
**Università di Pisa**

**Dipartimento di Matematica**

Corso di Laurea Magistrale in Matematica

# A systematic comparison of zero-knowledge proof systems across theory, implementations, and applications

*Candidato:*

Francesco Baldino

*Relatrici/Relatori:*

Prof.ssa Laura Emilia Maria Ricci

Dott. Matteo Nardelli

*Controrelatore:*

Prof. Davide Lombardo

Anno Accademico 2025-2026

## Abstract

Zero-knowledge proofs of knowledge and zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK) are a rapidly evolving field of study which is enabling breakthroughs in decentralised applications, verifiable computation, self-sovereign identity, as well as machine learning. As such, zkSNARKs are being applied in scenarios where security and privacy are the primary concern. However, with protocol definitions which span multiple fields in mathematics and theoretical computer science, it remains challenging for a developer of such applications to assess the security assumptions, efficiency trade-offs, and suitability of different protocols. Some libraries such as arkworks and snarkjs exist to simplify the workflow of developers in implementing zkSNARK protocols in various applications, but they usually treat the backend protocol as a black-box, focusing on simplicity of use for non-cryptographers rather than clarity of the underlying cryptographic assumptions.

This work addresses this gap by providing a systematic and comparative analysis of modern zero-knowledge proof systems. We present an in-depth analysis of the most prominent zkSNARKs in the current literature, with a theoretical comparison focusing on security analysis, cryptographic assumptions, and computational costs of proof size and prover/verifier time. We then analyse the most notable libraries for implementing zkSNARK protocols, highlighting the mismatch between available protocol implementations and the abstractions exposed to developers. Finally, we analyse a range of real-world applications, to identify how protocol properties influence their suitability across different scenarios. Building on this analysis, we present a developer-friendly guide to support informed selection of zero-knowledge protocols while preserving awareness of their security and performance implications.

# Contents

# Chapter 1

# Introduction

*Zero-knowledge proofs of knowledge* (ZKP), introduced by Goldwasser, Micali and Rackoff in 1985 [1], are a type of probabilistic proof between a prover party $\mathcal{P}$ and a verifier party $\mathcal{V}$ designed to prove the validity of a statement and possession of a witness of its validity, while revealing no information about the witness. The key result of zero-knowledge proof is to be able to provide both confidentiality of the witness for the prover and verifiability to the verifier, which are usually opposing qualities in security protocols. Among the different families of ZKP systems, which include proof systems for inner product arguments or range proofs such as Bulletproofs, we are interested in systems targeting the satisfiability of arbitrary arithmetic circuit, which can be used to implement proof systems for arbitrary computations. The most common type of ZKP systems developed for this purpose are *zero-knowledge Succinct Non-interactive Arguments of Knowledge* (zkSNARK) which grant two additional properties: succinctness of the proof size and verifier time (meaning sublinearity in computational complexity), and having a non-designated verifier, making it so that anyone in possession of the protocol transcript can verify the proof. For these reasons, ZKP systems and zkSNARKs are being applied in real-life scenarios where security and secrecy are the primary concern, such as payments and assets transfer in blockchains, authentication for Self-Sovereign Identity in decentralised applications, verifiable computation, and digital voting systems. As such, the usage of these systems requires critical understanding of the underlying protocols and required security assumptions.

In recent times, many libraries and *domain-specific languages* (DSL) have been developed to allow implementations of ZKP and zkSNARK systems in custom applications. These frameworks differ widely in target environment (generic purpose, on-chain contract, browser runtime), focus (efficiency vs usability), and required cryptographic knowledge, however not all of these have the required maturity, stability and documentation to be used in real applications. We focus on frameworks with a black-box use (or close to black-box use) of ZKP systems, which allow

developers not expert in cryptography to implement such systems for security-critical applications.

The considered frameworks in this work are snarkjs (JavaScript library), arkworks (ecosystem of Rust libraries), ZoKrates (Rust-like DSL) and Noir (Rust-like DSL), which cover the most common and mature solutions currently available. As for the ZKP systems, we consider the following zkSNARKs: Groth16 [2], Marlin [3], Plonk [4], Hyrax [5], and Virgo [6]. These protocols differ widely in properties such as computational costs, requirements, and security assumptions, making them suitable for different applications. We will see, however, that this wide range of properties available in the theoretical protocols given by the literature is not present in the list of protocols available in the frameworks. In particular the most prominent difference between the protocols, the type of setup required, is not well represented in the frameworks.

The main challenge in implementing such systems is making the correct choice of framework and protocol among the available ones. These solutions offer different efficiency, security, and target environment, making it critical to correctly identify the right choice of framework and protocol. Additionally, ZKPs and zkSNARKs usually span multiple fields of study, from number theory to cryptography to theoretical computer science, making it difficult for developers not specialised in cryptography to fully assess the requirements and whether a given protocol is the right fit to a given application.

To solve these issues, some works already exist in the literature comparing frameworks or protocols, to simplify the necessary assessments: Chen et al. [7] write an introduction to zkSNARK systems focused on the Groth16 protocol, exploring how it can be used for potential applications; Sheybani et al. [8] make a comprehensive comparison of the available solutions and frameworks, analysing usability, maturity, and computational results; Nainwal et al. [9] make an empirical comparison between Groth16 and zkSTARK [10]; El-Hajj and Bjorn [11] benchmark the efficiency of Groth16, zkSTARK, and Bulletproofs [12]; Baghery et al. [13] benchmark the efficiency of different trusted universal (updatable) zkSNARKs, including Sonic [14], Marlin, Plonk, LunarLite [15], and Basilisk [16]; Ernstberger et al. [17] define a novel framework for benchmarking generic zkSNARKs, and use it to compare Groth16 and Plonk. While similar works have already been proposed, all the comparisons in the literature focus on either the theoretical or the empirical aspects of the protocols and frameworks, and a unifying work taking into consideration both theoretical and practical aspects is still missing. This gap makes it difficult to make the correct assessments for all the relevant issues when choosing a solution for an application.

In this work, we aim to solve this gap in the literature by proposing a unified comparison of frameworks and solutions, as well as operative guidelines to help developers understand and choose the best fitting solution depending on the use case. We start by giving a detailed overview of ZKP systems in general, as well as the most commonly used zkSNARKs: Groth16, Marlin, Plonk, Hyrax, and Virgo. We then analyse the frameworks snarkjs, arkworks, ZoKrates, and Noir. Using these analyses, we compare theoretical and practical aspects of the different choices of protocols and frameworks, focusing on security aspects, computational efficiency, ease of usage and typical purpose.

Finally, we combine the theoretical comparison of the protocols together with the practical comparison of the libraries and DSLs to create a series of guidelines to be used for developers with little experience in cryptography and ZKP systems for a more complete understanding of the available solutions, as well as the respective trade-offs. We break down these guidelines based on a high-level view of the requirements and assumptions of the possible use cases, in order to make them applicable to a wide range of scenarios.

### Methodology

The thesis is structured as follows. In Chapter 2 we give preliminaries on notations and standard knowledge of group theory, probability and cryptography necessary to understand the work.

In Chapter 3 we give a general introduction of zero-knowledge cryptography. We start with the definition of the considered proof systems, starting from the generic zero-knowledge proofs of knowledge, to the non-interactive variant and the final succinct non-interactive form. These definitions are given for general relations, and also apply to other proof systems not considered in this work, such as ZKP systems for range proofs. We then specialise these relations to constraint systems deriving from arithmetic circuits, which can be used to represent arbitrary computations. We discuss the different and equivalent representations of these constraint systems which will be used by the proof systems mentioned above. In Chapter 4 we present common zero-knowledge protocols and subroutines which will be used, directly or with slight variations, by the proof systems. Here are defined the core protocols which make zkSNARKs possible.

In Chapters 5 and 6 we analyse Groth16, Marlin, Plonk, Hyrax, and Virgo. The analysis is divided in a chapter for trusted setup zkSNARKs and a chapter for transparent zkSNARKs. While this distinction in setup is the most evident difference between these protocols, we also analyse the respective inner workings,

to better understand the computational costs and security assumptions of each protocol. We also give a brief consideration to the Powers-of-tau protocol, which becomes relevant in implementations of some zkSNARKs which require a trusted setup procedure. In Chapter 7 we analyse the libraries snarkjs and arkworks, and the DSL ZoKrates and Noir, by comparing availability of protocols, ease of usage, and maturity of the environment.

In Chapter 8 we give a direct comparison of the considered frameworks and protocols. We start by analysing the practical implications of the current ecosystem of ZKP frameworks to the proof system choice. In particular, we highlight the discrepancy between the protocols offered by the current ecosystem, which is mostly restricted to trusted setup protocols, when compared to the available protocols defined in the literature. We then combine these practical considerations into the theoretical comparison of the protocols, comparing the relative strengths and weaknesses of the security assumptions and models, as well as the setup procedures and the computational costs. This comparison culminates in Chapter 9, where we combine the considerations of the previous chapters with a characterisation of the different requirements and assumptions of the possible scenarios for ZKP applications. We show how the properties of each protocol and choice of framework reflect on these requirements, and how this affects the choice of protocol for a given use case type. We give our conclusions and describe the open ends of this work in Chapter 10.

# Chapter 2

# Preliminaries

## 2.1 Variables and functions

We denote vectors and tuples in boldface, and components of vectors and tuples with a subscript of the index. For example, $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a tuple of elements and $x_i$ is the $i$-th component.

When writing multivariate functions, for the sake of notational simplicity we will sometimes treat them as univariate functions having tuples as inputs. In notation, if $f : X^n \to Y$ is a $n$-multivariate function, by letting $\boldsymbol{x} = (x_1, \ldots, x_n)$ we will use the following notation

$$f(\boldsymbol{x}) := f(x_1, \ldots, x_n)$$

We will use the notation $\boldsymbol{x} \,||\, \boldsymbol{y}$ to denote the concatenation of $\boldsymbol{x}$ and $\boldsymbol{y}$. When concatenating a scalar with a tuple, the scalar is to be intended as a 1-tuple, that is

$$1 \,||\, \boldsymbol{x} = (1, x_1, \ldots, x_n)$$

## 2.2 Probability

We use $\mathsf{Pr}\,[A]$ to express the probability of the event $A$.

When $\mathcal{D}$ is a probability distribution over a set $S$, we will use the notation $x \leftarrow \mathcal{D}$ to express the process of picking a random $x \in S$ following the probability distribution $\mathcal{D}$. We will use $\mathcal{U}(S)$ to represent the uniform distribution over the set $S$. We will use $x \leftarrow S$ as a shorthand for $x \leftarrow \mathcal{U}(S)$.

When $f$ is a function or an algorithm, we will use the notation $x \leftarrow f(a_1, \ldots, a_n)$ to express the assignment to $x$ of the output of $f$ when evaluated on $a_1, \ldots, a_n$.

While this notation overlaps with the notation for picking a random variable according to a distribution, one can resolve this collision issue by understanding a distribution $\mathcal{D}$ as an algorithm with output that follows the distribution $\mathcal{D}$.

We will use the notation

$$\left\{ \begin{array}{l} x_1 \leftarrow P_1 \\ x_2 \leftarrow P_2 \\ \quad \vdots \\ x_n \leftarrow P_n \end{array} : A(x_1, x_2, \ldots, x_n) \right\}$$

to express the event $A$ instantiated with the variables $x_1, \ldots, x_n$ obtained as in the (sequential top-to-bottom) process on the left. For example,

$$\left\{ \begin{array}{l} x \leftarrow \mathbb{Z}_p \\ y \leftarrow \mathbb{Z}_p \end{array} : x + y = 0 \right\}$$

expresses the event $x + y = 0$ with $x$ and $y$ being uniformly sampled in $\mathbb{Z}_p$. When writing the probability of an event defined with this notation, we will omit the braces as follows:

$$\mathsf{Pr}\left[ \begin{array}{l} x \leftarrow \mathbb{Z}_p \\ y \leftarrow \mathbb{Z}_p \end{array} : x + y = 0 \right] = \frac{1}{p}$$

## 2.3 Groups, finite fields and polynomials

In this work we use additive notation for groups unless explicitly stated. We use uppercase letters $G$ for groups and lowercase letters $g$ for group elements.

We will denote with $\mathbb{F}$ a generic finite field and with $\mathbb{F}_p$ the finite field of cardinality $p$.

When $G$ is a group of prime order $p$ generated by $g$, given $\alpha \in \mathbb{F}_p$ we use the notation $[\alpha]$ as a shorthand for $\alpha \cdot g$.

While the elements of $\mathbb{F}_p$ can be both added and multiplied, their corresponding elements through the map $\alpha \mapsto [\alpha]$ can only be added, and we will need *bilinear pairing* to circumvent this issue. Formally, let $G_1, G_2$ and $G_T$ be groups[1] of prime order $p$. A bilinear pairing on $G_1, G_2, G_T$ is a function $e : G_1 \times G_2 \to G_T$ such that

$$\forall \alpha, \beta \in \mathbb{F}_p \quad e([\alpha]_1, [\beta]_2) = \alpha\beta \cdot e([1]_1, [1]_2)$$

[1] In many works these groups are denoted with additive notation for $G_1, G_2$ and multiplicative notation for $G_T$. However, we chose to stick to additive notation for everything for sake of consistency

which is non-trivial and can be computed efficiently. Note that since $e$ is non-trivial and $G_T$ has order $p$, it follows that $e([1]_1, [1]_2)$ is not the identity element hence it is a generator of $G_T$. If we define (a posteriori of the pairing) $g_T := e([1]_1, [1]_2)$ then it trivially holds

$$e([\alpha]_1, [\beta]_2) = [\alpha\beta]_T$$

With these definitions, we can think of the computation of the bilinear pairing of two elements as a *kind of multiplication* between them.

Given a field $\mathbb{F}$, we will denote with $\mathbb{F}[x]$ the set of polynomials with coefficients in $\mathbb{F}$, with $\mathbb{F}^{\leq d}[x]$ the set of polynomials of degree at most $d$ with coefficients in $\mathbb{F}$, and with $\mathbb{F}^{<d}[x]$ the set of polynomials of degree strictly less than $d$ with coefficients in $\mathbb{F}$.

Similarly, we denote with $\mathbb{F}[x_1, \ldots, x_n]$ the set of multivariate polynomials with coefficients in $\mathbb{F}$ in the variables $x_1, \ldots, x_n$, and with $\mathbb{F}^{\leq d}[x_1, \ldots, x_n]$ the set of polynomials of degree at most $d$ with coefficients in $\mathbb{F}$ in the variables $x_1, \ldots, x_n$.

We will need to implement probabilistic checks on polynomials which follow from the following remark and its multivariate version, the Schwartz-Zippel lemma.

**Remark.** *For any polynomial $f \in \mathbb{F}_p^{\leq d}[x]$ and any subset $S \subset \mathbb{F}_p$, if $f \not\equiv 0$ then it holds*

$$\mathsf{Pr}\left[r \leftarrow S : f(r) = 0\right] \leq \frac{d}{|S|}$$

*where $r$ is sampled uniformly in $S$. Similarly, for any polynomials $f, g \in \mathbb{F}_p^{\leq d}[x]$ and any subset $S \subset \mathbb{F}_p$, if $f \neq g$ then it holds*

$$\mathsf{Pr}\left[r \leftarrow S : f(r) = g(r)\right] \leq \frac{d}{|S|}$$

This follows quickly from the fact that a polynomial $f \in \mathbb{F}^{\leq d}[x]$ has at most $d$ roots.

**Lemma 2.3.1** (Schwartz-Zippel[2])**.** *[19, 20] For any $f \in \mathbb{F}_p^{\leq d}[x_1, \ldots, x_k]$ and any subset $S \subset \mathbb{F}_p$, if $f \not\equiv 0$ then it holds*

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow S^k : f(\boldsymbol{r}) = 0\right] \leq \frac{d}{|S|}$$

*where $r_1, \ldots, r_k$ are sampled uniformly in $S$. Similarly, for any polynomials $f, g \in \mathbb{F}_p^{\leq d}[x_1, \ldots, x_k]$ and any subset $S \subset \mathbb{F}_p$, if $f \neq g$ then it holds*

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow S^k : f(\boldsymbol{r}) = g(\boldsymbol{r})\right] \leq \frac{d}{|S|}$$

[2]while this lemma is commonly known as the Schwartz-Zippel lemma, this result was independently discovered a year prior by DeMillo and Lipton [18]

We will use *multilinear extensions* of functions over $\{0,1\}^n$ in order to be able to perform polynomial checks over them. By *multilinear polynomial* we mean a multivariate polynomial which is linear in each of its variables. We will use the following definition and results.

**Definition 2.3.1** (Multilinear extension)**.** *Given $f : \{0,1\}^n \to \mathbb{F}_p$, the multilinear extension of $f$ is the unique multilinear polynomial $\widetilde{f} \in \mathbb{F}_p[x_1, \ldots, x_k]$ that verifies*

$$\forall \boldsymbol{x} \in \{0,1\}^k \quad \widetilde{f}(\boldsymbol{x}) = f(\boldsymbol{x})$$

**Remark.** *It is trivial to check that the multilinear extension operator satisfies the following properties*

$$\widetilde{f+g} = \widetilde{f} + \widetilde{g}$$
$$\widetilde{\lambda f} = \lambda \widetilde{f}$$
$$\widetilde{f} = \widetilde{g} \iff f = g$$

**Remark.** *Note that for $\boldsymbol{b}, \boldsymbol{x} \in \{0,1\}^k$, it holds*

$$(1 - x_i)(1 - b_i) + x_i b_i = \begin{cases} 1 & \text{if } x_i = b_i \\ 0 & \text{if } x_i \neq b_i \end{cases}$$

*which is a linear polynomial in $x_i$, so we can represent the indicator function $\mathbb{1}_{\boldsymbol{b}}(\boldsymbol{x})$ over $\{0,1\}^k$ with a multilinear polynomial*

$$\forall \boldsymbol{x} \in \{0,1\}^k \quad \prod_{i=1}^{k}((1 - x_i)(1 - b_i) + x_i b_i) = \mathbb{1}_{\boldsymbol{b}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} = \boldsymbol{b} \\ 0 & \text{if } \boldsymbol{x} \neq \boldsymbol{b} \end{cases}$$

*Knowing this, we can obtain the multilinear extension of a function by interpolation. Given $f : \{0,1\}^k \to \mathbb{F}_p$, its multilinear extension $\widetilde{f}$ is the polynomial*

$$\widetilde{f}(\boldsymbol{x}) := \sum_{\boldsymbol{b} \in \{0,1\}^k} \left[ f(\boldsymbol{b}) \cdot \prod_{i=1}^{k}((1 - x_i)(1 - b_i) + x_i b_i) \right]$$

*since $\forall \boldsymbol{x} \in \{0,1\}^k$ it holds*

$$\begin{aligned} \widetilde{f}(\boldsymbol{x}) &= \sum_{\boldsymbol{b} \in \{0,1\}^k} \left[ f(\boldsymbol{b}) \cdot \prod_{i=1}^{k}((1 - x_i)(1 - b_i) + x_i b_i) \right] = \\ &= \sum_{\boldsymbol{b} \in \{0,1\}^k} f(\boldsymbol{b}) \cdot \mathbb{1}_{\boldsymbol{b}}(\boldsymbol{x}) = f(\boldsymbol{x}) \end{aligned}$$

**Lemma 2.3.2** (Commutativity of multilinear extension operator with partial evaluations)**.**
*Let $f(\boldsymbol{a}, \boldsymbol{b}) : \{0,1\}^{k_1} \times \{0,1\}^{k_2} \to \mathbb{F}_p$. For any fixed value $\bar{\boldsymbol{b}} \in \{0,1\}^{k_2}$, the multilinear extension operator commutes with the partial evaluation of $f$ over $\boldsymbol{b} = \bar{\boldsymbol{b}}$, that is*

$$\forall \bar{\boldsymbol{b}} \in \{0,1\}^{k_2} \;\; \forall \boldsymbol{a} \in \mathbb{F}_p^{k_1} \qquad \widetilde{f}(\boldsymbol{a}, \bar{\boldsymbol{b}}) = \widetilde{f|_{\boldsymbol{b}=\bar{\boldsymbol{b}}}}(\boldsymbol{a})$$

*Proof.* For any fixed value of $\bar{\boldsymbol{b}} \in \{0,1\}^{k_2}$, then for all $\boldsymbol{a} \in \{0,1\}^{k_1}$ it trivially holds by definition that $\widetilde{f}(\boldsymbol{a}, \bar{\boldsymbol{b}}) = f(\boldsymbol{a}, \bar{\boldsymbol{b}})$.

Similarly, for all $\boldsymbol{a} \in \{0,1\}^{k_1}$ it also holds that

$$\widetilde{f|_{\boldsymbol{b}=\bar{\boldsymbol{b}}}}(\boldsymbol{a}) = f|_{\boldsymbol{b}=\bar{\boldsymbol{b}}}(\boldsymbol{a}) = f(\boldsymbol{a}, \bar{\boldsymbol{b}})$$

meaning that $\widetilde{f}(\boldsymbol{a}, \bar{\boldsymbol{b}})$ and $\widetilde{f|_{\boldsymbol{b}=\bar{\boldsymbol{b}}}}(\boldsymbol{a})$ are multilinear polynomials in $\boldsymbol{a}$ which agree on the hypercube $\{0,1\}^{k_1}$. The equality over $\mathbb{F}_p^{k_1}$ follows from the uniqueness of the multilinear extension. □

By applying the Schwarz-Zippel lemma to multilinear extensions we get the following useful property, called distance-amplification of the multilinear extensions. Informally, this property guarantees that functions which differ even by just one element get extended to functions which differ on a much wider range of inputs, effectively amplifying the distance between the two.

