-knowledge proof. However, note how in this last definition, the prover aims to prove that he knows something; this is more than proving that a statement is true. To fully understand the difference, let us first define a witness.

Definition 2.3. A witness is a piece of information that allows anyone to verify that a statement is true.

For instance, a witness in the quadratic residuosity example (see Subsection 2.2.1) would be the value z such that $y \equiv z^2 \bmod x$. Another example of a witness can be found in the example in Subsection 2.2.2. The isomorphism ϕ such that $G_1 = \phi(G_0)$ is that witness.

Given the definition of a witness, we can see how in the more general zero-knowledge proofs, the prover proves that there exists a witness. On a proof of knowledge, and in addition to this, the prover shows that he knows the value of the witness. Most examples presented

when zero-knowledge proofs were introduced were actually proofs of knowledge, although they were not defined as such back then.

In summary, a proof of knowledge is a zero-knowledge proof where the prover shows

I (the prover) know a witness for a true statement.

Before formally defining this idea, we must present what a knowledge extractor is.

Definition 2.4. A knowledge extractor is a verifier with special capabilities that interacts with the prover. It mimicks the communication a normal verifier would have with the prover during the usual execution of the protocol. The special capabilities of this verifier are described here.

- **Polynomial time.** The time complexity (see Section 1.6) is bounded by a polynomial dependent on a chosen security parameter. From this property, we will simply focus on the fact that its execution time is always bounded.
- **Access to randomness.** The knowledge extractor can make random choices during the interaction. This is used when simulating a real verifier whose choices are random when following the protocol. For example, the verifiers in each of the examples given in Subsection 2.2.1 and Subsection 2.2.2 make this random choice in the second step of the protocols. Therefore, it is sensible to expect the simulator to have access to randomness, too.
- **Rewinding capabilities.** The interaction with the prover can be rewound to a specific state. In other words, it can go back to a previous state of the communication protocol and make a different random choice, consequently generating a new answer from the prover. We will see how this works in real examples when presenting *Sigma protocols* in Section 2.6.

Once the knowledge extractor concept is understood, we can provide a more precise definition of a proof of knowledge.

Definition 2.5. A proof is a proof of knowledge of a witness if a probabilistic polynomial-time knowledge extractor can extract the witness from the prover.

It is not hard to see that a proof of knowledge is a much stronger proof than a basic zero-knowledge proof. Let us imagine the following example. Let G be a cyclic group of prime order p generated by g. Let $L = \{x \in G$ such that there exists $w \in \{0, ..., p-1\}$ such that $x = g^w\}$. The language L is trivial as every element in G belongs to L (any element of a finite prime cyclic group can be expressed as a power of the generator).

Consider now two different proof systems where the prover and verifier share the common input (G, g, x), and the prover aims to show that $x \in L$ or, in other words, there exists $w \in \{0, ..., p-1\}$ such that $x = g^w$.

- In the first proof system, the prover does not send anything to the verifier. The verifier checks if the element x belongs to G and accepts if this occurs. Because $x \in L$, the verifier always accepts.
- In the second one, the prover uses the witness to show that x belongs to G without disclosing w. This can be done as shown in the Schnorr protocol we will explain in Subsection 2.6.3. However, we are now only interested in assuming the prover can do this. Again, because $x \in L$, the verifier always accepts.

In conclusion, both proof systems are trivial as $x \in G$ implies $x \in L$. However, the proof of the second proof system is a proof of knowledge, while the first is not. The prover of the second proof system has demonstrated more than knowing the statement's veracity; he has proven he knows *why* the statement is true.

To conclude this section, we must note that the usual soundness property explained previously is modified when working with proofs of knowledge. The modified property associated with proofs of knowledge is usually called *special soundness* in literature.

- **Special soundness**. Only a prover who knows a witness can convince the verifier.

Proving special soundness does imply soundness because the former is a stronger property. Special soundness was defined as such because most proofs' soundness properties were proven by demonstrating the special soundness property was true. However, this does not apply to all proofs. For instance, the soundness property given in Subsection 2.2.2 shows this.

Although we have presented the general structure of a proof of knowledge, we will focus on a specific type of them called *Sigma protocols*. They set the fundamentals of many of the most common zero-knowledge protocols nowadays, including zk-SNARKs (although these are their non-interactive succinct counterparts), presented in Chapter 3. Before studying Sigma protocols, we must present a basic concept used in many cryptographic tools. This concept is called a *commitment scheme*.

2.5. Commitment schemes

As we will explain in Section 2.6, the first step in Sigma protocols consists of the prover sending a *commitment* to the verifier. In this section, we will understand what a commitment is and how it is used.

Definition 2.6. A commitment scheme is a process in which a party can compromise to one value while still being kept hidden from the others to reveal later the value to which he was committed.

The prover is the party that will usually commit to a value during the protocols we are studying. The cryptographic mechanism used to do this is called the commitment and must satisfy two main properties.

- **Binding.** Once the prover commits to a value, they cannot change this value. This ensures that the prover is bound to the committed value, providing security against cheating.
- **Hiding.** The committed value remains hidden from the verifier until the prover reveals it. This ensures that the prover's secret remains confidential until the prover decides otherwise.

Example 2.1. Imagine Bob had a dream in which he learned which number would win the lottery next month. He wants to tell his friends but does not want them to know which exact number it is. How can he do it then? He goes to a stationary shop and buys a box and a lock. He writes the number on a sheet of paper, puts it in the box, and locks it with the lock. Then, he gives the box to his friends, who keep it until the lottery number has been announced. That day, Bob meets his friends and gives them the lock's key. His friends open the box and

see that Bob had written the correct number a month before. Therefore, assuming Bob is not some kind of magician, Bob must have known the lottery number from the beginning.

As the example has shown, there are two phases in a commitment scheme.

1. **The commit phase**. During this phase, the value is chosen and committed to. This is represented in our example by Bob writing the number on paper and locking it in the box.
2. **The reveal phase**. During this phase, the value is revealed by the party committing to it, and the receiver checks its authenticity. In our example it will be Bob's friends opening the lock.

Once commitment schemes are understood, we can fully understand Sigma protocols. In the following section, we will present their structure, describe the properties they must satisfy and verify some of the examples we presented before follow this protocol.

2.6. Sigma protocols

Sigma protocols are a type of proof of knowledge (see Section 2.4). The scheme is called a Sigma protocol because the Greek letter Σ represents its 3-move structure. Let us describe each of those three steps.

1. The prover starts the protocol by sending a message a, called the commitment, to the verifier.
2. The verifier chooses a random message c, called the challenge, from a finite challenge space and sends it to the prover.
3. The prover sends a final message t, called the response to the verifier.

At last, the verifier decides if it has been convinced of the statement based on the final transcript (a, c, t). The value whose knowledge wants to be proven is called the *witness*. The witness is only known to the prover. For instance, in the 'Where's Waldo?' example, the witness would be the location coordinates of Waldo in the picture, that is $(x, y) = (10, -5)$. Note that this protocol can be repeated m times to make the probability of the verifier being cheated almost negligible. We will also refer to this repetition of the three steps as a Sigma protocol with m iterations.

As described in [ZCYW23], these protocols have special properties that make them highly valuable proof systems.

- **High computational efficiency.** The number of steps of each protocol iteration is constant (three). The absence of conditional clauses on the verifier's side, excluding the proof's correctness verification, further pushes this efficiency.
- **Fixed proof size.** We define the proof size of a Sigma protocol as the length in bits of all the messages sent during the protocol. This is the same as the length in bits of the generated transcript (a, c, t), where a is the commitment, c the challenge and t the response, multiplied by the number of iterations. The size of each of the transcripts' elements will vary depending on the specific sigma protocol, but they will not vary

between iterations of the same protocol. For instance, if the commitment, challenge and response are integers expressed in binary using 256 bits, they will also be 256 bits in length in the following iterations.

- **Sigma protocols are closed under composition.** Sigma protocols can be easily combined to prove compound statements. An example is provided at the end of the discrete logarithm problem example in Subsection 2.6.3.
- **Simpler soundness property.** The special soundness property, presented in a more general context in Section 2.4, is simpler to work with than the classic soundness property. We will see this simplicity later in this section.
- **Almost direct transformation to non-interactivity.** The use of the Fiat-Shamir heuristic (see Subsection 2.8.1) trivializes transforming Sigma protocols in non-interactive systems.

Let us study what properties must hold to ensure a Sigma protocol is a zero-knowledge proof of knowledge.

2.6.1. Sigma protocols properties

Slight modifications of the properties given for interactive proof systems in Section 2.2 must be satisfied to establish correct Sigma protocols.

- **Completeness**. Identical to the completeness property defined in Section 2.2.
- **Special soundness**. For any witness w, and any pair of accepting transcripts (a_1, c_1, t_1) and (a_2, c_2, t_2) which share the commitment, $c_1 = c_2$, there exists an efficient knowledge extractor that can compute w.
- **Honest verifier zero-knowledge proof**. Instead of the classic zero-knowledge proof we first presented, the verifier is assumed to be honest in this type of proof (see Section 2.7 to understand the consequences of this assumption).

The conceptually hardest of these properties is the special soundness. Let us reexamine some of the examples presented previously and verify the special soundness property holds in those examples.

2.6.2. Revisiting previous examples

Let us first reexamine the example of the graph isomorphism (see Subsection 2.2.2). We must check the knowledge extractor's existence to verify the special soundness property. Let (π, b_1, G_{b_1}) and (π, b_2, G_{b_2}) be two different transcripts. Because they are different and the commitment is common to both, $b_1 \neq b_2$. If this would not occur, b_1 would be equal to b_2, and the transcripts would be identical. As there are only two possibilities for the value of b, assume $b_1 = 1$ and $b_2 = 0$. The data in possession of the knowledge extractor (apart from the transcripts) is given in (2.2).

$$\begin{cases} H = (\pi \cdot \phi)(G_0), \\ H = \pi(G_1). \end{cases} \tag{2.2}$$

Then, the knowledge extractor can compute the value of ϕ by doing $\phi = \pi^{-1} \cdot (\pi \cdot \phi)$. This is because $(\pi \cdot \phi)(G_0) = \pi(G_1)$, and therefore, $\pi^{-1} \cdot (\pi \cdot \phi)(G_0) = G_1$ (the order in which the isomorphism are applied is fixed in the protocol).

We can also see the special soundness property holds for the quadratic residuosity example given in Subsection 2.2.1. Let (u, b_1, w_{b_1}) and (u, b_2, w_{b_2}) be two different transcripts. We can again assume $b_1 = 1$ and $b_2 = 0$. The data in possession of the knowledge extractor is given in (2.3) (note that a is not known and therefore cannot be used by the knowledge extractor).

$$\begin{cases} u \in \mathbb{Z}_x^* \text{ such that there exists } a \in \mathbb{Z}_x^* \text{ such that } a^2 \equiv u \pmod{x}, \\ w_{b_1} = w_1 \in \mathbb{Z}_x^* \text{ such that } {w_1}^2 \equiv u \pmod{x}, \\ w_{b_2} = w_0 \in \mathbb{Z}_x^* \text{ such that } {w_0}^2 \equiv uy \pmod{x}. \end{cases} \tag{2.3}$$

Then, the knowledge extractor can compute the value of $z \in \mathbb{Z}_x^*$ such that $z^2 \equiv y \pmod{x}$. First, using that ${w_0}^2 \equiv uy \pmod{x}$ and ${w_1}^2 \equiv u \pmod{x}$, then ${w_0}^2 \equiv {w_1}^2 y \pmod{x}$. Then, using the previous expression and $z^2 \equiv y \pmod{x}$, ${w_0}^2 \equiv {w_1}^2 z^2 \pmod{x}$ is obtained. Note that here, z is used as a variable whose value the knowledge extractor wants to find. Now, it is direct to obtain $w_0 \equiv \pm w_1 z \pmod{x}$. Finally, the variable z can be left on the left side of the equivalency equation, obtaining $z \equiv \pm {w_1}^{-1} w_0 \pmod{x}$.

One of the most famous Sigma protocols is the discrete logarithm problem, also called the *Schnorr protocol*. It gained most of its popularity due to its use as a signature protocol. This example is very useful to understand how a *hard* problem can protect the witness of the proof from being disclosed. This same principle will be used when using elliptic curves in the Groth16 protocol in Section 3.3.

2.6.3. Example: The discrete logarithm problem or the Schnorr Protocol

Claus Schnorr first described this protocol in [Sch90]. The methodology followed to prove the belonging of this problem to the sigma protocols class has been extracted from Jorge L. Villar's lecture notes [Vil23]. The common input to both parties (prover and verifier) is (G, q, g) and y, being G a cyclic group of prime order q generated by g and $y = g^x$. The prover's private input, also called witness, is the discrete logarithm x (see Section 1.5). The knowledge of this witness and the fact that $y = g^x$ is what the prover aims to demonstrate.

Let us see the three steps of the Sigma protocol.

1. The prover generates a commitment choosing a random $r \in \mathbb{Z}_q$ and computing $a = g^r$. Note how G is the finite challenge space in this problem.
2. The verifier receives the commitment and sends a random challenge $c \in \mathbb{Z}_q$ to the prover.
3. The prover computes the response $t = r + cx$. Note how the witness's knowledge is crucial during this step.

Now, the verifier needs to check if the proof of the statement is correct by checking if $g^t = ay^c$. The completeness of the proof system is directly proven by seeing $ay^c = g^r y^c = g^r g^{xc} = g^{r+xc} = g^t$.

To check the special soundness property, we must extract the witness from two accepting transcripts that share the commitment value but have different challenges. Assume we have two such transcripts (a, c_1, t_1) and (a, c_2, t_2). Therefore, $t_1 = r + c_1 x \pmod{q}$ and $t_2 = r + c_2 x$

$(\text{mod } q)$. Note that r is the same for both cases as a was set to be the same too. Therefore, we have $x = \frac{t_2 - t_1}{c_2 - c_1} \pmod{q}$. Note that this is well defined because we assumed the transcripts were different and thus $c_1 \neq c_2$.

Finally, we can check the honest verifier zero-knowledge property. But this is also direct as any simulator can choose for a random challenge $c \in \mathbb{Z}_q$ a random response t and compute the commitment $a = g^t y^{-c}$. Therefore, identically distributed transcripts can be easily computed without the witness's knowledge. It is clear that proving the zero-knowledge of the proof when assuming the verifier is honest is not complex.

Something worth noting is that this protocol is usually implemented to prove compound statements. For instance, suppose the prover is called Alice, and the verifier is called Bob. Alice wants to prove the knowledge of the discrete logarithms of $y_1, y_2 \in G$ modulo g.

1. Alice picks a combined commitment constructed with random $r_1, r_2 \in \mathbb{Z}_q$ and sends $a = (a_1, a_2) = (g^{r_1}, g^{r_2})$ to Bob.
2. Bob sends a single random challenge c to Alice.
3. Alice computes a combined response such that $t_1 = r_1 + x_1$ and $t_2 = r_2 + x_2$, and sends $t = (t_1, t_2)$ to Bob.
4. Bob checks if $g^{t_1} = a_1 y_1^c$ and $g^{z_2} = a_2 y_2^c$.

This same methodology can be easily generalized to a finite number of statements in any Sigma protocol and still follows the original three-move structure.

Another example that is as widely used in the literature is the graph colouring example. It has been proven that any NP problem has a zero-knowledge proof by reducing the NP problem to the graph colouring one.

2.6.4. Example: The graph colouring problem

Let $G = (V, E)$ be a graph and $\phi\colon V \to (R, G, B)$ a 3-colouring of the graph known by the prover. We will again refer to the prover as A and B to the verifier. Note we will be using B to refer to the verifier and B to refer to the blue colour in the (R, G, B) space. Context should sufficiently tell the difference between both. The following is done m times, each time using independent coin tosses:

- A selects a random permutation $\pi \in \text{Sym}(R, G, B)$ and computes $\pi(\phi(V))$. Then, the prover commits to $\pi(\phi)$ using a commitment scheme of his choice.
- B selects a random ordered pair (i, j) from E and sends the choice to A. This will be the challenge.
- A shows the colours used to colour the vertices i and j. These will be $\pi(\phi(i))$ and $\pi(\phi(j))$ respectively. A also shows the randomness used in the commitment scheme.

If the two colours received are the same, then B stops and rejects; else, he continues.

To prove the completeness, special soundness and honest verifier zero-knowledge of this example, we operate similarly to what we did with the graph isomorphism problem with a slight difference in considering the desired commitment scheme.

As we mentioned before, assuming the honesty of the verifier simplifies checking the zero-knowledge property of a proof. However, we have not studied the consequences of this assumption on the security of the protocols. Let us present some examples to show how a dishonest verifier could benefit from an honest-verifier zero-knowledge proof.

2.7. Honest-verifier Zero-Knowledge Proofs (HVZKPs)

Honest-verifier zero-knowledge proofs receive their name because the zero-knowledge property only holds when the verifier is assumed to be honest. What does honest mean in this context? It refers to a verifier who faithfully follows the protocol as specified without attempting to deviate from the prescribed steps or manipulate the process in any way. A dishonest verifier could be able to extract information from any zero-knowledge proof whose zero-knowledge property has been proven under the honest-verifier assumption. It has not yet been found (to the best of our knowledge) any proof or description of such information for the examples we have presented in this dissertation. However, we can see how the zero-knowledge property of an example, the Schnorr protocol, fails if a dishonest verifier interacts with it.

Suppose now the verifier does not choose the challenge randomly from $\mathbb{Z}_q$ and does choose each challenge based on the random commitment the prover chooses. In particular, the verifier always chooses the challenge equal to the commitment, such that $c = r$. Then, we could not create the simulator for this verifier as the prover only sends a such that $a = g^r$, not the random r. Being able to choose $c = r$ would imply the verifier could compute the discrete logarithm of a base g. In that case, he may as well directly compute the discrete logarithm of y, and he would not need to interact with the prover to learn anything. This shows that this example would not occur in a real-life scenario. We will look further into the Schnorr protocol case in Subsection 2.7.2.

Another example based on the same principle can be found in the quadratic residuosity problem (see Subsection 2.2.1). If the verifier sends his response based on the value a such that $a^2 \equiv u \pmod{x}$, then a simulator would not be able to simulate this same verifier with a probability different to $\frac{1}{2}$. The probability would be equal to $\frac{1}{2}$ because the simulator would have to randomly guess the bit chosen by the verifier. This guess would be random because the simulator would not have access to the value of a and consequently would not understand the verifier's choice logic. The bit's dependency on the u value could even be as simple as expressing u in binary and selecting bit as the last bit of that binary number. As we saw in the previous two examples, the dishonest verifier could indeed obtain the witness of the protocol but would have to be able to solve the problem in the first place.

It is not simple to see how a real (not computationally unbounded) dishonest verifier could benefit from being dishonest in obtaining any information from the proof in an already defined problem. For instance, there is no (known) attack against the zero-knowledge property of the quadratic residuosity problem. However, there is no security proof either, and the honest-verifier proof does not necessarily work for the dishonest-verifier proof. We can present a new zero-knowledge proof that illustrates this case.

Let us create a new zero-knowledge proof based on any of the sigma-protocol examples we have previously studied. Given one of them, we can add an extra initial round in which the verifier sends a 1 to the prover. Then, if the prover receives a 1, he runs the original zero-knowledge proof. In case he receives a 0, he reveals the witness. The proof is zero-knowledge if the verifier follows the protocol, in other words, behaves honestly. If it is a dishonest verifier, he could send a 0, leaking the witness and breaking the zero-knowledge property. Therefore, the protocol is zero-knowledge when the verifier is assumed to be honest but not when the verifier misbehaves. In Subsection 2.7.3, we will see that some authors solved this problem under certain assumptions.

Continuing with the particular case of Sigma protocols, due to their conceptual simplicity and wide usage, we proceed to see how a simulator would have to work when the verifier is

not assumed to behave honestly.

2.7.1. How does a simulator work when the verifier can behave maliciously?

When we presented the zero-knowledge property in Section 2.2, we indicated that the way of doing so is by generating transcripts that are *indistinguishable* from the original protocol transcripts. As we have seen in the Schnorr and quadratic residuosity examples studied in this section, these transcripts can not be generated when the verifier is not assumed to behave honestly. In those cases, we supposed the verifier chose the challenge based on the prover's commitment instead of choosing it randomly.

Let us suppose a dishonest verifier, one whose challenges depend on the prover's commitment, participates in a Sigma protocol. The way a simulator would work in this case is as follows.

- The simulator guesses in advance a random challenge c^* the verifier could send to the prover. He does this first because the verifier will choose the challenge based on the prover's commitment, which means the choice is not random. Therefore, the probability ensemble generated by the simulator would not be statistically indistinguishable from the one generated by the real pair of interacting Turing machines.
- Then, the simulator receives the commitment a from the prover. Sends him the challenge c^* he created in the previous step and receives a response t from the prover.
- Once that is done, the simulator interacts with the malicious verifier and sends him the prover's commitment a.
- The malicious verifier receives the commitment a and responds with the challenge c (based on the commitment's value).
- If the challenge the malicious verifier generates matches the one the simulator guessed ($c^* = c$), the transcript (a, c, t) is correctly generated by putting together the commitment, the challenge and the response. However, if the challenge the malicious verifier generates does not match the simulator's guess ($c^* \neq c$), the simulator must rewind the verifier to a previous state and repeat the whole process again.

Rewinding the verifier if the challenges do not match is necessary because the created transcript (a, c, t) would not be valid. Only (a, c^*, t) would be valid because a and t were generated for the specific challenge c^*.

The simulator's guess probability of being right depends on the challenge sample space size. The larger the sample size, the lower the probability of randomly guessing the challenge the real verifier would generate.

Let us now see how the sample space size affects the zero-knowledge property of the Schnorr protocol example when the verifier is not assumed to behave honestly.

2.7.2. Dishonest verifiers and the Schnorr protocol

Consider now the example provided in Subsection 2.6.3. If the sample space $\mathbb{Z}_q$ is big, the chances of guessing the challenge are $\frac{1}{q}$. Therefore, the chance of finding an efficient (see Remark 1.4) simulator exponentially drops when the sample space size grows. An

efficient simulator will always have non-negligible chances of right-guessing the challenge ($1-(1-\frac{1}{q})^k$ where k is the number of tries) in polynomial time.

Reducing the sample space to $\{0,1\}$ would increase the chances of the simulator guessing the challenge to $\frac{1}{2}$. However, chances of being fooled would rise to $\frac{1}{2}$, negatively affecting the soundness of the protocol. This negative effect can be solved by repeating the protocol m iterations. This is what was done in the Ali Baba example.

Such a drastic reduction of the sample space would mean that the number of iterations needed to obtain a similar soundness to the original problem would be huge. For that reason, a bigger sample space could be selected as long as the efficiency of the simulator would not drop exponentially.

Considering the soundness increases with the sample space size, it could be thought that increasing the sample size to infinity would allow us to complete the proof in only one iteration. This is untrue and can be explained with a simple example Wenbo Mao described in [Mao03].

Let us imagine a Schnorr protocol where *Alice* is the prover and *Bob* is the verifier. We assume the sample space $\mathbb{Z}_q$ size is very large. Let prf be a pseudorandom function onto $\mathbb{Z}_q$, g the generator of a finite cyclic group G and y a random element in G whose knowledge of the discrete logarithm modulo g Alice wants to demonstrate.

Alice sends a commitment a to Bob. Suppose that Bob behaves dishonestly, does not follow the protocol, and tries to trick Alice into disclosing some information. Bob generates $c = \text{prf}(M||a)$, such that M is defined as

$$M = \text{'Meaningful transcript, signed by Alice'},$$

and a will satisfy $g^t = ay^c \pmod{p}$ where t is the response Alice will generate after receiving c. Note t and c have not yet been generated; however, the commitment a will always satisfy this equation based on the definition of the protocol on the prover's side.

By generating this challenge, Bob has not followed the protocol and has behaved dishonestly. He has generated a challenge that is not a random choice amongst the elements of $\mathbb{Z}_q$. He has done this to break the zero-knowledge property of the protocol.

While challenge c has been generated by Bob, the commitment a has been generated by Alice, and this commitment has a uniquely defined relation with the values of c and, more importantly, t. Therefore, Alice has unintentionally helped Bob to construct the challenge $c = \text{prf}(M||a)$.

What has Bob achieved by constructing this challenge c? Let us assume the transcript (a, c, t) is seen by a third party that knows the problem being proven. Then, he could conclude either of two things.

1. The equation c verifies ($g^t = ay^c \pmod{p}$) was set up by Alice using her private input (there is no other possible way of determining the value of t otherwise). In other words, Alice discloses that she knows the witness and has interacted with Bob.

2. Bob has successfully broken the pseudo-random function prf of the large sample space Z_q. He has determined which value a needs to be concatenated with M and fed into the prf function to determine c.

The second of the two options is considered *hard* (see Section 1.6) because the first assumption we made was that the sample space $\mathbb{Z}_q$ was very large. The problem hardness allows the third party to assume the first of the two provided options is true. Thus, the fact that

Alice knows the witness has been leaked. Alice has unintentionally created a digital signature that implies her involvement and compliance with the protocol, indirectly suggesting her knowledge of the secret being proven: the witness.

The information leakage is not too severe as the witness has not been disclosed, but leaking that Alice knows the secret could be dangerous. For instance, Alice could have proved to Bob that her private key is associated with a wallet (see Definition 4.6) containing millions of dollars. This could attract potential robbers, endangering Alice.

This example shows how the dishonesty of a verifier makes the zero-knowledge property verification more complex and that modifying the sample space size intuitively does not always provide the expected results. In this case, it enables the malicious verifier, Bob, to leak information.

Despite these not-so-positive remarks, the same authors who presented zero-knowledge proofs in 1985 were able to prove that the honesty of the verifier can always be assumed when proving the zero-knowledge property.

2.7.3. Honest-verfier zero-knowledge implies zero-knowledge

Oded Goldreich, Amit Sahai and Salil Vadhan proved in [GSV98] that, under some light assumptions, Theorem 2.1 holds.

Theorem 2.1. *Any interactive proof system that is statistical zero-knowledge with respect to the honest-verifier can be transformed into a proof system that is statistical zero-knowledge for any verifier.*

The assumptions made were not based on the complexity of the verifier nor the computational capabilities as previous authors had done before. Although the details of this theorem escape the scope of this dissertation, it is important to have this theorem always in mind. Before this was known, the benefits of having an honest-verifier zero-knowledge proof of a protocol could be doubted. Affirming that one can derive the other shows the power of the verifier's honesty assumption.

2.8. Non-Interactive ZKPs (NIZKs)

We have presented several different proof systems throughout this dissertation. All of them had their interactivity in common. However, these interactions can cause considerable overhead on any network when the number of simultaneous proofs being done increases. This overhead dramatically decreases the potential applicability of zero-knowledge proofs in technologies such as Blockchain, where the immediacy of operations is primordial. Non-interactive zero-knowledge proofs were presented to improve the applicability of these protocols.

Fiat and Shamir proposed in [FS87] a conversion method that transforms interactive proof systems into non-interactive ones. In particular, they presented an example that will be described in the following section.

2.8.1. Fiat-Shamir conversion

Although this conversion has been widely used to simplify and eliminate the interactivity of many zero-knowledge proofs, this was not the authors' original intention. The main objective of Fiat and Shamir was to describe simple identification and signature schemes that could enable any user to prove his identity and the authenticity of his message to any other user

without needing to use shared or public keys. For instance, this technology could be used in passports, credit cards and military control systems. The assumptions they made were that factoring is difficult and that the pseudorandom function being used is indistinguishable from a truly random function by a polynomially bounded computation.

The identification protocol they presented in [FS87] was a combination of a zero-knowledge interactive proof and an identity-based scheme given in [Sha85] by Shamir. The scheme assumed the existence of a trusted center that would generate smart cards for users after checking their physical identity. No more interaction with the trusted center would be needed. Furthermore, the increase in the number of users in the system would not degrade the system, and the number of times the cards were used would not affect its properties as no information was leaked.

Before the trusted center issues a card, it chooses and makes public a modulus n and a pseudorandom function f such that $f\colon S \times \mathbb{N} \to [0, n)$ where S is the space of strings of arbitrary size. The second input term of the pseudorandom function will be used to indicate the seed being used. The value n is the product of two big prime numbers p and q, but only the trusted center knows the factorization. The steps the trusted center performs are the following.

1. After an eligible user applies for a smart card, the center prepares a string I with all his relevant information. Such information could include name, address, ID, physical description, and security clearance. The string also includes information about the card, such as expiration date or limitations.

2. Compute the values $v_j = f(I, j)$ being j the seed of the pseudorandom function until at least k of them are quadratic residues.

3. Pick k values of j for which v_j is a quadratic residue mod n. Compute the smallest square root s_j of $v_j^{-1} \pmod{n}$. Considering that the center knows n's factorization, this will not be complicated.

4. Issue a smart card that contains I, the k s_j values and their indices.

We assume the first k indices are used $j = 1, 2, ..., k$. There are other subtle assumptions to achieve robustness that we will not indicate here for simplicity reasons.

The smart card (prover) must prove that it knows the values $s_1, ..., s_k$ without giving any information about their values. Let us now see what steps occur when a smart card is inserted into a verifier. Let us assume that A is the prover and B the verifier.

1. A sends I to B.

2. B generates $v_j = f(I, j)$ for $j = 1, ..., k$.

   Repeat steps 3 to 6 for $i = 1, ..., t$.

3. A picks a random integer $r_i \in [0, n)$ and sends $x_i = r_i^2 \pmod{n}$ to B.

4. B sends a random binary vector $(e_{i1}, ..., e_{ik})$ to A.

5. A sends y_i to B, such that
$$y_i = r_i \prod_{e_{ij}=1} s_j \pmod{n}.$$

6. B uses that y_i to check that

$$x_i = y_i^2 \prod_{e_{ij}=1} v_j \pmod{n}.$$

It is direct to check the completeness of this proof system. If both A and B follow the protocol, B always accepts the proof as valid. This is because $y_i^2 \prod_{e_{ij}=1} v_j \pmod{n}$ is equal to $r_i^2 \prod_{e_{ij}=1} \left(s_j^2 v_j\right) \pmod{n}$. By definition, s_j are square roots of ${v_j}^{-1}$, and substituting we obtain that $r_i^2 \prod_{e_{ij}=1} \left(s_j^2 v_j\right) \pmod{n} = r_i^2 \prod_{e_{ij}=1} \left(v_j^{-1} v_j\right) \pmod{n} = r_i^2 \pmod{n}$. Finally, r_i^2 is congruent with x_i modulo n and the completeness is verified. The zero-knowledge property of the scheme is easily proven as no information about the s_j is revealed apart from revealing random square roots mod n of their inverses.

Let us now see how this proof scheme is transformed into a non-interactive one. Suppose that f is a pseudorandom function now defined as $f\colon S \times [0,n)^t \to [0,n)$ where t in an integer (the greater the t is, the more secure the protocol is). We obtain a signature scheme from an identification scheme by doing the following.

***A* sings a message *m*.**

1. A picks random integers $r_1, ..., r_t \in [0,n)$ and computes $x_i = r_i^2 \pmod{n}$.
2. A computes $f(m, x_1, ..., x_t)$ and uses its first $k \cdot t$ bits as e_{ij} values $(1 \leq i \leq t, 1 \leq j \leq k)$.
3. A computes y_i such that

$$y_i = r_i \prod_{e_{ij}=1} s_j \pmod{n} \quad \text{for } i = 1, ..., t.$$

and sends I, m, the e_{ij} matrix and all the y_i to B.

Note how A repeats the same steps seen before the conversion with the only difference that the random matrix $(e_{ij})_{i=1,...t;j=1,...k}$ is now computed using the pseudorandom function which is public to everyone.

***B* verifies *A*'s signature on *m*.**

1. B computes $v_j = f(I, j)$ for $j = 1, .., k$.
2. B computes z_i such that

$$z_i = y_i^2 \prod_{e_{ij}=1} v_j \pmod{n} \quad \text{for } i = 1, ..., t.$$

3. B verifies that the first $k \cdot t$ bits of $f(m, z_1, ..., z_t)$ are e_{ij}.

Again, it is direct to check the proof's completeness. By definition $z_i = y_i^2 \prod_{e_{ij}=1} v_j \pmod{n}$. Substituting the value of y_i, $r_i^2 \prod_{e_{ij}=1} (s_j^2 v_j) \pmod{n}$ is obtained. At last, the definition of s_j as square roots of ${v_j}^{-1}$ is used, and therefore $r_i^2 \prod_{e_{ij}=1} (v_j^{-1} v_j) \pmod{n}$. Finally, $r_i = x_i \pmod{n}$. This concludes that $f(m, z_1, ..., z_t) = f(m, x_1, ..., x_t)$.

Once the example given by Fiat-Shamir has been understood, we will see how this conversion can be applied to the Sigma protocol proof of the discrete logarithm (see Subsection 2.6.3).

2.8.2. A known sigma protocol made non-interactive through the Fiat-Shamir conversion

Consider the interactive proof presented in Subsection 2.6.3 for the discrete logarithm problem. The commitment a and response t are generated as in the interactive protocol. A hash function $H\colon \{0,1\}^* \to \mathbb{Z}_q$ is now part of the public information and, consequently, known by the prover and verifier. The main difference is found in the challenge, which the verifier does not randomly choose but instead is computed as $c = H(g||y||a)$ where g,y and a are expressed in binary and concatenated. In the non-interactive version, the prover sends (a,t), and the verifier accepts the proof if $g^t = ay^{H(g||y||a)}$, where g is again the generator of a cyclic group of prime order q and $y = g^x$.

To prove the honest verifier zero-knowledge property is simply needed to see that the simulator can generate $a = g^t y^{-H(g||y||a)}$ from a random response t.

2.8.3. The common Reference String Non-Interactive Model

In [BFM88], Blum, Feldman and Micali presented another Non-interactive Zero-Knowledge Proofs paradigm. The prover and the verifier can access a common string in this scheme. This string allows the prover to send a single non-interactive proof convincing the verifier. Depending on how this string is generated, there are two types of strings. *Uniformly random (URS)* or *structured (SRS)*. The structure reference strings use a secret s, which can act as a trapdoor to regenerate the desired reference string. This secret is eliminated once the string is created. Although having a trapdoor could be seen as a vulnerability, it is used by the prover simulator to prove that the system is indeed zero knowledge. This is done by generating the desired reference string so the simulated prover can prove the statement without prior knowledge of any witness.

This concludes the in-depth presentation of zero-knowledge proofs and the mathematical fundamentals used to support them. The following chapter will present Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs). The introduction of this technology aims to improve the applicability of zero-knowledge proofs, reducing the proof sizes and computational overhead during the verifying process. This technology heavily relies on elliptic curve cryptography and bilinear mapping, presented in Chapter 1. This study is needed to fully comprehend the most up-to-date protocols being used in many Blockchain solutions, specifically in the ones we will present in Chapter 5.

3. zk-SNARKs: Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge

This chapter introduces zk-SNARKs. From their definition and reason to be, to showing a simple construction example following the widely used Groth16 protocol. We provide a simplified version of the code in Appendix A and Appendix B, which we derived from the MoonMath manual in [Aut24]. The original Python notebooks are found in GitHub.

3.1. Arguments of knowledge, zk-SNARGs and zk-SNARKs

Similarly to what we studied in Section 2.8, authors have tried to improve the applicability of zero-knowledge proofs to exploit all their potential. The two main applicability problems researchers had to deal with were the high stringency of the prover's computational requirements and the size of the proofs. zk-SNARKs aim to tackle both of these problems simultaneously. Let us first relax the requirements on the proofs we will working with.

Applying the theory we have studied to real-life scenarios has positive and negative aspects. The main downside is that we must consider space and time complexity when implementing the protocols. On the other hand, we can consider attackers or dishonest parties to have limited resources and time. For this reason, perfect zero-knowledge proofs are rarely seen in real applications.

Therefore, as we did with the zero-knowledge property in Section 2.3, we can also consider statistical and computational soundness. Any proof with statistical soundness, that is, when any computationally unbounded prover cannot convince a verifier of a false statement, is considered a proof. On the other hand, any proof where we only require that a polynomial prover cannot convince a verifier of a false statement is considered an *argument*.

In [BCC88], Gilles Brassard demonstrated all NP-problems have perfect zero-knowledge arguments or, as the authors in that paper called them, *minimum disclosure* because some information is always leaked, such as the fact that wants to be proven. This demonstrates that applicability-wise arguments are as or more powerful than the original proofs. But why would we use these protocols if the soundness property is weaker than their predecessors? The answer to this question is the reduced communication complexity associated with arguments.

For instance, and developing the work of Brassard, Joe Kilian demonstrated in [Kil92] that given a family of intractable hash functions and parametrized by a security parameter l, there exists an argument for a language membership problem ($x \in L$) such that for a given constant c the size of the argument is $O((\log(c)|x|)l)$. For $|x|$, the length in bits of the element x whose belonging to the formal language L we wanted to prove. We understand the size of the argument as the number of bytes that the prover needs to send to the verifier to complete the protocol. Note that the common reference string length in bytes is not included here. We will not study how this was done because the size complexity of proofs has been reduced many times following that one. For that reason, we will present an example of how that is currently being implemented in zk-SNARKs following a protocol called *Groth16*.

Reducing the byte length of the proofs was an important step towards improving non-interactive arguments of knowledge applicability. The shorter the argument, the faster it can be sent through the network.

Many more authors, like Jens Groth in [Gro09] or Jae Hong Seo in [Seo11], both of whom provided solutions for linear algebra proofs, improved the technology in this sense, reducing the size complexity of arguments up to linear or even square root of the boolean circuit size (which is equivalent to the length of x). Interestingly enough, some of these authors, such as Jens Groth, are currently working in verifiable computation companies such as Nexus (see [Nex24]) due to the importance zero-knowledge proofs, specifically succinct ones, have in this field.

Clearly, the relaxation of the computational ability of provers led to a reduction in the size of arguments compared to the proof sizes. However, it was not until 2012 when Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer introduced zk-SNARKs, which led to an even greater reduction in size and verification time. This technology is currently being used in Blockchain, as we will later study in depth in Chapter 5.

3.1.1. zk-SNARGs and zk-SNARKs

So far in this dissertation, we have represented proof systems with the use of Turing machines, following what the authors in the literature did when presenting the technology. However, zero-knowledge proof systems are nowadays presented with the use of arithmetic circuits due to their simpler implementation. This represents a change of paradigm we must bear in mind during the following sections of the dissertation. Let us first talk about languages associated with boolean circuits.

Let us consider the language of Boolean circuit satisfiability, which is formed by the binary strings such that there exists an auxiliary input that makes the Boolean circuit output 1 when both values are fed to the circuit. More formally we express it as $L_c = \{x = \{0,1\}^n$ such that there exists $w \in \{0,1\}^m$ such that $C(x,w) = 1\}$, where n is the size of the statements, m the size of the witness and C is the boolean circuit. We can define a language membership problem from this language such that a prover wants to prove that a predefined $x \in \{0,1\}^n$ satisfies the circuit when a precise $w \in \{0,1\}^m$ is simultaneously fed to the circuit C. This is called the Boolean circuit satisfiability problem.

A trivial (non-zero-knowledge) proof system for this problem with a predefined $x \in \{0,1\}^n$ would consist of the prover sending w to the verifier and the verifier checking whether $C(x,w) = 1$ or not. The size of this proof system would be $|w|$, because that is the length in bytes of the message the prover would need to send to the verifier. Note this is the exact definition we previously used to define the size of an argument. It is possible to reduce the size of this proof system even more, and we use the lower bound of this reduction to define what we understand as succinctness. We say that a non-interactive proof (or argument) system is succinct for L_c if

$$\begin{cases} \text{the length of the proof } \pi \text{ satisfies that } |\pi| = \text{poly}(\lambda, \log_2(C)), \text{ and} \\ \text{the verification time is } \text{poly}(\lambda, |x|, \log_2(|C|)). \end{cases}$$

Where $\text{poly}(\lambda, \log_2(|C|))$ is a polynomial on two variables and $\text{poly}(\lambda, |x|, \log_2(|C|))$ is a polynomial on three variables, λ being a constant or security parameter, $|x|$ the size in bits of the statement and $\log_2(|C|)$ the log of the size of the boolean circuit (log of the number of bits needed to represent the circuit). Then, a SNARG is a succinct non-interactive argument

where succinctness is defined as described above and argument as explained in Section 3.1.

Some readers may wonder why we have defined succinctness solely based on this particular language problem. This is because the Boolean circuit satisfiability problem is NP-complete, which means that any NP problem can be reduced to it, and consequently, the results obtained here apply to any NP problem.

Once we have understood what SNARGs are, we can define SNARKs as SNARGs that are not only arguments but arguments of knowledge. This means, similarly to what we saw with proofs in Section 2.4, that the prover proves that a statement is true and that he knows a witness for that statement. Consequently, zk-SNARKs are succinct, as defined above, arguments of knowledge where minimum information is disclosed by the proof.

There are many available protocols that implement zk-SNARKs, such as Pinocchio, Groth16 or PLONK. In this dissertation, we will present Groth16, explaining how it works and understanding its mathematical fundamentals. We will also attach the code used to obtain the polynomials, elliptic curve and final proof (following the Groth16 protocol) of an example problem we will define.

3.2. Modelling problems to construct zk-SNARKs

Before implementing and explaining the Groth16 protocol, we will formalize how we will present statements and proofs. Throughout the following sections, we will use Dan Boneh's Stanford lecture notes on zk-SNARKs ([Bon23b, Bon23a]) and the MoonMath manual found in [Aut24] provided by the ZKProof organization.

3.2.1. Statements and proofs

As we have seen throughout the dissertation, in a zero-knowledge proof of knowledge, we aim to prove the knowledge of a statement's truth without disclosing any information about why the statement is true. This last part implies that we cannot plainly show the witness to prove the statement. Therefore, from now on, we will consider splitting the proof into two parts. A public part or instance, and a hidden or private part called the witness.

We define a decision function as $\mathrm{R} : \sum_I \times \sum_W \ \rightarrow \{true, false\}$ where $\sum_I$ and $\sum_W$ are two alphabets from which the instance and the witness are respectively extracted. This would be the x and the w referred to in the boolean circuit satisfiability language previously presented.

We define a *statement* S as a claim that, provided an instance $i \in \sum_I$, there exists a witness $w \in \sum_W$ such that $R(i, w) = true$. Let us see this with an example. Imagine we know that a number $x \in \mathbb{Z}_{19}$ can be expressed as the product of five numbers where one of them is subtracted 2 and the product is added to three times a sixth number, i.e.,

$$\text{there exists } x_1, x_2, x_3, x_4, x_5, x_6 \in \mathbb{Z}_{19} \text{ such that } (x_1 - 2) \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 + 3 \cdot x_6 = x. \quad (3.1)$$

Then we could choose (this choice is arbitrary) the witness to be $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ and the instance to be x. The statement would be, 'Given the field element x, there are six elements such that the product of five of them, where one is subtracted 2, added to the triple of the sixth, is equal to x'. An instance of this problem could be $i = \{1\}$ and a witness $w = \{4, 3, 3, 5, 1, 2\}$. We could also have chosen the witness to be $\{x_1, x_2, x_6\}$ and the instance to be $\{x, x_3, x_4, x_5\}$. Some particular instance of this problem could now be $i = \{4, 3, 2\}$ and a witness $w = \{1, 3, 5, 1\}$.

The decision function above must be translated into a model representing bounded computations. The two most commonly used models are Arithmetic circuits and R1CS.

3.2.2. Rank-1 Constraint Systems or R1CS

The decision function for the problem presented above could be written in code as shown in Listing 3.1, and we would like to know if there exist elements in $\mathbb{Z}_{19}$ for variables x_1 to x_6 such that $f(x_1, x_2, x_3, x_4, x_5, x_6) = x$ where x is a predefined element in $\mathbb{Z}_{19}$.

Listing 3.1: Decision function for the example above assuming all values and operations are in $\mathbb{Z}_{19}$

```
def f(x1,x2,x3,x4,x5,x6):
    return (x1 - 2) * x2 * x3 * x4 * x5 + 3 * x6
```

Assuming $f : {\mathbb{Z}_{19}}^6 \to \mathbb{Z}_{19}$ and $x \in \mathbb{Z}_{19}$ are predefined, we will have a valid proof if the prover provides values such that the program defined in Listing 3.2 returns *True*. The proof would consist of the values of variables x_1 to x_6 (witness), assuming the program and the instance x are part of the common information shared by the prover and the verifier.

Listing 3.2: Program associated to the decision problem of the example

```
w = [x1,x2,x3,x4,x5,x6]
def proof(f,x,w):
    return f(w) == x
```

We aim to provide this program, written in a specific programming language (in this case, Python), with a mathematical structure. This will allow the proof to be understood independently of the programming language or the exact definition of the function f. We will use *Rank-1 (quadratic) Constraint Systems* to achieve this.

Rank-1 Constraint Systems, or R1CS, are systems of quadratic equations that keep track of all the possible values variables can take during computations for later enforcing the program execution to behave precisely as expected. Thus, solutions to those constraint systems are proofs of proper program execution.

Given a field $\mathbb{F}$, three positive integers k, n, m and elements ${a_j}^i, {b_j}^i, {c_j}^i \in \mathbb{F}$ such that $1 \leq j \leq n + m$ and $1 \leq i \leq k$, being k the number of equations or restrictions, n the number of instances and m the number of witnesses, a R1CS is given in (3.2).

$$(a_0^i + \sum_{j=1}^{n} a_j^i \cdot I_j + \sum_{j=1}^{m} a_{n+j}^i \cdot W_j) \cdot (b_0^i + \sum_{j=1}^{n} b_j^i \cdot I_j + \sum_{j=1}^{m} b_{n+j}^i \cdot W_j) = c_0^i + \sum_{j=1}^{n} c_j^i \cdot I_j + \sum_{j=1}^{m} c_{n+j}^i \cdot W_j$$
$$\text{for } 1 \leq i \leq k. \tag{3.2}$$

Let us consider the provided example in $\mathbb{Z}_{19}$. As R1CS are quadratic equation systems, products of three or more elements are not accepted computations. Transforming non-valid equations into valid-format equations is called *flattening*. In the original problem (with the original instance and witness choices), the restriction was defined by $(W_1 - 2) \cdot W_2 \cdot W_3 \cdot W_4 \cdot W_5 + 3 \cdot W_6 = I_1$. We flatten it by creating the auxiliary variables W_7, W_8 and W_9 in (3.3).

$$\begin{aligned} (W_1 - 2) \cdot W_2 &= W_7 && \text{constraint 1,} \\ W_7 \cdot W_3 &= W_8 && \text{constraint 2,} \\ W_8 \cdot W_4 &= W_9 && \text{constraint 3,} \\ W_9 \cdot W_5 &= I_1 - 3 \cdot W_6 && \text{constraint 4.} \end{aligned} \tag{3.3}$$

To verify this system satisfies (3.2), we simply have to find ${a_j}^i, {b_j}^i, {c_j}^i \in \mathbb{Z}_{19}$ such that (3.3) can be written in the R1CS format. In this case, the choices are given in Table 3.1, Table 3.2 and Table 3.3.

i	a_0^i	a_1^i	a_2^i	a_3^i	a_4^i	a_5^i	a_6^i	a_7^i	a_8^i	a_9^i	a_{10}^i
1	-2	0	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	0	1

Table 3.1.: Coefficients of the first factors of the products.

i	b_0^i	b_1^i	b_2^i	b_3^i	b_4^i	b_5^i	b_6^i	b_7^i	b_8^i	b_9^i	b_{10}^i
1	0	0	0	1	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0

Table 3.2.: Coefficients of the second factors of the products.

i	c_0^i	c_1^i	c_2^i	c_3^i	c_4^i	c_5^i	c_6^i	c_7^i	c_8^i	c_9^i	c_{10}^i
1	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	-3	0	0	0

Table 3.3.: Coefficients of the right side of the equalities.

This process can also be done in the opposite direction, such that from an R1CS, we can define a decision problem and, consequently, a formal language. The statement that we would obtain would be 'Given an instance I, there is a witness W such that $(I; W)$ is a solution to the R1CS'.

Auxiliary structures called arithmetic circuits are used to compute the solutions of these systems efficiently and are one of the two most used models aforementioned.

3.2.3. Arithmetic cicuits

While on boolean circuits, gates could only represent AND or OR operations, arithmetic operations can be included on arithmetic circuits. We fix a finite field $\mathbb{F}$. We define an arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ as a directed acyclic graph whose nodes are labelled with the mathematical operators $\{+, -, \times\}$ and the inputs are labelled $\{1, x_1, ..., x_n\}$ where $n \in \mathbb{N}$.

Note the $\{-\}$ operator is sometimes not considered valid, although we will assume it is. We can make this assumption because a circuit that returns the difference of two numbers can be easily built using only one $\{+\}$ gate and one $\{\times\}$ gate. Each circuit uniquely defines an n-variate polynomial.

The inputs of the arithmetic circuit are a set of elements in $\mathbb{F}$. In the example we have been working on, we have set these inputs to be the witness of the problem. We want to verify whether the output of the arithmetic circuit is equal to the instance of the problem. To define a circuit, we must write the function that defines the decision problem as a concatenation of sums, differences and multiplications. We demonstrate how is done in (3.4) and use ADD to denote sums, SUB to denote difference and MUL to denote multiplications.

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5, x_6) &= (x_1 - 2) \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 + 3 \cdot x_6 \\ &= (((x_1 - 2) \cdot x_2) \cdot x_3) \cdot x_4) \cdot x_5 + 3 \cdot x_6 \\ &= ADD(MUL(MUL(MUL(MUL(SUB(x_1, 2), x_2), x_3), x_4)x_5), MUL(3, x_6)). \end{aligned} \tag{3.4}$$

We are now ready to define the arithmetic circuit in Figure 3.1, making the same choice of instance and witness as before. We can obtain the R1CS associated with this circuit by following the inverse process (but we will not focus on that in this dissertation). It is known that every R1CS has an arithmetic circuit associated with it, and every arithmetic circuit has an R1CS associated with it. This association does not necessarily have to be unique.

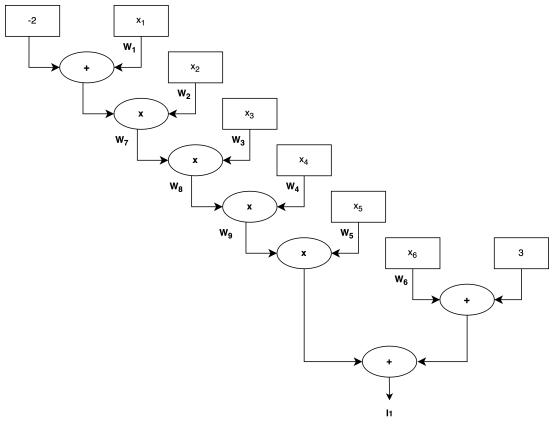


Figure 3.1.: An arithmetic circuit representing the example whose R1CS we found. Source: Elaborated by the author.

Any of these two models (R1CS and arithmetic circuits) can be transformed into another computational model called *Quadratic Arithmetic Program* or *QAP*. The reason why this is done is to achieve the desired succinctness. R1CS implies several matrices pointwise multiplications (Hadamard product), which are inefficient and would make the proof size grow parallel to the number of constraints, the cardinality of the instance, and the witness's cardinality. Instead, we want to summarize the information using a single list of polynomials for each table above. When evaluated at predefined points, such polynomials will take the values indicated in each table cell. For instance, looking at Table 3.1, the first polynomial of the first list (A_0) will take the value -2 when evaluated in the first point. Looking at Table 3.2, the

fifth polynomial of the second list (B_4) will take the value 1 when evaluated in the second point.

3.2.4. Quadratic Arithmetic Programs or QAPs

Let $\mathbb{F}$ be the same field defined above and R an R1CS over $\mathbb{F}$. We assume the number of constraints in the system is strictly smaller than the number of non-zero elements in $\mathbb{F}$. Furthermore, let ${a_j}^i, {b_j}^i, {c_j}^i \in \mathbb{F}$ for every $0 \leq j \leq n+m$ and $1 \leq i \leq k$ be the constants that form the quadratic equation system and let $\{m_1, ..., m_k\}$ be k random elements from $\mathbb{F}$. As we will be working with $\mathbb{Z}_{19}$, all these elements will be invertible, which is a requirement (if we were working with another field, we would have to make sure these elements had an inverse). We define a Quadratic Arithmetic Program as the set of polynomials over the field,

$$QAP(R) = \{T \in \mathbb{F}[x], \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m}\},$$

where $T(x) := \prod_{l=1}^{k}(x - m_l)$ is the called target polynomial of the QAP. A_j, B_j, C_j are three unique degree $k-1$ polynomials obtained by imposing $A_j(m_i) = {a_j}^i, B_j(m_i) = {b_j}^i, C_j(m_i) = {c_j}^i$ for all $j \in \{1, ..., n+m\}$ and $i \in \{1, ..., k\}$. It is worth noting that we can ensure the polynomials A_j, B_j, C_j are uniquely defined by these conditions because we are imposing their value in k points (the random predefined values m) and we can compute them by using Lagrange interpolation. The definitions of the $1 + 3(n+m)$ polynomials indicate the process that must be followed to compute the QAP of our R1CS defined by Table 3.1, Table 3.2 and Table 3.3 and the equations in (3.2) for $k = 4$.

Let us now start by choosing 4 random invertible elements from $\mathbb{Z}_{19}$, let $\{m_1, m_2, m_3, m_4\} = \{11, 5, 7, 3\}$.

Now, we can compute $T(x) = (x-11)(x-5)(x-7)(x-3) \pmod{19} = x^4 - 26x^3 + 236x^2 - 886x + 1155 \pmod{19} = x^4 + 12x^3 + 8x^2 + 7x + 15$. Once this is defined, we impose the conditions on the polynomials A_j, B_j and C_j as seen in Tables Table 3.4, Table 3.5 and Table 3.6.

Now, we can use Lagrange interpolation to obtain the polynomials that are not zero. From looking at the tables, we can derive that the only non-zero polynomials are $A_0, A_2, A_8, A_9, A_{10}$, $B_3, B_4, B_5, B_6, C_1, C_7, C_8, C_9, C_{10}$ and C_9. The resulting polynomials are given in Table 3.7 and were computed using the code in Appendix A.