**Lemma 2.3.3** (distance-amplifying property of the multilinear extension operator)**.** *For any functions $f, g : \{0,1\}^k \to \mathbb{F}_p$, if $f \neq g$ then*

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow \mathbb{F}_p^k : \widetilde{f}(\boldsymbol{r}) = \widetilde{g}(\boldsymbol{r})\right] \leq \frac{1}{p}$$

By choosing a large $p$, the conversion to multilinear extension reduces significantly the probability of collision of the two functions, especially so in cases where $f$ and $g$ were constructed to collide on all except a few of the inputs.

We also use the following result, which shows that one can mask any random variable $X$ over a group $G$ by adding to it a uniform variable $Y$. While the proof of the following lemma is trivial (and true even in more general scenarios), this is a crucial lemma that we will use to hide values which follow non-uniform distributions (which might leak some kind of information) by the addition of some uniform random noise which effectively makes the resulting value meaningless to any kind of information retrieval, as it now follows a uniform distribution.

**Lemma 2.3.4** (masking of random variables)**.** *Let $G$ be a finite group and let $X$, $Y$ be random variables over $G$, such that $Y$ is uniformly distributed and independent from $X$, then $X + Y$ has uniform distribution*

*Proof.* We will show that

$$\forall g \in G \quad \mathsf{Pr}\left[g \leftarrow X + Y\right] = \frac{1}{|G|}$$

Indeed, for any $g \in G$ it holds

$$\begin{aligned}
\mathsf{Pr}\left[g \leftarrow X + Y\right] &= \sum_{h \in G} \mathsf{Pr}\left[h \leftarrow Y \ \wedge \ g \leftarrow X + Y\right] \\
&= \sum_{h \in G} \mathsf{Pr}\left[h \leftarrow Y \ \wedge \ g - h \leftarrow X\right] \\
&= \sum_{h \in G} \mathsf{Pr}\left[h \leftarrow Y\right] \mathsf{Pr}\left[g - h \leftarrow X\right] \\
&= \sum_{h \in G} \frac{1}{|G|} \mathsf{Pr}\left[g - h \leftarrow X\right] \\
&= \frac{1}{|G|} \sum_{h \in G} \mathsf{Pr}\left[g - h \leftarrow X\right] \\
&= \frac{1}{|G|} \sum_{h' \in G} \mathsf{Pr}\left[h' \leftarrow X\right] \\
&= \frac{1}{|G|}
\end{aligned}$$

where we used the independence of $X$ and $Y$ to split the probability, and used the fact that $h \mapsto g - h$ gives a bijection of $G$ □

## 2.4 Cryptography

We use the abbreviation *PPT* to say that an algorithm is probabilistic and runs in polynomial-time in the size of its inputs. We will use $\lambda$ to denote the security parameter. In general, any algorithm in a context where a security parameter $\lambda$ is defined implicitly receives as part of its inputs a string[3] of length $\lambda$, so that the algorithm is allowed to run at least in polynomial time with respect to $\lambda$.

Given two algorithms $\mathcal{A}$ and $\mathcal{B}$, we note with $\mathcal{A}^{\mathcal{B}}$ to express that we allow $\mathcal{A}$ to use $\mathcal{B}$ as a subroutine.

We say that a function $f(x) : \mathbb{R}_+ \to [0, 1]$ is *negligible* if it satisfies

$$\forall c \in \mathbb{N} \ \exists \bar{x} \text{ s.t. } \forall x \geq \bar{x} \quad f(x) \leq \frac{1}{x^c}$$

[3]usually denoted as $1^\lambda$ to denote the concatenation of $\lambda$ occurrences of "1"

and we will use this notion as a benchmark for the probability of events which we want to be "unlikely". Informally, a function is negligible if it vanishes faster than any inverse polynomial, such as the inverse of an exponential. We use $\mathsf{negl}(x)$ to express *some* negligible function in $x$.

We say that the probability of an event $A$ is *negligible* if it is negligible as a function of the security parameter, that is

$$\mathsf{Pr}\left[A\right] = \mathsf{negl}(\lambda)$$

where the dependence of $A$ on $\lambda$ is left implicit. We say that the probability of an event is *overwhelming* if the probability of the complementary event is negligible. Note that *negligible* and *overwhelming* probabilities, while not carrying the same certainty as respectively 0 and 1, are the most we can require from any reasonable cryptographic system. Indeed, even the most secure cryptographic protocol will be protected by some kind of secret key (which allows the honest user to participate in the protocol and prevents an adversary to do the same), and any adversary has *some* non-zero probability of correctly guessing the secret key.

We now define what we intend when we say that two distributions $\mathcal{D}_0, \mathcal{D}_1$ are *indistinguishable*. Let $\mathcal{A}$ be a PPT challenger, that is a probabilistic polynomial-time algorithm which takes in input a value $x$ which was either obtained by distribution $\mathcal{D}_0$ or distribution $\mathcal{D}_1$ and has to guess from which distribution it was obtained, by outputting $i = 0$ or $i = 1$ expressing that it claims it was produced by $\mathcal{D}_i$. Then, we say that two distributions $\mathcal{D}_0, \mathcal{D}_1$ are indistinguishable if for any PPT challenger $\mathcal{A}$, the challenger is unable to distinguish the distributions, that is

$$\mathsf{Pr}\left[x \leftarrow \mathcal{D}_0 : 0 \leftarrow \mathcal{A}(x)\right] = \mathsf{Pr}\left[x \leftarrow \mathcal{D}_1 : 0 \leftarrow \mathcal{A}(x)\right]$$

A concept which we will use throughout the work is that of *interactive machines*. This definition is possibly redundant, as the concept of interactive machines is often clear enough to be left implicit, but we make it explicit to fix the notation which we will use.

**Definition 2.4.1** (Interactive machines)**.** *An interactive machine is an algorithm $\mathcal{A}$ which takes as input some initial value $x$ (potentially a tuple of values) and a list of incoming messages $(m_1, \ldots, m_k)$ belonging to an message space (left implicit), and outputs a new message $m_{k+1} \leftarrow \mathcal{A}(x; m_1, \ldots, m_k)$.*

*When running a pair of interactive machines $\mathcal{A}, \mathcal{B}$ on initial inputs $x, y$, we let $\mathcal{A}$ generate the first message*

$$a_0 \leftarrow \mathcal{A}(x)$$

*and then let $\mathcal{A}$ and $\mathcal{B}$ exchange messages sequentially*

$$a_{i+1} \leftarrow \mathcal{A}(x;\ a_0,\ b_0,\ \ldots,\ a_i,\ b_i)$$
$$b_{i+1} \leftarrow \mathcal{B}(y;\ a_0,\ b_0,\ \ldots,\ a_i,\ b_i,\ a_{i+1})$$

*Implicitly, we assume that the messages alphabet includes a special "terminating" symbol $\perp$ which terminates the interaction.*

*We note with*

$$\mathsf{output} \leftarrow \langle \mathcal{A}(x), \mathcal{B}(y) \rangle$$

*the process of running the interactive pair $\langle \mathcal{A}, \mathcal{B} \rangle$ on respective inputs $x, y$ until it terminates with last message* output*. When the interactive pair has a designated terminator ($\mathcal{A}$ or $\mathcal{B}$), we express this by saying that $\langle \mathcal{A}, \mathcal{B} \rangle$ terminates by $\mathcal{A}$ (resp. $\mathcal{B}$) outputting* output*.*

Informally, by interactive machines we mean a pair of stateful parties which exchange messages one after the other following some kind of protocol.

Our goal is to study protocols which can verify the validity of statements such as "the property $P(\mathbb{x})$ is true" for some defined property $P$, where $\mathbb{x}$ is called the input, or sometimes the statement itself. To do this, we take inspiration from NP relations, which bind a statement $\mathbb{x}$ to a witness $\mathbb{w}$ which allows to efficiently verify the validity of the statement.

**Definition 2.4.2** (Relations)**.** *A relation $\mathcal{R}$ over values $\mathbb{X}$ and witnesses $\mathbb{W}$ is a subset $\mathcal{R} \subset \mathbb{X} \times \mathbb{W}$ of values $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ together with a verifying boolean function*

$$i_{\mathcal{R}} : \mathbb{X} \times \mathbb{W} \to \{0, 1\}$$

*such that $i_{\mathcal{R}}(\mathbb{x}, \mathbb{w}) = 0 \iff (\mathbb{x}, \mathbb{w}) \in \mathcal{R}$.*

We will sometimes slightly abuse the notation, leaving implicit the function $i_{\mathcal{R}}$, and write $\mathcal{R}(\mathbb{x}, \mathbb{w}) = 0$ to say $i_{\mathcal{R}}(\mathbb{x}, \mathbb{w}) = 0$.

One should imagine a relation $\mathcal{R}$ as to be describing the validity of a certain property $P$ over some set $\mathbb{X}$, meaning that for any $\mathbb{x}$ such that $P(\mathbb{x})$ is true, there should exist a valid witness $\mathbb{w}$ which testifies that $P(\mathbb{x})$ is true, for example an encoding of a proof of $P(\mathbb{x})$. However, this intuition might become circular reasoning as we will mostly consider properties *defined* by some function which in practice plays the role of $i_{\mathcal{R}}$, such as

$$P(\mathbb{x}) := \text{"there exists an auxiliary value } \mathbb{w} \text{ such that } f(\mathbb{x}, \mathbb{w}) = 0\text{"}$$

for some fixed function $f$.

We will sometimes write $\mathbb{x} \in \mathcal{R}$ to say that there exists a witness $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, saying that $\mathbb{x}$ is "valid". Similarly, we will write $\mathbb{x} \notin \mathcal{R}$ to say that there exists no witness $\mathbb{w}$ for $\mathbb{x}$, and we say that $\mathbb{x}$ is "not valid". Note that for a given value $\mathbb{x}$, there might be multiple witnesses $\mathbb{w}_1, \mathbb{w}_2$ such that $(\mathbb{x}, \mathbb{w}_1) \in \mathcal{R}$ and $(\mathbb{x}, \mathbb{w}_2) \in \mathcal{R}$.

In this context, an NP relation is a relation where the verifying function $i_{\mathcal{R}}$ runs in polynomial time of its inputs. For example, we might express the property of "not being a prime number" as an NP relation, by considering an algorithm which takes as input a number $\mathbb{x}$ and a claimed factorization $\mathbb{w} = (p_1, \ldots, p_k)$ of $\mathbb{x}$, and checks whether the factorization $\mathbb{w}$ multiplies to $\mathbb{x}$. Then, given a non-prime number $\mathbb{x}$ we can have as a witness its factorization $\mathbb{w} = (p_1, \ldots, p_k)$, which the verifying algorithm accepts, so we consider it as "valid". On the other side, given a prime number $\mathbb{x}'$, there exist no valid factorization for $\mathbb{x}'$ so it is "not valid".

In this work we will mostly consider relations which describe the satisfiability of some constraint systems. A constraint system is a function $f$ over some values $(a_1, \ldots, a_n)$,[4] and we say that the constraint system is satisfied if $f(a_1, \ldots, a_n) = 0$. Then, given a partial input $\mathbb{x} := (a_1, \ldots, a_k)$, we can define the property

$$P(\mathbb{x}) := \text{“}\exists\, \mathbb{w} = (a_{k+1}, \ldots, a_n) \quad \text{s.t.} \quad f(\mathbb{x}||\mathbb{w}) = 0\text{”}$$

which gives the relation

$$\mathcal{R} := \{(\mathbb{x}, \mathbb{w}) \mid f(\mathbb{x}||\mathbb{w}) = 0\}$$

where the valid values $\mathbb{x}$ are the values for which the constraint system can be satisfied.

Ideally, when considering relations we would also want for it to be hard to decide whether a given value $\mathbb{x}$ is valid without having access to any witness. However, this is not required by the definition.

We list some standard families of cryptographic tools which we will need throughout the work

**Definition 2.4.3** (Commitment scheme)**.** *A commitment scheme is a protocol composed of two algorithms* Commit *and* Open *with the following interface:*

$\mathsf{com}_a, \mathsf{open} \leftarrow \mathsf{Commit}(a)$ *: creates a commitment* $\mathsf{com}_a$ *for the message* $a$*, together with any information* open *necessary for the opening step*

[4] We leave implicit the codomain of $f$ as it varies depending on the context.

$\mathsf{accept/reject} \leftarrow \mathsf{Verify}(a, \mathsf{com}, \mathsf{open})$ *: checks whether the claimed commitment* $\mathsf{com}$ *really corresponds to the message* $a$*.*

*which satisfy the* ***correctness*** *property, that is*

$$\forall a \quad \mathsf{Pr}\left[\mathsf{com}_a, \mathsf{open} \leftarrow \mathsf{Commit}(a) : \mathsf{accept} \leftarrow \mathsf{Verify}(a, \mathsf{com}_a, \mathsf{open})\right] = 1$$

*where* $\lambda$ *is the security parameter. Additionally, can satisfy the following properties*

**Hiding** *For any two messages* $a \neq b$*, the distributions of* $\mathsf{Commit}(a)$ *and* $\mathsf{Commit}(b)$ *are indistinguishable by any PPT adversary*

**Binding** *It must be hard to open* $\mathsf{com}_a$ *to some message* $b \neq a$*, meaning that for any PPT adversary* $\mathcal{A}$ *it holds*

$$\mathsf{Pr}\left[\begin{array}{c} \mathsf{com}, a, b, \\ \mathsf{open}_a, \mathsf{open}_b \end{array} \leftarrow \mathcal{A}() : \begin{array}{c} a \neq b \\ \mathsf{accept} \leftarrow \mathsf{Verify}(a, \mathsf{com}, \mathsf{open}_a) \\ \mathsf{accept} \leftarrow \mathsf{Verify}(b, \mathsf{com}, \mathsf{open}_b) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

*The commitment scheme might additionally require a* $\mathsf{Setup}$ *algorithm which generates public parameters* $\mathsf{pp}$ *given as input to both* $\mathsf{Commit}$ *and* $\mathsf{Verify}$

We will use (variants of) commitment schemes in scenarios where, for example, we require a party to successfully respond to a second party's challenge: by committing the terms of the challenge before the challenge is issued, the party tasked to respond cannot "change cards" which could make it trivial to respond successfully to the challenge.

Note that the hiding property *requires* the $\mathsf{Commit}$ algorithm to be non-deterministic.

Finally, we define some standard assumptions and models which will be used in the security proofs

**Definition 2.4.4** (Discrete Logarithm assumption)**.** *For any security level* $\lambda$*, the discrete logarithm assumption (DL) states that there are groups* $G$ *of prime order* $p$ *with generator* $g$ *such that it is hard to compute the discrete logarithm in* $G$*, meaning that for any PPT adversary* $\mathcal{A}$ *and for any* $\tau \in \mathbb{F}_p$*, the adversary* $\mathcal{A}$ *given access to* $[1] = g$ *and* $[\tau] = \tau \cdot g$ *cannot find* $\tau$ *except that with negligible probability*

$$\mathsf{Pr}\left[\tau \leftarrow \mathcal{A}([1], [\tau])\right] \leq \mathsf{negl}(\lambda)$$

We note that elliptic curves are usually great candidates for groups on which the discrete logarithm is indeed hard, and there exist multiple standardized elliptic curve groups to choose from when implementing cryptographic applications.

Under the discrete logarithm assumption, we can think of the map $\alpha \mapsto [\alpha]$ as a sort of homomorphic commitment scheme. Indeed, it is binding (as for a cyclic group of order $p$, given a commitment $h \in G$ there exists exactly one $\alpha \in \mathbb{F}_p^*$ that can open to h) and while not hiding (since it is deterministic) it is at least computationally hard to invert under the DL assumption. Additionally, it is homomorphic in the sense that the commitment commutes with addition, that is

$$\forall \alpha, \beta \in \mathbb{F}_p \quad [\alpha + \beta] = [\alpha] + [\beta]$$

$$\forall \alpha, \beta, \gamma \in \mathbb{F}_p \quad [\gamma] = [\alpha] + [\beta] \iff \gamma = \alpha + \beta$$

Note that by using a bilinear pairing $e : G_1 \times G_2 \to G_T$ for groups $G_1, G_2, G_3$ of prime order $p$, we somewhat get an additional homomorphism property relative to the multiplication in $\mathbb{F}_p$. It is not true that the map $\alpha \mapsto [\alpha]_i$ commutes with multiplication as there is no meaning to the product of the two group elements $[\alpha]_1$ and $[\beta]_1$, but we can use a pairing to work around this limitation. Indeed, if $g_1 = [1]_1$ and $g_2 = [1]_2$ are generators of $G_1$ and $G_2$ respectively, then it holds

$$\forall \alpha, \beta, \gamma, \delta \in \mathbb{F}_p \quad e([\alpha]_1, [\beta]_2) = e([\gamma]_1, [\delta]_2) \iff \alpha\beta = \gamma\delta$$

which gives us *some* way to check identities on products of elements of $\mathbb{F}_p$ through the committed values.

An extremely common type of commitment scheme which holds under the Discrete Logarithm assumption is the Pedersen commitment [21].

**Definition 2.4.5** (Pedersen commitment)**.** *Let $G$ be a cyclic group of order $p$ such that the Discrete Logarithm assumption holds. Let $g, h \in G$ be two distinct generators of $G$. The Pedersen commitment scheme for messages in $\mathbb{F}_p$ is defined as follows*

$\mathsf{com}_a, \mathsf{open} \leftarrow \mathsf{Commit}(a)$ *:*

1. *sample* $s \leftarrow \mathbb{F}_p$
2. *return* $\mathsf{com}_a := a \cdot g \ + \ s \cdot h, \ \mathsf{open} := s$

$\mathsf{accept/reject} \leftarrow \mathsf{Verify}(a, \mathsf{com}, \mathsf{open})$ *:*

1. *if* $a \cdot g \ + \ \mathsf{open} \cdot h \stackrel{?}{=} \mathsf{com}$
2. *return* accept
3. *else*
4. *return* reject

The proof of hiding and binding of this protocol is simple, and we give an intuition: for the hiding property, $s \cdot h$ has uniform distribution over $G$ meaning that using Lemma 2.3.4 the output of $\mathsf{Commit}$ also has uniform distribution over $G$ independently of the committed value; for the binding property, if an adversary could open to $\mathsf{com}, a, b, s_a, s_b$ such that the verify checks pass, then $\frac{a-b}{s_b-s_a}$ is the discrete logarithm of $h$ in base $g$ meaning the adversary can win at the discrete logarithm which by assumption happens with negligible probability.

Note that while the Pedersen commitment is not exactly homomorphic in the same sense as the simple map $\alpha \mapsto [\alpha]$, it has an homomorphic property. If $\mathsf{com}_a, \mathsf{com}_b, \mathsf{com}_c$ where honestly computed and the values satisfy $a + b = c$, then the committer can prove that they correspond to values which satisfy $a + b = c$ without opening any commitment. If $s_a, s_b$ and $s_c$ are the random salts sampled for the respective commitments, then the committer can send $s_c - (s_a + s_b)$ and the receiver can check that

$$\mathsf{com}_c \stackrel{?}{=} \mathsf{com}_a + \mathsf{com}_b + (s_c - (s_a + s_b)) \cdot h$$

which should convince the receiver that the values behind the commitment do indeed satisfy the claimed relation (or otherwise the committer was able to solve the discrete logarithm as in the proof of the binding property).

We also give a batched version of the discrete logarithm assumption

**Definition 2.4.6** ($q$-DLOG)**.** *For any integer $q$ and any security level $\lambda$, the $q$-DLOG assumption states that there are groups $G_1, G_2$ of prime order $p$ with respective generators $g_1, g_2$ such that, for any PPT adversary $\mathcal{A}$ and any exponent $\tau \in \mathbb{F}_p^*$, the adversary $\mathcal{A}$ given access to*

$$[1]_1 = g_1, [\tau]_1, \ldots, [\tau^q]_1,\ [1]_2 = g_2, [\tau]_2, \ldots [\tau^q]_2$$

*cannot find $\tau$ except that with negligible probability*

$$\mathsf{Pr}\left[\tau \leftarrow \mathcal{A}([1]_1, [\tau]_1, \ldots, [\tau^q]_1,\ [1]_2, [\tau]_2, \ldots [\tau^q]_2)\right] \leq \mathsf{negl}(\lambda)$$

**Definition 2.4.7** ($q$-Strong Bilinear Diffie-Hellman [22])**.** *For any integer $q$ and any security level $\lambda$, the $q$-Strong Bilinear Diffie-Hellman ($q$-SBDH) assumption states that there is a group $G$ of prime order $p$ with generator $g$ and a pairing $e : G \times G \to G_T$ for some group $G_T$ such that for any PPT adversary $\mathcal{A}$ and any exponent $\tau \in \mathbb{F}_p^*$, the adversary $\mathcal{A}$ given access to $[1], [\tau], \ldots, [\tau^q]$ cannot compute $\left[\frac{1}{\tau-u}\right]_T := \frac{1}{\tau-u} \cdot e(g, g)$ for any $u \in \mathbb{F}_p \setminus \{\tau\}$.*

$$\mathsf{Pr}\left[u,\ \left[\frac{1}{\tau - u}\right]_T \leftarrow \mathcal{A}([1], [\tau], \ldots, [\tau^q])\right] \leq \mathsf{negl}(\lambda)$$

Notice that the assumptios we just defined are in increasing order of strength, that is

$$q\text{-SBDH} \implies q\text{-DLOG} \implies \text{DL}$$

meaning that it might be the case that the DL assumption holds while the $q$-SBDH does not. For this reason, a *stronger* assumption is worse security-wise, while a *weaker* assumption is better. At the same time, it is easier to prove the security of a protocol using a stronger assumption than it is with a weaker assumption.