We can conclude that the obtained $QAP(R)$, where R is the R1CS of the problem presented in (3.1), is $QAP(R) = \{x^4 + 12x^3 + 8x^2 + 7x + 15, \{A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m}$ with $\{A_j, B_j, C_j\}$ as described in Table 3.7.

m_i	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
Evaluated in $x = 11$	-2	0	1	0	0	0	0	0	0	0	0
Evaluated in $x = 5$	0	0	0	0	0	0	0	0	1	0	0
Evaluated in $x = 7$	0	0	0	0	0	0	0	0	0	1	0
Evaluated in $x = 3$	0	0	0	0	0	0	0	0	0	0	1

Table 3.4.: Imposing the values the polynomials of the first factor of the product side must take.

m_i	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
Evaluated in $x = 11$	0	0	0	1	0	0	0	0	0	0	0
Evaluated in $x = 5$	0	0	0	0	1	0	0	0	0	0	0
Evaluated in $x = 7$	0	0	0	0	0	1	0	0	0	0	0
Evaluated in $x = 3$	0	0	0	0	0	0	1	0	0	0	0

Table 3.5.: Imposing the values the polynomials of the second factor of the product side must take.

m_i	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
Evaluated in $x = 11$	0	0	0	0	0	0	0	0	1	0	0
Evaluated in $x = 5$	0	0	0	0	0	0	0	0	0	1	0
Evaluated in $x = 7$	0	0	0	0	0	0	0	0	0	0	1
Evaluated in $x = 3$	0	1	0	0	0	0	0	-3	0	0	0

Table 3.6.: Imposing the values the polynomials of the right side of the equality must take.

i	$A_i(x)$	$B_i(x)$	$C_i(x)$
0	$18x^3 + 15x^2 + 5x + 10$	0	0
1	0	0	$8x^3 + 6x^2 + 6x + 17$
2	$10x^3 + 2x^2 + 7x + 14$	0	0
3	0	$10x^3 + 2x^2 + 7x + 14$	0
4	0	$4x^3 + 11x^2 + 11x + 7$	0
5	0	$16x^3 + 14x + 1$	0
6	0	$8x^3 + 6x^2 + 6x + 17$	0
7	0	0	$14x^3 + x^2 + x + 6$
8	$4x^3 + 11x^2 + 11x + 7$	0	$10x^3 + 2x^2 + 7x + 14$
9	$16x^3 + 14x + 1$	0	$4x^3 + 11x^2 + 11x + 7$
10	$8x^3 + 6x^2 + 6x + 17$	0	$16x^3 + 14x + 1$

Table 3.7.: Polynomials obtained using Lagrange interpolation.

3.2.5. Verifying proofs given in QAPs

Provided a QAP and a pair of instance-witness $(I;W)$ such that the statement is satisfied, one can compute a proof of knowledge. The proof consists of the polynomial in (3.5). Note that this polynomial is formed using the polynomials previously computed and compresses all the information of (3.2), that had k equations, into a single one. This polynomial will be 0 in every one of the k random invertible values selected in the previous step and, therefore, will be divisible by the target polynomial $T(x)$.

$$P_{(I;W)} = (A_0 + \sum_{j=1}^{n} A_j \cdot I_j + \sum_{j=1}^{m} A_{n+j} \cdot W_j) \cdot (B_0 + \sum_{j=1}^{n} B_j \cdot I_j + \sum_{j=1}^{m} B_{n+j} \cdot W_j) \\ -(C_0 + \sum_{j=1}^{n} C_j \cdot I_j + \sum_{j=1}^{m} C^i_{n+j} \cdot W_j). \tag{3.5}$$

In our example, $(I = \{1\}; W = \{4,3,3,5,1,2\})$ will be the original instance and witness of the problem. Using (3.2), we can derive that $\{W_1, W_2, W_3, W_4, W_5, W_6, W_7, W_8, W_9\} = \{4,3,3,5,1,2,6,18,14\}$ and compute $P_{(\{1\},\{4,3,3,5,1,2,6,18,14\})}(x)$ resulting in the polynomial $17x^6 + 5x^5 + 18x^4 + 3x^3 + 17x^2 + 4x + 2$. We can check that

$$\text{the remainder of dividing } T(x) \text{ by } P_{(\{1\};\{6,3,4,2,1,8,12,14,11\})}(x) \text{ is } 0.$$

However, choosing a different pair instance-witness that does not verify (3.2) produces a division with a remainder different to 0. For instance,

$$\text{the remainder of dividing } T(x) \text{ by } P_{(\{1\};\{6,3,4,2,1,8,12,10,1\})}(x) \text{ is } 2x^3 + 11x^2 + 11x + 9,$$

because $(6-2) \cdot 3 \cdot 4 \cdot 2 \cdot 1 + 3 \cdot 8 \pmod{19} \equiv 6 \not\equiv 0$, and therefore it is an invalid proof of knowledge.

The $QAP(R)$ is assumed to be known by the prover and verifier. It nails the computation based on the random invertible values $\{m_1, ..., m_k\}$. Also, note that $(I;W)$ now contains all the variables needed to compute the solution following the R1CS structure (we have added the intermediate auxiliary variables). Thus, the prover must have flattened the problem and computed all the intermediate variables needed to verify that the pair satisfies the statement from which R is derived. However, the prover now needs a way of proving the knowledge of the quotient between $T(x)$ and $P_{(I;W)}(x)$, let us call it $H(x)$, and $(I;W)$ without revealing their values. This may seem hard, but it is not, based on the Schwartz-Zippel lemma.

Lemma 3.1. *If two polynomials are randomly evaluated at the same value, and their images match, their probability of being different is almost negligible.*

Two assumptions are made in this lemma. The sample space is large enough, and the polynomials have a bounded degree. Provided these two conditions hold, we can use this result to find whether $P_{(I;W)}(x)$ is equal to $H(x) \cdot T(x)$ by evaluating them in random points. It is important neither the prover nor the verifier, know in which points these polynomials are being evaluated. For that reason, we will use encrypted polynomial evaluation using elliptic curve cryptography such that, given a polynomial $P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_n \cdot x^n$ and an elliptic curve subgroup generator g, $g^{P(\tau)} = [P(\tau)]g = g^{a_0 + a_1 \cdot \tau + a_2 \cdot \tau^2 + ... + a_n \cdot \tau^n} = [g]a_0 + [\tau g]a_1 \cdot + [\tau^2 g]a_2 + ... + [\tau^n g]a_n$ where $[\tau^j g]$ is known for $j \in \{1, ..., n\}$.

3.3. Providing zero-knowledge to the proof. Phases of the Groth16 protocol

There are several flavors of zk-SNARKs, which differ in the steps taken to provide zero knowledge to the proof derived from the QAP. As we mentioned before, we will focus on the Groth16 protocol, presented by Jens Groth in 2016 [Gro16], because of its wide popularity across many Blockchain solutions. This technology heavily relies on the use of bilinear pairings (see Definition 1.21) and elliptic curves (see Section 1.10).

The parameters needed to compute the proof and execute a proving-verifying process using this method were obtained using the program provided in Appendix A and Appendix B. We used some snippets provided in [Aut24] to generalize the steps followed to create a suitable elliptic curve (following the method proposed by Barreto et al. in [BLS02]) for any prime order base field. We later used the protocol's specification to implement it step by step. It allows anyone to change the desired prime and build the subgroups, obtain their generators and build the Weil pairing from scratch, ensuring the curve has an embedding degree of 6 (which provides efficient computations). Therefore, if any reader wants to implement a different problem to the one presented in (3.1), built over a different prime field, the changes needed in the code of the second program are minimal.

The parameters needed to define the protocol are six,

$$\text{Groth}_{16}\text{-Param}(R) = (t, \mathbb{G}_1, \mathbb{G}_2, e(\cdot,\cdot), g_1, g_2),$$

and to continue in future steps studying the practical example, we will consider the particular value they take in the example presented in (3.1).

1. A prime number t defining the order of the finite field over which the polynomials of the QAP were built. In the example $t = 19$.

2. Two finite cyclic subgroups of order t named $\mathbb{G}_1$ and $\mathbb{G}_2$. In the example

$$\mathbb{G}_1 = \left\{ \begin{array}{c} \omega, (25136134, 25958637), (7275896, 13973809), (13312868, 13973809), \\ (25099705, 3908357), (15849615, 22464570), (26454473, 3908357), \\ (14805829, 25958637), (32934795, 10479742), (21322580, 3908357), \\ (21322580, 32530022), (32934795, 25958637), (14805829, 10479742), \\ (26454473, 32530022), (15849615, 13973809), (25099705, 32530022), \\ (13312868, 22464570), (7275896, 22464570), (25136134, 10479742) \end{array} \right\},$$

$$\mathbb{G}_2 = \left\{ \begin{array}{c} \omega, (1210439v^4, 12881022v^3), (17333865v^4, 35359426v^3), (29341190v^4, 35359426v^3), \\ (24534295v^4, 14278112v^3), (26201703v^4, 1078953v^3), (24328005v^4, 14278112v^3), \\ (17507987v^4, 12881022v^3), (17719953v^4, 23557357v^3), (24014458v^4, 14278112v^3), \\ (24014458v^4, 22160267v^3), (17719953v^4, 12881022v^3), (17507987v^4, 23557357v^3), \\ (24328005v^4, 22160267v^3), (26201703v^4, 35359426v^3), (24534295v^4, 22160267v^3), \\ (29341190v^4, 1078953v^3), (17333865v^4, 1078953v^3), (1210439v^4, 23557357v^3) \end{array} \right\}.$$

   These two subgroups come from an elliptic curve of prime order p, in our case $p = 36438379$ (see the code to find the source of this value). The particular curve of the

example is

$$BLS6 := \{(x, y) \text{ such that } y^2 = x^3 + 1 \text{ for all } x, y \in \mathbb{F}_{36438379}\}.$$

The first of the two subgroups is $\mathbb{G}_1[19]$ obtained by finding a point of order 19 in the elliptic curve. The second of the two subgroups is $\mathbb{G}_2[19]$, which is defined by those elements of the full 19-torsion group that are mapped to p times themselves under the Frobenius endomorphism as seen in (1.4).

3. A Weil pairing such that $e(\cdot, \cdot) : \mathbb{G}_1[t] \times \mathbb{G}_2[t] \to \mathbb{F}_{p^k}^*$ where p is a prime number. This is computed using SageMath's built-in function *weil_pairing()*, although it can be done manually following Miller's Algorithm.

4. A generator of each of the finite cyclic groups of order r named g_1 and g_2 respectively. In the example $g_1 = (25136134, 25958637)$ and $g_2 = (1210439v^4, 12881022v^3)$. Because they are finite cyclic prime order groups, any element of the groups (different to the point at infinity) is a generator. The subgroups $\mathbb{G}_1[19]$ and $\mathbb{G}_2[19]$ can also be expressed as follows, denoting each d-multiple of the generator as $[d]g_i = [d]_i$ for $i \in \{1, 2\}$.

$$\mathbb{G}_1 = \left\{\omega, ([j]_1)_{j=1}{}^{18}\right\},$$
$$\mathbb{G}_2 = \left\{\omega, ([j]_2)_{j=1}{}^{18}\right\}.$$

As the name indicates, zk-SNARKs are Non-Interactive and therefore, a set-up phase needs to precede the proving phase to allow that non-interactivity. During that phase, the so-called Common Reference String or CRS is created (see Subsection 2.8.3). This implies the necessity to rely on a third-party trusted setup to carry out this process. Most of the most up-to-date blockchain solutions implement multi-party computation to reduce the risks of trusting the third party. However, we will assume this is not the case in this example.

3.3.1. Setup phase

The first step of this phase is to sample five random elements from the elliptic curve. These elements are called the *simulation trapdoor* and are denoted as $ST = (\alpha, \beta, \gamma, \delta, \tau)$. In particular, τ is called the *secret evaluation point*. This is because the CRS provides enough information to compute the evaluation of any polynomial P at the point τ without knowing τ. We saw how this evaluation works below Lemma 3.1, assuming the representation of some powers of tau on the elliptic curve were known.

The simulation trapdoor is also called *toxic waste*, as it must be discarded after the setup phase has been completed for security reasons. Being in possession of these values and the Common Reference Strings generated in the setup phase would allow malicious parties to forge proofs of knowledge without having the knowledge proven. We will note the values (following the mathematical notation provided in the MoonMath Manual [Aut24], which closely follows the notation used in [Gro09]) as seen in (3.6).

$$CRS_{G_1} = \left\{ \begin{array}{l} g_1^{\alpha}, g_1^{\beta}, g_1^{\delta}, \left(g_1^{\tau^j}, \dots\right)_{j=0}^{\deg(T)-1}, \left(g_1^{\frac{\beta \cdot A_j(\tau)+\alpha \cdot B_j(\tau)+C_j(\tau)}{\gamma}}, \dots\right)_{j=0}^{n} \\ \left(g_1^{\frac{\beta \cdot A_{j+n}(\tau)+\alpha \cdot B_{j+n}(\tau)+C_{j+n}(\tau)}{\delta}}, \dots\right)_{j=1}^{m}, \left(g_1^{\frac{\tau^j \cdot T(\tau)}{\delta}}, \dots\right)_{j=0}^{\deg(T)-2} \end{array} \right\}, \tag{3.6}$$

$$CRS_{G_2} = \left\{ g_2^{\beta}, g_2^{\gamma}, g_2^{\delta}, \left(g_2^{\tau^j}, \dots\right)_{j=0}^{\deg(T)-1} \right\}.$$

As previously mentioned, the information provided allows us to prove the evaluation value of a polynomial. In particular, $g_i^{P(\tau)} = g_i^{a_0 \cdot \tau^0} \cdot g_i^{a_1 \cdot \tau^1} \cdot g_i^{a_2 \cdot \tau^2} \cdots g_i^{a_k \cdot \tau^k} = (g_i^{\tau^0})^{a_0} \cdot (g_i^{\tau^1})^{a_1} \cdot (g_i^{\tau^2})^{a_2} \cdots (g_i^{\tau^k})^{a_k}$ where $i \in \{1,2\}$ and $P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_k \cdot x^k$ with $k < \deg(T)$.

We know this may seem very confusing at first, but we will explain the origin of this Common Reference String step by step. To do this, let us remember what the problem whose proof we are looking for is.

Given an R1CS R and instances I, the objective of the prover is to demonstrate the knowledge of the witness W such that $(I; W)$ is a solution to the R1CS. As studied in Subsection 3.2.4, these constraints are transformed into a set of polynomials denoted as $QAP(R)$. A new polynomial can be computed from these polynomials, containing all the information desired to be proven. We called this polynomial $P_{(I;W)}(x)$. We also defined $H(x)$ as $P_{(I;W)}(x)/T(x)$ where $T(x)$ was the target polynomial.

We can now explain why and how the parameters in (3.6) are defined the way they are.

The simplest way of defining the CRS would be by making the prover and verifier share the values in (3.7). Note they are simplified terms of the more complex Common Reference String presented in (3.6).

$$CRS_{simple} = \left\{ \left(g_1^{\tau^j}, \dots\right)_{j=0}^{\deg(T)-1}, \left(g_2^{\tau^j}, \dots\right)_{j=0}^{\deg(T)-1}, \left(g_1^{\tau^j \cdot T(\tau)}, \dots\right)_{j=0}^{\deg(T)-2} \right\}. \tag{3.7}$$

These values would allow the prover to generate g_1^A, g_1^B and ${g_1}^{C^*}$ as shown in (3.8), where

$$\begin{aligned} A(x) &= A_0(x) + \sum_{j=1}^{n} A_j(x) \cdot I_j + \sum_{j=1}^{m} A_{n+j}(x) \cdot W_j, \\ B(x) &= B_0(x) + \sum_{j=1}^{n} B_j(x) \cdot I_j + \sum_{j=1}^{m} B_{n+j}(x) \cdot W_j, \\ C(x) &= C_0(x) + \sum_{j=1}^{n} C_j(x) \cdot I_j + \sum_{j=1}^{m} C_{n+j}(x) \cdot W_j, \end{aligned}$$

and $C^*(x) = C(x) + H(x) \cdot T(x) = C(x) + P_{(I;W)}(x)$. Note that $A_j(x) = \sum_{k=0}^{deg(T)-1} a_{j,k} \cdot x^i$ where $a_{j,k}$ are the polynomial coefficients, and that we have only fully developed g_1^A in (3.8), as the calculi for g_1^B and ${g_1}^{C^*}$ are analogous. Note that we had noted $a_{j,k}$ as ${a_j}^k$, but we have changed the notation to avoid confusion with exponentiation in the elliptic curve.

$$
\begin{aligned}
g_1^A &= g_1^{A_0(\tau)} + \sum_{j=1}^{n} g_1^{A_j(\tau)} \cdot I_j + \sum_{j=1}^{m} g_1^{A_{n+j}(\tau)} \cdot W_j \\
&= \sum_{i=1}^{deg(T)-1} \left(g_1^{\tau^i}\right)^{a_{0,i}} + \sum_{j=1}^{n} I_j \cdot \sum_{i=1}^{deg(T)-1} \left(g_1^{\tau^i}\right)^{a_{j,i}} + \sum_{j=1}^{m} W_j \cdot \sum_{i=1}^{deg(T)-1} \left(g_1^{\tau^i}\right)^{a_{n+j,i}}, \\
g_2^B &= g_2^{B_0(\tau)} + \sum_{j=1}^{n} g_2^{B_j(\tau)} \cdot I_j + \sum_{j=1}^{m} g_2^{B_{n+j}(\tau)} \cdot W_j, \\
{g_1}^{C^*} &= g_1^{C_0(\tau)} + \sum_{j=1}^{n} g_1^{C_j(\tau)} \cdot I_j + \sum_{j=1}^{m} g_1^{C_{n+j}(\tau)} \cdot W_j + g_1^{H(\tau)\cdot T(\tau)}.
\end{aligned}
\tag{3.8}
$$

To complete the process, the verifier would only need to see if

$$
e(g_1^A, g_2^B) = e({g_1}^{C^*}, g_2). \tag{3.9}
$$

held, where the generator g_2 would be acting as the identity. Remember that $e(g_1^A, g_2^B) = e(g_1, g_2)^{A\cdot B}$ and $e({g_1}^{C^*}, g_2) = e(g_1, g_2)^{C^*\cdot 1}$. Therefore, the verifier needs to check whether $A(\tau)\cdot B(\tau) = C^*(\tau)$ where $C^*(\tau) = C(\tau) + P_{(I;W)}(\tau)$, or identically, whether $A(\tau)\cdot B(\tau) - C(\tau) = P_{(I;W)}(\tau)$. If the equality holds, both polynomials are identical because of the Schwartz-Zippel lemma (see Lemma 3.1). The prover must have known the value of $g_1^{H(\tau)}$ (and consequently $P_{(I;W)}$) based on the hardness of the discrete logarithm problem on elliptic curves.

This implementation has a major flaw. What happens if the prover acts dishonestly, picking three polynomials, such as the product of the first two equals the third one? Because the Common Reference String has not fixed the polynomials $A(x)$, $B(x)$ and $C^*(x)$, the prover could have chosen any three polynomials such that the product of the first two would be equal to the third one. Those polynomials could not relate to the statement being proven, and the verifier would not know this. Then, the verifier could not be sure whether the proof was valid. Groth16 solves this problem by fixing those polynomials into the Common reference string. However, ${g_1}^A$, ${g_2}^B$ and ${g_1}^{C^*}$ are not introduced in the Common Reference String directly but slightly modified with an offset. The representation of the polynomials on the elliptic curve is now unknown to both the prover and verifier. Furthermore, the public (calculated using the instance) and private (calculated using the witness) parts are separated and scaled by different constants that will cancel each other out when the verification is carried out. At last, random values are used during every setup phase to make every proof unique. Let us see how these steps are exactly taken.

1. Two random shifts, α and β, are introduced to the polynomials A and B, such that we define $A_{new} = A_{old} + \alpha$ and $B_{new} = B_{old} + \beta$ based on their old values and obtain the polynomials with coefficients on the elliptic curve presented in (3.10).

$$
\begin{aligned}
g_1^{A_{new}} &= g_1^{A_{old}} + g_1^{\alpha}, \\
g_2^{B_{new}} &= g_2^{B_{old}} + g_2^{\beta}.
\end{aligned}
\tag{3.10}
$$

   We can now compute the product of A_{new} times B_{new} and see what polynomial they are equal to. First we can express $A_{new}\cdot B_{new} = (A_{old} + \alpha)\cdot(B_{old} + \beta)$. Then is direct to obtain $A_{new}\cdot B_{new} = \alpha\beta + \beta A_{old} + \alpha B_{old} + A_{old}\cdot B_{old}$. Now, using (3.5) and $P_{(I;W)} =$

$H \cdot T$, it is direct to see $A_{new} \cdot B_{new} = \alpha\beta + \beta A_{old} + \alpha B_{old} + C + H \cdot T$, and consequently $A_{new} \cdot B_{new} = \alpha\beta + \beta A_{old} + \alpha B_{old} + C^*$.

Now the pairing in (3.9), has gone from checking if $e(g_1^A, g_2^B) = e({g_1}^{C^*}, g_2)$ to checking if

$$e(g_1^A, g_2^B) = e(g_1^\alpha, g_2^\beta) \cdot e({g_1}^{C^{**}}, g_2),$$

where $C^{**} = \beta A_{old} + \alpha B_{old} + C^*$. Forging using $e({g_1}^A, {g_2}^B)$ becomes significantly more complex (as long as the discrete logarithm problem is *hard* on elliptic curves) because the simulation trapdoor, containing the values of α and β, is deleted (and not known by anyone) after the CRS is computed. We will not study how forgeries are still possible (although very complex) and refer curious readers to [Gro16]. Note that we will now refer to A_{new} and B_{new} as A and B respectively.

2. As a further step, the public (instance) and private (witness) part of the polynomial C are separated such that the pairing becomes

$$e(g_1^A, g_2^B) = e(g_1^\alpha, g_2^\beta) \cdot e(g_1^{\sum_{j=0}^{n} I_j(\beta A_j + \alpha B_j + C_j)}, g_2) \cdot e(g_1^{\sum_{j=1}^{m} W_j(\beta A_{j+n} + \alpha B_{j+n} + C_{j+n}) + H \cdot T}, g_2).$$

This reduces the proof size and the computational effort the prover must do because the proof does not need to include computations using the instance.

The verifier must now use the set of instances $\{I\}_{j=0}^{n} = \{1, I_1, I_2, ..., I_n\}$ (note we use the convention of $I_0 = 1$), and the data provided by the trusted setup (in particular, the set of points in the bottom left of CRS_{G_1} in (3.6)), to compute the public part of C^{**}.

3. An extra layer of security is included to prevent the prover from forging a proof by using the public inputs. To do this, the verifier (public) and the prover (private) part are divided by the values γ and δ, respectively. Their encrypted versions are included in the CRS, so both parties can cancel them out if they behave honestly. The pairing now becomes

$$e(g_1^A, g_2^B) = e(g_1^\alpha, g_2^\beta) \cdot e(g_1^{\sum_{j=0}^{n} I_j \frac{\beta A_j + \alpha B_j + C_j}{\gamma}}, g_2^\gamma) \cdot e(g_1^{\sum_{j=1}^{m} W_j \frac{\beta A_{j+n} + \alpha B_{j+n} + C_{j+n}}{\delta} + \frac{H \cdot T}{\delta}}, g_2^\delta).$$

The inclusion of this security feature completes the unfeasibility of forgery. This is intuitive, although a rigorous proof is given [Gro16].

4. At last, two random values r and t are sampled from the base field and used to make two proofs of the same statement using the same pair $(I; W)$ to be completely different in two different executions. This provides full zero-knowledge as an attacker in possession of the $(I; W)$ pair, and its associated proof cannot verify whether the pair's proof is indeed the one he has because he does not know the values r and t. The polynomials will end up being the ones presented in (3.11).

$$\begin{aligned} A(x) &= \alpha + \sum_{j=0}^{n} A_j(x) \cdot I_j + \sum_{j=1}^{m} A_{n+j}(x) \cdot W_j + r\delta, \\ B(x) &= \beta + \sum_{j=0}^{n} B_j(x) \cdot I_j + \sum_{j=1}^{m} B_{n+j}(x) \cdot W_j + s\delta, \\ C(x) &= \beta A(x) + \alpha B(x) + H(x) \cdot T(x) + s \cdot A(x) + r \cdot B(x) - rs\delta. \end{aligned} \tag{3.11}$$

Keep in mind that the product of $A(x)$ and $B(x)$ is equal to $C^{**}(x)$ such that

$$\begin{aligned} C^{**}(x) = \alpha\beta + H(x)\cdot T(x) \\ + \beta\left(\sum_{j=0}^{n} A_j(x)\cdot I_j + \sum_{j=1}^{m} A_{n+j}(x)\cdot W_j\right) + \alpha\left(\sum_{j=0}^{n} B_j(x)\cdot I_j + \sum_{j=1}^{m} B_{n+j}(x)\cdot W_j\right) \\ + s\left(\alpha + \sum_{j=0}^{n} A_j(x)\cdot I_j + \sum_{j=1}^{m} A_{n+j}(x)\cdot W_j\right)\delta \\ + r\left(\beta + \sum_{j=0}^{n} B_j(x)\cdot I_j + \sum_{j=1}^{m} B_{n+j}(x)\cdot W_j\right)\delta \\ + rs\delta^2. \end{aligned} \tag{3.12}$$

To obtain the expression above we have used that $H(x)\cdot T(x)$ is equal to the product

$$\left(\sum_{j=0}^{n} A_j(x)\cdot I_j + \sum_{j=1}^{m} A_{n+j}(x)\cdot W_j\right)\cdot\left(\sum_{j=0}^{n} B_j(x)\cdot I_j + \sum_{j=1}^{m} B_{n+j}(x)\cdot W_j\right).$$

We have also extracted $s\delta$ and $r\delta$ as common factors to obtain the fifth and sixth terms.

This, using the product property of bilinear mapping e, leads to the final pairing $e(g_1^A, g_2^B)$ needing to be equal to the product of the following factors that will be found on the right side of the equality that must be satisfied during the verification process.

a) $e(g_1^{\alpha}, g_2^{\beta})$ associated with the first term of (3.12).

b) $e(g_1^{\sum_{j=0}^{n} I_j \frac{\beta A_j + \alpha B_j + C_j}{\gamma}}, g_2^{\gamma}) \cdot e(g_1^{\sum_{j=1}^{m} W_j \frac{\beta A_{j+n} + \alpha B_{j+n} + C_{j+n}}{\delta} + \frac{H\cdot T}{\delta}}, g_2^{\delta})$ associated with the second, third and fourth terms of (3.12).

c) $e((g_1^A)^s, g_2^{\delta}) \cdot e(g_1^{-\delta rs}, g_2^{\delta})$ associated with the fifth term of (3.12).

d) $e((g_1^B)^r, g_2^{\delta}) \cdot e(g_1^{-\delta rs}, g_2^{\delta})$ associated with the sixth term of (3.12).

e) $e(g_1^{\delta rs}, g_2^{\delta})$ associated with the last term of (3.12).

In conclusion,

$$\begin{aligned} e(g_1^A, g_2^B) = e(g_1^{\alpha}, g_2^{\beta}) \cdot e(g_1^{\sum_{j=0}^{n} I_j \frac{\beta A_j + \alpha B_j + C_j}{\gamma}}, g_2^{\gamma}) \cdot e(g_1^{\sum_{j=1}^{m} W_j \frac{\beta A_{j+n} + \alpha B_{j+n} + C_{j+n}}{\delta} + \frac{H\cdot T}{\delta}}, g_2^{\delta}) \\ \cdot e((g_1^A)^s, g_2^{\delta}) \cdot e((g_1^B)^r, g_2^{\delta}) \cdot e(g_1^{-\delta rs}, g_2^{\delta}). \end{aligned} \tag{3.13}$$

With this expression in mind and carefully looking in (3.15) at the parameters the prover must compute (as defined by the protocol), the CRS provides the exact parameters needed to compute the proof and later verify it.

Let us now return to our particular example. We obtain the Common Reference String values using the implemented code provided in Appendix B and defining $ST = (\alpha, \beta, \gamma, \delta, \tau) = (6, 5, 4, 3, 2)$. Note that this choice of parameters is arbitrary. We invite readers to copy the code provided in Appendix B, modify the values and verify that a valid proof is generated. The obtained CRS is given in (3.14).

$$
\begin{aligned}
CRS_{\mathbb{G}_1} &= \left\{ \begin{array}{r} (26454473, 3908357), (15849615, 22464570), (13312868, 13973809), \\ \{(25136134, 25958637), (7275896, 13973809), (25099705, 3908357), \\ (32934795, 10479742)\}, \{(15849615, 13973809), (25099705, 32530022)\}, \\ \{(13312868, 22464570), (25099705, 3908357), (25136134, 25958637), \\ (21322580, 32530022), (26454473, 3908357), (13312868, 22464570), \\ (32934795, 25958637), (25136134, 10479742), (26454473, 32530022)\}, \\ \{(14805829, 25958637), (15849615, 13973809), (21322580, 3908357)\} \end{array} \right\}, \\
CRS_{\mathbb{G}_2} &= \left\{ \begin{array}{r} (26201703v^4, 1078953v^3), (24534295v^4, 14278112v^3), \\ (29341190v^4, 35359426v^3), \{(1210439v^4, 12881022v^3), \\ (17333865v^4, 35359426v^3), (24534295v^4, 14278112v^3), \\ (17719953v^4, 23557357v^3)\} \end{array} \right\}.
\end{aligned}
\tag{3.14}
$$

We can again express the elements in (3.14) as multiples of the generators of the corresponding subgroups.

$$
\begin{aligned}
CRS_{\mathbb{G}_1} &= \left\{ \begin{array}{l} [6]_1, [5]_1, [3]_1, \{[1]_1, [2]_1, [15]_1, [8]_1\}, \{[14]_1, [15]_1\}, \\ \{[16]_1, [4]_1, [1]_1, [10]_1, [6]_1, [16]_1, [11]_1, [18]_1, [13]_1\}, \{[7]_1, [14]_1, [9]_1\} \end{array} \right\}, \\
CRS_{\mathbb{G}_2} &= \left\{ [5]_2, [4]_2, [3]_2, \{[1]_2, [2]_2, [4]_2, [8]_2\} \right\}.
\end{aligned}
$$

The proof can be built once the prover and verifier share these values.

3.3.2. Proving phase

The coefficients the prover needs to compute are described in (3.15). Note that they are coherent with the expression in Ecuación 3.13.

$$
\begin{aligned}
g_1^W &= \left(g_1^{\frac{\beta \cdot A_{1+n}(\tau) + \alpha \cdot B_{1+n}(\tau) + C_{1+n}(\tau)}{\delta} W_1} \right) \cdots \left(g_1^{\frac{\beta \cdot A_{m+n}(\tau) + \alpha \cdot B_{m+n}(\tau) + C_{m+n}(\tau)}{\delta} W_m} \right), \\
g_1^A &= g_1^{\alpha} \cdot g_1^{A_0(\tau)} \cdot \left(g_1^{A_1(\tau)} \right)^{I_1} \cdots \left(g_1^{A_n(\tau)} \right)^{I_n} \cdots \left(g_1^{A_{n+1}(\tau)} \right)^{W_1} \cdots \left(g_1^{A_{n+m}(\tau)} \right)^{W_m} \cdot \left(g_1^{\delta} \right)^{r}, \\
g_1^B &= g_1^{\beta} \cdot g_1^{B_0(\tau)} \cdot \left(g_1^{B_1(\tau)} \right)^{I_1} \cdots \left(g_1^{B_n(\tau)} \right)^{I_n} \cdots \left(g_1^{B_{n+1}(\tau)} \right)^{W_1} \cdots \left(g_1^{B_{n+m}(\tau)} \right)^{W_m} \cdot \left(g_1^{\delta} \right)^{s}, \\
g_2^B &= g_2^{\beta} \cdot g_2^{B_0(\tau)} \cdot \left(g_2^{B_1(\tau)} \right)^{I_1} \cdots \left(g_2^{B_n(\tau)} \right)^{I_n} \cdots \left(g_2^{B_{n+1}(\tau)} \right)^{W_1} \cdots \left(g_2^{B_{n+m}(\tau)} \right)^{W_m} \cdot \left(g_2^{\delta} \right)^{s}, \\
g_1^C &= g_1^W \cdot g_1^{\left(\frac{H(\tau) \cdot T(\tau)}{\delta} \right)} \cdot \left(g_1^A \right)^{s} \cdot \left(g_1^B \right)^{r} \cdot \left(g_1^{\delta} \right)^{-r \cdot s}.
\end{aligned}
\tag{3.15}
$$

Note how the random shifts (g_1^{α}, g_1^{β} and g_2^{β}) on the elliptic curve made to the polynomials ${g_1}^A$, ${g_1}^B$ and ${g_2}^B$, are common to the same setup process, however, $\left(g_1^{\delta}\right)^r$, $\left(g_1^{\delta}\right)^s$ and $\left(g_2^{\delta}\right)^s$ are random shifts depending on the random values chosen during the proving phase. In our example, we use the code in Appendix B and compute these values, obtaining the partial

values given in (3.16). The two random values used were $r = 10$ and $s = 5$.

$$\begin{aligned} g_1^W &= (7275896, 22464570) = [17]_1, \\ g_1^A &= (25099705, 3908357) = [4]_1, \\ g_1^B &= (32934795, 10479742) = [8]_1, \\ g_2^B &= (17719953v^4, 23557357v^3) = [8]_2, \\ g_1^C &= \omega = [0]_1. \end{aligned} \tag{3.16}$$

In summary, we have represented the evaluation in τ of the polynomials $\{P_{I;W}, A_j, B_j, C_j \in \mathbb{F}[x]\}_{j=0}^{n+m}$ in the r-torsion groups by exponentiating their generators and using the information provided in the Common Reference String. Note that the values of the ST are not used directly (only through the CRS) as they are supposed to be destroyed after the CRS is computed.

The resulting zero-knowledge succinct non-interactive argument of knowledge is represented by

$$\pi = (g_1^A, g_1^C, g_2^B).$$

In the example on which we have been working, we obtain

$$\pi = ([4]_1, \omega, [8]_2).$$

Provided $(I; W) = (\{1\}, \{4, 3, 3, 5, 1, 2, 6, 18, 14\}$, choosing $ST = (6, 5, 4, 3, 2)$ during the setup phase and $(r, s) = (10, 5)$ during the proving phase.

3.3.3. Verifying phase

In this phase, a verifier, who is in possession of the set of the Groth16 parameters, the R1CS, the CRS generated in the setup phase, the set of instances I and the proof π provided by the prover, aims to check whether the proof is valid. In other words, aims to check whether the prover knows a witness W such that the R1CS returns *True* when fed $(I; W)$. A positive result of this verification ensures the statement being proven is true, and the prover has a witness of such statement. It needs to be assumed the verifier is able to efficiently compute the Weil pairing over the two r-torsion groups ($\mathbb{G}_1$ and $\mathbb{G}_2$) provided as Groth16 parameters. Because of that, we made a special effort to follow the method from [BLS02] (as described in [Aut24]) to obtain a curve with a small embedding degree.

The verifier needs to compute the point on the curve given by

$$g_1^I = \left(g_1^{\frac{\beta \cdot A_0(\tau) + \alpha \cdot B_0(\tau) + C_0(\tau)}{\gamma}}\right) \cdot \left(g_1^{\frac{\beta \cdot A_1(\tau) + \alpha \cdot B_1(\tau) + C_1(\tau)}{\gamma}}\right)^{I_1} \cdots \left(g_1^{\frac{\beta \cdot A_n(\tau) + \alpha \cdot B_n(\tau) + C_n(\tau)}{\gamma}}\right)^{I_n}.$$

Using the elements previously calculated, some elements of the CRS and the Weil Pairing, the verification the verifier must do consists of checking whether

$$\mathrm{e}(g_1^A, g_2^B) = \mathrm{e}(g_1^\alpha, g_2^\beta) \cdot \mathrm{e}(g_1^I, g_2^\gamma) \cdot \mathrm{e}(g_1^C, g_2^\delta). \tag{3.17}$$

This is evident by comparing (3.13) and the above expression considering the definitions provided in (3.15).

In the case of our example, $g_1^I = (21322580, 32530022) = [10]_1$. Using that the Weil Pairing is a bilinear mapping, the verification that needs to occur is equivalent to checking if

$$A \cdot B \equiv \alpha \cdot \beta + I \cdot \gamma + C \cdot \delta \pmod{19}.$$

However, note that this is not the equation the verifier will check as none of the simulation trapdoor values are known by him. Furthermore, the discrete logarithm problem would be hard if we had chosen a larger prime number and therefore, the values of A, B and C would be impossible to compute. The verifier will always have to check whether the proof is valid or not by verifying if (3.17) holds. For this reason, and as we mentioned before, the verifier must be able to compute the Weil Pairing.

We can now finish this section by verifying the proof provided by the prover in the example is valid. Summarising all the results obtained for the example so far, and using that each d-multiple of the generator was denoted as $[d]g_i = [d]_i$ for $i \in \{1, 2\}$, we have (3.18).

$$\begin{aligned} A &= 4, \\ B &= 8, \\ I &= 10, \\ C &= 0. \end{aligned} \tag{3.18}$$

We also know the simulation trapdoor was $ST = (\alpha, \beta, \gamma, \delta, \tau) = (6, 5, 4, 3, 2)$. Using that information, we can demonstrate the equation would hold as

$$4 \cdot 8 \equiv 6 \cdot 5 + 10 \cdot 4 + 0 \cdot 3 \pmod{19},$$

and consequently, the verifier will accept the proof as valid.

This concludes this section on zk-SNARKs, and we can proceed to study how these tools are used in Blockchain after presenting the fundamentals that support this technology.

4. Blockchain

4.1. Introduction

A blockchain is a digitally distributed public ledger containing transactions between participating parties of a network. These transactions are recorded in blocks linked to each other following a specific structure and rules. The most characteristic trait of a blockchain is that blocks can only be altered once they have been added to the chain if all the blocks added later to this one are also modified.

The initial stages of the technology were presented in David Chaum's 1982 Ph.D. dissertation at the University of California Berkeley: 'Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups' [Cha82]. Wei Dai's b-money [Dai] was the first proposal for creating money by solving computational challenges. It also presented the idea of achieving consensus through the interaction of individuals. However, this proposal lacks details on how this technology could be implemented. It was not until 2005 when Hal Finney [Fin] presented a functional Proof-of-Work system that depended on a trusted authority in a central server. The decentralization and use of computational challenges were finally put together theoretically and practically in Bitcoin. In 2008, an anonymous author or group of authors working under the name of Satoshi Nakamoto presented the cryptocurrency in [Nak]. A cryptocurrency is a digital currency implemented in Blockchain, and it is the technology's main and most developed use case. Consequently, when studying Blockchain, cryptocurrencies are a crucial focus and will be pivotal in this dissertation, too.

Bitcoin was the first widely known cryptocurrency whose main objective was to eliminate the need for a trusted third party when doing transactions between individuals. Following Bitcoin, several other popular cryptocurrencies were created, such as Ethereum, Cardano, and Monero. Each cryptocurrency has strengths and weaknesses based on the possibilities it exploits in Blockchain. This section will first present the main technical and mathematical tools used to implement their basic characteristics. Furthermore, the main processes of Bitcoin and Ethereum will be explained. Finally, privacy and scalability will be presented as the main problems of cryptocurrencies and the approaches taken to solve these problems.

4.2. Prior knowledge needed

As we did in Chapter 1, we will present a few basic concepts related to cryptography and Blockchain. These are general concepts that will be used throughout the following sections. More precise definitions of cryptocurrencies, specifically Bitcoin and Ethereum, will be given in Section 4.3. Let us start by presenting the main objective of cryptocurrencies: reducing the necessity of a central authority.

4.2.1. Decentralized systems

In the context of cryptocurrencies such as Bitcoin or Ethereum, a decentralized system is the main characteristic that differs from classic currency forms.

Definition 4.1. A decentralized system is one in which its members operate on local information to achieve global goals without a governing central authority.

For instance, colonies of bees or ants make decentralized decisions that achieve great results when combined with the work of their peers. A single bee cannot build a hive; however, thousands of bees can do it when their achievements are combined.

A particular example of decentralized systems are P2P systems. This is the paradigm followed by the cryptocurrencies Bitcoin and Ethereum.

Definition 4.2. A peer-to-peer (P2P) network consists of a group of computers or individuals interacting with each other, having similar permissions and responsibilities when processing data.

In this network, there are no devices whose only purpose is to serve or receive data. All of the *nodes* in the network, which is how the participants in the network are called, are clients and servers simultaneously.

A decentralized system needs a protocol for authentication, privacy and non-repudiation. Here is where public-key cryptography comes into play.

4.2.2. Public-key cryptography

In 1976, Whitfield Diffie and Martin Hellman in[DH76], and Ralph Merkleor invented Public-key cryptography while working on a project for his Berkeley Computer Science course. The curious reader may find it interesting to read in [Mek] about the story of how Ralph Merkleor invented this powerful technology and was still rejected by, first, his professor and, later, the publisher. That *rejected* technology, which is usually called *Asymmetric cryptography*, is one of the most crucial advancements ever to occur in the area. Public-key cryptography is based on the use of pairs of related keys.

Definition 4.3. A private key is essentially a randomly generated secret number. The private key is only known by the person who generated it. It is used to decrypt a message encrypted with the public key. It is also used to create a digital signature.

Every private key has a unique public key (another number) associated with it. It is usually widely distributed. Any user uses it to encrypt a message sent to the public/private key pair's owner. It is also used to verify the owner's digital signature.

Digital signatures have been mentioned in Subsection 2.8.1 in the first chapter of this dissertation. Their uses are widely spread across many identification schemes and provide integrity in Blockchain.

Definition 4.4. A digital signature is the encrypted hash of a message. It allows the reader to verify the owner of the signature. It also lets the reader check if the message has been altered after the message's signature was created.

As we mentioned before, cryptocurrencies aim to avoid the need for a central authority governing the transactions between individuals. This is not as simple to implement as it sounds because users need to trust each other when doing transactions. The way of imposing this trust is by making the *nodes* or participants of the network agree on which transactions are valid and make them immutable. Let us present some definitions first before studying how this agreement occurs.

4.3. Important cryptocurrency terminology

Some important definitions and ideas that will be used throughout this part of our work must be given. These terms and ideas are recurrent when studying cryptocurrencies, and it will be necessary to understand these concepts fully to follow more complex topics later in this text. Let us begin by providing some basic definitions.

4.3.1. Basic definitions

Let us start by defining what is usually called an *address* or *account*.

Definition 4.5. An address is the hash of a public key. It can be viewed as a virtual location where some coins can be sent. However, there is no physical movement of coins from one address to another.

It consists of 26 to 35 alphanumeric characters.

Example 4.1. There are several online tools that allow you to generate the pairs of keys (public-private) after introducing some natural entropy. The entropy used to create the following pair was generated by randomly moving the mouse on the screen in [poi23].

$$\begin{cases} \text{Public address: } 127mSypGAgkRnms7H5BmHNPTirJ3pdwjS7. \\ \text{Private key: } KyHtbjqUgzHEzXMg1XihHi8cSFGuBKyXi9KF4YZu53RPgSfPm5oN. \end{cases}$$

The hash function used to generate the Bitcoin address is SHA-256. Other cryptocurrencies may use other hash functions. For instance, Ethereum uses Keccak-256.

Definition 4.6. A wallet is a software program that stores the user's private keys and enables him to access his coins. It knows which private key is associated with each public address.

They are usually structured files or simple databases, and there are two basic types. Deterministic (seeded) wallets are generated by repeatedly hashing a private key. Non-deterministic (random) wallets generate random private keys for each address. Deterministic wallets are the most space-efficient option because it is only necessary to have a backup of the original private key that is subsequently being hashed. On the other hand, random wallets need to have backups of each of the keys. If the user does not want to reuse addresses (which is extremely dangerous), this option becomes unviable due to the great amount of storage it requires. The previously mentioned danger resides in the fact that the user can be easily tracked, discovering his wealth and becoming a target for theft or other malicious activities. We will see how that is exactly done in Section 4.6.

Definition 4.7. A transaction is a piece of information indicating the transfer of tokens from one address to another signed by the sender's private key. If the receiver does not have the private key associated with the receiver's address, those funds will be inaccessible.

Example 4.2. Many websites give users access to transactions that aim to join the blockchain. Figure 4.1 shows a transaction in one of those webs.

Definition 4.8. A block is a data structure that contains transactions. Blocks are *linked* to one another by hashing the latest block of a chain of blocks and including that hash in the following block (see Figure 4.5).

```
{
  "txid": "d94b0765585f44fdc3acbccd9ebbf88710af13d52a0d026c8daba09d2df38775",
  "size": 192,
  "version": 2,
  "locktime": 0,
  "fee": 22034,
  "inputs": [
    {
      "coinbase": false,
      "txid": "7cd38606d84147dd58bd1989d456b5080b34f16d18711ede139015f86435d4ef",
      "output": 8,
      "sigscript": "",
      "sequence": 4294967295,
      "pkscript": "001447d6ddca9a2e70f5866175ebe0e043879d5fb643",
      "value": 139300,
      "address": "bc1qgltdmj569ec0tpnpwh47pczrs7w4ldjr83ppym",
      "witness": [

"3045022100d9f001e02ad4c5c47d4cf495b54de05cefd857559cc503eb653c3695ee1d72e7022069f0509521e5f18255ede9fd5c35102a04
7c2033cc816a70f6e2bd4d43fde4f801",
        "039e94228a6a29607b3e778a50e4db5a472ab2b54737a8e4cafe9897583dbfa477"
      ]
    }
  ],
  "outputs": [
    {
      "address": "bc1qt74grdduzt0aacn6h08tv8lw9l6cjyqvsz9kq5",
      "pkscript": "00145faa81b5bc12dfdee27abbceb61fee2ff589100c",
      "value": 117266,
      "spent": false,
      "spender": null
    }
  ],
  "block": {
    "mempool": 1712939093
  },
  "deleted": false,
  "time": 1712939093,
  "rbf": false,
  "weight": 438
}
```


Figure 4.1.: A transaction extracted the 12th of April 2024. Source: [Blo24a].

Blocks are added to the blockchain by nodes. Depending on each node's specific function on the network, the software and hardware capabilities required from them vary. However, apart from the individual nodes' infrastructure restrictions, the network does not restrict any user or node from participating as any other does. This means anyone can set up their computer as a node with the same capabilities as any other person.

To achieve decentralization, all cryptocurrency users should theoretically run a *full node*. We use the word *full* to refer to a node with all the capabilities (and responsibilities) a node can have. We will not delve into the different types of nodes that exist and how they work, as it is a topic that deviates from the interest of this dissertation. We will simply mention some of the nodes' characteristics when needed.

Despite the importance of decentralizing the system, most users nowadays access their funds from cryptocurrency exchanges such as Binance [Bin24] or Coinbase [Coi24].

Definition 4.9. A cryptocurrency exchange is a digital platform or service that allows users to buy, sell, trade, and store cryptocurrencies. Users of these tools do not need to run their own full nodes, as the exchange manages the underlying blockchain interactions on their behalf.

Although common users are unaware of this, the process from when a transaction order is sent to when the funds have changed from one address to another is quite complex. This process is completed when a new block is added to the blockchain, and it has different names depending on the *consensus algorithm* being followed. A consensus algorithm is a mechanism that ensures agreement on a single data value among distributed processes or systems [Awa24]. In the case of Blockchain, nodes of the decentralized system agree on which block should be added to the blockchain next. There are many variations for this idea, but in this dissertation, we will focus on two of them: *proof of work* (Bitcoin's algorithm) and *proof of stake* (Ethereum's algorithm).

4.3.2. Proof of work

Proof of work is a mechanism used in modern-day cryptocurrencies to achieve consensus when adding blocks to a blockchain. The protocol was initially introduced by Cynthia Dwork and Moni Naor in [DN93]. The most attentive readers will have noted that this is the same author who created the 'Where is Waldo?' example in Subsection 2.1.1. The idea of the authors in [DN93] was to introduce a computational problem or challenge that must be completed before emailing someone. This would slightly increase the time it takes to send the email but significantly reduce the spam users receive.

In Blockchain context, the computational challenge consists of hashing a block's header and a nonce.

Definition 4.10. A nonce is a cryptographic term used to refer to a 'number used once'. It is usually a pseudorandom-generated number whose strength resides in that its value never (or hardly ever) repeats during a protocol.

The hashed pair (the nonce and the block-header) must verify a strict condition to satisfy the computational challenge correctly. This condition is called the *difficulty* of the computational challenge.

Definition 4.11. The difficulty is the number of zeroes that must be found at the beginning of the hash. The network automatically adjusts the difficulty to ensure the rate of blocks added to the network is constant at any time.

The nodes that add blocks to the network following a proof of work consensus algorithm are called *miners*. These miners must collect transactions from pools of transactions, although usually referred to as *mempools*, which is short for memory pools. These transaction pools store the transactions nodes have created and desire to add to the network. As we mentioned before, Blockchain is a decentralized system, and it is for that reason that these transaction pools are distributed amongst many nodes. We will not delve into the details of how these pools of transactions are kept secure and updated, but we will assume they do.

Continuing the process, a miner who has collected enough transactions verifies they are correct and builds a block containing them. Once that is done, the miner that has built that block must compute a hash containing the block's headers that satisfies the difficulty imposed by the network. In other words, the generated hash must have enough zeroes at its beginning.

Following this, the network verifies that the imposed difficulty (number of leading zeroes the hash must contain) has been achieved and validates the block. This means verifying that all its transactions are correct and coherent with the blockchain's state. For instance, a wallet that has no funds has not participated as a sender in any transaction.

Once the validation is complete, the block is considered the one at the top of the chain of blocks. Consequently, miners building a new block must include that block's hash (the hash from the block at the top of the chain) in their blocks. This ensures the chain continues growing, and the transactions included in that block become immutable.

Because two miners can add two different blocks to the top of the blockchain simultaneously, a *fork* can occur. A fork means two different versions of the blockchain are available. One of these blockchains will become invalid once a miner adds a block to the other chain, consequently becoming the longest one. The transactions that were included in the block at the top of the discarded chain return to the transaction mempool and are included in the following block.

If a malicious party wanted to change a transaction added in a past block, he would have to recompute all the blocks' challenges from all the blocks added to the chain after that block was added. This is computationally unfeasible and ensures the blocks added to a blockchain are immutable.

Proof of work solves the problem of determining representation in majority decision-making. If the mechanism was one-IP-address-one-vote, it would imply that anyone allocating many IPs could decide which blocks were added to the blockchain and which were not. Using proof-of-work, the majority decision is represented by the chain whose computational effort is greater and, therefore, the longest available chain. This computational effort requires a great amount of hardware and electricity resources. The hardware resources, which mainly consist of high GPUs and cooling systems, are used to compute several thousands of hashes per second to solve the computational challenge imposed by the network. This enormous use of resources has made other options, such as *proof of stake*, increasingly popular.

4.3.3. Proof of stake

Proof of stake is a consensus algorithm that demonstrates a *validator*, a node in the network that replaces the proof of work miner, has deposited something of value into the blockchain as insurance that prevents him from acting dishonestly. The validator is responsible for checking that the block that will be added to the blockchain is valid. A block is valid when all its transactions are consistent with the current Blockchain state. If a validator misbehaves, some of his deposited funds will be destroyed.

Different from the growth rate management of the blockchain that takes place in proof of work, time is divided into slots. For every slot, a random validator is responsible for creating a new block and sending it to the network. Committees of random validators are formed to vote on the validity of the block's transactions, deciding whether it should be added to the network or not.

The steps that occur regarding validation when a transaction gets executed depend on the specific blockchain implementation. However, we can study how it works in Ethereum, which defined the paradigm most other cryptocurrencies follow. The following steps have been summarised from [Wac24b, @om24, Eth24].

1. A user creates, signs and submits the transaction to an execution client, which verifies its validity. The execution client is the name a part of an Ethereum node receives (see Figure 4.2). Its job is to listen to new transactions (see Definition 4.7) and verify they are signed with the correct key.

2. If it is valid, the transaction is added to a list of pending transactions and broadcast to all the other nodes participating in the validation. When these nodes listen to this broadcast, they add the transaction to their local *transaction pool*. A transaction pool is a local data storage containing all the transactions that have been broadcast to the nodes in the network.

3. The validator who has been randomly selected as the block proposer is responsible for building the next block and updating the state of the network. As it will be discussed in detail in Section 4.5, transactions can contain code that needs to be executed to generate the state change. The block validator is responsible for executing that code to update the accounts that participate in the transactions. Then, the block is broadcast to the consensus clients in the network (see Figure 4.2), passing it to their execution clients.

These re-execute the code locally, verifying the transaction's validity. After that, the node votes on the transaction's validity. The way nodes *vote* is by adding the verified block to their view of the chain with the greatest number of total votes (or attestations) from the beginning of its history.

4. The block becomes valid when it becomes part of the largest chain after a predefined period of time passes.

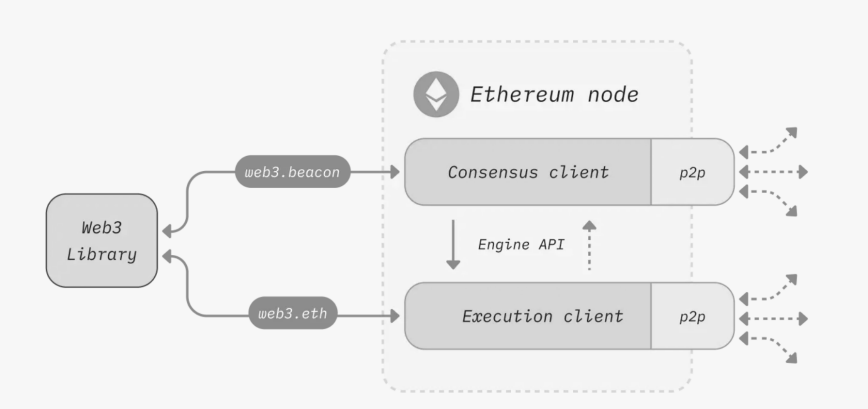


Figure 4.2.: An Ethereum full node. Source: [@om24].