As for the models, first we note that we suppose to work in the *Structured Reference String* model (SRS), that is we suppose that one can generate a string $\mathsf{srs}$ that is structured, which in our case means composed of encodings of group elements, which each party of the described protocols has access to[5]. The $\mathsf{srs}$ will usually be the public parameters generated in the setup step.

Some protocols will require the *Random Oracle Model* (ROM), which assumes that hash functions behave like random (but deterministic) functions. This is an extremely widespread standard model for cryptographic protocols and has been well tested.

Additionally, some of the security proofs will require the *Algebraic Group Model* (AGM) [23], which restricts the ability of any adversary of producing arbitrary group elements.

**Definition 2.4.8** (Algebraic Group Model)**.** *Let $G_1, G_2$ be groups, and let $\mathsf{srs} = (\mathsf{srs}_1, \mathsf{srs}_2)$ where $\mathsf{srs}_i$ is composed of elements of the group $G_i$. In the Algebraic Group Model (AGM), any adversary $\mathcal{A}$ which outputs an element $h$ of the group $G_i$ must also output a vector $\boldsymbol{v}$ such that*

$$h = \langle \boldsymbol{v}, \mathsf{srs}_i \rangle = \sum_j v_j \cdot \mathsf{srs}_{i,j}$$

The sense of the AGM is that we restrict the security analysis by considering only *algebraic* adversaries, that is adversaries able to generate group elements only by a linear combination of the elements they are given (notice that the $\mathsf{srs}$ will usually include $[1]_1$ and $[1]_2$ so these combination can be thought of affine combinations of the other elements, which also allows the adversary to potentially produce arbitrary elements based on the generator). This is weaker (hence better) assumption than the *Generic Group Model* which treats the groups as black-boxes which the adversary cannot inspect and only use to query evaluations of the group operation.

[5]This means, for example, that we are ignoring potential man-in-the-middle attacks as they can obviated with other well-studied methods.

# Chapter 3

# Introduction to Zero-knowledge proofs

The first goal of this work is to analyse and compare zero-knowledge proof systems. Zero-knowledge proofs (ZKP) are a type of probabilistic proofs, which are a proof system where the "proof" should not be interpreted in the mathematical sense, rather an argument or a response to a challenge that should be hard to pass if the claimed statement was false. This makes a probabilistic proof convincing up to a certain probability, which can be hopefully made be arbitrarily close to 1.

The type of statements which we will consider for the proofs are statement of satisfiability of constraint system. A constraint system over variables $z_1, \ldots, z_n$, in its most generic form, is some function $f(z_1, \ldots, z_n)$, and we say that the constraint system is satisfied if it holds

$$f(z_1, \ldots, z_n) = 0$$

and we express the constraint system as

$$f(z_1, \ldots, z_n) \stackrel{?}{=} 0$$

Constraint systems are extremely powerful tools which can be used to express any kind of computation, and we will focus on constraint systems deriving from arithmetic circuits, or systems which are equivalent to these.

## 3.1 Proof systems

### 3.1.1 Zero-knowledge proofs

We start by giving the definition of interactive proofs.

**Definition 3.1.1** (Interactive proof)**.** *Let $\mathcal{R}$ be a relation and let $\lambda$ be a security parameter. An interactive proof for $\mathcal{R}$ (with security parameter $\lambda$) is a pair of interactive machines composed of a prover $\mathcal{P}$ and a PPT verifier $\mathcal{V}$, where the interaction terminates by $\mathcal{V}$ returning either* accept *or* reject*, which satisfy the following properties:*

**Correctness** *For any valid value $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, $\mathcal{V}$ always accepts the validity of $\mathbb{x}$, that is*

$$\forall (\mathbb{x}, \mathbb{w}) \in \mathcal{R} \quad \mathsf{Pr}\left[\mathsf{accept} \leftarrow \langle \mathcal{P}(\mathbb{x}, \mathbb{w}), \mathcal{V}(\mathbb{x}) \rangle\right] = 1 \tag{3.1}$$

**Soundness** *For any adversary $\mathcal{P}^*$, and invalid value $\mathbb{x} \notin \mathcal{R}$, the verifier $\mathcal{V}$ rejects with overwhelming probability*

$$\forall \mathcal{P}^*,\ \forall \mathbb{x} \notin \mathcal{R} \quad \mathsf{Pr}\left[\mathsf{accept} \leftarrow \langle \mathcal{P}^*(\mathbb{x}), \mathcal{V}(\mathbb{x}) \rangle\right] \leq \mathsf{negl}(\lambda) \tag{3.2}$$

*The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ may additionally take as input some parameters generated by a setup algorithm.*

The intuition behind this concept is that we have to imagine the prover making some initial claim, to which the verifier sends one or multiple challenges to which the prover should be able to reply successfully only if $\mathbb{x} \in \mathcal{R}$. The fact that the prover was able to pass the challenges convinces the verifier that $\mathbb{x} \in \mathcal{R}$, up to the soundness error.

Note that we do not assume the prover $\mathcal{P}$ to be computationally constrained. Additionally, some authors discriminate further and use the term "interactive proof" when referring to protocols in which Soundness holds even for unconstrained adversaries, while using the term "interactive argument" when Soundness holds only for polynomial-time adversaries. However, this distinction is not exactly a standard, and we will not use it.

While interactive proofs already offer interesting properties, we will focus on a specific type of interactive proofs, namely zero-knowledge proofs of knowledge (ZKP). The goal of this type of proof is to provide two guarantees:

- On the prover side, $\mathcal{P}$ might want to convince $\mathcal{V}$ of the validity of $\mathbb{x}$ while revealing no information about $\mathbb{w}$. To achieve this, we want the protocol to satisfy an additional property which shows that the interaction leaks no information about $\mathbb{w}$.
- On the verifier side, the soundness property only guarantees (with high probability) that a valid witness $\mathbb{w}$ for the value $\mathbb{x}$ exists, while making no claim on whether the prover actually knows the valid $\mathbb{w}$. We want an additional property which guarantees that a prover which can convince an honest verifier must necessarily "know" a valid witness.

This goal is clearly complicated, as the concepts of "leaking information" and "knowing information" are not well defined. We now give an intuition on how we can reflect these concepts into mathematically definable properties.

One way to express that the transcript of the interaction does not leak any kind of information on the witness $\mathbb{w}$ is to say that the transcript is completely independent from the witness $\mathbb{w}$. To express this property, we imagine a simulator which, without knowledge of $\mathbb{w}$, produces outputs that look like a transcript of a real execution. If the simulator is able to produce an output distribution that is indistinguishable from the distribution of transcripts of real executions, then for a given adversary, obtaining real transcripts is equivalent to obtaining transcripts from the simulator distribution, and the latter cannot possibly leak any information on $\mathbb{w}$ since they were created without having $\mathbb{w}$ as input.

As stated right now, the simulator would prove no leakage only against eavesdropper adversaries which obtain transcripts of honest executions. While this is certainly an improvement, we would like to be able to prove that the protocol leaks no information even when the adversary is a malicious or even *honest-but-curious* verifier taking part in the execution. To obtain this, we should make the simulator produce a distribution of transcripts indistinguishable from the transcripts of executions with the adversary verifier, for which we need to give the simulator access to the specific adversary, for example to be able to at least reproduce its distribution of challenges.

As for the knowledge aspect, we need to define what it means to "know" something. For starters, we can say with confidence that if a party (honest prover or adversary) recieves some value $\mathbb{z}$ as input, then it "knows" $\mathbb{z}$. Additionally, while it might be hard to decide whether a party knows or not some value $\mathbb{z}'$ not given as part of its inputs, it is easier to express that the given party has *the means to know* $\mathbb{z}'$ if they wanted to: for example, if there exist a publicly known efficient algorithm to obtain $\mathbb{z}'$ from $\mathbb{z}$, then any party that knows $\mathbb{z}$ from its inputs than has the means to also know the value $\mathbb{z}'$.

This is the approach used to define the property of "knowing" a valid witness: suppose we have a party that, without receiving the witness for a value $\mathbb{x}$ as part of its inputs, is able to convince a verifier of the validity of $\mathbb{x}$; if we are able to exhibit an algorithm that can produce a valid witness from the behaviour of said party (basically extracting the witness from it), then said party would have the means to know a valid witness. In this sense, that party "knows" a valid witness.

We formalize these concepts in the following definitions.

**Definition 3.1.2** (Zero-knowledge proof, proof of knowledge)**.** *We say that an interactive proof is a zero-knowledge proof if it satisfies the following property:*

**Zero-knowledge** *For any adversary $\mathcal{V}^*$ there exists a PPT algorithm $\mathcal{S}$ called simulator such that for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, $\mathcal{S}^{\mathcal{V}^*}$ produces an output distribution which is indistinguishable from the distribution of the transcript of $\langle \mathcal{P}(\mathbb{x}, \mathbb{w}), \mathcal{V}^*(\mathbb{x}) \rangle$*

*We say that an interactive proof is a proof of knowledge if it satisfies a stronger version of the soundness property, namely:*

**Knowledge-soundness** *For any adversary $\mathcal{P}^*$ there exists a PPT algorithm $\mathcal{E}$ called extractor such that for any value $\mathbb{x}$, either $\mathcal{P}^*$ fails to fool the verifier $\mathcal{V}$ or the extractor is able to extract a valid witness for $\mathbb{x}$, except that with negligible probability.*

$$\mathsf{Pr}\left[\mathbb{w} \leftarrow \mathcal{E}^{\mathcal{P}^*}(x) : \begin{array}{c}\mathsf{accept} \leftarrow \langle \mathcal{P}^*(\mathbb{x}), \mathcal{V}(\mathbb{x}) \rangle \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}\end{array}\right] \leq \mathsf{negl}(\lambda) \tag{3.3}$$

*Finally, we say that an interactive proof is a zero-knowledge proof of knowledge if it satisfies both zero-knowledge and knowledge-soundness.*

Zero-knowledge proofs of knowledge are extremely powerful tools for in contexts where privacy is crucial. For example, consider a situation where a user wants to convince a provider of their identity. The naive way would be to have the user send their credentials to the provider, however this would put the user at risk in case the provider turned out to be malicious (or the connection was insecure). A better implementation would be to use a zero-knowledge proof of knowledge protocol where the witness $\mathbb{w}$ is composed of the credentials of the user: in this case, the user has the guarantee that even when interacting when malicious providers no information is leaked from their credentials, and the provider has the guarantee that (with overwhelming probability) any user which can convince knowing the credentials really does know the credentials.

### 3.1.2 Non-interactive proofs

One of the main restrictions of ZKPs in the current definition is that the verifier is *designated*, in the sense that only the verifier which actually took part in the execution of the protocol can verify the proof given by the prover. More precisely, the reason why the verifier is convinced at the end of the protocol is that the prover was able to successfully respond to its randomly generated challenges, while to any third-party the challenges of the verifier might as well be maliciously crafted, so the inspection of the transcript without any interaction has no reason

to convince the third-party of anything.

To solve this issue we define a new feature which a given ZKP system can satisfy: we would like the protocol to be *non-interactive*, in the sense that messages should only flow from $\mathcal{P}$ to $\mathcal{V}$ and not both ways. This way, any party receiving the messages of the prover can play the role of the verifier, and be convinced of the prover's claim.

We give the following definition for a non-interactive zero-knowledge proof of knowledge, which are sometimes referred to as NIZK proofs in the literature.

**Definition 3.1.3** (Non-interactive zero-knowledge proof of knowledge, NIZK)**.** *Let $\mathcal{R}$ be a relation and let $\lambda$ be a security parameter. A non-interactive zero-knowledge proof of knowledge is an interactive protocol composed of the following interface:*

$\mathsf{pp} \leftarrow \mathsf{Setup}()$ *: a setup function generating public parameters* $\mathsf{pp}$*.*

$\pi \leftarrow \mathsf{Prove}(\mathsf{p}, \mathbb{x}, \mathbb{w})$ *: the algorithm run by the prover to produce a proof $\pi$ sent to the verifier*

$\mathsf{accept}/\mathsf{reject} \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi)$ *: the check run by the verifier to test the validity of the proof*

*which satisfy correctness, zero-knowledge and knowledge-soundness defined as follows:*

**Correctness** *For any valid value $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$,* Verify *always accepts a correctly computed proof, that is*

$$\forall (\mathbb{x}, \mathbb{w}) \in \mathcal{R} \;\; \mathsf{Pr}\left[\begin{matrix}\mathsf{pp} \leftarrow \mathsf{Setup}() \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbb{x}, \mathbb{w})\end{matrix} : \mathsf{accept} \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi)\right] = 1 \tag{3.4}$$

**Zero-knowledge** *There exists a PPT algorithm* Sim *called simulator such that for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$,* Sim *produces an output distribution which is indistinguishable from the distribution of the proofs honestly generated by* Prove*, that is for any challenger $\mathcal{A}$ which tries to distinguish between proofs generated by* Sim *and proofs generated by* Prove*, and for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ it holds*

$$\begin{aligned}&\mathsf{Pr}\left[\mathsf{pp}, \pi \leftarrow \mathsf{Sim}(\mathbb{x}) : 0 \leftarrow \mathcal{A}(\mathsf{pp}, \mathbb{x}, \pi)\right] \\ &\quad = \mathsf{Pr}\left[\begin{matrix}\mathsf{pp} \leftarrow \mathsf{Setup}() \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbb{x}, \mathbb{w})\end{matrix} : 0 \leftarrow \mathcal{A}(\mathsf{pp}, \mathbb{x}, \pi)\right]\end{aligned} \tag{3.5}$$

**Knowledge-soundness** *For any adversary $\mathcal{P}^*$ there exists a PPT algorithm $\mathcal{E}$ called extractor such that for any value $\mathbb{x}$, either $\mathcal{P}^*$ fails to pass the check* Verify *or the extractor is able to extract a valid witness for $\mathbb{x}$, except that with negligible probability.*

$$\mathsf{Pr}\left[\begin{array}{c}\mathsf{pp} \leftarrow \mathsf{Setup}() \\ \pi \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathbb{x}) \\ \mathbb{w} \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathsf{pp}, \mathbb{x})\end{array} : \begin{array}{c}\mathsf{accept} \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi) \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}\end{array}\right] \leq \mathsf{negl}(\lambda) \tag{3.6}$$

At this point one might rightfully wonder why should a non-interactive proof convince a verifier of anything, since we argued that the convincing part of an interactive proof is the ability to successfully respond to a challenge issued by the verifier. The idea is that the prover is now responding to a challenge prescribed by the Setup step: as an example which will make more sense later, the prover might have to evaluate some polynomials at a *random* point chosen in the Setup step. This approach, however, is open to trust issues: the verifier will find the proof convincing only if they can trust the source of the challenge. Notice that while we have been explicit about definition of the properties of non-interactive proofs, as well as how these proofs should be created and verified, we have not yet discussed by who and how should the Setup step be run. Indeed, different protocols have different requirements on the Setup procedure, for example requiring that it should be run by a trusted third party, or allowing the setup to be run by any party. There are three main types of setup procedures in the current landscape of non-interactive proofs: trusted setup, universal trusted setup and transparent setup.

**Definition 3.1.4** (Trusted, Universal, Transparent)**.** *A* Setup *procedure is trusted if it requires to be run by a trusted third party. A* Setup *procedure is transparent if it can be run by any party. Additionally, a trusted setup can be universal, if it can be divided in two sub-procedures $\mathsf{S}_1$ and $\mathsf{S}_2$ such that*

$$\mathsf{pp}_1 \leftarrow \mathsf{S}_1() \qquad \mathsf{pp} \leftarrow \mathsf{S}_2(\mathcal{R}, \mathsf{pp}_1)$$

*where the first sub-procedure $\mathsf{S}_1$ is the only part which requires a trusted third party and is not relation-specific, while the second sub-procedure specializes the public parameters to the given relation and is not trusted, meaning that it can be run by any party.*

Universal trusted setups are also sometimes called *updatable* trusted setups.

### 3.1.3 Succinctness and zkSNARKs

An important issue for non-interactive zero-knowledge proofs of knowledge is efficiency. Indeed, it is easy to define protocols which are correct and sound in

theory, but computationally prohibitive in practice. As a baseline for feasibility of a protocol we require that the size of the proof and the running time of the verifier should be limited, at least sublinear in the size of the given statement, and we call this property *succinctness*.

Before giving the formal definition of succinctness, we introduce a small abuse of notation by denoting the "size" of the relation with $|\mathcal{R}|$ whenever it makes sense. For example: if $\mathcal{R}$ is defined by a circuit $\mathcal{C}$ (as discussed more in depth in the following section), we denote by $|\mathcal{R}|$ the size of the circuit $|\mathcal{C}|$; if $\mathcal{R}$ is a generic NP relation, we can think of $|\mathcal{R}|$ as the running time of the verification algorithm for the relation, which is itself polynomial in $|\mathbb{x}|$ and $|\mathbb{w}|$.

**Definition 3.1.5** (zero-knowledge Succinct Non-interactive Argument of Knowledge, zkSNARK)**.** *Let $\mathcal{R}$ be a relation and let $\lambda$ be a security parameter. A non-interactive zero-knowledge proof of knowledge composed of algorithms:*

$\mathsf{pp} \leftarrow \mathsf{Setup}()$

$\pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbb{x}, \mathbb{w})$

$\mathsf{accept/reject} \leftarrow \mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi)$

*is a zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) if the size of the proof $\pi$ and the running time of* Verify *are sublinear*[6] *in the size of the input $|\mathbb{x}|$, in the size of the witness $|\mathbb{w}|$, and in $|\mathcal{R}|$.*

We note that the requirements for zero-knowledge, knowledge-soundness and succinctness are independent from each other, and for this reason some literature focuses only on SNARKs which do not satisfy necessarily the zero-knowledge property.

## 3.2 Circuits and constraint systems

We have defined zero-knowledge proofs of knowledge and zkSNARKs for generic relations. In order to apply these constructions to concrete problems, we need to define the concrete problems into relations. Our main focus will be on *arithmetic circuits*, however many zkSNARKs work on equivalent constraints, namely Rank-1 Constraint Systems (R1CS) and Quadratic Arithmetic Programming (QAP).

[6]Some authors specify different requirements for succinctness, with common choices being constant size, polylogarithmic size and sublinear size. We consider a sublinear requirement and discuss later on the results of different protocols

### 3.2.1 Arithmetic Circuits

An arithmetic circuit is a way to formalize some kind of arithmetic computation. In general, a circuit $\mathcal{C}$ is a directed acyclic graphs meant to represent the computation. In this context, we call *wire* any edge of the graph, *input nodes* the source vertices of the graph, *output nodes* the sink vertices and *gate* any other vertex.

When talking about a gate, the fan-in number is the number of incoming wires that it expects, and the fan-out number is the number of outgoing wires from the gate. We say that the size of a circuit, denoted by $|\mathcal{C}|$, is the number of total nodes in the circuit.

**Definition 3.2.1** (Arithmetic circuit)**.** *An arithmetic circuit over a field $\mathbb{F}$ is a circuit $\mathcal{C}$ where the inputs and outputs are elements of $\mathbb{F}$ and the gates are addition or multiplication gates over $\mathbb{F}$ with a fan-in of 2 and unbounded fan-out.*

If one thinks an arithmetic circuit with $k$ inputs and $m$ outputs as a function from $\mathbb{F}^n$ to $\mathbb{F}^m$, it is trivial to show that the set of functions representable by an arithmetic circuit is the set of functions of the form

$$\boldsymbol{z} \longmapsto (p_1(\boldsymbol{z}),\ \ldots,\ p_m(\boldsymbol{z}))$$

where $\boldsymbol{z} = (z_1, \ldots, z_n) \in \mathbb{F}^n$ and each $p_i$ is a polynomial $p_i \in \mathbb{F}[x_1, \ldots, x_n]$.

We can give additional structure to a circuit by giving it a layering, defined as follows.

**Definition 3.2.2** (Layered circuit)**.** *A layer circuit of depth $D$ is a circuit where the set of nodes is partitioned into $D+1$ subsets such that for every layer $i = 0, \ldots, D$ and every gate in the $i$-th layer, every incoming wire of the gate comes from a gate in layer $i+1$ and every outgoing wire of the gate goes to a gate in layer $i-1$. With this convention, layer 0 is the layer of outputs and layer $D$ is the layer of inputs*[7]

Arithmetic circuits can be used to express a wide variety of operations, many of which have a natural representation as a layered circuit with repeating sub-blocks which iterate multiple times, such as many families of hash functions. We note that, while circuits are directed and acyclic hence have a natural ordering on the nodes, not every circuit admits a layering by default. This can be done by adding "dummy" gates which behave like the identity function, to pass an earlier value

[7] This choice of notation gives a bottom-up representation of the circuit, instead of a top-down representation. While this notation might be a bit counter-intuitive, it is the notation used by the protocols we will analyse [24, 25, 6], as it does simplify the description of the recursive process used by these protocols.

down the layers, though this comes at a cost of additional gates. *We will see that the computational cost and sizes of the protocols will heavily depend on the number of gates.* To give an example of sizes, the sha256 hash function (the de-facto standard hash function currently in use for most applications) can be represented as a boolean circuit, which can be thought[8] as an arithmetic circuit over $\mathbb{F}_2$, with 116245 gates [26].

Circuits can be thought of as constraint systems when constraining the evaluation to some fixed value. More precisely, let $\mathcal{C}$ be a circuit with inputs $(z_1, \ldots, z_n)$, then we can define a natural constraint system given by

$$\mathcal{C}(z_1, \ldots, z_n) \stackrel{?}{=} 0$$

where the right-hand side should be interpreted as the $\mathbf{0} = (0, \ldots, 0)$ if $\mathcal{C}$ has multiple outputs[9].