This validation of blocks is facilitated by the use of several cryptographic tools. One of the most important of these are Merkle trees, which significantly increase the efficiency of transaction validation.

4.3.4. Merkle trees

Merkle trees are binary trees whose nodes are labelled using hash values. Every non-leaf node is labelled with the hash computed from the hash of its two child nodes. To demonstrate that a leaf node belongs to a Merkle tree, only a proportional to $\log_2(n)$ number of hashes needs to be computed, where n is the number of leaf nodes. This is an improvement in efficiency compared to other structures, such as hash lists. Furthermore, changing the hash of a node will affect the hashes of the nodes situated in the upper layers of the same branch, consequently changing the hash of the root node.

The hash of a block is really the hash of its header. This header contains a timestamp, a nonce (although it is usually considered external to the header), the previous block hash, and the root hash of the Merkle tree storing all transactions in the block. Therefore, changing any transaction's data would change the Merkle tree's root hash (see Figure 4.3) and consequently change the hash of the whole header. This change would imply that any node retrieving the chain would see a discordance between that block and the following because the chain of hashes would not be correct. As we mentioned, nodes can be of various types, and some, as the one here mentioned, are responsible for storing a copy of the longest blockchain at all times.

Merkle trees allow nodes to retrieve information about blocks in pieces from the network. For instance, if they want to verify a transaction is in a particular block, they can retrieve the

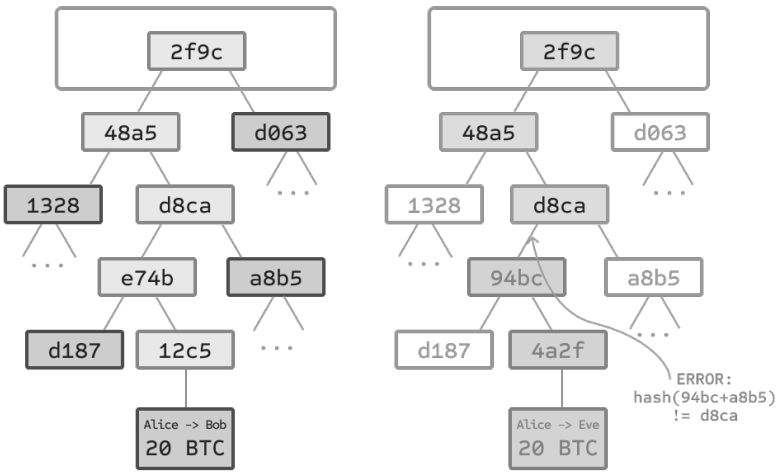


Figure 4.3.: Comparing the Merkle tree associated with the correct transaction (Alice sends BTC to Bob) to the one with the fraudulent transaction (Alice sends BTC to Eve) and seeing how the modified hash propagates upwards. Source: [But14].

header from one source and a small part of the tree relevant to them from another. Retrieving all the siblings of a leaf node l, from that leaf to the root of the tree, is called finding the *opening* of l, also noted as $O(l)$. To verify a transaction is in a Merkle tree, the node would have to compute the hash of the transaction, find the leaf l that contains it and verify $O(l)$ allows him to retrieve tree's root by computing the successive hashes from the leaf to that root. If, after successively hashing the siblings and parents of the leaf up to the root, the obtained hash is different from the tree's root hash, the transaction must be fraudulent.

Once these commonly used cryptocurrency terms are understood, we proceed to study Bitcoin, the first and most famous cryptocurrency that we introduced at the beginning of this chapter.

4.4. Bitcoin

Although thousands of different cryptocurrencies exist, each with its particular modifications of blockchain structure, they all follow, in one way or another, the basic model introduced by Bitcoin.

The idea behind this technology is very simple and relies on asymmetric encryption (see Subsection 4.2.2). Imagine Alice wants to transfer a coin to Bob. Then Alice would write a message: 'I, Alice, transfer a coin to Bob' and would later encrypt the message with her private key and publish it. In this case, she would add it to the blockchain. Bob, or anybody who would like to check the transaction, would use Alice's public key to decrypt the message and verify that Alice had indeed signed that transaction. That would be the receipt that ensures that Bob possesses the coin now. The currency which is used to represent these coins is noted as BTC.

If Bob wanted to transfer the coin to Carol, he would follow the same procedure. If Carol wanted proof of her possession of the coin, she would simply need to use the chain of signatures (Bob's and Alice's). The coin is defined as a chain of electronic signatures in this

system. Each owner transfers the currency to the next by signing a hash of the previous transaction and the next owner's public key (see Figure 4.4). However, there is no way of ensuring that the coin has not been double-spent (double-spending is the action of using the same coin twice in two different transactions).

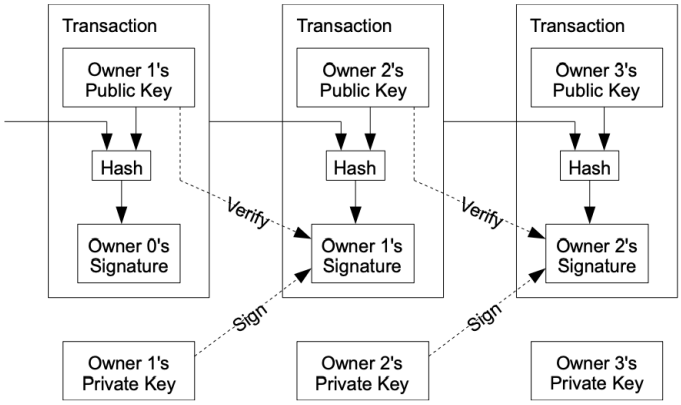


Figure 4.4.: Chain of digital signatures initially proposed for Bitcoin. Source: [Nak].

To ensure that the payee can verify that one of the owners did not double-spend, Bitcoin proposes forming a chain of hashes in which every hash has the previous hash and a block, or group of transactions, as input (see Figure 4.5). Proof-of-work (see Subsection 4.3.2) is used to implement this idea on a distributed peer-to-peer basis (see Subsection 4.2.1).

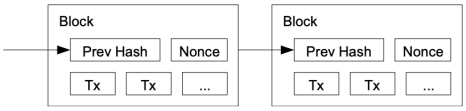


Figure 4.5.: Chain of blocks including nonces to implement Proof-of-Work and ensure not double-spending. Source: [Nak].

The whole system works according to this structure.

1. New transactions are broadcast to all nodes in the network.
2. New transactions are collected into blocks by every node. On average, a block contains 2,500 transactions.
3. Each node proceeds to solve the proof-of-work challenge of their block.
4. When a challenge is solved, it broadcasts the block to all nodes.
5. Nodes accept the block if all transactions are valid and not already spent. Their acceptance is expressed by using the hash of that block as the previous hash when creating a new block. This is the same as linking the block they are constructing to the block they just received.

The absence of a centralized authority implies new ways of introducing currency into the system must be used. In Bitcoin, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. Added to the fees associated with transactions, this mechanism is used to introduce coins in the network. Adding a new block to the network is called *mining*; consequently, nodes that add blocks to the network are called miners (as we explained in Subsection 4.3.2).

Mining is a highly computationally intense activity that requires many resources, mainly electricity. For this reason, other ways of achieving consensus on the main blockchain chain are used in other cryptocurrencies, such as Ethereum, which uses proof of stake.

The transactions added to the blocks by the miners are made up of *UTXOs* or Unspent Transaction Outputs. They are the pieces of Bitcoin found in users' wallets, and we have previously referred to them as coins. They are created during transactions and are used to prevent double-spending. Each UTXO has a denomination, for instance, 20 BTC, and an owner indicated by its public address. Users have the right to include UTXOs in new transactions if they are the UTXOs' owners. Once an UTXO has been included in a transaction and the transaction is added to a block by a miner, the UTXO owner changes. This avoids double-spending.

The rigidity UTXOs introduced in Bitcoin was one of the reasons why Ethereum was created. The impossibility of defining more than a binary state for UTXOs (*spent* or *unspent*), the lack of fine-grained control over their denomination and the lost opportunity of including the already existent randomness provided by the nonces were reasons to support the creation of a new cryptocurrency paradigm. Before talking about Ethereum, the cryptocurrency that shifted the direction of Blockchain, let us talk about how Bitcoin keys are generated, as most Blockchain ecosystems rely on the same method.

4.4.1. How are keys generated?

Several steps occur to generate a Bitcoin (or other cryptocurrency) key pair. We have extracted them from [Ant24] and explained some mathematical concepts associated with each step.

1. First, entropy must be generated from a natural source, such as by randomly pressing the keys of our keyboard or scrolling the mouse around. Entropy is used to refer to the lack of order or randomness.

2. This generates a random number fed to a hash function (SHA-256 in the case of Bitcoin) that transforms it into a hash value (see Section 1.9). If the obtained value is smaller than a predefined amount (around 2^{256}), the value is accepted. Otherwise, the process is repeated.

3. Once the private key has been generated, the public key is generated using elliptic-curve cryptography (see Section 1.10). Precisely, Public key $=$ [Generator]Private key.

   The curve the Bitcoin protocol uses is defined as $y^2 \equiv (x^3 + 7)$ over $\mathbb{F}_p$. It is named *secp256k1* by the National Institute of Standards and Technology (NIST), which is an American government institution that, amongst many other tasks, sets up standards across many industries. The value p is a prime number defined as $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. The curve is defined over a finite field, unlike the example given in Figure 1.1. Therefore, the representation would be a set of discrete points plotted in a two-dimensional grid.

The generator point on the curve is always the same for all Bitcoin private keys and is defined as part of the standard *secp256k1*. In particular, the value of the generator is 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8. This (very) long value is a pair of points (x, y) expressed in uncompressed hexadecimal format. The multiplication of this value and the private key returns the value of the public key on the elliptic curve.

Disregarding the particular values and standardized protocols, this key generation method is common to many other cryptocurrencies that use elliptic curve cryptography. Some of those other cryptocurrencies are as popular as Ethereum, the biggest Bitcoin competitor in the market.

4.5. Ethereum

Ethereum is the second most known cryptocurrency and was presented by Vitalik Buterin in 2015 in [But14]. It has more than ninety-six million wallets containing a total of more than four hundred billion dollars. Ethereum's scope of applicability is significantly larger than that of Bitcoin, and that is exactly why Vitalik chose to build an independent blockchain instead of a cryptocurrency on top of Bitcoin.

The state of the blockchain is defined by the objects called *accounts*. These are similar to the wallet addresses seen in Bitcoin but have some differences. Each Ethereum account is made up of four fields. A nonce, which makes each transaction unique, the ether balance, the account's contract code, and the account's storage. Accounts can be divided into *externally owned accounts* (EOA) and *contract accounts* or *smart contract*.

The first of these two types is what we could call a *classic* type of account. Its owner can send messages by creating and signing transactions. Transactions are signed messages containing (in very basic terms) the following fields.

1. **Recipient**. Account (or address) of the user (or smart contract) that will receive the message.
2. **Signature**. It uniquely identifies the sender.
3. **ETH**. Amount of ether to be transferred.
4. **STARTGAS**. It stands for the maximum number of computational steps the transaction execution can take. Consequently, we call *gas* the required fee that must be paid to execute a transaction.
5. **GASPRICE**. It is the price the sender pays per computational step used to execute the transaction and the recipient's smart contract code. When the recipient is not an EOA, we can assume this contract does not do anything.

Every time a transaction is created, some code is executed as part of the transition function. This function changes the state of the whole blockchain each time a new block is added. Consequently, nodes must be aware of the result of its execution at any time. For that reason, the code whose execution is triggered by a transaction is executed by all nodes that download and validate the block in which it is contained. It is worth noting that STARTGAS and

GASPRICE are included in the message to avoid *Denial of Service attacks (DoS)*. If attackers wanted to make some transaction state change code loop forever, flooding the network and making it non-functional, they would need to own lots of ether. This makes these types of attacks unprofitable.

Returning to the types of accounts, we also mentioned *contract accounts*. Contract accounts are controlled by a piece of code that activates every time a message is received. This code can read and write the account storage, create other contracts, or send messages. These contract accounts or *smart contracts* (as they are commonly known) are one of the main features of Ethereum. They behave similarly to real contracts because some clauses are executed when certain conditions are met. However, they take the funds in advance, are unmodifiable, and stay in the blockchain forever. Furthermore, they include functions other accounts can invoke by sending transactions to the smart contract. An example of a smart contract simulating a vending machine, extracted from [Wac24a], is given in Appendix C. Smart contracts are a very interesting topic that could have a full dissertation dedicated to studying them due to their complexities and wide applicability.

Similarly to what we saw in Section 4.4, we can study the transaction state function (see an example of how the state changes in Figure 4.6) that is executed when a transaction occurs.

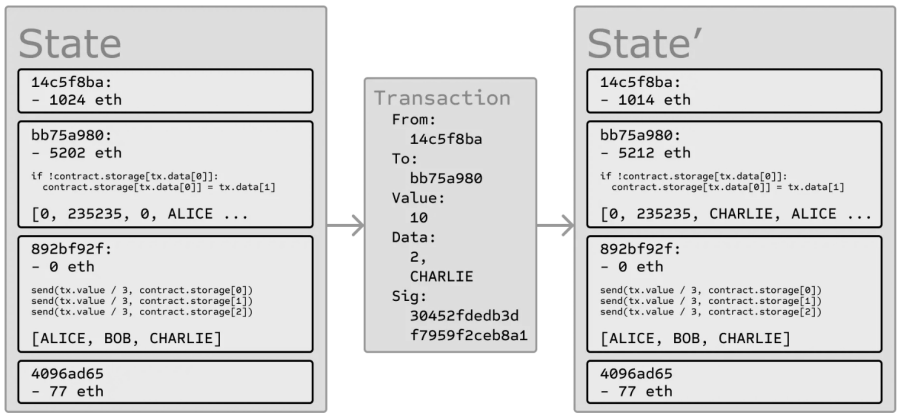


Figure 4.6.: Ethereum State Transaction Function example. Source: [But14].

1. The transaction is validated using the proof of stake consensus algorithm (see Subsection 4.3.3). Ethereum originally followed the same mining algorithm introduced by Bitcoin; however, it switched to this most sustainable alternative in September of 2022 [Pab22].

2. Calculate the transaction fee by multiplying GASPRICE times STARTPRRICE, subtracting the amount from the sender's account, and incrementing the nonce of the sender's account. This will ensure each transaction is unique. If the sender's funds are not sufficient, return an error.

3. Subtract the corresponding gas spent on the message size from the STARTGAS amount. Storage is a valuable resource in Ethereum. For that reason, the amount in bytes a message occupies is paid when a transaction is executed.

4. Transfer the ether from one account to the other. If the receiving account does not exist, create it. If the receiving account is a contract, execute it until completion or until the execution runs out of gas.

5. Transfer the ether from one account to the other. If the receiving account does not exist, create it. If the receiving account is a contract, execute it until completion or until the execution runs out of gas.

6. If the transfer failed because of lack of funds, revert the changes except the fees paid to the validator. Otherwise, refund the remaining unspent gas to the sender and pay the corresponding fees to the validator.

At last, we must mention that Ethereum's network is divided into two levels or layers: *Layer 1* and *Layer 2*. Layer 1 refers to the *Mainnet* or main Ethereum blockchain. On the other hand, Layer 2 refers to the secondary frameworks or protocols built on top of the main Ethereum blockchain to improve its scalability and efficiency. We will study some Layer 2 protocols in Section 4.7.

4.6. Privacy

Despite most users believing transactions are fully *private* when using cryptocurrencies, this is far from the truth. This section will describe how privacy is violated and which proposals have been made to solve this problem. Most of the literature on this topic has explored Bitcoin and its privacy vulnerabilities because most other cryptocurrencies follow Bitcoin's paradigm (including Ethereum from a fundamental point of view). Therefore, the ideas provided here should be considered fully accurate for Bitcoin and highly accurate for any other cryptocurrency. Let us start by providing a formal definition for *privacy*.

Definition 4.12. Privacy is the assurance that the confidentiality and accessibility to certain information about an entity are protected.

As explained in Section 4.4, transactions are broadcast to all nodes in the network. This means that any participating node can see the payer's public key, the payee's public key and the amount being transferred. This transparency is necessary to enable any node to verify the correctness of the transaction before adding it to the block it is mining. Bitcoin achieves some kind of privacy by keeping public keys anonymous. Anyone can see the public keys being used, but there is no direct correlation between the person using the public key and the public key itself. Bitcoin also recommends generating a new key pair for each transaction to keep them from being linked to a common owner. However, this system has a big flaw. Once a public key becomes associated with a particular individual, it compromises the privacy of all transactions involving that public key. Adversaries can then trace the flow of funds to and from that public key, potentially revealing sensitive financial information and compromising the user's privacy. This problem equally affects Ethereum because the fundamental basis of transactions is common to both cryptocurrencies.

4.6.1. Blockchain Analysis

Through analyzing the transactions added to the blockchain, attackers, governments, or any interested party can link the physical person or organization to a particular Bitcoin (or

any other cryptocurrency) address. As explained in [CSKLR18], a highly detailed survey on security and privacy issues found in Bitcoin, there are three main pre-processing steps required to analyze the blockchain.

1. **Transaction graph**. An acyclic graph represents the transactions' flow in the blockchain. The weights of the edges represent the number of bitcoins included as the transaction's input.

2. **Address graph**. By studying the transaction graph, a new graph can be built inferring relationships between several input and output addresses.

3. **User/entity graph**. The previous two graph models added to the use of two heuristics derived from Bitcoin protocols are used to group addresses belonging to the same user. The two heuristics were introduced in [AKR+13] and are the following.

   - **Several input addresses.** The same user generates all inputs of a transaction. Users rarely contribute to form a common transaction due to its structural difficulties.
   - **Change addresses.** Input addresses that have transactions associated with output addresses that have never appeared in the blockchain in the past nor will be seen to receive more transactions in the future belong to the same user. This is because Bitcoin creates those addresses to deposit the change from the extra Bitcoin that will not be transferred to the output address after a transaction. Change addresses are also referred to in the literature as *shadow addresses*.

   Although these heuristics are imperfect and generate many false positives, slight refinements make them great tools for finding connections between users and public addresses.

Apart from using these graphs, several other online tools diminish the theoretical *anonymity* (we define anonymity as the users' identity privacy) that cryptocurrencies offer. Some of them are outlined here.

- **Web Crawlers.** Programs that search social networks for public addresses of Bitcoin and directly connect them to the users who leaked them.

- **Blockchain Analyzers**, such as, *BitIodine*. Michele Spagnuolo presented in [Spa13] a software capable of tracing scammers and other cybercriminals through heuristics, clustering and leaks on the deep web.

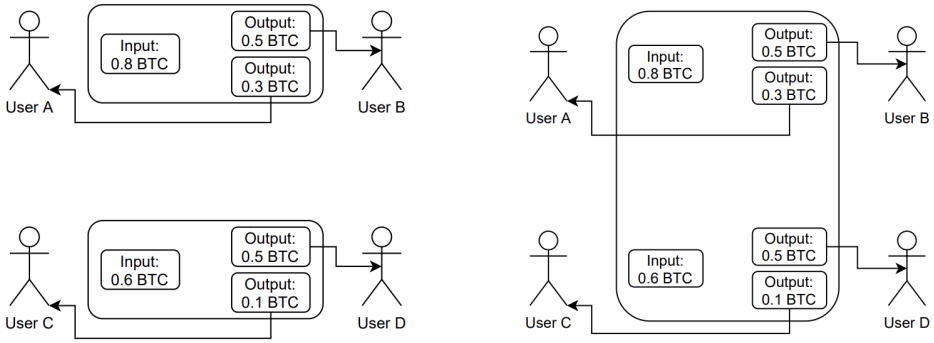
- **Transaction Visualizers**, such as, *BitConeView*. A more visually enticing approach was presented in [DBDDP+15], representing graphs of transactions occurring in the blockchain using advanced graph analysis. Although the tool is no longer available online, some slides on the visualization process can be found in [DBDDP+].

4.6.2. Different proposals to improve anonymity

As shown in the previous section, many tools are available to track transactions and easily find connections that could potentially leak the identity of the Bitcoin network users. Many authors and organizations have found solutions using different techniques to tackle this lack of privacy. The ones that have been more successful in doing so are showcased here.

1. **Peer-to-peer mixing protocols**. Service providers such as *DiceMix* [RMSK17] provide a platform where parties interact with one another, scatterring their transaction trails. This mechanism divides, mixes, and redistributes a user's coins between many other users, hindering blockchain analyzers' use. All users end up receiving the same amount of coins they paid. Tracking the identity of the account owners becomes significantly harder when the coins' ownership is repeatedly permuted.

   Effectively contradicting the first heuristic presented in Subsection 4.6.1, *CoinJoin* [gBfu] allows users to mix their transactions' inputs in a single transaction that maintains the original output addresses and amounts (see Figure 4.7). Due to the vulnerability caused by having to manage the signatures of all participating parties, *CoinShuffle* [RMSK14] was presented. CoinShuffle merged mixing techniques with the fundamentals of CoinJoin, eliminating the necessity of a central server.



(a) Two regular Bitcoin transactions.

(b) A CoinJoin transaction.

Figure 4.7.: Side by side comparison between regular traceable transactions and CoinJoin merged transactions. Source: [SHMM21].

   To give some context on how valuable these mixing technologies are for the Bitcoin community, in 2022, around 227,480.84 Bitcoin coins had been mixed [SHMM21]. This amount of coins is equivalent to ca. 4.74 B USD using the current exchange rate (1 BTC = 70.464,50 USD).

2. **Distributed mixing networks**. Third parties such as *MixCoin* [BNM+14] provide public addresses, or *mixers*, that receive funds from address A and transfer them to address B, charging a fee. A and B could belong to the same user who simply wants to remove her association with the funds. There are two problems with this system. First, the user must trust the mixer not to steal her coins. Second, the mixer could discover the user's identity as owning the address$A$.

   Because of the impossibility of eliminating these threats, MixCoin proposed mixing with accountability. Accountability means that individuals or organizations are responsible for their actions and can be held answerable for them. This means that if the mixer misbehaves, the user receiving its service will publicly present a warranty, showing that the mixer cannot be trusted. Consequently, the mixer would run out of business. Assuming all parties aim to maximize profit, mixers would prefer to behave honestly rather than dishonestly. Although out of the scope of this dissertation, this specific idea of maximizing profits is widely used in cryptocurrencies when designing

protocols. We recommend the paper 'SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies' [BMC+15] by Joseph Bonneau et al. to interested readers because of its clarity when explaining these and other fundamental ideas surrounding cryptocurrencies.

3. **Bitcoin extensions or *Altcoins***. Altcoins are other currencies inspired by Bitcoin and usually work as extensions or full-fledged currencies. *Zerocoin* [MGGR13] is an example of an Altcoin that addresses the anonymity problem using zero-knowledge proofs. In Zerocoin, fully encrypted transactions can be verified as valid without accessing the transactions themselves. *Zerocash* is an evolution of Zerocoin that uses zk-SNARKs (see Section 3.1). In Section 5.1, we will explain the fundamentals of *ZCash*, which is an evolution of Zerocash.

   Another branch of Altcoins includes those built in completely independent blockchains such as Ethereum. These cryptocurrencies natively add many other technologies that the original Bitcoin blockchain does not have, such as smart contracts.

Those smart contracts are why scalability, a problem that already existed in Bitcoin, very strongly affects many Altcoins such as Ethereum.

4.7. Scalability

While the privacy problem developed in Section 4.6 affected Bitcoin, Ethereum and many other cryptocurrencies equally, there is a problem that, while present in all of them, specially affects those cryptocurrencies that implement smart contracts. This problem is called scalability.

Definition 4.13. Scalability is the ability of a computer application or protocol to maintain its performance when its context's size changes in size.

In the previous definition, *context's size* refers to the number of concurrent users and computational effort required to run the network. This computational effort is considerably bigger in Ethereum (and other similar cryptocurrencies) because of the implementation of smart contracts. These contracts are pieces of code that must be run by validators to verify the correctness of the transactions or Ethereum messages. This is why scalability poses a greater problem in cryptocurrencies implementing smart contracts and similar complex structures.

As we mentioned at the beginning of Section 4.6, we were going to focus on Bitcoin, although the ideas were easily generalizable to most other cryptocurrencies. Conversely, in this section, we will focus on Ethereum and study the proposals to solve this problem in that specific cryptocurrency. This is because scalability solutions are not as general, unlike the previously presented privacy solutions and depend heavily on the underlying cryptocurrency. For instance, it is common to delegate some work on off-chain solutions (Layer 2 solutions as referred in Section 4.5), and these solutions change considerably depending on the kind of transactions or messages that are interchanged with the main network. The specific cryptocurrency protocol indicates which kind of messages can be interchanged and, consequently, how the off-chain solutions must be built. Once the reason behind our focus shift is clear, let us study when scalability issues arise.

Ethereum blockchain overhead is significantly greater than Bitcoin's because of all the computations derived from smart contract maintenance. The Ethereum network tries to cope

with this overhead by increasing the gas price (the price of each computation unit). However, this price can skyrocket when high peaks of demand occur. We can see in the following graph how, in 2020 and 2022, these peaks were evident (see Figure 4.8). To provide some context, the price per gas unit was around 700 GWEI, which at today's exchange rate is around $0.0027. A basic transaction usually takes around 20000 GWEI. Thus, the price was $54 for a single transaction. This price makes it significantly expensive to transfer funds and unfeasibly costly to interact with any smart contract, making the cryptocurrency practically unusable.

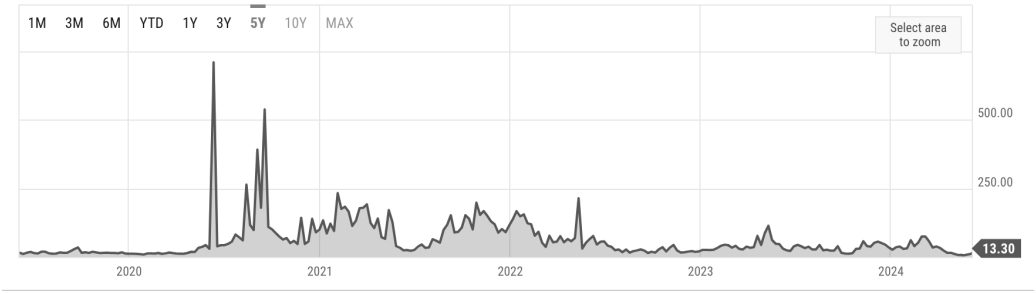


Figure 4.8.: Gas price in GWEI graph showing major peaks in various dates between 2020 and 2021. Source: [YCh24].

4.7.1. Different proposals to improve scalability

To solve scalability, Ethereum presented a solution called *sharding* when they switched from Ethereum to Ethereum 2.0 (this occurred in 2022 and introduced proof of stake to the cryptocurrency). Sharding is a Layer 1 solution that consists of splitting the blockchain into discrete pieces to be validated by groups of validators. This was the initial option proposed by Ethereum to face its scalability challenge. However, due to their better results, they took a turn towards scaling solutions on Layer 2 (as the Ethereum organization recognizes in [Smi24]).

Layer 2 solutions aim to reduce the overhead of the main Ethereum network by receiving transactions that would otherwise be directly sent to the Mainnnet. Eventually, these transactions are sent and recorded in batches in the Mainnet, reducing the number of validations needed.

The way to achieve this is by having users have two separate accounts. One is in Layer 1 (the typical cryptocurrency address), and the other is in Layer 2. Intermediary parties between user's accounts in Layer 2 or *operators* are responsible for running the protocol. These operators have an associated smart contract in the Mainnet used to receive messages from users in Layer 1.

Before interacting with these operators, users must *lock* funds in their accounts. Locking funds involves transferring them from their Ethereum accounts to the operator's smart contract. This operator credits the funds in the users' Layer 2 accounts, allowing them to interact with the funds as if they were the original. We can visualize users locking their funds in Figure 4.9. When the users wish to recover their funds, they send a message to the smart contract requiring it. After some time (called the *challenge period*), used to ensure the transaction proofs were added to the Mainnet and were valid, users receive their funds in

their Layer 1 account.

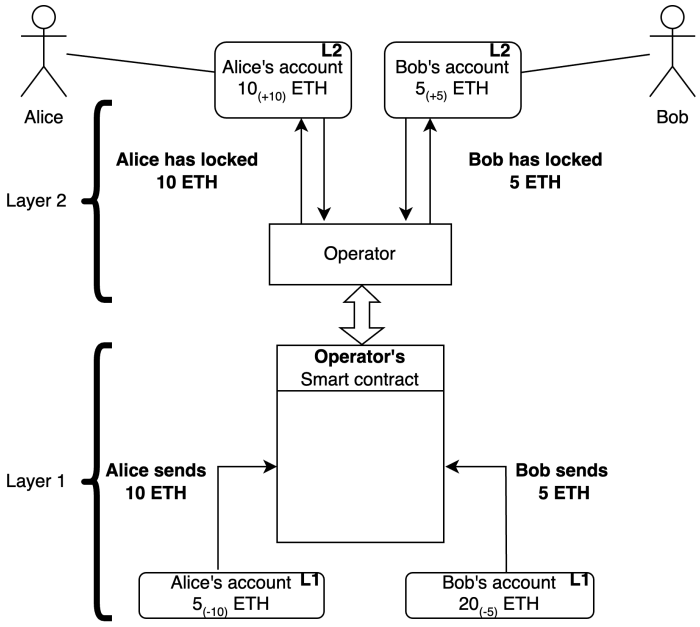


Figure 4.9.: Example in which Alice and Bob lock 10 and 5 ETH, respectively, in their Layer 2 accounts. Source: Elaborated by the author.

Let us see an example of a hypothetical Layer 2 solution. Imagine Alice offers 1 MB of internet browsing at high speed for $0.0001. Bob takes the offer and issues a ticket indicating the amount he owns every time he uses a new MB. This would lead to several transactions in the Mainnet. However, the Layer 2 solution stores all the transactions and simply adds the one with the maximum amount at the end of the usage period. Some security measures would be in place to ensure both users behave honestly, but the main idea can be understood from this example.

The most popular Layer 2 solutions currently being used in Ethereum are *rollups*.

4.7.2. Rollups

Rollups consist of grouping transactions into batches or bundles and executing them off-chain. This means doing transaction executions in Layer 2 but maintaining data on-chain (Layer 1) where consensus is reached, ensuring the data is as secure as it was when transactions were done in the Mainnet. There are two types of rollups: *zk-Rollups* and *optimistic rollups*. Both are controlled by a smart contract operating in the Mainnet, but they have some differences.

1. **Optimistic rollups**. These rollups always include a compressed version of a bundle of transactions executed off-chain. After the computation, the operators interact with a smart contract in Layer 1 and send a compressed version of the transactions. These transactions are not validated and considered valid unless another node indicates they are not. This is why they are called optimistic; the validity of the transactions is assumed.

If a node reports the blockchain state changes were fraudulent, the network (Layer 1) must validate all the batch transactions and decide whether the rollup operator or the node is right. Both parties must provide a bond (some predefined amount of ETH) before participating in the protocol. This enables the network to punish the one who behaves fraudulently or inappropriately. In other words, when the operator is trying to validate a fraudulent batch, or the node is calling fraudulent a correct batch of transactions. This ensures both parties are interested in behaving well. This system has been proven secure when at least one node is reporting fraudulent changes in the network's state.

2. **zk-Rollups**. These solutions interest us the most and follow a completely different paradigm from the optimistic rollups proposed. In optimistic rollups, operators do not need to prove the validity of their claims and off-chain transactions are assumed to be valid. On the other hand, in zk-Rollups, operators need to demonstrate that the proposed state changes are correct because the transactions were correctly executed. The way of doing so is by providing a *validity proof*. Here is where zk-SNARKs provide a great tool to ensure the transactions were correctly validated in a succinct way, reducing the overall overhead.

   In zk-Rollups, apart from the main smart contract storing the roll-up blocks, tracking deposits, and monitoring state updates (this would be the contract shown in Figure 4.9 and it is called *tracking contract*), there is a second smart contract, the *verifier contract* or *verifying contract*. Its task is to verify the zero-knowledge proofs submitted from Layer 2. In summary, an independent off-chain virtual machine executes the transactions and provides validity proofs. Those are later verified in Ethereum to add the transactions to the main blockchain.

We have completed our study of Blockchain, including the mathematical fundamentals supporting it, the two most prominent cryptocurrencies and two of their intrinsic problems: privacy and scalability. In the following chapter, we will study how zero-knowledge proofs have been used to solve these problems.

5. Zero-Knowledge Proofs in Blockchain

The initial chapters of this dissertation were focused on the theoretical and mathematical fundamentals that support zero-knowledge proofs and, in particular, zk-SNARKs. We have also studied mathematical fundamentals supporting Blockchain, such as elliptic curves, hash functions and Merkle trees.

In this chapter, we will study some implementations of zk-SNARKs in Blockchain and describe their internal functioning and benefits in combating privacy and scalability issues. We will not study how the zk-SNARKs are precisely defined because that could occupy a full dissertation itself. However, we will indicate the statements or conditions that want to be proven and where the proving and verifying steps occur.

We will present three solutions, one of them associated with Bitcoin and the other two related to Ethereum.

1. **ZCash.**
2. **Tornado Cash.**
3. **zk-Rollups.**

We will provide a diagram for each solution presented to enable the reader to understand the protocols' general structure rapidly. Let us begin with ZCash, the oldest of three solutions.

5.1. ZCash

One of the first approaches to achieving cryptocurrency privacy was Zerocoin, which quickly evolved into Zerocash. Zerocash is an Altcoin that enables users to transfer funds and validate transactions without disclosing some transaction information in the blockchain. This Alcoin was rapidly surpassed by ZCash; Zerocash's evolution that was presented by Zooko Wilcox and some professors from Johns Hopkins University.

Although we will first outline which features ZCash offers, we will describe how Zerocash and its zero-knowledge proofs were originally presented. This is because ZCash is a very sophisticated evolution of Zerocash, and describing its protocol would require a very comprehensive study that would be very difficult to explain in this dissertation succinctly. In addition to this, ZCash is fundamentally similar to Zerocash. For that reason, we will present Zerocash in Subsection 5.1.1 as a very primigenial ZCash version (which it really is). Let us begin by outlining ZCash's main features.

First, we must mention that because ZCash is an Altcoin, it has its own coin denomination: ZEC. The current exchange rate is around 1 BTC=2000 ZEC.

We move on to describe ZCash's principal feature: *shielding transactions*. Shielding consists of not disclosing the information of the participants or the data being transmitted while posting the transactions in the blockchain as usual.

Shielding is not what occurs in the original Bitcoin protocol. We already provided in Figure 4.1 a random transaction that we could extract from the blockchain. This transaction

discloses all the operation's data, such as the transferred amount. This is ideal for validation purposes. Any node can retrieve the block in which the transaction is contained and verify the transaction is correct. However, it poses a problem in privacy. zk-SNARKs provide the perfect tool to verify the correctness of a computation without disclosing the witnesses to the statement being proven. It provides privacy but allows for an easy validation process.

Once shielding is understood, we can differentiate between the different types of accounts in ZCash.

1. **Transparent accounts or t-addresses.** These addresses are similar to the classic Bitcoin addresses in that transactions between them are published in the network, and all their information fields are visible to every node.

2. **Shielded accounts or z-addresses.** Transactions between these types of addresses also appear in the blockchain. However, the addresses and transaction amounts are encrypted and not publicly visible.

A key feature that ZCash includes is called *memo fields*. The memo field of a transaction allows the sender to send some information to the receiver and is especially useful when making shielded transactions. The receiver of a shielded transaction does not know who sent him the money or the history of that money. Adding a brief description, such as an invoice or order number, allows the sender to indicate the reason for the payment to the receiver.

Another important feature available to z-address owners is the possibility of sharing a *view key*. This view key grants read access to anyone possessing the key but does not grant spending authority. This is referred to as *selective disclosure* by ZCash [ZCa24b] and provides fine-grain control over the information users can disclose.

Implementing the above-mentioned protocol properties is complex and scapes this dissertation's scope. Because of this, we will present a simplified version of the ZCash protocol heavily based on its Zerocash predecessor. We have used [Zer14], [DEHW24] and [BCT20] to dive into how this technology works.

5.1.1. How is privacy improved?

We separate the process into several phases. The first of these phases is the setup phase. As we saw in the Groth16 protocol, we first need to generate the common reference strings that will be used for the zero-knowledge proofs. This is done by the protocol itself, and there has been a big ongoing debate about whether the simulation trapdoor used during the setup phase of ZCash was indeed destroyed after the setup phase. Even so, Electric Coin Co. (the organization behind ZCash) has a dedicated section on their website specifically confirming the destruction of such trapdoor (see [ZCa24a]).

Once the setup phase is completed, addresses must be created. Any user can create a public-private key pair (see Subsection 4.4.1), let us call it (a_{pk}, a_{sk}), that will be used to make transactions.

Then, the *minting* phase can take place. Minting consists of creating new coins belonging to a predefined address by converting non-anonymous Bitcoins [Zer14]. Minting a new coin of value v works in the following way.

1. The user generates the coin's serial number using a pseudorandom function PRF_{seed}, a random value ρ, and his public key as seed such that

$$sn := \mathrm{PRF}_{a_{sk}}(\rho).$$

2. The user commits (see Section 2.5) to his public key and the random ρ using another random value r which concatenates to the input of a commitment scheme function h defined in the protocol. The image of the concatenated input, when fed to the function, is
$$k_1 := h(a_{pk}||\rho||r).$$
He then commits to that value k_1, and the amount v using now a new random value s, such that
$$k_2 := h(v||k_1||s).$$

3. The coin is created as the following vector
$$c := (a_{pk}, v, \rho, r, s, k_2),$$
and a record of the transaction (*mint transaction*) is defined as
$$tx_{mint} := (v, k_1, k_2, s).$$

It is important to highlight two aspects of this process. First, and most importantly, tx_{mint} will not be accepted by the network infrastructure or public ledger unless the user deposits v BTC. In other words, new coins can not be created out of nothing and must be supported by real funds.

Second, this double commitment scheme allows any user to verify k_2 is a commitment of value v. However, it does not allow us to discern who the owner is (finding the public key) or the coin's serial number. Thus, privacy has been achieved (and zk-proofs have not been needed).

The phase that follows minting is called *pouring*. During this phase, coins are spent by creating new coins at different addresses.

1. The owner of the coin extracts two new random values ρ_i where $i \in \{1, 2\}$.

2. After sampling random values r_i, s_i where $i \in \{1, 2\}$, he computes
$$k_1^i := h(a_{pk,i}||\rho_i||r_i), \text{ and}$$
$$k_2^i := h(v_i||k_1^i||s_i).$$

3. Two new coins are created, defined as the vectors described in (5.1).
$$\begin{aligned} c_1 &:= (a_{pk,1}, v_1, \rho_1, s_1, k_2^1), \\ c_2 &:= (a_{pk,2}, v_2, \rho_2, s_2, k_2^2). \end{aligned} \tag{5.1}$$
Now, the owner of the coins produces a zk-SNARK, let us call it π_{pour}, that proves his knowledge of c, c_1, c_2 and a_{sk} verifying that some conditions hold. A function problem is generated in which the public key and coins act as *witness* and *instance*, and the following conditions are satisfied when those exact values are used as inputs of the function.
   a) The coins are well formed (the commitments are indeed computed as the protocol states they are).

b) The secret key matches the public key (they belong to the same pair) and the serial number is correctly computed such that $sn = \text{PRF}_{a_{sk}}(\rho)$.

c) The coin is unspent; that is, the commitment to the old coin k_2 is still in the list of coin commitments.

d) No funds have been created, only transformed $v = v_1 + v_2$.

4. The protocol concludes when a pour transaction, $tx_{pour} := (sn, k_2^1, k_2^2, \pi_{pour})$ is added to the ledger (list of transactions) once the network verifies the proof is correct. The verification is done by network validators, similar to miners or full nodes in Bitcoin before it can be added to the ledger.

   Any user can verify the proof at any time without having to directly interact with the previous owner of the coin.

Let us highlight some important details to be considered if a user A wants to send funds to another user B whose public-private key pair is $(a_{pk,1}, a_{sk,1})$

- A must know $a_{pk,1}$ (B's public address). A should also send the the last four elements of c_1 $(v_1, \rho_1, s_1, k_2^1)$ in order to *transfer* the coin's ownership to B.
- B must verify the correctness of the data received because A could behave dishonestly and send fake data (not associated with any funds).
- If A wanted to spend the coin already transferred, he would need to have $a_{sk,1}$, which we assume belongs to B (as B owns the associated public key).
- The parameters associated with the second index $(i = 2)$ are used to return the exceeding funds to A (therefore $a_{pk,2}$ is assumed to be a_{pk}).
- At last, A cannot know when the coin is used as sn_1 is unknown to him.

In Figure 5.1, we present a use case where Alice wants to send 2 BTC to Bob without leaving a record of the transaction. We assume the exchange rate is 1 BTC = 2000 ZEC, (a_{pk}, a_{sk}) and (b_{pk}, b_{sk}) are key pairs, $\{s, s_1, s_2\}$ are random values and $\{k_1, k_2, k_1^1, k_1^2, k_2^1, k_2^2\}$ are commitments computed as explained above. We assume all validations and verifications are correct. We also assume that both users recover their Bitcoins after the transaction is completed. In Figure 5.2, we show the evolution of the unspent coins list.

5.1.2. Benefits of zk-SNARKs in the technology

As studied in Subsection 4.6.2, several mixing coin technologies have been presented to achieve privacy in the network. However, the solutions were complex and inefficient until zero-knowledge proofs were used because mechanisms such as multi-party signing or central authorities were needed. As Chapter 2 has shown, zero-knowledge proofs are a technology with a long history at its back. However, introducing succinct proofs significantly increased their usability. Anonymity and succinctness are exactly the two aspects mixers have aimed to obtain since their conception. Providing privacy to users should not come at the price of efficiency and transaction speed.

It is worth noting that without anonymity, mixers would not make sense. The objective is to hide the tracks the coins have followed to avoid malicious parties from discovering who owns the coins (or how they are used). Proving proof of ownership of a coin that is found

amongst others was originally done by multi-party signature schemes which require a great amount of effort and trust on a third party. Using zero-knowledge proofs, users are sure no one participating in the protocol can extract any information.

The succinctness of the proof allows this mixing process to be quick and seamless. Mixer's number of users would decline if the process were too burdensome or time-consuming. This ease has made the popularity of these technologies rapidly grow, as we will see it happenned with Tornado Cash, which is the next application we will present.

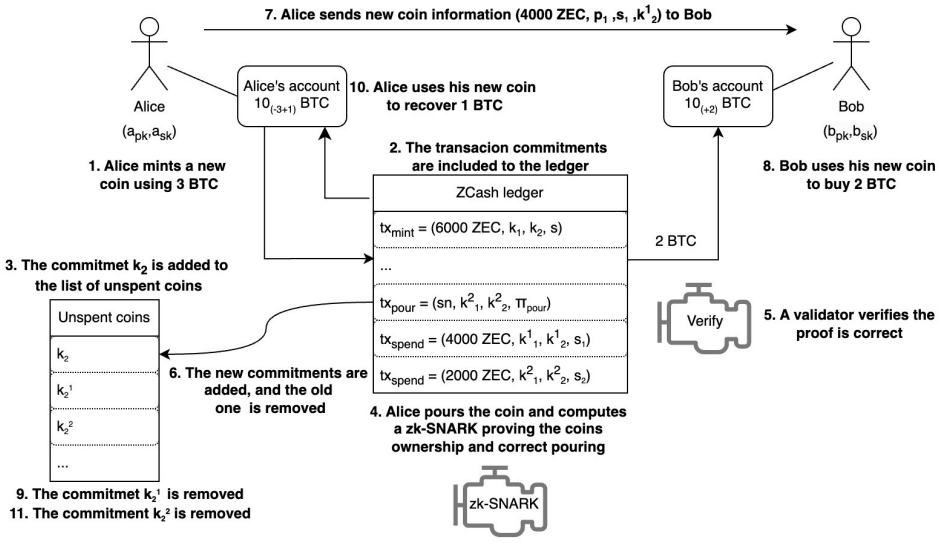


Figure 5.1.: A simplified ZCash transaction following the paradigm presented by Zerocash. Source: Elaborated by the author.

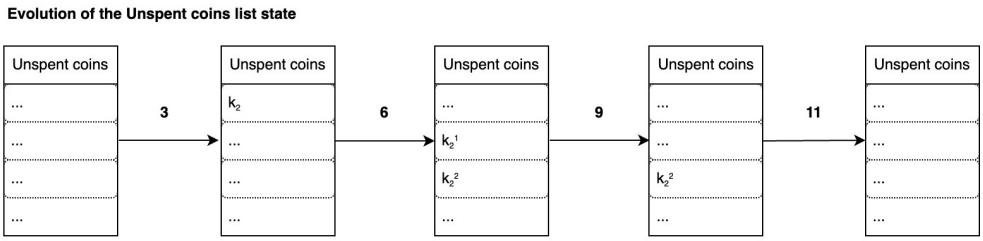


Figure 5.2.: Evolution of the unspent coins list associated with Figure 5.1. The numbers above the arrows indicate their corresponding step in the previous figure. Source: Elaborated by the author.

5.2. Tornado Cash

Tornado Cash is a mixing solution that differs from ZCash because of its direct implementation in the Ethereum network instead of being built on a layer above the cryptocurrency

(Bitcoin in the case of ZCash). This layer is the one in which Altcoins are built.

While solutions like ZCash seem highly beneficial in achieving privacy, some downsides exist, such as the effort users need to make to adopt a new cryptocurrency with its own infrastructure. Adopting a new currency has another downside, such as the risk of being affected by an unexpected exchange rate change between the currency and a *stablecoin* such as Bitcoin. We use the term stablecoins to refer to Bitcoin and Ethereum due to their lower volatility than other cryptocurrencies.

Solutions like Tornado Cash are built on top of widely used cryptocurrencies like Ethereum. While some kind of centralization is needed to mix the coins, users are already used to working with the infrastructure of the original cryptocurrency. Furthermore, there is no risk of losing funds due to the drop in price of an alternative cryptocurrency. These aspects make mixers like Tornado Cash the preferred mixing choice for many users across the Blockchain environment.

5.2.1. How is privacy improved?

Tornado Cash fundamentals are a simple set of premises presented in its white paper [AP19]. It consists of a smart contract found in the Mainnet that is responsible for receiving funds, recording the owners of these coins, verifying proofs of ownership and sending funds to the corresponding addresses at a later moment in time.

Before fully describing the protocol, we must note that the simplest use case of Tornado Cash implies there is no intermediate party between the smart contract and the user withdrawing the funds. That intermediate party would be called a *relayer* and its implementation is slightly more complex to understand. Let us now develop the basic use case scenario process.

First, users send some of their Ethereum funds, say N ETH, to a smart contract running in the Ethereum main blockchain or Mainnet. To do this, and provided there are two predefined hash functions H_1 and H_2 in the protocol, the user generates two random values. These values are $k, r \in \mathbb{B}^{248}$ where $\mathbb{B} = \{0, 1\}$. Then, the user computes the hash of their concatenation $H_2(k||r)$ and attaches this hash to the message sent to the smart contract containing the funds.

If the contract's Merkle trees are not full (storage is available), the hash is included in the tree as a leaf, and its position and the hash root of the tree are returned; let us call them R and l. Otherwise, the protocol is aborted.

To withdraw the funds identified by $H_2(k||r)$, the user must select a recipient address and the amount of funds he desires to withdraw. Then, he selects the root of the tree stored in the contract that contains his hash. Once the root R is fetched, the opening $O(l)$ is computed (see Subsection 4.3.4).

In addition to this, a *nullifier* $h = H_1(k)$ needs to be computed. The nullifier is another hash that indicates that the funds associated with the hashed key have been withdrawn once the process is complete. This nullifier must be added to a list of hashes that indicate the funds that have been withdrawn. Double spending would be possible without this list, making other users lose part of their coins.

One key aspect of this privacy solution is that the association between users (or accounts depositing ether in the smart contract) and their funds needs to be untraceable. To ensure this occurs, zk-SNARKs are used. The user withdrawing the funds (prover) needs to compute

a proof of the funds' ownership. In particular, the statement being proven is

$$S[R,h,A] = \{\text{ I know } k,r \in \mathbb{B}^{248}, l \in \mathbb{B}^{16}, O \in \mathbb{Z}_p{}^{16}, \text{ such that } h = H_1(k) \\ \text{and } O \text{ is the opening of } H_2(k||r) \text{ at position } l \text{ to } R\}. \tag{5.2}$$

Where A is the recipient address, $\mathbb{B}$,H_1 and H_2 are defined as before, $\mathbb{Z}_p$ is a finite prime order field and R and l are the previously defined root of the Merkle tree and position of the hash $H_2(k||r)$ in it.

The prover computes a zk-SNARK of the statement provided in (5.2) using the public parameters computed by a trusted setup for the statement. Let us call this proof P. At last, the prover sends an Ethereum transaction to the smart contract supplying $\{R,h,A,P\}$. In the other slightly more complex alternative, this would be sent to the relayer and include the relayer address (t) and the amount of ETH that could be spent in fees (f).

To conclude the procedure, the contract verifies the proof and uniqueness of the nullifier hash. In case everything is correct, it must send N ETH to A (and f to the relayer if used) and add h to the nullifier hashes to prevent double-spending.

In Figure 5.3, we present a use case where Alice wants to store 3 ETH to redeem them from another account later. She wants to do this in order to avoid being tracked by potential hackers who want to steal her funds. Other users also use the same contract, enabling her funds to get *lost* between them. We assume all validations and verifications are correct.

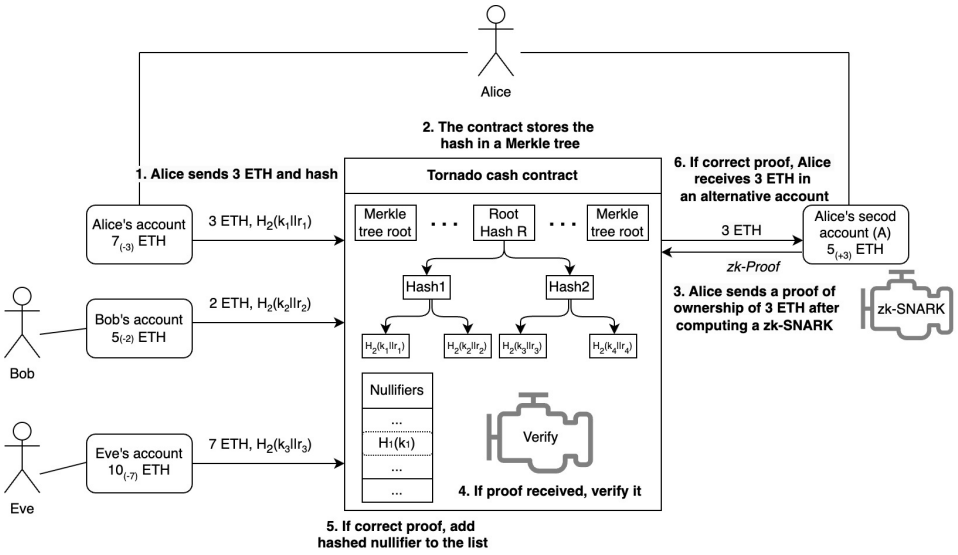


Figure 5.3.: A transaction of 3 ETH between Alice and a Tornado cash contract aiming to hide the tracks of the funds. Source: Elaborated by the author.

5.2.2. Benefits of zk-SNARKs in the technology

Tornado Cash is an example of a technology that would simply not exist if succinct zero-knowledge proofs were not as developed as they are. This is because being built on Ethereum and not being another full cryptocurrency has its benefits but it also has some downsides. One of them is that on-chain computations are pricy, and maintaining a smart contract

constantly verifying the proofs provided by users can be very expensive. However, the short verification time associated with zk-SNARKs lowers the price of this kind of solution. It makes it affordable for many more users who aim to achieve privacy in the network.

The clearest example of Tornado Cash achieving its purposes is the high number of deposits Tornado Cash contracts received between 2020 and 2022 (see Figure 5.4).

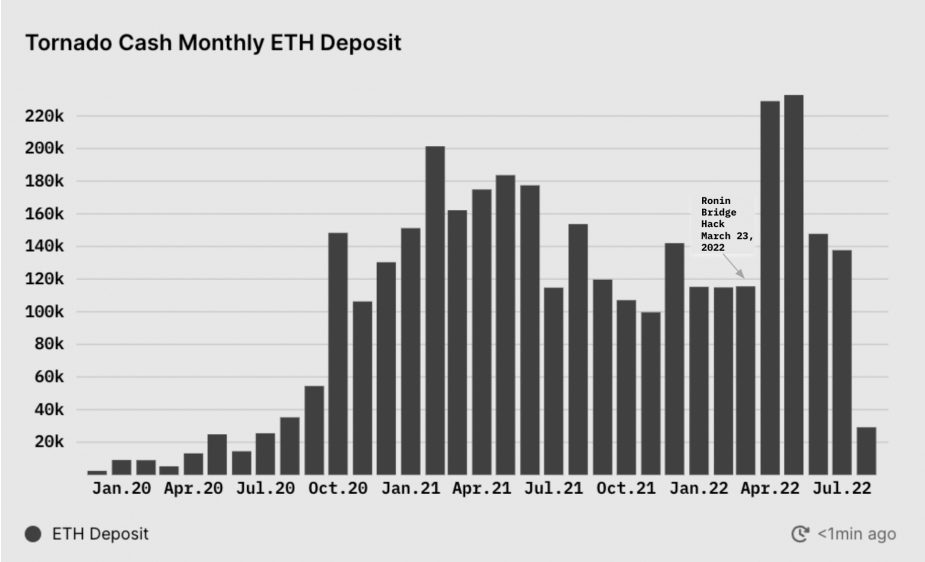


Figure 5.4.: Number of ETH deposited in Tornado Cash monthly between 2020 and 2022 amounting to a total of approximately $4,4B. The graph also shows the event that began the investigation against Alexey. Source: [De22].

However, this popularity has come at a high price to the creator of this technology, Alexey Pertsev. At the time of writing this dissertation, Alexey has been declared guilty of money laundering due to the use of Tornado Cash (not by Alexey himself but by criminals) to launder $1.2 billion in 36 illicit transactions [Blo24b]. This caused the technology to be blacklisted by the U.S. Government in 2022 and is proof of the tension between fully privacy-preserving technologies and regulatory efforts to avoid illicit activities.