In the contexts we are interested in, circuits usually have a part of public inputs, which give information about the claimed statement, and private inputs, which are necessary to check that the statement is true. For example, suppose that a prover $\mathcal{P}$ wants to convince a verifier $\mathcal{V}$ that they know a valid sha256 preimage for a public hash result. We might consider a circuit $\mathcal{C}$ which takes as input public values representing an sha256 result, and private values representing the claimed preimage, then computes the sha256 of the given claimed preimage and compares it to the given result.

Up to reordering, we will always suppose that the inputs of the circuit are partitioned in a first set of public inputs $\mathbb{x} = (z_1, \ldots, z_k)$ and a remaining set of private inputs $\mathbb{w} = (z_{k+1}, \ldots, z_n)$. Additionally, we will always suppose that circuits take a special input fixed to the value $z_0 := 1$ , in order to allow constants in the body of the circuit. With this division of the inputs, we will write $\mathcal{C}(\mathbb{x}, \mathbb{w})$ as a nicer notation for $\mathcal{C}(1 \,||\, \mathbb{x} \,||\, \mathbb{w})$. We can define a relation for the circuit satisfiability as follows

$$\mathcal{R} := \{(\mathbb{x}, \mathbb{w}) \in \mathbb{F}^k \times \mathbb{F}^{n-k} \mid \mathcal{C}(\mathbb{x}, \mathbb{w}) = 0\}$$

For these types of relations, we let $|\mathcal{R}|$ be the size of the circuit $|\mathcal{C}|$, meaning that the size and time constraint for a zkSNARK must be sublinear in $|\mathcal{C}|$.

[8] The only technical difference is that in a boolean circuit we expect to use AND and XOR, which behave respectively as multiplication and addition over $\mathbb{F}_2$, as well as the unary INV gates. To fix this, we can suppose that boolean gates always have a fixed input node with value 1, with which we can write the INV using a XOR gate wired to the 1 input.

[9] A more flexible system would let the right-hand side be any arbitrary fixed value. However, up to adding an additional layer before the output which subtracts the intended result, the two systems are equivalent

### 3.2.2 Rank-1 Constraint Systems

While arithmetic circuits are extremely versatile and expressive, many zkSNARKs use a different, and in certain aspects simpler, constraint system, called *Rank-1 Constraint System* (R1CS). R1CSs reduce much of the description complexity of a circuit, and are used as an internal middleware by basically any library used to implement zkSNARK proof systems.

**Definition 3.2.3** (Rank-1 Constraint System)**.** *A Rank-1 Constraint System (R1CS) is a set of equations depending on input values*

$$\boldsymbol{z} = (z_0 := 1, z_1, \ldots, z_n)$$

*and coefficients $a_{i,j}, b_{i,j}, c_{i,j}$ for $i = 0, \ldots, n$ and $j = 1, \ldots, m$ of the form*

$$\text{for } j = 1, \ldots, m \qquad \sum_{i=0}^{n} a_{i,j} z_i \cdot \sum_{i=0}^{n} b_{i,j} z_i \stackrel{?}{=} \sum_{i=0}^{n} c_{i,j} z_i \tag{3.7}$$

*R1CSs are usually represented in matrix form as*

$$A\boldsymbol{z} \circ B\boldsymbol{z} \stackrel{?}{=} C\boldsymbol{z} \tag{3.8}$$

*for matrices $A, B, C \in \mathbb{F}^{m \times (n+1)}$, where $\circ$ represents the Hadamard product (element-wise product).*

Similarly to what we had with arithmetic circuits, the convention of fixing the first value $z_0 = 1$ allows to write affine combinations of the input values, instead of just linear combinations, while allowing the succinct matrix notation.

By dividing the inputs into a set of public inputs $\mathbb{x} = (z_1, \ldots, z_k)$ and private inputs $\mathbb{w} = (z_{k+1}, \ldots, z_n)$ we can define a relation for the satisfiability of a R1CS in the same fashion as the circuit satisfiability, as follows

$$\mathcal{R} := \{(\mathbb{x}, \mathbb{w}) \in \mathbb{F}^k \times \mathbb{F}^{n-k} \mid A\boldsymbol{z} \circ B\boldsymbol{z} = C\boldsymbol{z},\ \boldsymbol{z} := \mathbb{x} \,||\, \mathbb{w}\}$$

We note that while composed of different types of constraint, arithmetic circuits and R1CS are equivalent in the sense that one can transform one into the other such that the satisfiability relation of the original and the satisfiability relation of the result always agree on the validity of any input $\mathbb{x}$. That is to say, if $\mathcal{R}_1$ is the satisfiability relation for the original system (circuit or R1CS) and $\mathcal{R}_2$ is the satisfiability relation of the transformed system (R1CS or circuit), then it holds

$$\forall \mathbb{x} \quad \mathbb{x} \in \mathcal{R}_1 \iff \mathbb{x} \in \mathcal{R}_2$$

Note that we do not require that the value $\mathbb{x}$ should have the same witness both for $\mathcal{R}_1$ and $\mathcal{R}_2$, as the conversion from one system to the other might (and will, in one direction) require to create additional values that encode internal states, and because of this will have to make part of the witness.

**Theorem 3.2.1** (Equivalence of circuit satisfiability and R1CS satisfiability)**.** *Given a R1CS over $\mathbb{F}$ with public inputs $(z_1, \ldots, z_k)$ and private inputs $(z_{k+1}, \ldots, z_n)$, there exists an arithmetic circuit $\mathcal{C}$ over $\mathbb{F}$ such that, by letting $\mathcal{R}_{R1CS}$ be the satisfiability relation for the R1CS and $\mathcal{R}_{\mathcal{C}}$ be the satisfiability relation for $\mathcal{C}$, it holds*

$$\forall \mathbb{x} \in \mathbb{F}^k \quad \mathbb{x} \in \mathcal{R}_{R1CS} \iff \mathbb{x} \in \mathcal{R}_{\mathcal{C}}$$

*Viceversa, given an arithmetic circuit $\mathcal{C}$ over $\mathbb{F}$ with public inputs $(z_1, \ldots, z_k)$ and private inputs $(z_{k+1}, \ldots, z_n)$, there exists a R1CS over $\mathbb{F}$ such that, by letting $\mathcal{R}_{\mathcal{C}}$ be the satisfiability relation for $\mathcal{C}$ and $\mathcal{R}_{R1CS}$ be the satisfiability relation for the R1CS, it holds*

$$\forall \mathbb{x} \in \mathbb{F}^k \quad \mathbb{x} \in \mathcal{R}_{\mathcal{C}} \iff \mathbb{x} \in \mathcal{R}_{R1CS}$$

*Proof.* We start with the proof of the first claim, which is trivial. By focusing on each row $j$ of the R1CS, we can write the constraint as

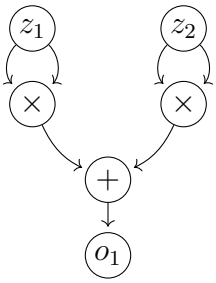
$$\sum_{i=0}^{n} a_{i,j} z_i \cdot \sum_{i=0}^{n} b_{i,j} z_i - \sum_{i=0}^{n} c_{i,j} z_i \stackrel{?}{=} 0$$

The left-hand side of the constraint is a polynomial $p_j \in \mathbb{F}[z_1, \ldots, z_n]$ (recalling that by convention $z_0 := 1$) with the property that $p_j(\boldsymbol{z}) = 0$ if and only if the $j$-th row of the R1CS is satisfied. We can represent the multi-function $(p_1, \ldots, p_m)$ as an arithmetic circuit $\mathcal{C}$. It is then clear that for a given $\boldsymbol{z} \in \mathbb{F}^n$ then

$$\mathcal{C}(\boldsymbol{z}) = 0 \iff A\boldsymbol{z} \circ B\boldsymbol{z} - C\boldsymbol{z}$$

hence (independently from how the inputs are divided in public and private) $\mathcal{R}_{\text{R1CS}}$ and $\mathcal{R}_{\mathcal{C}}$ are the exact same relation.

The proof of the second claim is a bit more complicated. To better understand the idea behind the proof, we start by considering the following toy example[10]




[10]Note that we said that by convention, the first input of an arithmetic circuit is always the fixed value $a_0 := 1$. We omit it in the description of this circuit as it is an unused input, but it will come back when converting to an R1CS.

which represents the constraint $z_1^2 + z_2^2 = 0$. While this constraint cannot be directly represented as a R1CS, we can add auxiliary variables $z_3, z_4$ and $z_5$ corresponding to each of the internal gates. Then, we can transform the original constraint into the following set of constraints

$$\begin{aligned} z_1 \cdot z_1 &= z_3 \\ z_2 \cdot z_2 &= z_4 \\ z_3 + z_4 &= z_5 \\ z_5 &= 0 \end{aligned}$$

which can be represented as a R1CS as follows[11]

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} \circ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix}$$

This gives a recipe for transforming an arithmetic circuit constraint into a R1CS. We start with an empty R1CS and by defining auxiliary variables $z_{n+1}, \ldots, z_{n+t}$ one for each gate of the circuit. Then for each gate, corresponding to some variable $z_i$, we add a row to the R1CS as follows

- if the current gate is a multiplication gate with inputs $z_{i_l}, z_{i_r}$, we add a row to $A$ with a 1 in the $i_l$-th entry, a row to $B$ with a 1 in the $i_r$-th entry, and a row to $C$ with a 1 in the $i$-th entry
- if the current gate is an addition gate with inputs $z_{i_l}, z_{i_r}$, we add a row to $A$ with ones in the $i_l$-th and $i_r$-th entry, a row to $B$ with a 1 in the 0-th entry, and a row to $C$ with a 1 in the $i$-th entry

Finally, for each output coming from some gate $z_i$ we add a row to $A$ with a 1 in the $i$-th entry, a row to $B$ with a 1 in the 0-th entry, and and empty row to $C$, asserting that $z_i$ evaluates to 0.

The resulting R1CS takes as input values $z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+t}$, which is not the same input space as the original circuit $\mathcal{C}$. If the inputs of $\mathcal{C}$ are divided in public inputs $\mathbb{x} = (z_1, \ldots, z_k)$ and private inputs $\mathbb{w} = (z_{k+1}, \ldots, z_n)$, then we consider for our R1CS the division of inputs given by public inputs $\mathbb{x} = (z_1, \ldots, z_k)$ and

[11] note that for the sake of simplicity, we omitted the conventional "1" as initial input both for the circuit as well as the R1CS

private inputs $\widetilde{\mathbb{w}} = (z_{k+1}, \ldots, z_n, z_{n+1}, \ldots, z_{n+t})$. By construction, then, it holds

$$\begin{aligned}
\mathbb{x} \in \mathcal{R}_{\mathcal{C}} &\iff \exists \mathbb{w} \in \mathbb{F}^{n-k} \text{ s.t. } \mathcal{C}(\mathbb{x}, \mathbb{w}) = \mathbf{0} \\
&\iff \exists \mathbb{w} \in \mathbb{F}^{n-k}, \exists (z_{n+1}, \ldots z_{n+t}) \in \mathbb{F}^t \text{ s.t. } \mathcal{C}(\mathbb{x}, \mathbb{w}) = \mathbf{0} \\
&\qquad \text{and the gates of } \mathcal{C}(\mathbb{x}, \mathbb{w}) \text{ evaluate to } (z_{n+1}, \ldots z_{n+t}) \\
&\iff \exists \widetilde{\mathbb{w}} \in \mathbb{F}^{n-k+t} \text{ s.t } A\boldsymbol{z} \circ B\boldsymbol{z} = C\boldsymbol{z} \text{ where } \boldsymbol{z} = 1||\mathbb{x}||\widetilde{\mathbb{w}} \\
&\iff \mathbb{x} \in \mathcal{R}_{\text{R1CS}}
\end{aligned}$$

□

The process we followed in the end of the proof is called an *arithmetization*, that is the encoding of a computation into an algebraic constraint, which is more suitable for the types of probabilistic proofs we are studying.

There are a few points worth mentioning that arise from proof of the theorem. First of all, we notice that the number of private inputs, hence the size of the witness of our relation, increased by the number of gates[12] of the circuit. This not only has implications on the computational costs on the protocol, but also implies that a prover $\mathcal{P}$ with a witness $\mathbb{w}$ for a value $\mathbb{x}$ for the circuit satisfiability cannot use directly the same witness $\mathbb{w}$ for the same value in the R1CS satisfiability. Indeed, the prover has to compute the "extended" witness by evaluating the circuit and retrieving the value of each gate to obtain the values of the auxiliary variables. This is what happens under the hood when libraries such as snarkjs or ZoKrates run the "generation of the witness" step.

Additionally, we notice that the matrices of the resulting R1CS are sparse, which is not necessarily the case for any given R1CS. We point this out as the density of the matrices will become relevant in the comparison of some of the protocols we will see.

### 3.2.3 Quadratic Arithmetic Programs

*Quadratic Arithmetic Programs* (QAP) are a different, possibly more efficient, type of arithmetization for arithmetic circuits. They were proposed by Gennaro et al. in 2013 [27] and they resembles R1CS in the form, but instead of working with multiple equation checks expressed in matrix form, they work with polynomials. Some protocols such as the Groth16 analysed later are defined over QAP relations rather than R1CS relations, as it is usually easier to perform checks over polynomials.

[12]Note that while the process used in the proof does indeed add as many inputs as the number of gates, when actually converting a circuit into a R1CS some tricks can be used to reduce the number of necessary auxiliary variables

**Definition 3.2.4** (Quadratic Arithmetic Program)**.** *Let $\mathbb{F}$ be a field. Let $r_1, \ldots, r_m \in \mathbb{F}$ be distinct points and let $t(x) = \prod_{j=1}^{m}(x - r_j)$ be the vanishing polynomial over $\{r_1, \ldots, r_m\}$, which we will call the target polynomial. For $i = 0, \ldots, n$, let $u_i, v_i, w_i \in \mathbb{F}^{<m}[x]$. Finally, let $z_0 := 1, z_1, \ldots, z_n$ be inputs in $\mathbb{F}$.*

*A Quadratic Arithmetic Program (QAP) is an equation of the form*

$$\sum_{i=0}^{n} z_i u_i(x) \cdot \sum_{i=0}^{n} z_i v_i(x) \stackrel{?}{=} \sum_{i=0}^{n} z_i w_i(x) \mod t(x) \tag{3.9}$$

The equation (3.9) can equivalently be expressed in the following form

$$\exists q \in \mathbb{F}^{\leq m-2}[x] \text{ s.t. } \sum_{i=0}^{n} z_i u_i(x) \cdot \sum_{i=0}^{n} z_i v_i(x) \stackrel{?}{=} \sum_{i=0}^{n} z_i w_i(x) + q(x)t(x) \tag{3.10}$$

or as

$$\text{for } j = 1, \ldots, m \quad \sum_{i=0}^{n} z_i u_i(r_j) \cdot \sum_{i=0}^{n} z_i v_i(r_j) \stackrel{?}{=} \sum_{i=0}^{n} z_i w_i(r_j) \tag{3.11}$$

where for this last equivalence we used the fact that if $t(x)$ is a vanishing polynomial over a set, then any polynomial is identically null modulo $t(x)$ if and only if it vanishes on the same set.

In practice, the vanishing set is almost always taken to be the set of the $m$-th roots of unity $\{\omega, \omega^2, \ldots, \omega^m\}$ as it results in more efficient interpolations at computation time[13]. The resulting target polynomial is $t(x) = x^m - 1$. We will stick to this choice of target polynomial and vanishing set.

**Theorem 3.2.2** (Conversion of R1CS to QAP)**.** *Given a R1CS over $\mathbb{F}$ with public inputs $(z_1, \ldots, z_k)$ and private inputs $(z_{k+1}, \ldots, z_n)$, there exists a QAP over $\mathbb{F}$ such that, by letting $\mathcal{R}_{R1CS}$ be the satisfiability relation for the R1CS and $\mathcal{R}_{QAP}$ be the satisfiability relation for the QAP, it holds*

$$\forall \mathbb{x} \in \mathbb{F}^k \quad \mathbb{x} \in \mathcal{R}_{R1CS} \iff \mathcal{R}_{QAP}$$

*Viceversa, given a QAP over $\mathbb{F}$ with public inputs $(z_1, \ldots, z_k)$ and private inputs $(z_{k+1}, \ldots, z_n)$, there exists a R1CS over $\mathbb{F}$ such that, by letting $\mathcal{R}_{QAP}$ be the satisfiability relation for the QAP and $\mathcal{R}_{R1CS}$ be the satisfiability relation for the R1CS, it holds*

$$\forall \mathbb{x} \in \mathbb{F}^k \quad \mathbb{x} \in \mathcal{R}_{QAP} \iff \mathcal{R}_{R1CS}$$

[13] Of course, in order to be able to take this choice one should require that the $m$-th root of unity exists inside the field $\omega \in \mathbb{F}$. However, as we will see later, we are usually not targeting a specific $m$ rather a "large enough" valid $m$, which simplifies the requirement. Additionally, there are a list of standardized fields commonly used for cryptographically applications which satisfy these requirements from which one can choose to instantiate a protocol.

*Proof.* The proof is straightforward. To construct an equivalent QAP from a R1CS, we define the polynomials $u_i$, $v_i$ and $w_i$ by interpolation on the columns of the matrices $A, B$ and $C$ that define the R1CS, as follows

$$\begin{array}{l} u_i(\omega^j) := a_{i,j} \\ v_i(\omega^j) := b_{i,j} \\ w_i(\omega^j) := c_{i,j} \end{array} \qquad \text{for} \quad \begin{array}{c} i = 0, \ldots, n \\ j = 0, \ldots, m \end{array}$$

With these choice of polynomials the equivalent formulation of the QAP constraint in Eq. 3.11 becomes precisely the R1CS constraint in Eq. 3.7, which means they have the same satisfiability relations $\mathcal{R}_{\text{R1CS}} = \mathcal{R}_{\text{QAP}}$.

For the second claim, it suffices to construct matrices where the coefficients are the evaluations of the polynomials $u_i, v_i$ and $w_i$ at the points $\omega^j$. □

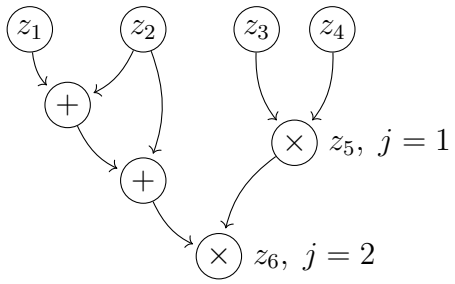
The advantage of QAP over R1CS is that, as we will see later, some components of probabilistic proofs work especially well with polynomials.
We will now show a particular type of arithmetization from a circuit to a QAP[14] which ends up being particularly efficient as it only scales in the number of multiplication gates of the circuit, instead of the total number of gates of the circuit, and is the approach used in many implementations of zkSNARK systems that reply on Groth16 which will be presented later on. This particular arithmetization is the reason why additions are said to be "free" in Groth16.

**Theorem 3.2.3.** *There exists an efficient arithmetization from arithmetic circuits to QAP such that the number of additional auxiliary private inputs depends only on the number of multiplication gates of the circuit*

*Proof.* Let $\mathcal{C}$ be an arithmetic circuit over $\mathbb{F}$ with public inputs $z_1, \ldots, z_k$ and private inputs $z_{k+1}, \ldots, z_n$. We start by fixing an enumeration of the multiplication gates of the circuit $\mathcal{C}$ and creating auxiliary variables $z_{n+1}, \ldots, z_{n+m}$, one for each of the $m$ multiplication gates.

We want to define *selector* polynomials, which specify the inputs of each of the multiplication gates. For $j = 1, \ldots, m$ corresponding to the $j$-th gate, we start by defining its left and right inputs. Notice that normally a multiplication gate has only one left and one right input. However, we want to aggregate all the values $z_i$ that arrive to the $j$-th gate through addition gate. To make this process more clear, consider the following circuit:

[14]Note that as far as the construction goes, nothing forces us to use QAP instead of R1CS as they are practically the same constraint from a different perspective. However, the protocol we will consider, Groth16, is designed on QAP so we will show this representation




The circuit has "original" inputs $z_1, \ldots, z_4$ and auxiliary input $z_5, z_6$ corresponding to the first and second multiplication gate, that is the gate with $j = 1, 2$. In this scenario,

- the left input of the gate $j = 1$ is $z_3$, while the right input is $z_4$, each with multiplicity 1.
- the left inputs of the gate $j = 2$ are $z_1$ with multiplicity 1 and $z_2$ with multiplicity 2, while the right input is $z_6$ with multiplicity 1.

That is, we are defining "inputs to a multiplication gate" up to intermediate addition gates and counting multiplicity.

For $i = 0, \ldots, n + m$ we define the *left selector* polynomial $l_i \in \mathbb{F}^{<m}[x]$ which encodes whether the value $z_i$ is a left input of the $j$-th gate (for $j = 1, \ldots, m$) in the following way

$$l_i(\omega^j) := \begin{cases} \text{multiplicity of } z_i & \text{if } z_i \text{ is a left input of the } j\text{-th gate} \\ 0 & \text{otherwise} \end{cases}$$

Similarly, we define the *right selector* polynomial $r_i \in \mathbb{F}^{<m}[x]$ which encodes whether the value $z_i$ is a left input of the $j$-th gate.
Finally, we define the *output selector* polynomial $o_i \in \mathbb{F}^{<m}[x]$ which encodes whether the value $z_i$ is the value associated to the $j$-th gate, meaning it is the output of the $j$-th gate, as follows[15]

$$o_i(\omega^j) := \begin{cases} 1 & \text{if } z_i \text{ is the auxiliary value associated to the } j\text{-th gate} \\ 0 & \text{otherwise} \end{cases}$$

We can then write the QAP equation

$$\sum_{i=0}^{n+m} z_i l_i(x) \cdot \sum_{i=0}^{n+m} z_i r_i(x) \stackrel{?}{=} \sum_{i=0}^{n+m} z_i o_i(x) \mod (x^m - 1)$$

[15]Notice that by construction the $j$-th gate is the auxiliary value $z_{n+j}$, so one could more easily define $o_i(\omega^j) := \mathbb{1}_{i-n}(j)$

or equivalently, in the same form as Eq. 3.11

$$\text{for } j = 1, \ldots, m \quad \sum_{i=0}^{n+m} z_i l_i(\omega^j) \cdot \sum_{i=0}^{n+m} z_i r_i(\omega^j) \stackrel{?}{=} \sum_{i=0}^{n+m} z_i o_i(\omega^j)$$

Notice that when evaluating at $\omega^j$, the first sum of the left-hand side becomes exactly the sum of the left inputs of the $j$-th gate (with the multiplicity accounted for), similarly the second sum becomes exactly the sum of the right inputs of the $j$-th gate, and the right-hand side becomes $z_{n+j}$. Thus, the correctness of the $j$-th equation of this QAP is equivalent to the correctness of the evaluation of the $j$-th gate of the circuit $\mathcal{C}$. Then it holds

$$\begin{aligned}
\mathbb{x} \in \mathcal{R}_{\mathcal{C}} &\iff \exists \mathbb{w} \in \mathbb{F}^{n-k} \text{ s.t. } \mathcal{C}(\mathbb{x}, \mathbb{w}) = \mathbf{0} \\
&\iff \exists \mathbb{w} \in \mathbb{F}^{n-k}, \exists (z_{n+1}, \ldots z_{k+t}) \in \mathbb{F}^m \text{ s.t. } \mathcal{C}(\mathbb{x}, \mathbb{w}) = \mathbf{0} \\
&\qquad \text{and the multiplication gates of } \mathcal{C}(\mathbb{x}, \mathbb{w}) \\
&\qquad \text{evaluate to } (z_{n+1}, \ldots z_{k+t}) \\
&\iff \exists \widetilde{\mathbb{w}} \in \mathbb{F}^{n-k+t} \text{ s.t } \boldsymbol{z} := 1 || \mathbb{x} || \widetilde{\mathbb{w}} \text{ satisfies the QAP} \\
&\iff \mathbb{x} \in \mathcal{R}_{\text{QAP}}
\end{aligned}$$

□

Notice that this construction can be further optimized in specific cases, such as in the case where some value $z_i$ arrives to the $j$-th gate not only through sums but also through a constant multiplication (which would slightly complicate the description of the selector polynomials by changing the definition of "multiplicity" to take account of these constant multiplications).

# Chapter 4

# Common protocols and gadgets

Before jumping to real zkSNARK protocols and their analysis, we present some common protocols and gadgets which are either present in full in the zkSNARKs we will consider, or the basis for variants used by them. In this context, a gadget is a subroutine which can be invoked inside of a larger-scope protocol and used as a black-box.

We note that many of the protocols considered in this section are interactive, hence not directly suitable for zkSNARK protocols. However, as it is common practice in these situations, they can be transformed into non-interactive protocols through the Fiat-Shamir transformation [28].

## 4.1 Checks on polynomials

At the heart of every probabilistic proof based on polynomials there is a gadget to check the equality of two polynomials, based on the Schwartz-Zippel lemma, which we now recall

**Remark** (Schwartz-Zippel Lemma 2.3.1)**.** *For any $f \in \mathbb{F}_p^{\leq d}[x_1, \ldots, x_k]$ and any subset $S \subset \mathbb{F}_p$, if $f \not\equiv 0$ then it holds*

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow S^k : f(\boldsymbol{r}) = 0\right] \leq \frac{d}{|S|}$$

*where $r_1, \ldots, r_k$ are sampled uniformly in $S$. Similarly, for any polynomials $f, g \in \mathbb{F}_p^{\leq d}[x_1, \ldots, x_k]$ and any subset $S \subset \mathbb{F}_p$, if $f \neq g$ then it holds*

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow S^k : f(\boldsymbol{r}) = g(\boldsymbol{r})\right] \leq \frac{d}{|S|}$$

This gives an upper bound on the probability of two different polynomials $f$ and $g$ passing the check $f(\boldsymbol{r}) \stackrel{?}{=} g(\boldsymbol{r})$. By opportunely choosing the parameters[16], for example by assuring that $\log(|S|) \geq \log(d) + \lambda$, we get that for $f \neq g$ then it holds

$$\mathsf{Pr}\left[\boldsymbol{r} \leftarrow S^k : f(\boldsymbol{r}) = g(\boldsymbol{r})\right] \leq \mathsf{negl}(\lambda)$$

This allows us to write an efficient check for a verifier to control whether two polynomials are equal or not. We suppose that the verifier has oracle access to evaluations of the polynomial, meaning that the verifier has not direct access to the polynomials but can query verified evaluations of the polynomials at any time. This is usually instantiated through a polynomial commitment scheme or some kind of homomorphic commitment scheme.

**Definition 4.1.1** (Polynomial Equality Check)**.** *Let $\mathbb{F}_p$ be a finite field for a prime $p$ and let $f, g \in \mathbb{F}_p^{\leq d}[x_1, \ldots, x_k]$, such that $\frac{d}{p} = \mathsf{negl}(\lambda)$.*

*Let $\mathcal{V}$ be a verifier and $\mathcal{O}_{f,g}$ be an oracle for evaluations of the polynomials $f$ and $g$. We define the polynomial equality check as the following protocol:*

PolyEqCheck *:*

1. $\mathcal{V}^{\mathcal{O}_{f,g}}$ *samples a random* $\boldsymbol{r} \leftarrow \mathbb{F}_p^k$
2. $\mathcal{V}^{\mathcal{O}_{f,g}}$ *queries the evaluations* $f(\boldsymbol{r}), g(\boldsymbol{r})$ *to* $\mathcal{O}_{f,g}$
3. $\mathcal{V}^{\mathcal{O}_{f,g}}$ *outputs* accept *if* $f(\boldsymbol{r}) = g(\boldsymbol{r})$*, and* reject *otherwise*

**Proposition 4.1.1.** *In the Polynomial Equality Check, the verifier accepts the equality of $f \neq g$ only with negligible probability, that is*

$$f \neq g \implies \mathsf{Pr}\left[\mathsf{accept} \leftarrow \mathsf{PolyEqCheck}\right] \leq \mathsf{negl}(\lambda)$$

*Proof.* Follows trivially from the Schwartz-Zippel lemma □

Similarly, we can construct a protocol which allows to check whether a given polynomial $f \in \mathbb{F}_p^{\leq d}[x]$ vanishes over a set of points (distinct) $r_1, \ldots, r_k$. We simplify this requirement to a single equality by using the vanishing polynomial of the set $r_1, \ldots, r_k$ defined as $z(x) = \prod_{i=1}^{k}(x - r_i)$. It holds that

$$\forall i \; f(r_i) = 0 \iff \exists q \in \mathbb{F}_p^{d-k} \text{ s.t. } f(x) = q(x)z(x)$$

Then, if a prover wants to show that a given polynomial $f$ vanishes over $r_1, \ldots, r_k$, it can compute the quotient polynomial $q$ and prove the equality $f(x) = q(x)z(x)$ through a polynomial equality check.

[16] In practice we usually pick $S = \mathbb{F}_p$ with $p \approx 2^{256}$ and around $d \approx 2^{20}$ for real use cases, which leaves plenty of breathing room for the standard security level $\lambda = 128$

**Definition 4.1.2** (Polynomial Vanishing Check)**.** *Let $\mathbb{F}_p$ be a finite field for a prime $p$. Let $r_1, \ldots, r_k \in \mathbb{F}_p$ be distinct points, let $f \in \mathbb{F}_p^{\leq d}[x]$ and $q \in \mathbb{F}_p^{\leq d-k}$ such that $\frac{d}{p} = \mathsf{negl}(\lambda)$.*

*Let $\mathcal{V}$ be a verifier and $\mathcal{O}_{f,q}$ be an oracle for evaluations of the polynomials $f$ and $q$. We define the polynomial vanishing check as the following protocol:*

PolyVanishCheck *:*

1. $\mathcal{V}^{\mathcal{O}_{f,q}}$ *samples a random* $\boldsymbol{r} \leftarrow \mathbb{F}_p^k$
2. $\mathcal{V}^{\mathcal{O}_{f,q}}$ *queries the evaluations* $f(\boldsymbol{r}), q(\boldsymbol{r})$ *to* $\mathcal{O}_{f,q}$
3. $\mathcal{V}^{\mathcal{O}_{f,q}}$ *computes manually* $z(\boldsymbol{r})$
4. $\mathcal{V}^{\mathcal{O}_{f,q}}$ *outputs* accept *if* $f(\boldsymbol{r}) = q(\boldsymbol{r})z(\boldsymbol{r})$*, and* reject *otherwise*

**Proposition 4.1.2.** *In the Polynomial Vanishing Check, the verifier accepts the vanishing of a non-vanishing polynomial $f$ only with negligible probability, that is*

$$\exists i \text{ s.t. } f(r_i) \neq 0 \implies \mathsf{Pr}\left[\mathsf{accept} \leftarrow \mathsf{PolyVanishCheck}\right] \leq \mathsf{negl}(\lambda)$$

*Proof.* If $\exists i$ such that $f(r_i) \neq 0$ then for any polynomial $q$ (in particular the one with oracle access) it must hold

$$f(x) \neq q(x)z(x)$$

as the right-hand side evaluates to 0 at $r_i$ while the left-hand side does not. The rest follows trivially from the Schwartz-Zippel lemma. □

## 4.2 Polynomial commitments and the KZG protocol

Similarly to how we defined a commitment scheme in the preliminaries, we now define a polynomial commitment scheme with knowledge-soundness, which can be used to obtain verified evaluations of a committed polynomial

**Definition 4.2.1** (Polynomial commitment scheme)**.** *Let $\mathbb{F}$ be a finite field. A polynomial commitment scheme is a protocol composed of algorithms* Setup*,* Commit*,* Eval*,* VerifyCommit *and* VerifyEval *with the following interface:*

$\mathsf{pp} \leftarrow \mathsf{Setup}(\mathbb{F})$ *: creates public parameters* pp *for the protocol*

$\mathsf{com}_p, \mathsf{open} \leftarrow \mathsf{Commit}(\mathsf{pp}, p)$ *: creates a commitment for the polynomial $p \in \mathbb{F}[x]$ together with any opening information* open *necessary for the verification step of the commitment*

$v, \pi \leftarrow \mathsf{Eval}(\mathsf{pp}, p, u)$ *: returns the claimed evaluation* $v = p(u)$ *together with additional information required to prove that indeed* $v = p(u)$

$\mathsf{accept/reject} \leftarrow \mathsf{VerifyEval}(\mathsf{pp}, \mathsf{com}_p, u, v, \pi)$ *: checks whether the claimed evaluation* $(u, v)$ *is compatible with the claimed commitment* $\mathsf{com}_p$

$\mathsf{accept/reject} \leftarrow \mathsf{VerifyCom}(\mathsf{pp}, \mathsf{com}, p, \mathsf{open})$ *: checks whether the claimed commitment* com *really corresponds to the polynomial* $p$

*which satisfy the following properties*

**Correctness** *For any* $p \in \mathbb{F}[x]$ *and* $u \in \mathbb{F}$

$$\mathsf{Pr}\left[\begin{array}{c}\mathsf{pp} \leftarrow \mathsf{Setup}(\mathbb{F}) \\ \mathsf{com}_p, \mathsf{open} \leftarrow \mathsf{Commit}(\mathsf{pp}, p) \\ v, \pi \leftarrow \mathsf{Eval}(\mathsf{pp}, p, u)\end{array} : \begin{array}{c}\mathsf{accept} \leftarrow \mathsf{VerifyEval}(\mathsf{pp}, \mathsf{com}_p, u, v, \pi) \\ \mathsf{accept} \leftarrow \mathsf{VerifyCom}(\mathsf{pp}, \mathsf{com}_p, p, \mathsf{open})\end{array}\right] = 1$$

**Knowledge-soundness** *For any PPT adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ *there exists an extractor* $\mathcal{E}$ *such that either* $\mathcal{A}$ *fails to pass the* VerifyEval *or the extractor is able to extract a polynomial which satisfies the commitment and evaluation, except that with negligible probability*

$$\mathsf{Pr}\left[\begin{array}{c}\mathsf{pp} \leftarrow \mathsf{Setup}(\mathbb{F}_p) \\ \mathsf{com} \leftarrow \mathcal{A}_0(\mathsf{pp}) \\ p, \mathsf{open} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{com}) \\ u, v, \pi \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathsf{com})\end{array} : \begin{array}{c} p(u) \neq v \\ \mathsf{accept} \leftarrow \mathsf{VerifyEval}(\mathsf{pp}, \mathsf{com}, u, v, \pi)\end{array}\right] = \mathsf{negl}(\lambda)$$

One of the most widely used polynomial commitment schemes is the KZG protocol proposed by Kate et al. [22] in 2010, which is a constant-size polynomial commitment scheme used by many zkSNARKs for its properties. It relies on the Algebraic Group Model and bilinear pairings, and checks similar to the polynomial equality check to obtain extremely efficient proof size and verification time.

The main idea behind the KZG protocol is to prove the evaluation of a certain polynomial $p \in \mathbb{F}_p^{\leq d}[x]$ at a given point $u$ as $v = p(u)$ by showing that $p(x) - v$ can be factorised as

$$p(x) - v = (x - u) \cdot q(x)$$

for some polynomial $q(x) \in \mathbb{F}_p^{<d}[x]$. This equality is checked in similar fashion to the polynomial equality check by evaluating both the left-hand side and the right-hand side at a random point $\tau \in \mathbb{F}_p$. The random point $\tau$ will be baked into the public parameters as the group elements

$$[1], [\tau], \ldots, [\tau^d]$$

which allow the evaluation of $[p(\tau)]$ without direct access to $\tau$. Indeed, if we let $p(x) = \sum_{i=0}^{d} p_i x^i$ then one can compute

$$[p(\tau)] = \sum_{i=0}^{d} p_i [\tau^i]$$

The whole protocol is defined as follows

**Definition 4.2.2** (KZG protocol)**.** *Let $\mathbb{F}_p$ be a finite field of prime order $p$, and let $e : G \times G \to G_T$ be a bilinear pairing on groups $G, G_T$ of prime order $p$, where $G$ is generated by $g$. Let $d$ be an integer.*

*The* KZG *protocol is a polynomial commitment scheme on $\mathbb{F}_p^{\leq d}[x]$ defined by the following algorithms (we assume that all the algorithms that receive* pp *as part of their arguments start by parsing* pp *into its components):*

Setup() *:*

1. *sample a random $\tau \in \mathbb{F}_p^*$*
2. *output* pp $:= ([1], [\tau], \ldots, [\tau^d])$

Commit(pp, $p$) *:*

1. *output* $\mathsf{com}_p := [p(\tau)]$ *(and* open $:= \varnothing$ *as it is not needed)*

Eval(pp, $p, u$) *:*

1. *compute $v = p(u)$*
2. *compute $q \in \mathbb{F}_p^{<d}[x]$ such that $p(x) - v = (x - u) \cdot q(x)$*
3. *compute $[q(\tau)]$*
4. *output $v, \pi := [q(\tau)]$*

VerifyEval(pp, com, $u, v, \pi$) *:*

1. *compute $[-v]$ and $[-u]$*
2. *compute $[\tau - u]$ as $[\tau] + [-u]$*
3. *if $e(\mathsf{com} + [-v], [1]) \stackrel{?}{=} e([\tau - u], \pi)$*
4. *output* accept
5. *else*
6. *output* reject

VerifyCom(pp, com, $p, \varnothing$) *:*

1. *compute $[p(\tau)]$*

2. *if* $\mathsf{com} \stackrel{?}{=} [p(\tau)]$
3. *output* accept
4. *else*
5. *output* reject

We remark some of the aspects of the given protocol. First of all, the protocol results in a remarkable constant proof size of only one group element $\pi = [q(\tau)]$ for any input polynomial up to degree $d$. Secondly, we note that the protocol is not hiding, as the Commit algorithm is deterministic.

Lastly, we note that the setup process is a trusted setup process, meaning that it must be carried out by a trusted party, the reason being that if the prover gets access to $\tau$ directly instead of $[\tau]$ and its powers, they would be able to prove false evaluations of the polynomial. This then implies that the protocol relies on the $q$-DLOG (with $q = d$) for security.

**Theorem 4.2.1.** *If $e$ is a bilinear pairing such that the generalised $q$-SBDH assumption holds for $q = d$, then* KZG *is correct and knowledge-sound in the Algebraic Group Model*

*Proof.* Correctness follows from inspection of the protocol. For knowledge-soundness, recall that an algebraic adversary $\mathcal{A}$, when outputting $\mathsf{com} \in G$, must also construct an $\boldsymbol{a} \in \mathbb{F}_p^d$ such that

$$\mathsf{com} = \langle \boldsymbol{a}, \mathsf{pp} \rangle = \sum_{i=0}^{d} a_i [\tau^i]$$

which the extractor $\mathcal{E}^{\mathcal{A}}$ can obtain since it has access to $\mathcal{A}$. We let the extractor output the polynomial

$$p(x) := \sum_{i=0}^{d} a_i x^i$$

and we show that, if indeed $p(u) \neq v$, then the probability of $\mathcal{A}$ passing the VerifyEval check is upperbounded by the probability of any adversary winning at the generalised $q$-SBDH challenge, which we assumed to be negligible. To do so, we will show that with access to such an adversary $\mathcal{A}$ and the extractor $\mathcal{E}^{\mathcal{A}}$, one can construct another adversary $\mathcal{B}$ for the $q$-SBDH challenge such that if $\mathcal{A}$ passes the VerifyEval check then $\mathcal{B}$ wins at the $q$-SBDH challenge. We define $\mathcal{B}$ as follows

$\mathcal{B}([1], \ldots, [\tau^d])$ :

1. set $\mathsf{pp} := ([1], \ldots, [\tau^d])$

2. compute $\mathsf{com} \leftarrow \mathcal{A}_0(\mathsf{pp})$ and $p \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{com})$
3. compute $u, v, \pi \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathsf{com})$
4. compute the quotient polynomial $q$ such that $p(x) - p(u) = q(x)(x - u)$
5. compute $[q(\tau)]$ from the public parameters
6. compute $\delta$ and $h$ as

$$h := q(\tau) = e([1], \pi) - e([1], [q(\tau)])$$
$$\delta := p(u) - v \neq 0$$

7. return $u, \delta, h$

We argue that $h = \left[\frac{\delta}{\tau - u}\right]_T$ if and only if the outputs of $\mathcal{A}_0, \mathcal{A}_1$ pass the $\mathsf{VerifyEval}$ check. Recall that the $\mathsf{VerifyEval}$ check is $e(\mathsf{com} + [-v], [1]) \stackrel{?}{=} e([\tau - u], \pi)$ and notice that by construction, $\mathsf{com} = [p(\tau)]$, which means we can rewrite the $\mathsf{VerifyEval}$ check as

$$\begin{aligned}
& e([\tau - u], \pi) \stackrel{?}{=} e(\mathsf{com} + [-v], [1]) \\
\iff\ & e([\tau - u], \pi) \stackrel{?}{=} e([p(\tau) - v], [1]) \\
& \qquad = e([p(\tau) - p(u) + p(u) - v], [1]) \\
& \qquad = e([q(\tau)(\tau - u) + \delta], [1]) \\
& \qquad = e([q(\tau)(\tau - u)], [1]) + e([\delta], [1]) \\
\iff\ & e([\delta], [1]) \stackrel{?}{=} e([\tau - u], \pi) - e([q(\tau)(\tau - u)], [1]) \\
& \qquad = (\tau - u) \cdot \big(e([1], \pi) - e([q(\tau)], [1])\big) \\
& \qquad = (\tau - u) \cdot h \\
\iff\ & \frac{\delta}{\tau - u} \cdot e([1], [1]) \stackrel{?}{=} h \\
\iff\ & \left[\frac{\delta}{\tau - u}\right]_T \stackrel{?}{=} h
\end{aligned}$$

Then the probability of $\mathcal{A}$ passing the check is equal to the probability of $\mathcal{B}$ winning the $q$-SBDH challenge which is upper bounded by $\mathsf{negl}(\lambda)$ □

There exist variants of the $\mathsf{KZG}$ protocol, some of which proposed in the same work as the original [22] with different properties. For example, by masking the polynomial $p$ with a random polynomial $q$ in the $\mathsf{Commit}$ (and adding $q$ to the $\mathsf{open}$ information) we get a protocol that is hiding. As another example, one can modify the protocol to obtain a batched version of the $\mathsf{KZG}$ which allows one to open multiple evaluations of multiple polynomials with a single proof of a single group element (at a small cost of soundness error).

## 4.3 Sumcheck protocol

The Sumcheck protocol is an interactive protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ to reduce the verified evaluation of a specific sum of a polynomial to a verified evaluation of the same polynomial at a random verifier-chosen point.

Given a $k$-variate polynomial $g \in \mathbb{F}^{\leq d}[x_1, \ldots, x_k]$, consider the following expression

$$\sum_{\boldsymbol{b} \in \{0,1\}^k} g(b_1, \ldots, b_k)$$

that is the sum of the evaluations of $g$ at the vertices of the $k$-hypercube. The Sumcheck protocol starts with the prover $\mathcal{P}$ sending a claimed value for the evaluation of Eq. 4.1, to which the verifier $\mathcal{V}$ recursively responds with challenges to reduce the claim to a single verified evaluation $g(\boldsymbol{r})$. The recursion is done on the variables of the polynomial, decreasing the number of variables of the polynomial at each recursive step by generating an univariate polynomial that the verifier can evaluate manually. The validity of the univariate polynomial is then checked using the same technique as the polynomial equality check by evaluation at a random verifier-chosen point.