Some other authors do not like to mention the obvious application of this kind of technology in money laundering, but it is a fact that must be addressed. The most avid readers may have realized this illicit application of the technology since we mentioned Alice was trying to *avoid being tracked*. We are not doubting Alice's good intentions. However, many criminals have used ZCash and Tornado Cash to avoid leaving tracks of how illicit money was earned or spent. Many exchanges (platforms where users can buy, sell, and trade cryptocurrencies), such as Binance, threatened to delist the so-called *privacy coins* because of fear of receiving backlash from the verdict of criminal investigations.

In conclusion, zk-SNARKs' benefits on mixers such as ZCash and Tornado Cash are more than evident. However, these benefits affect everyone in the same way and can help criminals get away with their crimes. We will not take a position in this battle between fully privacy-preserving technologies and crime-fighting on the net. However, we believe the users of these technologies should be aware of the kind of coins theirs are getting mixed with.

5.3. Ethereum zk-Rollups

zk-Rollups are rollups (see Subsection 4.7.2) that provide a validity proof of the transactions being grouped in batches when added to the blockchain. In this section, we will focus on how these are actually implemented and the benefits they have when solving Ethereum's scalability problem.

5.3.1. How is scalability improved?

Let us now study how zk-Rollups use zk-SNARKs to reduce the validation overhead. First, it is worth noting that operators in Layer 2 transmit state changes of the rollup to Layer 1 by using *calldata*. The *calldata* is a data area in smart contracts whose modification does not affect the chain's state. On the other hand, the batches of transactions are not validated in Layer 1 until an operator in Layer 2 has received enough transactions from users to build a block (similar to how it works in Layer 1).

The state of the zk-Rollup, including the accounts in Layer 2, is represented by a Merkle tree (see Subsection 4.3.4). Rollup operators compute a new state root every time a new state transition begins. This occurs when a transaction is received by an operator.

Suppose the validity proof (zk-SNARK) the Layer 2 operator provided to the verifying contract is correctly verified. In that case, the Merkle root associated with that proof identifies the tree that becomes the one users can see in the Mainnet. The precise steps that occur in the rollup are described here.

1. A transaction is received by a Layer 2 operator. This operator performs the usual checks a block proposer would perform in Layer 1, but instead of checking the state of the whole blockchain, it simply checks the state of the Merkle tree of the rollup.
2. Once the operator has received enough transactions, it computes the inputs that will be used to build the zero-knowledge proof. These are the following.
   a) Merkle tree root summarising all transactions in the bundle.
   b) Merkle proofs (or transactions' hashes openings) to prove transactions are included in the batch.
   c) Merkle proofs for each sender-receiver pair (these prove they are part of the state tree describing rollup).
   d) Intermediate state roots that prove the state changes were included in the batch when executing the transactions.

   It is worth noting that every change in the state tree root represents a state change and, therefore, needs to be included in the zk-SNARK.
3. After all these inputs are computed, a zk-SNARK can be built, and the latest Merkle tree root becomes the root of the state tree of the rollup.
4. The operator now submits the proof to the verifier contract in Layer 1. This contract verifies the validity of the proof. The public inputs used to carry out the verification are the state root before the rollup occurred, the state root after the rollup, the root of the batch and the input data of the transactions. Suppose all these are correct and the proof verification is correct. In that case, the verification contract (considered a special type of account and, consequently, part of the Ethereum blockchain state) is updated by changing the rollup's state tree.

In Figure 5.5, we present a use case where Alice wants to send 3 ETH to Bob using a rollup operator she trusts. She wants to do this to avoid the delay and the high gas fees that result from the network's saturation. We assume Alice had locked funds in her Layer 2 accounts before the transaction. We assume all validations and verifications are correct.

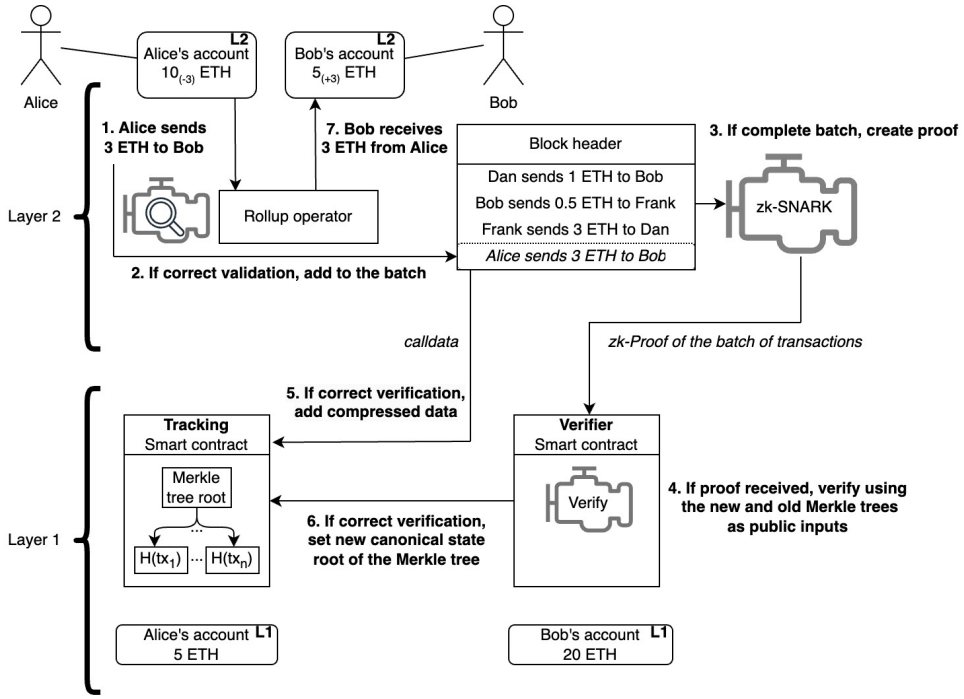


Figure 5.5.: A transaction between two users using an Ethereum zk-Rollup operator in Layer 2. Source: Elaborated by the author.

5.3.2. Benefits of zk-SNARKs in the technology

Storage and computation in Ethereum come at a price (a very expensive one). They require miners and validators to allocate resources to maintain and secure network data and execute smart contracts. Computation includes validating transactions and executing any code in a smart contract. Because of that, reducing the space transactions take and the Blockchain resources used in validating those transactions is highly valuable for the users.

As we explained in Section 3.1, succinctness, apart from the short length of the proof, refers to the short verification time of proofs. Without this last property, solving Ethereum's scalability issues using zero-knowledge proofs would be unfeasible. Transactions would take significantly longer than usual, and users would end up paying high gas fees in order to execute their transactions as fast as possible.

Besides reducing the storage and computational requirements of the Blockchain (these last part by offloading some validation tasks off-chain), zk-SNARKs also provide privacy to the users whose transactions are validated in Layer 2. The operator in Layer 2 indeed sees all the information as the Blockchain would if the transactions were to occur in Layer 1. However, this information is only recorded in the Blockchain (persistent storage) as a Merkle tree and

an associated proof. These allow users to prove transactions by opening the correct Merkle tree node while not showing anyone which addresses participated in each transaction and what amount was transferred.

This study of the benefits concludes the chapter on applications of zero-knowledge proofs in Blockchain. ZCash, Tornado Cash and Ethereum's zk-Rollups are three solutions that would not be possible without this technology. The succinctness of the proofs, in time and space, added to their capability of proving computations without leaking almost any information, has made zero-knowledge proofs a powerful tool for solving Blockchain's privacy and scalability problems.

6. Final summary and limitations

We summarize the most important points of the dissertation, addressing the topics of greatest significance.

6.1. Final summary

In this dissertation for our double degree in Computer Science and Mathematics, we have bridged the theoretical fundamentals of zero-knowledge proofs and their state-of-the-art applications in Blockchain. We began by detailing the mathematical foundations needed to comprehend the literature on zero-knowledge, tracing their development from the work of Goldwasser, Micali and Rackoff to the latest protocols such as Groth16. Classic examples like the 'Quadratic Residuosity Problem' and 'Discrete Logarithm Problem' make patent the necessity of having a good understanding of Number Theory and Abstract Algebra. Other examples, such as the 'Graph Isomorphism Problem' or the 'Graph Colouring Problem,' illustrate the broad mathematical knowledge required when dealing with zero-knowledge proofs. Topics in Complexity Theory and Language Theory frequently recur in the construction of efficient proofs tailored for cryptographic applications.

To reach zk-SNARKs, we had to eliminate the characteristic interactivity of the initial proofs. This paradigm change opened a window into the world of identity schemes, a field of Cryptography that demonstrated a clear connection with the zero-knowledge constructions of this dissertation. Upon reaching succinct arguments of knowledge, the complexity of state-of-the-art protocols became evident. A deep understanding of elliptic curves and bilinear mappings was necessary to grasp the rationale behind Jens Groth's choices in developing the Groth16 protocol.

Understanding Blockchain required fewer mathematically intense concepts, yet a solid grasp of hash functions, public key cryptography, and elliptic curves was essential to follow the basic protocols used in Bitcoin and Ethereum. From our analysis of these two cryptocurrencies and their technologies, we logically derived the challenges they face. This examination was supported by many external references on the problems, such as Blockchain analyzers and proposed solutions, including Altcoins, mixers, and rollups.

Finally, we materialized these proposed solutions in three specific examples: ZCash, Tornado Cash, and zk-Rollups. Each example was accompanied by a technical description, a descriptive diagram, and an analysis of the benefits of employing zk-SNARKs in the technology.

Additionally, a Python program using the Sagemath library (an open-source mathematics library that provides a comprehensive set of tools for Algebra, Calculus, and Number theory) was given in the appendix to allow readers to test and verify the values obtained in the proposed zero-knowledge example solved using Groth16.

6.2. Limitations

We did not cover certain topics in this dissertation for various reasons. We have grouped them here to outline potential improvements and areas for development in future work.

In the zero-knowledge proofs chapter, additional classic examples like the 'Graph Non-Isomorphism Problem' or the 'Hamiltonian Path Problem' could have been included. Moreover, the proof of Theorem 2.1 could have been provided if *Arthur-Merlin* protocols had been explained. However, this topic would have deviated from our clear path toward the Fiat-Shamir non-interactivity transformation.

The zk-SNARKs section extensively covered how statements and problems are defined and studied the Groth16 protocol in depth. Despite this, other state-of-the-art technologies such as zk-STARKs or Bulletproofs and protocols such as PLONK exist. Comparisons between these various options have widely been made in the literature, and it could have been an interesting addition to provide a brief overview of those ideas. Another aspect that we have not covered in this section is the use of *Circom* [Bay24a] and *snark.js* [Bay24b] for zk-SNARKs implementation. These are extensively used in practical implementations of zero-knowledge proofs; however, they introduce a level of abstraction that hinders the visualization of the theoretical fundamentals supporting the technology. Understanding the connection of these with the practical applications was the objective of our dissertation, and raising the level of abstraction would have distanced us from that objective.

The chapter on Blockchain covers the fundamental mathematical concepts and provides sufficient knowledge to understand the final chapter correctly. Concerning the last chapter, and connected to the limitation described in the previous paragraph, a study of the two mentioned programs could have allowed us to provide snippets of code of the actual implementations. The absence of these pieces of code is something we are aware of.

7. Conclusion and future directions

The increasing popularity of Blockchain has highlighted significant challenges cryptocurrencies face, particularly in the areas of privacy and scalability. Zero-knowledge proofs offer an effective solution to mitigate these issues. Specifically, the succinctness in proof size and verification time of zk-SNARKs enables developers to create solutions compatible with Blockchain's stringent resource constraints. These solutions either rely entirely on these proofs or significantly outperform their non-zero-knowledge counterparts.

Despite all this, the complexity of building simple zero-knowledge proofs demonstrates the hurdles that must be overcome when proving more elaborate statements. Furthermore, the lack of literature connecting the proofs' fundamentals and state-of-the-art proofs hinders this technology from being understood instead of studied as a black box.

This dissertation makes an attempt to bridge the gap between theoretical foundations and practical applications of zero-knowledge proofs in Blockchain. A precise description of the necessary mathematical tools to implement state-of-the-art proofs has demonstrated the wide range of mathematical machinery needed to fully comprehend the proofs being used.

A thorough examination of the early stages of the technology has naturally led to their contemporary implementations. Blockchain's inherent problems have necessitated the application of zk-SNARKs, showcasing the relevance of mathematical tools in modern applications such as Blockchain.

Examining the primary cryptocurrencies implemented within Blockchain, Bitcoin and Ethereum, has underscored the substantial complexity inherent in the cryptographic processes that support these systems. These range from fundamental public-key protocols to sophisticated multi-party mining and validation mechanisms. Often obscured from the average user, these protocols form the foundations of Blockchain and necessitate deep understanding to enhance their services, similar to what occurred with the proofs.

Our development of a code that emulates the Groth16 protocol's underlying process has also highlighted the critical role of mathematical structures, such as elliptic curves and bilinear mappings, in zk-SNARKs. It has also showcased why these proofs are computationally secure, contingent on the hardness of the discrete logarithm problem in elliptic curves.

Building on this dissertation's foundation, future research could further explore state-of-the-art zero-knowledge proofs, including *zk-STARKs* and *Bulletproofs*. Using tools such as *Circom* and *snark.js* to program proofs and empirically compare the efficiency of different protocols would be beneficial. This future work could be accompanied by the development of an identity-based scheme using other protocols such as PLONK. These protocols can be used to implement proofs for many more complex problems, which expand the applicability area of zero-knowledge proofs.

Another promising path for future research is the application of verifiable computing to artificial intelligence. During our research, we encountered numerous papers on *zero-knowledge machine learning*, a field gaining popularity in parallel to the rapid growth of artificial intelligence and generative models. Zero-knowledge proofs can ensure the correctness of model computations while maintaining data privacy, which is crucial for protecting sensitive information and harnessing models' insights.

For a potential PhD thesis, exploring the concept of *zero-knowledge trees* presents a novel research opportunity. Inspired by Ethereum's zk-Rollups Merkle trees, we could explore a different implementation approach in which each tree node would consist of a zero-knowledge proof, effectively building a set of recursive zero-knowledge proofs. Recent innovative literature [DD23] suggests that this structure offers reduced verification time and computational steps compared to existing alternatives, although it has not yet been implemented in zk-Rollups. Successfully implementing a zero-knowledge tree structure in an Ethereum rollup, reducing the communication and verification time, could be a groundbreaking contribution.

This dissertation has successfully achieved its objectives and equipped us with the knowledge and skills for a future research-oriented career on the topic. Our work serves as a comprehensive guide for mathematicians seeking to understand zero-knowledge proofs, their historical development, and their advanced implementations in Blockchain.

A. Appendix

Python code using the SageMath library that was used to obtain the polynomials of the example in Table 3.4, Table 3.5 and Table 3.6. See the full notebook in my GitHub.

```
# Define prime order field
q=19
F_q = GF(q)

# Define polynomial field
F_qt.<t> = F_q[]

# Define target polynomial
T = F_qt((t-11)*(t-5)*(t-7)*(t-3))


# Initialize non-zero polynomials
A0 = F_qt.lagrange_polynomial([(11,-2),(5,0),(7,0),(3,0)])
A2 = F_qt.lagrange_polynomial([(11,1),(5,0),(7,0),(3,0)])
A8 = F_qt.lagrange_polynomial([(11,0),(5,1),(7,0),(3,0)])
A9 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,1),(3,0)])
A10 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,0),(3,1)])

B3 = F_qt.lagrange_polynomial([(11,1),(5,0),(7,0),(3,0)])
B4 = F_qt.lagrange_polynomial([(11,0),(5,1),(7,0),(3,0)])
B5 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,1),(3,0)])
B6 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,0),(3,1)])

C1 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,0),(3,1)])
C7 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,0),(3,-3)])
C8 = F_qt.lagrange_polynomial([(11,1),(5,0),(7,0),(3,0)])
C9 = F_qt.lagrange_polynomial([(11,0),(5,1),(7,0),(3,0)])
C10 = F_qt.lagrange_polynomial([(11,0),(5,0),(7,1),(3,0)])

# Initialize the rest of polynomials
zero_poly = 19
A1,A3,A4,A5,A6,A7,B0,B1,B2,B7,B8,B9,B10,C0,C2,C3,C4,C5,C6 =
[0 for i in range(0,zero_poly)]

# Fill in the data: witness, instance, polynomials
W = [NaN,6,3,4,2,1,8,12,14,11]
I = [NaN,1]
A = [A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10]
B = [B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10]
```

```
C = [C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10]


n,m = len(I)-1,len(W)-1

# Define the polynomial expressions
A_poly = A[0] + sum([A[j]*I[j] for j in range(1,n+1)]) +
sum([A[j+n]*W[j] for j in range(1,m+1)])

B_poly = B[0] + sum([B[j]*I[j] for j in range(1,n+1)]) +
sum([B[j+n]*W[j] for j in range(1,m+1)])

C_poly = C[0] + sum([C[j]*I[j] for j in range(1,n+1)]) +
sum([C[j+n]*W[j] for j in range(1,m+1)])

# Calculate the final expression
P_final = A_poly * B_poly - C_poly

# Calculate the quotient polynomial
H = P_final/T
```

B. Appendix

Python code using the SageMath library that was used to obtain the elliptic curve and implementation of the Groth16 protocol for its use in Section 3.3. See the full notebook in my GitHub.

```
# Function defined to find the embedding degree of the elliptic curve of
# order prime_order with respect to prime_searched

def embedding_degree(prime_searched,prime_order):
    order = prime_order
    found = False
    # We iterate on k from 1 to order-1 because we know that k=order-1
    # verifies the equation because of Fermat's little theorem
    for k in range(1,prime_order-1):
        if (prime_order^k-1)%prime_searched == 0:
            found = True
            break

    if found==False:
        return -1
    return k

# Define limit of iterations and exit variable
max_lim = 100000
found = False

# Define the functions t(x) and p(x) as provided in the BLS6 method
def t(x):
    return x + 1

def p(x):
    return (1/3) * (1 - x)**2 * (x**2 - x + 1) + x

# Define prime value of the base prime field of the problem
q=19

# Iterate over values of x
for x in range(2, max_lim):
    p_x = p(x)
    t_x = t(x)
    expr = p_x + 1 - t_x
    # We ensure expr has q as a divisor, p_x is a prime integer and
```

```
    # embedding degree is 6
    if expr % q == 0 and p_x in ZZ is_prime(int(p_x)) and
    embedding_degree(q,p_x)==6:
        print(f"Taking␣x␣=␣{x}␣satisfies␣the␣condition␣with
␣␣␣␣␣␣␣␣p(x)␣=␣{p_x}␣and␣t(x)␣=␣{t_x}")
        found = True
        break;

if found == False:
    print(f"An␣x␣value␣verifying␣the␣conditions␣was␣not␣found.
␣␣␣␣Please␣change␣the␣prime␣q")

# Set up two random candidates following the BLS6 method
candidate_c2 = 2
candidate_c3 = 6

# Define the base field of order p_x
F_base = GF(p_x)

# Try if candidate_c2 is quadratic non-residue
c2 = F_base(candidate_c2)
try:
    c2.nth_root(candidate_c2)
except ValueError:
    print("c2␣is␣a␣quadratic␣non-residue")

# Try if candidate_c3 is cubic non-residue
c3 =F_base(candidate_c3)
try:
    c3.nth_root(candidate_c3)
except ValueError:
    print("c3␣is␣a␣cubic␣non-residue")

# As defined in BLS6: List of polynomials that need to be checked
# when j=0 and D =-3
polynomials = [[0,1],[0,c2^3],[0,c3^2],[0,c3^2*c2^3],[0,c3^(-2)],
[0,c3^(-2)*c2^3]]

# Reinitialize exit variable
found = False
# Iterate on the polynomials and find which one matches the order
for poly in polynomials:
    poly_used = poly
    # Define elliptic curve
    BLS = EllipticCurve(F_base,poly)
    # Verify it matches the desired order
    if BLS.order() == expr:
        found = True
```

```
        break;

# Print the exit variable, the found polynomial and the order of the curve
print(found,poly_used, BLS, BLS.order())

# Select a random point in the curve that will be used to find a generator
# of the cyclic subgroup G1

# Set random seed for reproducibility
set_random_seed(42)

# Define the point randomly or used the one used for obtaining the results
# presented in the dissertation: P1 = BLS(25734263,28283947,1)
P1 = BLS.random_point()
print(P1)

# We take the cofactor of the prime value whose cyclic group we aim to find
cofactor = factor(expr/q)
cofactor_value = cofactor.value()
print(cofactor, cofactor_value)

# We use the cofactor to obtain a generator of the cyclic group of order q
# It is an element of order q
g1 = int(cofactor_value)*P1
g1.xy()

# We return the subgroup of order q
G1_q = [x*g1 for x in range(0,q)]
print(G1_q)

# We now use the base field and define its extension
F_p = F_base
F_pt.<t> = F_p[]

# We find an irreducible polynomial of order the embedding degree k=6 to
# later define the extension of the curve
k=6
# We try first with a simple polynomial to ease the representation
poly  = F_pt(t^6+6)
# If this polynomial is not irreducible we find a random one
while poly.is_irreducible() == False:
    poly = F_pt.random_element(degree=k)

# We return the polynomial
print(poly)

# We define the curve extension
ExtBLS6 = EllipticCurve(F_pt_6,poly_used)
```

```
INF = ExtBLS6(0) # point at infinity

# We find the point that is applied to itself with the Frobenius isomorphism
for P in INF.division_points(q): # full q-torsion
    if P.order() == q: # exclude point at infinity
        piP = ExtBLS6([a.frobenius() for a in P])
        qP = p_x*P
        if piP == qP:
            break

# We define the generator of the second group of the Weil pairing domain
g2 = ExtBLS6(P.xy())
print(g2)

# We define the second cyclic group
G2_q = [ x*g2 for x in range(0,q) ]
print(G2_q)


# We intelligently define a set of values that we use to verify
# the Weil pairing  works as expected
index = [5,5,3,2] # 19

# We verify the values are well defined considering the
# chosen prime order q
if ((index[0]*index[1] - index[2]*index[3]) % q) != 0:
   print("The product of the indices are not congruent")

# We define the points in the respective groups and check the
# Weil pairings return the same value
Q = ExtBLS6(G1_q[index[0]].xy())
P = ExtBLS6(G2_q[index[1]].xy())

Q_1 = ExtBLS6(G1_q[index[2]].xy())
P_1 = ExtBLS6(G2_q[index[3]].xy())

# We are meant to obtain two identical values
print(Q.weil_pairing(P,q),Q_1.weil_pairing(P_1,q))

##### GROTH16 PROTOCOL - UNDER THE HOOD #####

## Including problem-specific data ##

# We define the base field of order q for the polynomials
# obtained in the previous program
Fq = GF(q)
Fqt.<t> = Fq[]
```

```
# Polynomials obtained in the previous program
# These must be changed in case the problem is changed
A = [Fqt(18*t^3 + 15*t^2 + 5*t + 10),Fqt(0),Fqt(10*t^3 + 2*t^2 + 7*t + 14),
Fqt(0),Fqt(0),Fqt(0),Fqt(0),Fqt(0),Fqt(4*t^3 + 11*t^2 + 11*t + 7),
Fqt(16*t^3 + 14*t + 1),Fqt(8*t^3 + 6*t^2 + 6*t + 17)]

B = [Fqt(0),Fqt(0),Fqt(0),Fqt(10*t^3 + 2*t^2 + 7*t + 14),
Fqt(4*t^3 + 11*t^2 + 11*t + 7),Fqt(16*t^3 + 14*t + 1),
Fqt(8*t^3 + 6*t^2 + 6*t + 17),Fqt(0),Fqt(0),Fqt(0),Fqt(0)]

C =[Fqt(0),Fqt(8*t^3 + 6*t^2 + 6*t + 17),Fqt(0),Fqt(0),Fqt(0),Fqt(0),
Fqt(0),Fqt(14*t^3 + t^2 + t + 6),Fqt(10*t^3 + 2*t^2 + 7*t + 14),
Fqt(4*t^3 + 11*t^2 + 11*t + 7),Fqt(16*t^3 + 14*t + 1)]

# Filling in the instance, witness, resulting polynomial, target polynomial
# and quotient polynomial
# These must be changed in case the problem is changed
I = [1]
W = [4,3,3,5,1,2,6,18,14]
P = Fqt(17*t^6 + 5*t^5 + 18*t^4 + 3*t^3 + 17*t^2 + 4*t + 2)
T = Fqt(t^4 + 12*t^3 + 8*t^2 + 7*t + 15)
H = Fqt(P/T)
total = [1]+ I + W

## Setup phase ##

# Initializing the common reference string
CRS = [[],[]]

# Setting up the simulation trapdoor
# These can be changed to test the program works for any
# sensible numeric choice
alpha = 6
beta = 5
gamma = 4
delta = 3
tau = 2

# We define the simulation trapdoor
ST = [alpha,beta,gamma,delta,tau]

# Setting up the two random values used for this execution
# These can also be changed
r = 10
s = 5

# We start by defining the first set of arguments of the first
# CRS vector
```

```
i = 0
deg = T.degree()
m = len(W)
n = len(I)

# Here is where the choices on alpha, beta and delta are first used
for g in (g1,g2):
    alpha_g = alpha * g
    beta_g = beta * g
    delta_g = delta * g
    gamma_g = gamma * g
    if  i==0:
        CRS[i].append([alpha_g,beta_g,delta_g])
    else:
        CRS[i].append([beta_g,gamma_g,delta_g])
    for j in range(0,deg):
        CRS[i][0].append(tau^j * g)
    i+=1

# We define empty arrays for the second, third and fourth arguments
# of the first CRS vector
second_arg = []
third_arg = []
fourth_arg = []

for j in range(0,n+1):
    g_1_power = g1 * ((beta * A[j](tau) + alpha * B[j](tau) +
    C[j](tau))/gamma)
    second_arg.append(g_1_power)

for j in range(1,m+1):
    g_1_power = g1 * ((beta * A[j+n](tau) + alpha * B[j+n](tau) +
    C[j+n](tau))/delta)
    third_arg.append(g_1_power)

for j in range(0,deg-1):
    g_1_power = g1 * ((tau^j * T(tau))/delta)
    fourth_arg.append(g_1_power)

# We append it to the CRS
CRS[0].append(second_arg)
CRS[0].append(third_arg)
CRS[0].append(fourth_arg)

## Proving phase ##

# Note that the values of A,B,C,T,P,H will not be used
# Only the values found in the CRS can be used because the
```

```
# Simulation trapdoor is unknown to the prover

# Note that the witness instance information is used to create the proof


# We initialize the rest of the elements that are used
# (directly or indirectly) in the CRS definition
g1_H,g1_W,g1_A,g1_B,g2_B,g1_C = 0,0,0,0,0,0

# Obtaining g1_W #

# Initializing
g1_W = 0
j=0

# Iterating on the CRS parameters and introducing the witness information
for c in CRS[0][2]:
    g1_W += W[j] * c
    j+=1
print(g1_W)

# Obtaining g1_H #

# Initializing
g1_H = 0
j = 0

# Iterating on H's coefficients introducing its information
for h in H.coefficients(sparse=False):
    g1_H += CRS[0][3][j] * h
    j+=1
print(g1_H)

# Obtaining g1_A #

# Initializing
j=tau_power=0
g1_A = CRS[0][0][0] # g1 * alpha

# Iterating on (1,instance,witness)
for iw in total:
    for a in A[j].coefficients(sparse=False):
        g1_A += CRS[0][0][3+tau_power] * a * iw
        tau_power+=1
    tau_power=0
    j+=1

# Adding last term
```

```
g1_A += r * CRS[0][0][2] # r * g1 * delta
print(g1_A)

# Obtaining g1_B #

# Initializing
j=tau_power=0
g1_B = CRS[0][0][1] # g1 * beta

# Iterating on (1,instance,witness)
for iw in total:
    for b in B[j].coefficients(sparse=False):
        g1_B += CRS[0][0][3+tau_power] * b * iw
        tau_power+=1
    j+=1
    tau_power=0

# Adding last term
g1_B += s * CRS[0][0][2] # s * g1 * delta
print(g1_B)

# Obtaining g2_B #

# Initializing
j=tau_power=0
g2_B = CRS[1][0][0] # g2 * beta

# Iterating on (1,instance,witness)
for iw in total:
    for b in B[j].coefficients(sparse=False): # We
        g2_B += CRS[1][0][3+tau_power] * b * iw
        tau_power+=1
    j+=1
    tau_power=0

# Adding last term
g2_B += s * CRS[1][0][2] # s * g2 * delta
print(g2_B)

# Obtaining g1_C
g1_C = g1_W
g1_C += g1_H
g1_C += s * g1_A
g1_C += r * g1_B
g1_C += (-r * s) * CRS[0][0][2]
print(g1_C)

# The generated proof is (g1_A,g1_C,g2_B)
```

```
## Verifying phase ##

# Note the verifier can only use (g1_A,g1_C,g2_B), the CRS,
# the Weil pairing and the computed g1_I

# Obtaining g1_I
j=0
g1_I = 0
I_1 = [1] + I
for c in CRS[0][1]:
    g1_I += I_1[j] * c
    j+=1
print(g1_I)

ExtBLS6(g1_A).weil_pairing(g2_B,q) ==
(ExtBLS6(CRS[0][0][0]).weil_pairing(CRS[1][0][0],q) *
ExtBLS6(g1_I).weil_pairing(CRS[1][0][1],q) *
ExtBLS6(g1_C).weil_pairing(CRS[1][0][2],q))

##### VERIFYING THE PROTOCOL IS CORRECT #####

## Defining a discrete log function using the baby step giant step ##

# Input: point and cyclic prime subgroup
# Output: the discrete log of the point modulo the subgroup's order

def discrete_log(P, G):
    # Obtain order and ceiling m
    n = G.order()
    m = int(n**0.5) + 1

    # Define the baby steps dicitionary
    baby_steps = {}
    for j in range(m):
        baby_steps[j * G] = j

    # Define the point m in the group
    mG = m * G

    # Search for the discrete log iterating
    for i in range(m):
        giant_step = P - i * mG
        if giant_step in baby_steps:
            return i * m + baby_steps[giant_step]

    return None  # The discrete logarithm was not found
```

```
# Make the verification

((discrete_log(ExtBLS6,g1_A,g1)*discrete_log(ExtBLS6,g2_B,g2))%q) ==
(discrete_log(ExtBLS6,CRS[0][0][0],g1)*discrete_log(ExtBLS6,CRS[1][0][0],g2) +
discrete_log(ExtBLS6,g1_I,g1)*discrete_log(ExtBLS6,CRS[1][0][1],g2) +
discrete_log(ExtBLS6,g1_C,g1)*discrete_log(ExtBLS6,CRS[1][0][2],g2))%q

# If the output is True the proof has been correctly built
```