We now describe formally the Sumcheck protocol.

**Definition 4.3.1** (Sumcheck protocol)**.** *Let $\mathbb{F}$ be a finite field and let $g \in \mathbb{F}^{\leq d}[x_1, \ldots, x_k]$ be a $k$-variate polynomial.*

*The* Sumcheck *protocol is an interactive protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ which reduces the problem of the verified evaluation of the following expression*

$$\sum_{\boldsymbol{b} \in \{0,1\}^k} g(b_1, \ldots, b_k) \tag{4.1}$$

*to the verified evaluation of $g$ at a single random verifier-chosen point $\boldsymbol{r} \in \mathbb{F}^k$. The polynomial $g$ is known by the prover $\mathcal{P}$ but not necessarily known by the verifier $\mathcal{V}$*[17]*.*

*For the sake of consistent notation in the recursion, let $g_k := g$. The* Sumcheck *protocol is defined as follows:*

Sumcheck *:*

[17] We do not require $g$ to be unknown by $\mathcal{V}$, which would not be the case in scenarios where $\mathcal{V}$ only wants to delegate the computation of the expression in Eq. 4.1 to a computationally-powerful party $\mathcal{P}$

**Setup** *:*

1. *$\mathcal{P}$ computes $c_k := \sum_{\boldsymbol{b} \in \{0,1\}^k} g_k(b_1, \ldots, b_k)$ and sends it to $\mathcal{V}$.*

**Recursive step, for i=k, ..., 1** *:*

1. *$\mathcal{P}$ computes the univariate polynomial*

$$s_i(x_i) := \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} g_i(b_1, \ldots, b_{i-1}, x_i)$$

   *and sends it to $\mathcal{V}$.*
2. *$\mathcal{V}$ receives $s_i$ and checks that $c_i \stackrel{?}{=} s_i(0) + s_i(1)$, aborting if the check does not hold.*
3. *$\mathcal{V}$ samples a random $r_i \leftarrow \mathbb{F}$, sets $c_{i-1} := s_i(r_i)$ and sends $r_i$ to $\mathcal{P}$.*
4. *$\mathcal{P}$ receives $r_i$ and sets*

$$g_{i-1}(x_1, \ldots, x_{i-1}) := g_i(x_1, \ldots, x_{i-1}, r_i)$$

*In the end, the initial claim is reduced to the claim that $c_0 \stackrel{?}{=} g(\boldsymbol{r})$ where $\boldsymbol{r} = (r_1, \ldots, r_k)$ is uniformly chosen by the verifier entry by entry in the recursion.*

In the setup step of the protocol, the prover computes and sends the claimed correct evaluation $c_k$ of the sumcheck expression (Eq. 4.1). Then, for $i = k, \ldots, 1$, the purpose of each recursive step is to reduce the following claim (which for $i = k$ ensures the correct evaluation of the sumcheck expression)

$$c_i \stackrel{?}{=} \sum_{\boldsymbol{b} \in \{0,1\}^i} g_i(b_1, \ldots, b_i)$$

to

$$c_{i-1} \stackrel{?}{=} \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} g_{i-1}(b_1, \ldots, b_{i-1})$$

This is done as follows. First, the prover computes a univariate polynomial $s_i$ and sends it to the verifier. This allows $\mathcal{V}$ to compute $s_i(0) + s_i(1)$ which, if both $c_i$ and $s_i$ were computed correctly, should equal $c_i$. Now if this check passes and the prover can convince the verifier that $s_i$ was computed correctly, then the verifier will be convinced that $c_i$ too was computed correctly. Hence, the verifier wants a proof that $s_i$ was computed correctly, meaning that indeed it holds

$$s_i(x_i) \stackrel{?}{=} \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} g_i(b_1, \ldots, b_{i-1}, x_i)$$

To do this, the protocol uses the polynomial equality check. The verifier picks a random point $r_i \in \mathbb{F}$ and, ideally, checks that the left-hand side and the right-hand side evaluate to the same value. The left-hand side can be computed locally and

efficiently by the verifier $\mathcal{V}$, which computes $c_{i-1} := s_i(r_i)$. Then, since the prover wants to prove that indeed $s_i$ was computed correctly, it wants to prove that the right-hand side of the previous sum indeed evaluates to $c_{i-1}$. Given the definition of $g_{i-1}$, the problem is indeed reduced to the following claim

$$c_{i-1} \stackrel{?}{=} \sum_{\boldsymbol{b} \in \{0,1\}^{i-1}} g_{i-1}(b_1, \ldots, b_{i-1})$$

In the end, the remaining claim is that $c_0 \stackrel{?}{=} g(r_1, \ldots, r_k)$, hence the sumcheck protocol reduces the claim of correct evaluations of the sumcheck expression to the claim of correct evaluation of the polynomial $g$ at a single random verifier-chosen point in $\mathbb{F}^k$.

**Theorem 4.3.1.** *The sumcheck protocol has a soundness error of $\frac{kd}{|\mathbb{F}|}$, meaning that a false claim can be reduced to a true claim at most with probability $\frac{kd}{|\mathbb{F}|}$.*

*Proof.* The proof is done by recursion on $k$. Let $k = 1$. Suppose that the original claim is false, namely

$$c_1 \neq \sum_{b \in \{0,1\}} g(b) = g(0) + g(1)$$

and that $\mathcal{V}$ does not abort. Since the verifier did not abort, it must hold

$$c_1 = s_1(0) + s_1(1)$$

which clearly implies that $s_1 \neq g$. Since $c_0 := s_1(r_1)$ for a random verifier-chosen $r_1$, then the claim is equivalent to $s_1(r_1) \stackrel{?}{=} g(r_1)$, and since they are different it holds

$$\mathsf{Pr}\left[r_1 \leftarrow \mathbb{F} : s_1(r_1) = g(r_1)\right] \leq \frac{d}{|\mathbb{F}|}$$

from the Schwartz-Zippel lemma.

Consider then a generic $k$. In order to reduce a false claim to a true claim, in at least one of the recursive steps there must be a reduction from a false claim to a true claim. We divide the analysis on whether this happens in the first step or not. More formally, suppose that $c_k \neq \sum_{\boldsymbol{b} \in \{0,1\}^k} g(\boldsymbol{b})$, and consider the following events

$$\begin{aligned}
A &:= \{c_0 = g(\boldsymbol{r})\} \\
B &:= \{c_{k-1} = \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g_{k-1}(b_1, \ldots, b_{k-1}) \ \wedge \ c_0 = g(\boldsymbol{r})\} \\
C &:= \{c_{k-1} \neq \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g_{k-1}(b_1, \ldots, b_{k-1}) \ \wedge \ c_0 = g(\boldsymbol{r})\}
\end{aligned}$$

where the event $A$ encodes the reduction of a false claim to a true claim. Clearly $A = B \cup C$ hence

$$\mathsf{Pr}[A] \leq \mathsf{Pr}[B] + \mathsf{Pr}[C]$$

Additionally, the event $C$ corresponds exactly to the event $A$ for $k-1$, so by induction we have that

$$\mathsf{Pr}[C] \leq \frac{(k-1)d}{|\mathbb{F}|}$$

As for $B$, first note that if $\mathcal{V}$ does not abort then it must hold $c_k = s_k(0) + s_k(1)$. Combining this with the fact that by hypothesis $c_k \neq \sum_{\boldsymbol{b} \in \{0,1\}^k} g(\boldsymbol{b})$, we get that

$$s(x) \neq \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g(b_1, \ldots, b_{k-1}, x)$$

Note that $B \subset \{c_{k-1} = \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g_{k-1}(b_1, \ldots, b_{k-1})\}$ hence

$$\begin{aligned}
\mathsf{Pr}[B] &\leq \mathsf{Pr}\left[c_{k-1} = \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g_{k-1}(b_1, \ldots, b_{k-1})\right] \\
&= \mathsf{Pr}\left[s_k(r_k) = \sum_{\boldsymbol{b} \in \{0,1\}^{k-1}} g(b_1, \ldots, b_{k-1}, r_k)\right] \\
&\leq \frac{d}{|\mathbb{F}|}
\end{aligned}$$

where the last inequality comes from the Schwartz-Zippel lemma. Summing the probabilities of $B$ and $C$ we get that

$$\mathsf{Pr}[A] \leq \frac{kd}{|\mathbb{F}|}$$

□

## 4.4 GKR protocol

The GKR protocol defined by Goldwasser et al. [24] is an interactive proof between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ for the correct evaluation of layered arithmetic circuits. We underline that this protocol is both interactive and not zero-knowledge, so it will need adaptations if we intend to use it as a building block for non-interactive zero-knowledge proofs. However, it provides succinct verifier time and interaction which are linear in the depth of the circuit instead of its size.

The high-level idea behind the GKR protocol is that it starts from the end layer of the circuit and then recursively reduces $\mathcal{P}$'s claim of the correct evaluation of the current layer to a claim on the previous layer using the sumcheck protocol. To do this, first we give a precise labeling of the circuit.

Given a layered arithmetic circuit $\mathcal{C}$ over $\mathbb{F}$ of size $C$ and depth $D$, let $S_i$ be the size of the $i$-th layer and let $s_i = \lceil \log S_i \rceil$. Notice that in the layer enumeration, $i = 0$ corresponds to the *output* layer of the circuit, and $i = D$ corresponds to the *input* layer of the circuit. For each layer $i$ we fix an enumeration that maps $\boldsymbol{b} \in \{0,1\}^{s_i}$ to the $\boldsymbol{b}$-th gate (interpreting $\boldsymbol{b}$ as the binary representation of a number) of the $i$-th layer and we call $\boldsymbol{b}$ the label of the $\boldsymbol{b}$-th gate. We also define a function

$$V_i : \{0,1\}^{s_i} \to \mathbb{F}$$

which maps $\boldsymbol{b}$ to the output of the $\boldsymbol{b}$-th gate of the $i$-th layer.

Let

$$\begin{aligned} \text{add}_{i+1} &: \{0,1\}^{s_i} \times \{0,1\}^{s_{i+1}} \times \{0,1\}^{s_{i+1}} \to \{0,1\} \\ \text{mult}_{i+1} &: \{0,1\}^{s_i} \times \{0,1\}^{s_{i+1}} \times \{0,1\}^{s_{i+1}} \to \{0,1\} \end{aligned}$$

be the *wiring predicates* of the circuit $\mathcal{C}$ at layer $i$, that is the functions that, given the gate $\boldsymbol{a}$ at layer $i$ and gates $\boldsymbol{b}, \boldsymbol{c}$ at layer $i+1$, output 1 if $\boldsymbol{a}$ is an addition (resp. multiplication) gate with left input $\boldsymbol{b}$ and right input $\boldsymbol{c}$, and output 0 otherwise.

With these definitions we can give an explicit definition of a correctly constructed $V_i$ in terms of $V_{i+1}$

$$\begin{aligned} V_i(\boldsymbol{a}) = \sum_{\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{s_{i+1}}} \Big[ &\text{add}_{i+1}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[V_{i+1}(\boldsymbol{b}) + V_{i+1}(\boldsymbol{c})\big] + \\ &+ \text{mult}_{i+1}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[V_{i+1}(\boldsymbol{b}) \cdot V_{i+1}(\boldsymbol{c})\big] \Big] \end{aligned} \tag{4.2}$$

Ideally, we would like to check if this equation holds by sampling at a random verifier-chosen point. The issue is that by doing this, a malicious prover could change the value of one of the outputs and with probability $\frac{S_0 - 1}{S_0}$ it would not be detected. Instead, we will take advantage of the distance-amplifying property of multilinear extensions.

While the multilinear extension of the left-hand side of Equation (4.2) is straightforward, we have to be more careful with the right-hand side. We are *not* taking the multilinear extension with respect to all the variables $\boldsymbol{a} \,||\, \boldsymbol{b} \,||\, \boldsymbol{c}$, rather just

$\boldsymbol{a}$. Additionally, with respect to just $\boldsymbol{a}$, the evaluations of $V_{i+1}(\boldsymbol{b}) + V_{i+1}(\boldsymbol{c})$ and $V_{i+1}(\boldsymbol{b}) \cdot V_{i+1}(\boldsymbol{c})$ for fixed values of $\boldsymbol{b}$ and $\boldsymbol{c}$ behave as constants in the multilinear extension, which we can extract by linearity of the operator. Lastly, using Lemma (2.3.2) we can directly use the multilinear extension of $\text{add}_{i+1}$ and $\text{mult}_{i+1}$ over all the variables, instead of just $\boldsymbol{a}$. This process induces the following equality as multilinear polynomials over $\mathbb{F}_p^{s_i}$

$$\widetilde{V_i}(\boldsymbol{a}) = \sum_{\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{s_{i+1}}} \Big[ \widetilde{\text{add}_{i+1}}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[V_{i+1}(\boldsymbol{b}) + V_{i+1}(\boldsymbol{c})\big] + \\ + \widetilde{\text{mult}_{i+1}}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[V_{i+1}(\boldsymbol{b}) \cdot V_{i+1}(\boldsymbol{c})\big] \Big]$$

Lastly, by noting that by definition $V_{i+1} = \widetilde{V_{i+1}}$ over $\{0,1\}^{s_{i+1}}$, we can use this substitution in the evaluations of $V_{i+1}$ to get the following equality as multilinear polynomials over $\mathbb{F}_p^{s_i}$, which holds if $V_i$ is correctly constructed:

$$\widetilde{V_i}(\boldsymbol{a}) = \sum_{\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{s_{i+1}}} \Big[ \widetilde{\text{add}_{i+1}}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[\widetilde{V_{i+1}}(\boldsymbol{b}) + \widetilde{V_{i+1}}(\boldsymbol{c})\big] + \\ + \widetilde{\text{mult}_{i+1}}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) \cdot \big[\widetilde{V_{i+1}}(\boldsymbol{b}) \cdot \widetilde{V_{i+1}}(\boldsymbol{c})\big] \Big] \tag{4.3}$$

We can now describe the recursive process of GKR, highlighting the first step which is slightly different to the remaining recursive steps:

**First step:** $\mathcal{V}$ wants to check that $V_0$ has been computed correctly. We will reduce this to a check on the evaluation of $\widetilde{V_1}$ at two random points $\boldsymbol{k}^{(1)}, \boldsymbol{h}^{(1)}$ uniformly chosen by $\mathcal{V}$

To verify this, $\mathcal{V}$ can instead check the equation $\widetilde{V_0}$, which if satisfied would imply that $V_0$ has been computed correctly. To verify the equation on $\widetilde{V_0}$ we can use the same trick as the polynomial equality check, that is evaluating the polynomial at a random point to assert the equality. $\mathcal{V}$ chooses a random point $\boldsymbol{r} \leftarrow \mathbb{F}^{s_i}$, evaluates $\widetilde{V_0}(\boldsymbol{r})$ (which can be done directly by $\mathcal{V}$ since it can be obtained from $V_0$ which is public), sends $\boldsymbol{r}$ to $\mathcal{P}$ and queries the evaluation of the right-hand side, concluding by checking the queried value.

First, let us define the following $2s_1$-variate polynomial to simplify the writing of the equation (we leave implicit the dependence on $\boldsymbol{r}$, as by now it is

fixed, to simplify notation)

$$f(\boldsymbol{b},\boldsymbol{c}) = \widetilde{\text{add}_1}(\boldsymbol{r},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_1}(\boldsymbol{b}) + \widetilde{V_1}(\boldsymbol{c})\right] + \\ + \widetilde{\text{mult}_1}(\boldsymbol{r},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_1}(\boldsymbol{b}) \cdot \widetilde{V_1}(\boldsymbol{c})\right]$$

Then we can rewrite the equality that $\mathcal{V}$ wants to verify as

$$\widetilde{V_0}(\boldsymbol{r}) \stackrel{?}{=} \sum_{\boldsymbol{b},\boldsymbol{c}\in\{0,1\}^{s_1}} f(\boldsymbol{b},\boldsymbol{c})$$

By interpreting the sum on "$\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{s_1}$" as a sum on "$(\boldsymbol{b}|\boldsymbol{c}) \in \{0,1\}^{2s_1}$", $\mathcal{V}$ can query the evaluation of the right-hand side of the equality through the sumcheck protocol, which reduces the verification to querying a verified evaluation of the $2s_1$-variate polynomial $f$ at a random verifier-chosen point $(\boldsymbol{k}^{(1)}|\boldsymbol{h}^{(1)}) \in \mathbb{F}^{2s_1}$, with $\boldsymbol{k}^{(1)},\boldsymbol{h}^{(1)} \in \mathbb{F}^{s_1}$.

Notice that the values of $\widetilde{\text{add}_1}(\boldsymbol{a},\boldsymbol{k}^{(1)},\boldsymbol{h}^{(1)})$ and $\widetilde{\text{mult}_1}(\boldsymbol{a},\boldsymbol{k}^{(1)},\boldsymbol{h}^{(1)})$ can be computed by $\mathcal{V}$ locally, so in order to obtain a verified evaluation of $f(\boldsymbol{k}^{(1)},\boldsymbol{h}^{(1)})$ it suffices to query a verified evaluation of $\widetilde{V_1}(\boldsymbol{k}^{(1)})$ and $\widetilde{V_1}(\boldsymbol{h}^{(1)})$ and then $\mathcal{V}$ can compute the final value.

Thus we have successfully reduced the problem of verifying the correct evaluation of $V_0$ to the problem of verifying

**Recursive step:** $\mathcal{V}$ wants to check the evaluation of $\widetilde{V_i}$ at two random points $\boldsymbol{k}^{(i)},\boldsymbol{h}^{(i)}$. We will reduce this to a check on the evaluation of $\widetilde{V_{i+1}}$ at two random points $\boldsymbol{k}^{(i+1)},\boldsymbol{h}^{(i+1)}$.

We cannot simply verify $\widetilde{V_i}(\boldsymbol{k}^{(i)})$ and $\widetilde{V_i}(\boldsymbol{h}^{(i)})$ with two sumchecks as, while it would work, the number of sumchecks required would grow exponentially in the depth $D$. Instead, we first reduce the two checks on $\widetilde{V_i}(\boldsymbol{k}^{(i)})$ and $\widetilde{V_i}(\boldsymbol{h}^{(i)})$ to a single sumcheck[18].

The idea is that instead of verifying $\widetilde{V_i}(\boldsymbol{k}^{(i)})$ and $\widetilde{V_i}(\boldsymbol{h}^{(i)})$ which would take two checks, we will verify a verifier-chosen random linear combination of these terms. First, $\mathcal{V}$ samples random coefficients $\alpha_i,\beta_i \in \mathbb{F}$ for the linear combination $\alpha_i \cdot \widetilde{V_i}(\boldsymbol{k}^{(i)}) + \beta_i \cdot \widetilde{V_i}(\boldsymbol{h}^{(i)})$.

[18]The original version of the GKR protocol actually uses a different version of two-to-one reduction based on the composition of a random verifier-chosen polynomial. Instead, we will use the random linear combination solution proposed by Chiesa et al. [29] as it adapts more nicely to satisfy the zero-knowledge property and is the version used by Libra [25] and Virgo [6]

We define a new helper $2s_{i+1}$-variate polynomial

$$\begin{aligned} g(\boldsymbol{b},\boldsymbol{c}) := \alpha_i \cdot \widetilde{\mathrm{add}_{i+1}}(\boldsymbol{k}^{(i)},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_{i+1}}(\boldsymbol{b}) + \widetilde{V_{i+1}}(\boldsymbol{c})\right] + \\ + \alpha_i \cdot \widetilde{\mathrm{mult}_{i+1}}(\boldsymbol{k}^{(i)},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_{i+1}}(\boldsymbol{b}) \cdot \widetilde{V_{i+1}}(\boldsymbol{c})\right] + \\ + \beta_i \cdot \widetilde{\mathrm{add}_{i+1}}(\boldsymbol{h}^{(i)},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_{i+1}}(\boldsymbol{b}) + \widetilde{V_{i+1}}(\boldsymbol{c})\right] + \\ + \beta_i \cdot \widetilde{\mathrm{mult}_{i+1}}(\boldsymbol{h}^{(i)},\boldsymbol{b},\boldsymbol{c}) \cdot \left[\widetilde{V_{i+1}}(\boldsymbol{b}) \cdot \widetilde{V_{i+1}}(\boldsymbol{c})\right] \end{aligned}$$

Then, if $V_i$ is correctly evaluated, using a similar procedure to the one used to produce Equation (4.3) we get the following equation:

$$\alpha_i \widetilde{V}_i(\boldsymbol{k}^{(i)}) + \beta_i \widetilde{V}_i(\boldsymbol{h}^{(i)}) = \sum_{\boldsymbol{b},\boldsymbol{c} \in \{0,1\}^{s_{i+1}}} g(\boldsymbol{b},\boldsymbol{c})$$

Similarly to the first step, we can see this sum as a sum over $(\boldsymbol{b}|\boldsymbol{c}) \in \{0,1\}^{2s_{i+1}}$ which allows $\mathcal{V}$ to verify it through a single invocation of the sumcheck protocol, which reduces the verification to verifying the evaluation of the $2s_{i+1}$-variate polynomial $g$ at a random verifier-chosen point $(\boldsymbol{k}^{(i+1)}|\boldsymbol{h}^{(i+1)}) \in \mathbb{F}^{2s_{i+1}}$, with $\boldsymbol{k}^{(i+1)}, \boldsymbol{h}^{(i+1)} \in \mathbb{F}^{s_{i+1}}$. Again, most of the evaluation can be done locally by $\mathcal{V}$ which then only needs to query verified evaluations of $\widetilde{V}_{i+1}(\boldsymbol{k}^{(i+1)})$ and $\widetilde{V}_{i+1}(\boldsymbol{h}^{(i+1)})$.

**Last step:** At the end of the recursion, the claim is reduced to a verified evaluation of $\widetilde{V}_D(\boldsymbol{k}^{(D)})$ and $\widetilde{V}_D(\boldsymbol{k}^{(D)})$.

This evaluation is done locally by $\mathcal{V}$ in the original version of the protocol as the input $V_D$ is considered to be known by the verifier, since the context in which it was proposed was as an application to delegation of computation.

The protocol can be extended to secret inputs by implementing some kind of oracle access to $\widetilde{V}_D(\boldsymbol{k}^{(D)})$ and $\widetilde{V}_D(\boldsymbol{k}^{(D)})$, for example through a zero-knowledge verifiable polynomial delegation scheme as done in Libra [25] and Virgo [6].

# Chapter 5

# Trusted setup zkSNARKs

## 5.1 Groth16

In 2016, J. Groth proposed a novel trusted setup zkSNARK for Quadratic Arithmetic Programs relations [2], which we will refer to as Groth16 following the example of most of the current literature. This protocol leverages the polynomial equality check to obtain an extremely impressive constant proof size of just three group elements and constant verification time of a single group pairing. However, despite the impressive results, Groth16 suffers from proof malleability as we will discuss later, which makes it not suitable for every scenario.

Recall that given a QAP relation, that is a relation of the form

$$\sum_{i=0}^{n} z_i u_i(x) \cdot \sum_{i=0}^{n} z_i v_i(x) = \sum_{i=0}^{n} z_i w_i(x) \mod (x^m - 1)$$

where $\boldsymbol{z} \in \mathbb{F}_p^n$ and $u_i, v_i, w_i \in \mathbb{F}_p^{<m}[x]$, the relation can be equivalently expressed as the alternative form Eq. 3.10, claiming that there exists a polynomial $q \in \mathbb{F}^{\leq m-2}[x]$ such that

$$\sum_{i=0}^{n} z_i u_i(x) \cdot \sum_{i=0}^{n} z_i v_i(x) \stackrel{?}{=} \sum_{i=0}^{n} z_i w_i(x) + q(x)(x^m - 1)$$

The basic idea behind Groth16 is that this relation can be efficiently checked using the polynomial equality check, which requires checking the equality at a single (random) point. Ideally, we would like to have a randomly sampled point $\tau \in \mathbb{F}_p$ and then prove through bilinear pairings that the following equality holds

$$\sum_{i=0}^{n} z_i u_i(\tau) \cdot \sum_{i=0}^{n} z_i v_i(\tau) \stackrel{?}{=} \sum_{i=0}^{n} z_i w_i(\tau) + q(\tau)(\tau^m - 1)$$

To understand the intuition behind the Groth16 protocol, we start by considering a case where the prover sets

$$A = \sum_{i=0}^{n} z_i u_i(\tau)$$
$$B = \sum_{i=0}^{n} z_i v_i(\tau)$$
$$C = \sum_{i=h+1}^{n} z_i w_i(\tau) + q(\tau) \cdot (\tau^m - 1),$$

and then sends $A, B$ and $C$ to the verifier which can then check[19] that the following equality holds

$$A \cdot B = C + \sum_{i=0}^{k} z_i w_i(\tau)$$

Of course, this approach is deeply flawed both for the zero-knowledge property and the knowledge soundness property. First of all, sending $A$ and $B$ as they are reveals information which depends on the witness. Secondly, if we just sample a random $\tau \in \mathbb{F}_p$ and send it to the prover, the prover would be able to forge invalid proofs by sampling random $A$ and $B$ and computing the $C$ that satisfies the previous equality.

To solve the zero-knowledge issue, we can add some random elements to $A$ and $B$ which, informally, will mask the values and reveal no information on the witness, and formally will make $A$ and $B$ follow uniform distribution so that they can be simulated in the simulation algorithm for the zero-knowledge property.

To solve the knowledge soundness issue, we can define a pair of prove key and verification key, created to a trusted setup, which allows the prover and the verifier to perform the respective computations without needing direct access to $\tau$ as in the KZG protocol.

This is exactly the approach of the protocol proposed by J. Groth in the original article [2] which we now define.

Note that there are subtle differences between the original presentation in [2] and the following presentation. First of all, the original protocol is defined for a generic vanishing polynomial $t(x)$ while the following definition chooses the fixed $x^m - 1$.

[19] Note that the sum is composed only of the public inputs $z_0 = 1, z_1, \ldots, z_k$ and the polynomials $w_0, \ldots, w_k$ which are public

This choice was taken considering that virtually all implementations of Groth16 choose $x^m - 1$ so that interpolation of polynomials can be done on the roots of unity which can be computed efficiently with the FFT.

Additionally, we slightly changed the signature of the algorithms to conform to the definition of zkSNARK which evolved with time, in the following way: the original presentation divides the public parameters into a prover key pk and a verifier key vk, while we condense them in the public parameters pp; the original presentation uses a definition that requires the simulator to use the public parameters pp while given access to a trapdoor (which in this case would be the value of $\tau$) while in this presentation we rewrote it so that the simulator generates its own $\tau$ and public parameters without needing a trapdoor. This is not an improvement or downgrade as much as a rewrite to conform to the definition.

Lastly, the elements given in the public parameters differ slightly as they reflect how one would actual implement Groth16 in an efficient implementation. Instead of giving elements $\{[\tau^j]_1\}_{j=0}^m$ and $\{[\tau^j]_2\}_{j=0}^m$, which would be used to construct the values $[u_i(\tau)]_*, [v_i(\tau)]_*$ and $[w_i(\tau)]_*$ we give directly these elements since they can be precomputed. Additionally, we also substitute the elements $\left[\frac{\tau^j(\tau^m-1)}{\delta}\right]_1$ with elements $\left[\frac{\mathcal{L}_j(\tau)(\tau^m-1)}{\delta}\right]_1$ where $\mathcal{L}_j$ is the $j$-th Lagrange-basis polynomial for $\omega, \ldots, \omega^m$ which can be used to compute $\left[\frac{q(\tau)(\tau^m-1)}{\delta}\right]_1$.

**Definition 5.1.1** (Groth16)**.** *Let $\mathbb{F}_p$ be a finite field of prime order $p$. Let $G_1, G_2$ be groups of order $p$ that admit a bilinear pairing $e : G_1 \times G_2 \to G_T$. Let $g_1, g_2$ be generators of $G_1$ and $G_2$ respectively.*

*Let $u_i, v_i, w_i \in \mathbb{F}_p^{<m}[x]$ for $i = 0, \ldots, m$ be public polynomials and let $z_0 = 1, z_1, \ldots, z_n \in \mathbb{F}_p$ scalars divided into public inputs $(z_1, \ldots, z_k)$ and witness $(z_{k+1}, \ldots, z_n)$.*

Groth16 *is a zkSNARK for the following QAP relation*

$$\sum_{i=0}^{n} z_i u_i(x) \cdot \sum_{i=0}^{n} z_i v_i(x) = \sum_{i=0}^{n} z_i w_i(x) \quad mod\,(x^m - 1) \tag{5.1}$$

*defined by the following algorithms*

$\mathsf{pk}, \mathsf{vk} \leftarrow \mathsf{Groth16.Setup}()$ *:*

1. *sample $\alpha, \beta, \gamma, \delta, \tau \leftarrow \mathbb{F}_p^*$*

2. *return* pp *defined as*

$$\mathsf{pp} := \Bigg( [\alpha]_1,\ [\beta]_1,\ [\beta]_2,\ [\gamma]_2,\ [\delta]_1,\ [\delta]_2,\ [\alpha\beta]_T,$$
$$\{[u_i(\tau)]_1\}_{i=0}^{m},\ \{[v_i(\tau)]_1\}_{i=0}^{m},\ \{[v_i(\tau)]_2\}_{i=0}^{m},$$
$$\left\{\left[\tfrac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\gamma}\right]_1\right\}_{i=0}^{k},\ \left\{\left[\tfrac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\delta}\right]_1\right\}_{i=k+1}^{n},$$
$$\left\{\left[\tfrac{\mathcal{L}_j(\tau)(\tau^m-1)}{\delta}\right]_1\right\}_{i=0}^{m-2} \Bigg)$$

*where* $[\alpha\beta]_T := e([\alpha]_1, [\beta]_2)$

$\pi \leftarrow \mathsf{Groth16.Prove}(\mathsf{pp}, (z_1, \ldots, z_k), (z_{k+1}, \ldots, z_n))$ *:*

1. *pick* $r, s \leftarrow \mathbb{F}_p$
2. *compute the quotient polynomial* $q \in \mathbb{F}_p^{\leq m-2}[x]$
3. *compute the following values*

$$[A]_1 := [\alpha]_1 + r[\delta]_1 + \sum_{i=0}^{n} z_i [u_i(\tau)]_1$$
$$[B]_1 := [\beta]_1 + s[\delta]_1 + \sum_{i=0}^{n} z_i [v_i(\tau)]_1$$
$$[B]_2 := [\beta]_2 + s[\delta]_2 + \sum_{i=0}^{n} z_i [v_i(\tau)]_2$$
$$[C]_1 := \sum_{i=k+1}^{n} z_i \left[\frac{\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)}{\delta}\right]_1 + \left[\frac{q(\tau)(\tau^m - 1)}{\delta}\right]_1 + $$
$$+ s[A]_1 + r[B]_1 + rs[\delta]_1$$

4. *return* $\pi := ([A]_1, [B]_2, [C]_1)$

$\mathsf{accept/reject} \leftarrow \mathsf{Groth16.Verify}(\mathsf{pp}, (z_1, \ldots, z_k), \pi)$ *:*

1. *check the equality of*

$$e([A]_1,\ [B]_2) \stackrel{?}{=}$$
$$[\alpha\beta]_T + e\left(\textstyle\sum_{i=0}^{k} z_i \left[\frac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\gamma}\right]_1, [\gamma]_2\right) + e([C]_1, [\delta]_2)$$

2. *if the previous equality holds*
3. *return* accept

*4. else*

*5. return* reject

The properties of the Groth16 protocol are summarized in the following result.

**Theorem 5.1.1.** *There exists a parameter choice for which* Groth16 *is a trusted setup zkSNARK in the GGM for the Quadratic Arithmetic Program relation (Eq. 5.1) with times and sizes dominated by the quantities given in Table 5.1*

| | |
|---|---|
| Prover time | $\mathcal{O}(n+m)$ $\text{mul}_*$ |
| Proof size | 2 $G_1$ + 1 $G_2$ |
| Verifier time | $\mathcal{O}(k)$ $\text{mul}_*$ + 3 pair |

Table 5.1: For sizes, $G_i$ notates an element from $G_i$. For computational cost, $\text{add}_i$ notates addition in group $G_i$, $\text{mul}_i$ notates scalar multiplication in group $G_i$, pair notates a pairing evaluation

*Proof.* The claim on the proof size is evident. The verifier time is dominated by the evaluation of the sum (which itself is dominated by the number of scalar multiplications) and by the three pairings. The domination for the prover time requires some more involved tricks for which we refer to the original work [2], some of which are already accounted for given the precomputing done in the public parameters. Additionally, the quotient polynomial $q$ can be computed in $\mathcal{O}(m\log(m))$ via the FFT.

The correctness of the protocol follows from close inspection of the verification check. As for the zero-knowledge property, since the elements $[A]_1$ and $[B]_2$ get masked by $r[\delta]_1$ and $s[\delta]_2$ respectively and $r, s$ have uniform distribution, then (recalling Lemma 2.3.4) so do $[A]_1$ and $[B]_2$, which means that just by inspecting $\pi$ then the first and second entry of $\pi$ would have uniform distribution over the respective groups. We argue that the value of the third entry of $\pi$ is uniquely determined by the first two.

Let $\pi_1, \pi_2$ be the first two elements of the proof, which we just argued follow a uniform distribution, and $\pi_3$ the third element. Recall that the correspondence $\sigma \mapsto [\sigma]_i$ is a bijection, so while it is computationally difficult to obtain some $A$ such that $[A]_1 = \pi_1$, it is true that there must exist *some* $A$ such that $[A]_1 = \pi_1$ and that since $\pi_1$ is uniformly distributed and the correspondence is a bijection, then so does $A$, and $\pi_1$ is uniquely determined by this value of $A$. Same reasoning goes for the value $B$ such that $[B]_2 = \pi_2$ which is uniformly distributed. As for $\pi_3$, it is true that there must exist some value $C$ such that $[C]_1 = \pi_3$. We argue

that this value $C$ is uniquely identified by the values of $A$ and $B$, which in turn were uniquely identified by $\pi_1$ and $\pi_2$, which would make $\pi_3$ be uniquely identified (though not computationally accessible[20]) by the values of $\pi_1$ and $\pi_2$.

Indeed, $C$ must satisfy

$$C = \frac{AB - \alpha\beta - \sum_{i=0}^{k} z_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau))}{\delta}$$

which makes it uniquely identified by the values of $A$ and $B$. Hence, $\pi_3$ is uniquely identified by the values of $\pi_1$ and $\pi_2$.

This makes it clear how to define the $\mathsf{Sim}$ simulator algorithm, which is done as follows:

$\mathsf{Groth16.Sim}(z_1, \ldots, z_k)$ :

1. sample $\alpha, \beta, \gamma, \delta, \tau \leftarrow \mathbb{F}_p^*$
2. sample $A, B \leftarrow \mathbb{F}_p$
3. compute $C$ as above
4. return $\pi := ([A]_1, [B]_2, [C]_1)$ and $\mathsf{pp}$ constructed as in $\mathsf{Setup}$

The distribution of outputs of this simulator algorithm is then exactly identical to the distribution of real transcripts, which proves the zero-knowledge property.

The proof of knowledge soundness is more convoluted than the proof zero-knowledge, as constructing the extractor is usually more complicated than the simulator, and we refer to the original work [2] as explaining every detail of the proof would be outside the scope of this work. We will however highlight the most important steps as well as the security assumptions for the knowledge soundness.

As stated, the security proof for knowledge soundness holds in the Generic Group Model, which means that the elements of the proof $\pi := (\pi_1, \pi_2, \pi_3)$ must be constructed using the elements given in the public parameters. This means that, whatever the values $A, B$ and $C$ corresponding to $\pi_1, \pi_2$ and $\pi_3$ through the bijection, they must be a combination of $\alpha, \beta, \gamma, \delta$ and all the other field elements corresponding to the group elements of the public parameters $\mathsf{pp}$. Additionally, the fact that the verification check passes means, as discussed in the preliminaries, that the same equation must hold on the fields elements level. Then we interpret $A$, $B$ and $C$ as Laurent polynomials[21] in variables corresponding to said fields

[20] We stress this part of it being identified but not through any computationally feasible mean because if it were not so, then it would be computationally feasible to forge a proof

[21] A Laurent polynomial in $x$ can be seen as an element of $\mathbb{F}_p[x, x^{-1}]$

elements $\alpha, \beta, \ldots$ for some coefficients $A_\alpha, A_\beta, \ldots, B_\alpha, B_\beta, \ldots$ and $C_\alpha, C_\beta, \ldots$ and we interpret the equality as a polynomial equality check to assert that, except that with negligible probability, the coefficients of the polynomials on both sides of the equation must be equal. Then with some calculations one can show that the coefficients $C_{k+1}, \ldots, C_n$ corresponding to the field elements $\frac{\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau)}{\gamma}$ form a valid witness $\mathbb{w} = (C_{k+1}, \ldots, C_n)$. □

As we anticipated, despite the impressive results, Groth16 suffers from a security issue regarding the malleability of its proofs. By malleability we mean that a valid proof can be modified to create another valid proof. Indeed, it is easy to see that given elements $[A]_1, [B]_2$ and $[C]_1$ which pass the verification step, one can use any combination of the following transformations to obtain new elements $[A_{\text{new}}]_1, [B_{\text{new}}]_2$ and $[C_{\text{new}}]_1$ which also pass the verification step:

$$\begin{aligned} [A_{\text{new}}]_1 &= \sigma[A]_1 \\ [B_{\text{new}}]_2 &= \sigma^{-1}[B]_2 \qquad \text{for any } \sigma \in \mathbb{F}_p^* \\ [C_{\text{new}}]_1 &= [C]_1 \end{aligned}$$

or

$$\begin{aligned} [A_{\text{new}}]_1 &= [A]_1 \\ [B_{\text{new}}]_2 &= [B]_2 + \eta[\delta]_2 \qquad \text{for any } \eta \in \mathbb{F}_p^* \\ [C_{\text{new}}]_1 &= [C]_1 + \eta[A]_1 \end{aligned}$$

We note that this malleability is not necessarily an issue and can be used as a feature, such as in cases where unlinkability or untraceability of the prover is necessary, where it is referred to as *re-randomization* of the proof. However, it is a feature that one must be aware of when choosing Groth16 as a zkSNARK for an application. This issue can be mitigated by adding to the protocol a signature scheme to sign the proofs. Additionally, Groth and Maller proposed a new protocol, usually referred to as gm17, which takes inspiration from Groth16 and achieves similar results (with a proof size of 5 group elements instead of 3) while also guaranteeing a stronger version of knowledge-soundness called *simulation extractability*, which basically guarantees that even if an adversary sees an arbitrary number of valid proofs, they cannot produce another valid proof for a statement or else one is able to extract the witness.

## 5.2 Marlin

In 2019 Chiesa et al. [3] proposed Marlin, a zkSNARK for R1CS that is particularly efficient in setups where the R1CS is represented by sparse matrices, by the

means of a specialised indexer which the protocol introduces.

While the full implementation of Marlin is quite convoluted, we give an intuition of the involved processes. Recall the R1CS constraint

$$A\boldsymbol{z} \circ B\boldsymbol{z} \stackrel{?}{=} C\boldsymbol{z}$$

for matrices $A, B, C \in \mathbb{F}_p^{m\times(n+1)}$ and input vector $\boldsymbol{z} \in \mathbb{F}_p^{n+1}$ divided into public inputs $z_1, \ldots, z_k$ and private inputs $z_{k+1}, \ldots, z_n$ with $z_0 := 1$. In Marlin, this check is divided into the following two checks:

$$\begin{aligned} &\text{(lincheck)} && \forall M \in \{A, B, C\} \quad \boldsymbol{z_M} \stackrel{?}{=} M\boldsymbol{z} \\ &\text{(row check)} && \boldsymbol{z_A} \circ \boldsymbol{z_B} \stackrel{?}{=} \boldsymbol{z_C} \end{aligned}$$

with auxiliary variables $\boldsymbol{z_A}, \boldsymbol{z_B}, \boldsymbol{z_C} \in \mathbb{F}_p^m$.

These can be transformed into polynomial checks by defining polynomials $\widehat{z}, \widehat{z}_A, \widehat{z}_B$ and $\widehat{z}_C$ which interpolate the respective vectors at points $\omega, \ldots, \omega^m$. Additionally we define bivariate polynomials $\widehat{A}, \widehat{B}, \widehat{C}$ which interpolate entries $(i, j)$ of the respective matrices at points $(\omega^i, \omega^j)$. This allows to rewrite the the row check as

$$\forall i = 1, \ldots, m \qquad \widehat{z}_A(\omega^i) \cdot \widehat{z}_B(\omega^i) - \widehat{z}_C(\omega^i) \stackrel{?}{=} 0$$

which as we have already seen can be rewritten as

$$\exists\, q \in \mathbb{F}_p^{\leq m}[x] \text{ s.t.} \quad \widehat{z}_A(x) \cdot \widehat{z}_B(x) - \widehat{z}_C(x) = q(x)(x^m - 1)$$

which can be efficiently checked through the polynomial equality check. As for the lincheck, we can rewrite the lincheck for matrix $M$ as

$$\forall i = 1, \ldots, m \qquad \widehat{z}_M(\omega^i) = \sum_{j=1}^{m} \widehat{M}(\omega^i, \omega^j)\widehat{z}(\omega^j)$$

Rather than checking this equality with a polynomial zerocheck which would incur either in a huge number of verified evaluations ($m$ to be precise) or some other form of reduction, we define a polynomial $F(x)$ in a way[22] that $F(\omega^i) = 0$ if and only if the above check is satisfied at $i$, and then check that

$$\sum_{i=1}^{m} F(\omega^i) \stackrel{?}{=} 0$$

[22]For now one can think of $F(x)$ being the difference of the left-hand side and the right-hand side, though the actual implementation differs slightly

Note that similarly to the polynomial equality check, this reduction is a probabilistic reduction, which introduces a soundness error which, just like the case of the polynomial equality check, can be controlled with a compatible choice of parameters.

More precisely, the check is reduced via multiplication by $r$ evaluated at a random point, where $r(x,y) = \frac{(x^m-1)-(y^m-1)}{x-y}$ is the formal derivative of the vanishing polynomial $x^m - 1$, into the following check

$$\sum_{i=1}^{m} r(\alpha, \omega^i) \left[ \widehat{z}_M(\omega^i) - \sum_{j=1}^{m} \widehat{M}(\omega^i, \omega^j) \widehat{z}(\omega^j) \right] \stackrel{?}{=} 0$$

The reason behind the choice of $r(x,y)$ is two-fold: first, it can be computed efficiently; secondly, it satisfies the property that $\{r(x,\omega^i)\}_{i=1}^m$ are linearly independent polynomials. From the second property it shown in the original article that the soundness error induced by the reduction is $\frac{m}{p}$.

By simple indices permutation one can rearrange the left-hand side of the condition into

$$\sum_{i=1}^{m} r(\alpha, \omega^i) \widehat{z}_M(\omega^i) - r_M(\alpha, \omega^i) \widehat{z}(\omega^i) \stackrel{?}{=} 0$$

where $r_M(z,x) := \sum_{j=1}^{m} r(z, \omega^j) \widehat{M}(\omega^j, x)$. Then we reduced the original condition to the univariate sumcheck problem for the function $F_M(x) := r(\alpha, x)\widehat{z}_M(x) - r_M(\alpha, x)\widehat{z}(x)$ over the domain $\omega, \ldots, \omega^m$.