C. Appendix

Ethereum's smart contract simulating a vending machine extracted from [Wac24a].

```
pragma solidity 0.8.7;

contract VendingMachine {

    // Declare state variables of the contract
    address public owner;
    mapping (address => uint) public cupcakeBalances;

    // When 'VendingMachine' contract is deployed:
    // 1. set the deploying address as the owner of the contract
    // 2. set the deployed smart contract's cupcake balance to 100
    constructor() {
        owner = msg.sender;
        cupcakeBalances[address(this)] = 100;
    }

    // Allow the owner to increase the smart contract's cupcake balance
    function refill(uint amount) public {
        require(msg.sender == owner, "Only the owner can refill.");
        cupcakeBalances[address(this)] += amount;
    }

    // Allow anyone to purchase cupcakes
    function purchase(uint amount) public payable {
        require(msg.value >= amount * 1 ether,
        "You must pay at least 1 ETH per cupcake");
        require(cupcakeBalances[address(this)] >= amount,
        "Not enough cupcakes in stock to complete this purchase");
        cupcakeBalances[address(this)] -= amount;
        cupcakeBalances[msg.sender] += amount;
    }
}
```

Glossary of mathematical notation

Main mathematical symbols used throughout the dissertation.

$\mathbb{N}$ Set of natural numbers.

$\mathbb{N}_0$ Set of natural numbers including 0.

$\mathbb{Z}$ Set of integer numbers.

$\mathbb{Z}_n^*$ Set of units of $\mathbb{Z}_n$.

$\mathbb{Z}_n$ Ring of integers numbers modulo n.

$mod(n)$ Modulo n.

$\varphi(n)$ Totient function evaluated in n.

$\gcd(a, b)$ Greatest common divisor of a and b.

$\log(a)$ Logarithm of a (undefined base).

$\log_2(a)$ Logarithm to the base 2 of a.

$(\frac{a}{p})$ Legendre symbol.

$(\frac{a}{b})$ Jacobi symbol.

$\#G$ Cardinality of group G.

$\lfloor \gamma \rfloor$ Floor function evaluated in γ.

$\lceil \gamma \rceil$ Ceiling function evaluated in γ.

$\text{ord}(\alpha)$ Order of element alpha.

$G = \langle \alpha \rangle$ G is a cyclic group generated by α.

$|G|$ Cardinality of group G (alternative notation).

$\text{Sym}(V)$ Permutations group of set V.

prf Pseudo random function.

H Hash function.

L Formal language.

$O(\text{f})$ O notation for function f.

$\mathbb{F}$ Field.

$\mathbb{F}_p$ Field of size p.

$\mathbb{F}_p^*$ Group of units of field of size p.

$E_{a,b}(\mathbb{F})$ Elliptic curve over field $\mathbb{F}$ with equation $y^2 = x^3 + ax + b$.

$\mathrm{e}(p, q)$ Weil pairing of points p and q.

$\mathrm{E_{a,b}}(\mathbb{F}_p)[r]$ r-torsion group of a elliptic curve $E_{a,b}(\mathbb{F})$.

$k(r)$ Embedding degree of the elliptic curve $E_{a,b}(\mathbb{F})$ with respect to r.

$\mathrm{E_{a,b}}(\mathbb{F}_{p^{k(r)}})[r]$ Full r-torsion group of an elliptic curve defined on the finite field of size $p^{k(r)}$.

ω Point at infinity.

$\mathbb{G}[r]$ Subgroup of the r-torsion group of the elliptic curve.

$[g]a$ Point a evaluated in the elliptic curve generated by g.

g^A Polynomial A evaluated in the elliptic curve subgroup generated by g.

$[a]_i$ Value a represented in a elliptic subgroup $\mathbb{G}_i$.

$O(\mathrm{l})$ Opening of leaf l in a Merkle tree.

$a||b$ Concatenation of two binary strings a and b.

$(A, B)(x)$ Interactive proof system of two parties A and B and input x.

$S(x)$ Probabilistic polynomial-time simulator fed with input x.

$D(x)$ Probabilistic polynomial-time distinguisher fed with input x.

$Pr[\{a\}]$ Probability of a.

Glossary of zero-knowledge terms

Main zero-knowledge terminology used throughout the dissertation.

Interactive proof system Two interactive Turing Machines that would communicate, allowing one to prove a statement to the other (probabilistically) in a finite amount of time.

Completeness Property of an interactive proof system where all true statements can be proven using the proof system.

Soundness Property of an interactive proof system where a false statement can never be derived using the proof system.

Zero-knowledge Property of an interactive proof system where the verifier cannot prove anything about the statement to anyone once the proving protocol has been completed.

Proof of knowledge Protocol where a prover aims to prove to a verifier that he knows something.

Witness Piece of information that allows anyone to verify a statement is true.

Knowledge extractor Verifier with special capabilities that interacts with the prover mimicking the communication a verifier would have with the prover during the usual execution of the proving protocol.

Special soundness Property of an interactive proof system where only a prover who knows a witness can convince the verifier.

Commitment scheme Process in which a party can compromise to one value while still being kept hidden from the others to reveal later the value to which he was committed.

Sigma protocol Special type of proof of knowledge characterized by its 3-move structure.

Honest verifier zero-knowledge Property of an interactive proof system where the classic zero-knowledge property holds when the verifier behaves honestly.

CRS Common Reference String.

Argument Any proof where it is only required that a polynomial prover cannot convince a verifier of a false statement. In other words, being *tricked* by a computationally unbounded prover is acceptable.

Argument of knowledge An argument in which the prover proves the truth of a statement and the knowledge of a witness to that statement.

zk-SNARG Zero-knowledge Succinct Non-Interactive Argument.

zk-SNARK Zero-knowledge Succinct Non-Interactive Argument of Knowledge.

R1CS Rank-1 Constraint System.

QAP Quadratic Arithmetic Program.

Groth-16 State-of-the-art zk-SNARK protocol created by Jens Groth.

Glossary of Blockchain terms

Main Blockchain terminology used throughout the dissertation.

Blockchain Digitally distributed public ledger found in a network containing transactions between participating parties.

Cryptocurrency Digital currency implemented in Blockchain.

Decentralized system Structure in which its members operate on local information to achieve global goals without a governing central authority.

Peer-to-peer network Group of computers or individuals interacting with each other, having similar permissions and responsibilities when processing data.

Private key Randomly generated secret number, only known by the person who generated it, and used to decrypt a message encrypted with the public key and create digital signatures.

Public key Randomly generated secret number uniquely identified with a secret key, widely distributed, and used to encrypt a message sent to the public/private key pair's owner. It is also used to verify the owner's digital signature.

Address Hash of a public key that can also be viewed as a virtual location where some coins can be sent.

Wallet Software program that stores the user's private keys and enables him to access his coins. It knows which private key is associated with each public address.

Transaction Piece of information indicating the transfer of tokens from one address to another signed by the sender's private key.

Block A block is a data structure that contains transactions. Blocks are linked by hashing the latest block of a chain of blocks and including that hash in the following block.

Cryptocurrency exchange Digital platform or service that allows users to buy, sell, trade, and store cryptocurrencies.

Consensus algorithm Mechanism that ensures agreement on a single data value among distributed processes or systems.

Proof of work Consensus algorithm used in Bitcoin.

Miner Node that adds blocks to the network following a proof of work consensus algorithm.

Proof of stake Consensus algorithm used in Ethereum.

Validator Node that adds blocks to the network following a proof of stake consensus algorithm.

Gas Required fee that must be paid to execute a transaction in the Ethereum network.

Smart contract Special type of Ethereum account containing code that behaves as a real-life contract whose main property is its immutability.

Merkle tree Binary tree whose nodes are labelled using hash values.

Privacy Assurance that the confidentiality and accessibility to certain information about an entity are protected.

Mixer Third party that receives, divides, mixes, and redistributes users' coins between many other users, hindering blockchain analyzers from tracking users' transactions.

Altcoin Other currencies inspired by Bitcoin that usually work as extensions or full-fledged currencies.

Scalability Ability of a computer application or protocol to maintain its performance when its context's size changes in size.

Layer 1 Main Ethereum blockchain or Mainnet.

Layer 2 Secondary frameworks or protocols built on top of the main Ethereum blockchain to improve its scalability and efficiency.

Rollup Layer 2 solution that groups transactions into batches or bundles, executes them off-chain and sends them to the Mainnet to be added to the blockchain.

Bibliography

[AKR+13] Elli Androulaki, Ghassan O. Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security*, pages 34–51, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ant24] Andreas M. Antonopoulos. Mastering bitcoin 2. chapter 4. keys, addresses. https://www.oreilly.com/library/view/mastering-bitcoin-2nd/9781491954379/ch04.html, 2024. [Online resource, accessed on the 12th of April of 2024].

[AP19] Roman Storm Alexey Pertsev, Roman Semenov. Tornado cash privacy solution. https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf, 2019. [Online resource, accessed on the 6th of June of 2024].

[Aut24] Least Authority. The moonmath manual to zk-snarks. https://leastauthority.com/community-matters/moonmath-manual/, 2024. [Online resource, accessed on the 10th of May of 2024].

[Awa24] Rahul Awati. Consensus algorithm. https://www.techtarget.com/whatis/definition/consensus-algorithm#:~:text=A%20consensus%20algorithm%20is%20a,among%20distributed%20processes%20or%20systems, 2024. [Online resource, accessed on the 30th of March of 2024].

[Bar07a] Boaz Barak. Lecture 15 - zero knowledge proofs. https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec15.pdf, 2007. [Online resource, accessed on the 9th of June of 2024].

[Bar07b] Boaz Barak. Lecture 15 - zero knowledge proofs. https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec16.pdf, url=https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec16.pdf , 2007. [Online resource, accessed on the 9th of June of 2024].

[Bar07c] Boaz Barak. Lecture 17 - diffie-hellman key exchange, pairing, identity-based encryption and forward security. https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec17.pdf, url=https://www.cs.princeton.edu/courses/archive/fall07/cos433/lec17.pdf , 2007. [Online resource, accessed on the 9th of June of 2024].

[Bay24a] Jordi Baylina. Circom. https://github.com/iden3/circom, 2024. [Online resource, accessed on the 9th of June of 2024].

[Bay24b] Jordi Baylina. snarkjs. https://github.com/iden3/snarkjs, 2024. [Online resource, accessed on the 9th of June of 2024].

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BCT20] Aritra Banerjee, Michael Clear, and Hitesh Tewari. Demystifying the role of zk-snarks in zcash. *CoRR*, abs/2008.00881, 2020.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[Bin24] Binance. Binance. https://www.binance.com, 2024. [Online resource, accessed on the 24th of May of 2024].

[Bis14] Goutam Biswas. Basic properties of integers iv. https://cse.iitkgp.ac.in/~goutam/cnt/lect/Lect5.pdf, 2014. [Online resource, accessed on the 13th of July of 2024].

[BLN23] Aleksander Berentsen, Jeremias Lenzi, and Remo Nyffenegger. An introduction to zero-knowledge proofs in blockchains and economics. 105(4), 2023.

[Blo24a] Blockchain.com. Blockchain.com. https://www.blockchain.com/explorer/transactions/btc/d94b0765585f44fdc3acbccd9ebbf88710af13d52a0d026c8daba09d2df38775, 2024. [Online resource, accessed on the 12th of April of 2024].

[Blo24b] Blockhead. Tornado cash's alexey pertsev appeals guilty verdict after 64 months sentence. https://www.blockhead.co/2024/05/17/tornado-cashs-alexey-pertsev-appeals-guilty-verdict/, 2024. [Online resource, accessed on the 6th of June of 2024].

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees, 2002. https://eprint.iacr.org/2002/088.

[BMC+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015.

[BNM+14] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua Kroll, and Edward Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. pages 486–504, 03 2014.

[Bon23a] Dan Boneh. Building a snark. https://cs251.stanford.edu/lectures/lecture15.pdf, 2023. [Online resource, accessed on the 10th of May of 2024].

[Bon23b] Dan Boneh. Using zk-snarks for privacy on the blockchain. https://cs251.stanford.edu/lectures/lecture14.pdf, 2023. [Online resource, accessed on the 10th of May of 2024].

[But14] Vitalik Buterin. Ethereum white paper. https://ethereum.org/en/whitepaper/, url=https://ethereum.org/en/whitepaper/, note=[Online resource, accessed on the 24th of May of 2024], 2014.

[Cha82] David Lee Chaum. *Relaxation Effects for Coupled Nuclear Spins*. PhD thesis, University of California, Berkeley, Berkeley, CA, April 1982.

[Coi24] Coinbase. Coinbase. https://www.coinbase.com, 2024. [Online resource, accessed on the 24th of May of 2024].

[CSKLR18] Mauro Conti, E. Sandeep Kumar, Chhagan Lal, and Sushmita Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys & Tutorials*, 20(4):3416–3452, 2018.

[CXS+18] Wubing Chen, Zhiying Xu, Shuyu Shi, Yang Zhao, and Jun Zhao. A survey of blockchain applications in different domains. In *Proceedings of the 2018 International Conference on Blockchain Technology and Application*, ICBTA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[Dai] Wei Dai. B-money. http://www.weidai.com/bmoney.txt. [Online resource, accessed on the 27th of March of 2024].

[DBDDP+] Giuseppe Di Battista, Valentino Di Donato, Maurizio Patrignani, Maurizio Pizzonia, Vincenzo Roselli, and Roberto Tamassia. Bitconeview: Visualization of flows in the bitcoin transaction graph. https://dltgroup.it/DLTWorkshop/presentazioni%20DLT%20workshop/pizzonia1.pdf. [Online resource, accessed on the 27th of March of 2024].

[DBDDP+15] Giuseppe Di Battista, Valentino Di Donato, Maurizio Patrignani, Maurizio Pizzonia, Vincenzo Roselli, and Roberto Tamassia. Bitconeview: visualization of flows in the bitcoin transaction graph. In *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, 2015.

[DD23] Sai Deng and Bo Du. zkTree: A zero-knowledge recursion tree with ZKP membership proofs, 2023. https://eprint.iacr.org/2023/208.

[De22] Nikhilesh De. Crypto-mixing service tornado cash blacklisted by us treasury. https://www.coindesk.com/policy/2022/08/08/crypto-mixing-service-tornado-cash-blacklisted-by-us-treasury/, 2022. [Online resource, accessed on the 6th of June of 2024].

[DEHW24] Taylor Hornby Daira Emma Hopwood, Sean Bowe and Nathan Wilcox. Zcash protocol specification. https://zips.z.cash/protocol/protocol.pdf, 2024. [Online resource, accessed on the 6th of June of 2024].

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE TRANSACTIONS ON INFORMATION THEORY*, IT-22(6), Nov 1976.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. *Advances in Cryptology — CRYPTO' 92*, page 139–147, 1993.

[Em23] Emaxx.ru. Binary exponentiation. https://cp-algorithms.com/algebra/binary-exp.html, 2023. [Online resource, accessed on the 13th of July of 2024].

[Eth24] Ethereum. What is staking? https://ethereum.org/en/staking/, 2024. [Online resource, accessed on the 28th of May of 2024].

[Fin] Hal Finney. Rpow - reusable proofs of work. https://nakamotoinstitute.org/finney/rpow. [Online resource, accessed on the 27th of March of 2024].

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[gBfu] gmaxwell (Bitcointalk forum username). Coinjoin: Bitcoin privacy for the real world. https://bitcointalk.org/index.php?topic=279249.0. [Online resource, accessed on the 28th of March of 2024].

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[GMR89] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18(1):186–208, Feb 1989.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, volume 5677 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2009.

[Gro16] Jens Groth. *On the Size of Pairing-Based Non-interactive Arguments*. 1 2016.

[GSV98] Oded Goldreich, Amit Sahai, and Salil Vadhan. Honest-verifier statistical zero-knowledge equals general statistical zero-knowledge. *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, 1998.

[HW59] G. H. Hardy and E. M. Wright. An introduction to the theory of numbers. https://blngcc.wordpress.com/wp-content/uploads/2008/11/hardy-wright-theory_of_numbers.pdf, 1959. [Online resource, accessed on the 13th of July of 2024].

[KI90] Michael Rosen Kenneth Ireland. A classical introduction to modern number theory. https://link.springer.com/book/10.1007/978-1-4757-2103-4, 1990. [Online resource, accessed on the 13th of July of 2024].

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). STOC '92, page 723–732, New York, NY, USA, 1992. Association for Computing Machinery.

[Kob94] Neal Koblitz. A course in number theory and cryptography. https://link.springer.com/book/10.1007/978-1-4419-8592-7, 1994. [Online resource, accessed on the 13th of July of 2024].

[Leg98] A. M. Legendre. Essai sur la théorie des nombres. 1798.

[Lyn23] Ben Lynn. Quadratic reciprocity. https://crypto.stanford.edu/pbc/notes/numbertheory/quadrecip.html, 2023. [Online resource, accessed on the 13th of July of 2024].

[Lyn24] Ben Lynn. Number theory. https://crypto.stanford.edu/pbc/notes/numbertheory/, 2024. [Online resource, accessed on the 13th of July of 2024].

[Mao03] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.

[Mek] Ralph Mekleor. Publishing a new idea. https://www.ralphmerkle.com/1974/. [Online resource, accessed on the 26th of March of 2024].

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, 2013.

[Nak] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[Nex24] Nexus. Nexus. https://www.nexus.xyz/, url=https://www.nexus.xyz/, note=[Online resource, accessed on the 10th of May of 2024], 2024.

[NNR98] Moni Naor, Yael Naor, and Omer Reingold. Applied kid cryptography or how to convince your children you are not cheating. 1998.

[@om24] @omahs. Nodes and clients. https://ethereum.org/en/developers/docs/nodes-and-clients/, 2024. [Online resource, accessed on the 28th of May of 2024].

[Onl24] Math Online. Legendre symbols. http://mathonline.wikidot.com/legendre-symbols, 2024. [Online resource, accessed on the 13th of July of 2024].

[oS24] Web of Science. Web of science. , 2024. [Online resource, accessed on the 9th of June of 2024].

[Pab22] Sarah Pabst. Crypto's long-awaited 'merge' reaches the finish line. https://www.nytimes.com/2022/09/15/technology/ethereum-merge-crypto.html#:~:text=The%20Merge%20shifts%20Ethereum%20to,enters%20them%20in%20a%20lottery., 2022. [Online resource, accessed on the 24th of May of 2024].

[poi23] pointbiz. bitaddress.org. https://www.bitaddress.org/bitaddress.org-v3.3.0-SHA256-dec17c07685e1870960903d8f58090475b25af946fe95a734f88408cef4aa194.html, 2023. [Online resource, accessed on the 12th of April of 2024].

[Pop04] J. W. Pope. Proving a theorem in zero-knowledge. 2004.

[QG90] Jean-Jacques Quisquater and Louis Guillou. How to explain zero-knowledge protocols to your children. *Advances in Cryptology — CRYPTO' 89 Proceedings*, 435:628–631, 1990. In collaboration with Tom BERSON for the English version.

[RMSK14] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 345–364, Cham, 2014. Springer International Publishing.

[RMSK17] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2p mixing and unlinkable bitcoin transactions. 01 2017.

[San23] Jan R Sandbakken. Elliptic curve addition. https://www.geogebra.org/m/GCrVzWCV, 2023. [Online resource, accessed on the 11th of April of 2024].

[Sch90] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[Seo11] Jae Hong Seo. Round-efficient sub-linear zero-knowledge arguments for linear algebra. In *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, page 387. Springer, 2011.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. 1971.

[Sha85] Adi Shamir. Identity-based cryptosystems and signature schemes. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 47–53, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[SHMM21] Johann Stockinger, Bernhard Haslhofer, Pedro Moreno-Sanchez, and Matteo Maffei. Pinpointing and measuring wasabi and samourai coinjoins in the bitcoin ecosystem. *CoRR*, abs/2109.10229, 2021.

[Smi24] Corwin Smith. Scaling. https://ethereum.org/en/staking/, 2024. [Online resource, accessed on the 4th of June of 2024].

[Spa13] Michele Spagnuolo. Bitiodine: Extracting intelligence from the bitcoin network. Master's thesis, Politecnico di Milano, Milano, Italy, 2013.

[Sta24] Statista. Statista. https://www.statista.com/statistics/1269302/crypto-market-share/, 2024. [Online resource, accessed on the 3rd of July of 2024].

[Tre24] Google Trends. Google trends. https://trends.google.es/trends/explore?date=2020-01-06%202022-01-06&q=Bitcoin,Ethereum,%2Fm%2F025rs2z,Covid&hl=en, 2024. [Online resource, accessed on the 3rd of July of 2024].

[Vil23] Jorge L. Villar. Zero-knowledge proofs notes. https://web.mat.upc.edu/jorge.villar/doc/notes/DataProt/zk.html, 2023. [Online resource, accessed on the 30th of March of 2024].

[Wac24a] Paul Wackerow. Introduction to smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/, 2024. [Online resource, accessed on the 28th of May of 2024].

[Wac24b] Paul Wackerow. Proof of stake. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, 2024. [Online resource, accessed on the 28th of May of 2024].

[Wik24a] Proof Wiki. Properties of legendre symbol. https://proofwiki.org/wiki/Properties_of_Legendre_Symbol, 2024. [Online resource, accessed on the 13th of July of 2024].

[Wik24b] Wikipedia.com. Wikipedia.com. https://en.wikipedia.org/wiki/Computational_complexity_theory, url=https://en.wikipedia.org/wiki/Computational_complexity_theory, note=[Online resource, accessed on the 14th of April of 2024], 2024.

[Wu19a] David Wu. Fully homomorphic encryption. https://crypto.stanford.edu/cs355/19sp/lec16.pdf, 2019. [Online resource, accessed on the 9th of June of 2024].

[Wu19b] David Wu. Lecture 17: Succinct non-interactive arguments. https://crypto.stanford.edu/cs355/19sp/lec17.pdf, 2019. [Online resource, accessed on the 9th of June of 2024].

[YCh24] YCharts. Ethereum average gas price. https://ycharts.com/indicators/ethereum_average_gas_price, 2024. [Online resource, accessed on the 4th of June of 2024].

[ZCa24a] ZCash. Does zcash contain a backdoor? https://z.cash/learn/does-zcash-contain-a-backdoor/, 2024. [Online resource, accessed on the 6th of June of 2024].

[ZCa24b] ZCash. Zcash basics. https://zcash.readthedocs.io/en/latest/rtd_pages/basics.html, 2024. [Online resource, accessed on the 6th of June of 2024].

[ZCYW23] Min Zhang, Yu Chen, Chuanzhou Yao, and Zhichao Wang. Sigma protocols from verifiable secret sharing and their applications. Cryptology ePrint Archive, Paper 2023/1388, 2023. https://eprint.iacr.org/2023/1388.

[ZDS+24] Lu Zhou, Abebe Diro, Akanksha Saini, Shahriar Kaisar, and Pham Cong Hiep. Leveraging zero knowledge proofs for blockchain-based identity sharing: A survey of advancements, challenges and opportunities. *Journal of Information Security and Applications*, 80:103678, 2024.

[Zer14] Zerocash. How zerocash works. http://zerocash-project.org/how_zerocash_works, 2014. [Online resource, accessed on the 6th of June of 2024].

[ZWMT22] Yu Zhou, Zeming Wei, Shansi Ma, and Hua Tang. Overview of zero-knowledge proof and its applications in blockchain. In Yi Sun, Liang Cai, Wei Wang, Xianhua Song, and Zeguang Lu, editors, *Blockchain Technology and Application*, pages 60–82, Singapore, 2022. Springer Nature Singapore.