It is shown in previous work that, in general, the univariate sumcheck problem

$$\sum_{i=0}^{m} F(\omega^i) \stackrel{?}{=} \sigma$$

for some $\sigma$ is equivalent to the existence of polynomials $h, g$ such that

$$F(x) \stackrel{?}{=} h(x)(x^m - 1) + xg(x) + \frac{\sigma}{m}$$

which can then be checked via the polynomial equality check. Additionally, the protocol batches the lincheck for the three matrices $A, B$ and $C$ into a unique check through a random linear combination of the checks.

There are some considerations left about the protocol. First of all, the protocol as described is not zero-knowledge, and indeed will require some additional masking, precisely detailed in [3], to obtain the zero-knowledge property. Secondly, the protocol is inherently interactive (for example, the random point $\alpha$ is a challenge that

must be appear as random to the prover in order to show the soundness bounds) and must be made non-interactive through the Fiat-Shamir transformation [28].

Most importantly, though, the protocol is quite inefficient, or more precisely it can be made much more efficient through an optimization which is part of the principal results of the article. Indeed, it is shown that through the definition of a trusted indexer protocol, which defines an index $\mathfrak{i}$ for an efficient encoding of the matrices $A, B$ and $C$, the computations time for both the prover and the verifier can be significantly improved if the matrices are sparse, which is the case for matrices produced when converting an arithmetic circuit into R1CS.

We summarize the properties of the Marlin protocol in the following result, for the proof of which we refer to the original article [3]. We note that the definition of zkSNARK used for the Marlin differs slightly from the definition proposed by us in order to account for the indexing step. Additionally, the results account for a query-parameter $\mathsf{b}$ not mentioned in our discussion, which is the number of queries that the verifier is allowed to make and is necessary to take in account when deciding the size of the masking polynomials for zero-knowledge. However this parameter can bounded by $\mathcal{O}(b)$ and is left only for preciseness of the result. Finally, in order to account for the sparse property of the matrices, we let $h$ be the max of the number of non-zero entries of the matrices $A, B$ and $C$.

**Theorem 5.2.1.** *There exists a parameter choice for which* Marlin *is a universal trusted setup zkSNARK in the AGM for the Rank-1 Constraint System relation (Eq. 3.8) with times and sizes dominated by the quantities given in Table 5.2*

| | |
|---|---|
| Prover time | $\mathcal{O}((h + \mathsf{b}\log(h + \mathsf{b}))$ op |
| Proof size | 13 $G_1$ + 8 $\mathbb{F}_p$ |
| Verifier time | $\mathcal{O}(k + \log(h))$ op |

Table 5.2: For sizes, $G_1$ denotes elements of $G_1$ and $\mathbb{F}_p$ denotes elements of $\mathbb{F}_p$. For computational cost, op denotes field operations

Since the setup protocol is trusted universal, and the sizes used in the protocol are only an upper bound on the maximum allowed size for the R1CS constraint that one chooses to consider, a common practice is to set up in a trusted environment a partial public key that is "big enough", that is at least of the maximum size of R1CS that one think might need (though the bigger the partial public key the bigger the security parameters must be which imply heavier computation, so the bound better be reasonable) and then update it for any particular R1CS considered.

## 5.3 Plonk

In 2019 Gabizon et al. [4] proposed Plonk, a zkSNARK system which applies to arithmetic circuits with fan-in two and unlimited fan-out gates. While the main recipe for the proposed system is not inherently zero-knowledge, they give a zero-knowledge implementation for arithmetic circuits which is knowledge-sound in the Algebraic Group Model, linear prover time in the circuit size and constant proof size and verifier time, though sligthly larger than Groth16's.

While we will use Plonk to refer to the specific instantiation based on the KZG commitment scheme given in the original work by Gabizon et al., it is worth noting that the Plonk paradigm can be instantiated with different polynomial commitment schemes and can even be extended to arbitrary arithmetic gates, other than the standard addition and multiplication. Such protocols are often referred to as Plonkish or based on Plonkish arithmetization.

First, we define a constraint system[23] over vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \{1, \dots, m+n\}^{m+k}$ and $\boldsymbol{q_L}, \boldsymbol{q_R}, \boldsymbol{q_O}\boldsymbol{q_M}, \boldsymbol{q_C} \in \mathbb{F}_p^{m+k}$: we say that the vector $\boldsymbol{z} \in \mathbb{F}_p^{m+n}$ satisfies the constraint system if for $i = 1, \dots, m+k$ it holds

$$(q_L)_i \cdot z_{a_i} + (q_R)_i \cdot z_{b_i} + (q_M)_i \cdot (z_{a_i} z_{b_i}) + (q_O)_i \cdot z_{c_i} + (q_C)_i = 0 \tag{5.2}$$

We think of the vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ as the "wiring" vectors which specify inputs and outputs of the gates, and the vectors $\boldsymbol{q_L}, \boldsymbol{q_R}, \boldsymbol{q_O}\boldsymbol{q_M}, \boldsymbol{q_C}$ as "selector" polynomials which specify which part of the constraint should be activated for the given gate.

We use this constraint system to define an arithmetization for arithmetic circuits with $m$ gates and inputs $z_1, \dots, z_n$ of which $z_1, \dots, z_k$ are public and $z_{k+1}, \dots, z_n$ are private.

We consider a vector $\boldsymbol{z} \in \mathbb{F}_p^{m+n}$ which extends the original inputs and accounts for each value assumed by each gate. Then we add constraints to enforce the values of the public inputs, and this is done by setting the following values for the vectors defining the constraint system:

$$a_i = i, \quad (q_L)_i = 1, \quad (q_R)_i = 0, \quad (q_M)_i = 0, \quad (q_O)_i = 0, \quad (q_C)_i = 0$$

Note that if left as is, the constraint would enforce that the public input $z_i$ satisfies $z_i = 0$, which is restrictive. Indeed, later we will let the verifier add a shift to $\boldsymbol{q_C}$ which will make the constraint relative to the public inputs become $z_i = \bar{z}_i$ where $\bar{z}_i$ is the actual value assumed by the public input, which will actually enforce that

[23] We note that our terminology differs slightly from the terminology used in the article [4]. We discuss this difference in Appendix A.

the vector $\boldsymbol{z}$ is compatible with the public inputs. This is done so that compilation from arithmetic circuit to constraint system can be done without knowledge of the public inputs, which otherwise would have to be fixed a priori and baked into the constraint system.

Then for each other gate $i$ we let $a_i, b_i$ and $c_i$ be respectively the index of the left input, right input and output of the gate. If the gate is a multiplication gate, we set

$$(q_L)_i = 0, \quad (q_R)_i = 0, \quad (q_M)_i = 1, \quad (q_O)_i = -1, \quad (q_C)_i = 0$$

Vice versa, if it is an addition gate we set

$$(q_L)_i = 1, \quad (q_R)_i = 1, \quad (q_M)_i = 0, \quad (q_O)_i = -1, \quad (q_C)_i = 0$$

It is trivial to see that the resulting constraint are

$$\begin{aligned} \text{(multiplication gate)} \qquad & z_{a_i} z_{b_i} - z_{c_i} = 0 \\ \text{(addition gate)} \qquad & z_{a_i} + z_{b_i} - z_{c_i} = 0 \end{aligned}$$

In order to allow the verifier to efficiently check that the constraint in Eq. 5.2 holds, the vectors defining the constraint system are converted into polynomials: given a vector $\boldsymbol{z}$ which the prover claims to be the evaluations of the nodes of the circuit, we define polynomials $f_L, f_R, f_O \in \mathbb{F}_p^{\leq m+k}$ such that for $i = 1, \ldots, m+k$

$$f_L(\omega^i) := z_{a_i}, \quad f_R(\omega^i) := z_{b_i}, \quad f_O(\omega^i) := z_{c_i}$$

where $\omega$ is the $(m+k)$-th root of unity. Additionally, we let $q_L, q_R, q_O, q_M, q_C$ be the polynomials which interpolate the corresponding vectors on the powers of $\omega$ as $f_L, f_R, f_O$ do. Then we can rewrite the constraint in Eq. 5.2 as the existance of a quotient polynomial $h$ such that (hiding the dependence in $x$ of the polynomials for cleaner notation)

$$q_L f_L + q_R f_R + q_M f_L f_R + q_O f_O + q_C = h \cdot (x^{m+k} - 1)$$

As we mentioned, in order for this to be reflecting of the actual circuit with the given public inputs $z_1, \ldots, z_k$, we also define a polynomial $q_P$ which interpolates the public inputs on $\omega, \ldots, \omega^k$ and equals zero on the other powers of $\omega$, so that the check

$$q_L f_L + q_R f_R + q_M f_L f_R + q_O f_O + q_C - q_P = h \cdot (x^{m+k} - 1) \tag{5.3}$$

is equivalent to the circuit constraint.

However, there is a subtlety which we have not yet discussed, regarding the validity of the polynomials $f_L, f_R$ and $f_O$. Indeed, even if a prover is able to convince

a verifier that the polynomial equation holds, it still might be the case that $f_L, f_R$ and $f_O$ are not valid, meaning that they do not faithfully represent the values of the gates in the circuit, as nowhere in the check is enforced that the value of a gate used when checking its output and the value of the gate when used as an input are equal. When considering the situation where the output of gate $i$ is fed as left input of gate $j$, then we have $c_i = a_j$ which, as long as we only care about $\boldsymbol{z}$, implies $z_{c_i} = z_{a_j}$. However, once we transition to the polynomials $f_L$ and $f_O$, if these are not constructed honestly then it might happen that $f_O(\omega^i) \neq f_L(\omega^j)$ which does hold true in an honest computation.

In order to enforce that the polynomials $f_L, f_R$ and $f_O$ are compatible (that is, the value of the output of a gate remains the same when it is being used as an input to some other gate), the authors implement a new protocol called permutation check. The (extended) permutation check for polynomials is a protocol which does the following. Given polynomials $p_1, \ldots, p_k, q_1, \ldots, q_k \in \mathbb{F}_p^{\leq d}[x]$, values $r_1, \ldots, r_n$ and a permutation $\sigma$ over the indices $1, \ldots, kn$, let $p_{(1)}, \ldots, p_{(kn)}$ and $q_{(1)}, \ldots, q_{(kn)}$ be defined as

$$p_{((j-1)n+i)} := p_j(r_i), \qquad q_{((j-1)n+i)} := q_j(r_i)$$

Then the permutation check checks that, for every $t = 1, \ldots, kn$, it holds $q_{(t)} = p_{(\sigma(t))}$.

In order to understand the point of the permutation check (and which permutation we are considering), first we consider the following toy example.

Suppose we have values $\boldsymbol{r} = (r_1, \ldots, r_5)$ and let $\sigma = (1\ 2\ 3)(4\ 5)$. If we can prove that for $i = 1, \ldots, 5$ it holds $r_i = r_{\sigma(i)}$ then we get that

$$r_1 = r_2 = r_3, \qquad r_4 = r_5$$

We use a similar idea to force $f_\triangle(\omega^i) = f_\square(\omega^j)$ when necessary. We let $\boldsymbol{v} = \boldsymbol{a}||\boldsymbol{b}||\boldsymbol{c} \in \{1, \ldots, m+n\}^{3(m+k)}$, and for each $i = 1, \ldots, m+n$ we define

$$T_i := \{\ j\ \mid\ v_j = i\ \}$$

Then we define the permutation $\sigma$ over the indices $1, \ldots, 3(m+k)$ composed of $m+n$ cycles such that the $i$-th cycle is a $|T_i|$-cycle over the entries of $T_i$. Notice that with these definitions if for some $i_1, i_2$ it holds $a_{i_1} = b_{i_2}$, then these $a_{i_1}, b_{i_2}$ would correspond to some $v_{j_1}, v_{j_2}$ where $j_1$ and $j_2$ must belong to the same $T_i$. We argue that if we can prove that the polynomials $f_L, f_R, f_O$ satisfy the permutation check against themselves on points $\omega, \ldots, \omega^{m+k}$ with the $\sigma$ we constructed, we actually get the guarantee that were looking for.

Suppose that the permutation check passes. As an example, suppose that for some $i_1, i_2$ it holds $a_{i_1} = c_{i_2}$. Our goal is to show that it also must hold $f_L(\omega^{i_1}) = f_O(\omega^{i_2})$. By the definition of $\boldsymbol{v}$ we have that $a_{i_1}$ is the entry $v_{i_1}$ and $c_{i_2}$ is $v_{2(m+k)+i_2}$, and by construction of $\sigma$ the indices $i_1$ and $2(m+k)+i_2$ belong to the same cycle in $\sigma$, which means that up to iterating $\sigma$ it must hold $2(m+k)+i_2 = \sigma^*(i_1)$, which implies

$$f_L(\omega^{i_1}) =: f_{(i_1)} = f_{(\sigma(i_1))} = \cdots = f_{(\sigma*(i_1))} = f_{(2(m+k)+i_2)} := f_O(\omega^{i_2})$$

By applying the same strategy on the other pairs of polynomials and indices, we have shown that indeed the polynomials $f_L, f_R, f_O$ are compatible with each other in representing the values of the gates when being considered as inputs and outputs.

The Plonk protocol is then defined as follows (up to instantiating the actual permutation check and the polynomial commitment scheme used to obtain verified evaluations of the polynomials)

1. In the setup, public parameters for the commitment scheme and the permutation check are computed. When using the KZG commitment scheme, these includes values $[1], [\tau], \ldots, [\tau^d]$ for a random secret $\tau$ and in a group compatible with a bilinear pairing. Additionally, the selector polynomials and the wiring vectors are computed

2. The prover computes the polynomials $f_L, f_R, f_O$ and sends to the verifier commitments to these polynomials

3. The prover and the verifier run the permutation check protocol to verify the compatibility of the committed polynomials

4. The verifier computes $q_P$ corresponding to the public inputs

5. The prover computes the quotient polynomial $h$ in Eq. 5.3 and sends it to the verifier

6. The verifier checks the identity in Eq. 5.3 and returns accept/reject accordingly

We note that the protocol as defined is intrinsically interactive, and must be made non-interactive through the Fiat-Shamir transformation [28]. Additionally, we note that the protocol as defined now is not zero-knowledge, and in order to make it so we need to mask the prover's polynomial with some uniformly random multiples of $(x^{m+k} - 1)$ so that the resulting commitments become uniformly random but the constraints are still satisfied. The proof of security for this protocol are

not necessarily complicated but long, and pass through decomposing the protocol in simpler components which assume to have access to some "ideal" trusted functionality $\mathcal{I}$ which allows to do the verified polynomial evaluations, and the resulting constants are just a matter of summing the constants of the underlying protocols (verified evaluations through polynomial commitments and permutation check).

**Theorem 5.3.1.** *There exists a parameter choice for which* Plonk *is a universal trusted setup zkSNARK in the AGM for the* Plonk *arithmetization relation with times and sizes dominated by the quantities given in Table 5.3*

| | |
|---|---|
| Prover time | $11m\ G$ |
| Proof size | $7\ G + 6\ \mathbb{F}_p$ |
| Verifier time | 2 par + $\mathcal{O}(1)\ \mathbb{F}_p + \mathcal{O}(1)\ G$ |

Table 5.3: For sizes, $G$ denotes elements of $G$ (the group for the polynomial commitment scheme) and $\mathbb{F}_p$ denotes elements of $\mathbb{F}_p$. For computational cost, $G$ denotes operations (addition and scalar-multiplication) in $G$, $\mathbb{F}_p$ denotes operations in $\mathbb{F}_p$, par notates pairings

We note that while a numerical confrontation between these protocols is possible, it is not always straightforward or absolute. As pointed out by the authors of Plonk when confronting the protocol Marlin, they note that the source for the constants are inherently different, which makes one protocol perform better or worse than the other depending on the specific problem it is being tested on. Concretely, whether or not Plonk outperforms Marlin depends primarily on the ratio between the number of multiplication gates and addition gates of the circuit. At the extremes, in circuits with almost only multiplication gates, Plonk's prover time is a $\times 2$ improvement relative to Marlin's, while Marlin wins drastically in circuits composed of almost only addition gates.

Also similarly to Marlin, the setup procedure is trusted universal, which allows to create initial partial public keys composed of powers $[1], [\tau], \ldots, [\tau^d]$ for a "big enough" $d$, chosen based on the size of the circuits that one expects to work with, and then the partial public key gets updated for the circuit specific public parameters.

## 5.4 Powers-of-tau

All of the protocols described so far are trusted setup where the motivation for the trust requirement comes from the generation of the $q$-DLOG/$q$-SBDH basis

$[1], [\tau], \dots [\tau^d]$ for a randomly uniform $\tau$. The issue which makes a trusted setup necessary is that these group elements are used to emulate the evaluations of polynomials at a "random" point: if the point appears to be random (or unknown, here) to the prover, then by the Schwartz-Zippel lemma we get guarantees on the related polynomial equality with small and bounded soundness error; otherwise, if the prover knows the value of $\tau$, the prover can give valid proofs of false equalities. For this reason, the value of $\tau$ has to be kept secret, which means that the party running the setup procedure has to delete it afterwards and make it unretrievable. At the same time, the prover and verifier have to trust that the setup party did not leak the value of $\tau$. While it is true that the value of $\tau$ must be kept unknown or it would allow the prover to prove false statements, there are improvements to the setup procedure which somewhat relax the trusted requirement.

In 2022, Nikolaenko V. et al. [30] proposed the *Powers-of-tau* ceremony, which is a multiparty computation protocol for generating SRS of type $[1], [\tau], \dots [\tau^d]$. The description of the protocol is quite simple, and basically consists of passing around the SRS and updating it each time by having the current user give a contribution to the "value of tau". The first SRS is chosen by the first user of the multiparty computation round, then at each turn

1. receive $[1], [\tau], \dots, [\tau^d]$
2. sample a random $\delta \leftarrow \mathbb{F}_p^*$
3. for $i = 0, \dots, d$ compute $[\hat{\tau}^i] := \delta^i \cdot [\tau^i]$
4. output $[1], [\hat{\tau}], \dots [\hat{\tau}^d]$

It is easy to see that the "new" value of $\hat{\tau}$ corresponds to $\delta\tau$, which is uniformly random assuming that either one of $\delta$ and $\tau$ are uniformly random.

Each contribution has to be validated to make sure that the new output truly was built on the input and not produced artificially, otherwise, the last party of the round could output manually sampled $[1], [\beta], \dots, [\beta^d]$ for some chosen $\beta$ and would then be able to prove false statements. This is done by making the contributor output also a simple zero-knowledge proof based on hash functions to show that it knows a valid $\delta$ that corresponds to the contribution. Let $\mathcal{H}$ be an ideal hash function with output in (or mapped to) $\mathbb{F}_p$, let $P_1$ be the first (non-[1]) element received by the contributor, and let $P_1'$ be the first (non-[1]) element returned by the contributor. The proof and verification go as follows

**Proof generation** :

1. sample a random $z \leftarrow \mathbb{F}_p^*$

2. let $h \leftarrow \mathcal{H}(P_1'||P_1||z \cdot P_1)$
3. return $\pi := (z \cdot P_1, \;\; z + h\delta)$

**Proof verification** :

1. parse $\pi = (\pi_1, \pi_2)$
2. let $h \leftarrow \mathcal{H}(P_1'||P_1||\pi_1)$
3. check that $\pi_2 \cdot P_1 \stackrel{?}{=} \pi_1 + h \cdot P_1$

Actually, this only shows that the contributor knows a valid $\delta$ for the update of the first element, so an additional check is required to verify that the contributed SRS composed of elements which correspond to successive powers of some base.

If the verifications pass, then it holds that the final SRS is composed of $[1], [\tau], \ldots [\tau^d]$ where

$$\tau = \delta_1 \cdot \delta_2 \cdot \; \ldots \; \cdot \delta_k$$

is defined by the product of each contribution. Additionally, as long as at least *one* of the contribution is honest, then the resulting $\tau$ is uniformly random and unknown by any party. Additionally, in order for the prover and the verifier to trust that the generated public parameters correspond to an unknown $\tau$, they require trust in only *one* of the participant, which does not have to be the same participant for both prover and verifier. Indeed, if $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ take part in the ceremony, with the prover trusting $\mathcal{B}$ and the verifier trusting $\mathcal{C}$, then even if they are not trusting the *same* party, at the end of the ceremony they will both trust its output. This is a very interesting improvement relative to the original setup procedure, which required the existance of a party trusted both by the prover and the verifier.

This makes the powers-of-tau ceremony an extremely interesting alternative in contexts where multiparty computation is already accounted for, such as the case of decentralised applications and blockchain scenarios.

# Chapter 6

# Transparent zkSNARKs

All the protocols we have discussed so far had a trusted setup procedure, whether it was strictly trusted or trusted universal. Indeed, protocols like the polynomial equality check and zero check based on the Schwartz-Zippel lemma make it simpler to design polynomial constraint system which are efficient to verify with extremely succinct proofs, and the requirement for non-interactivity can be easily be resolved by hiding the randomness in the setup process which necessarily becomes trusted in some way. However, there are more techniques for dealing with supposed randomness, or at least randomness that cannot be controlled by the prover, such as using hash functions modelled in the Random Oracle Model. This solution was already being used in part by some protocols which were designed as interactive and then made non-interactive through the Fiat-Shamir transformation [28], which in short simulates the random challenges of a verifier by creating the challenges with an (ideal) hash function in such a way that the objects against which the challenge is being made are committed *before* the challenge is created (which in practice means adding the commitments to the objects are added to the input of the hash function), so that they cannot be changed afterwards[24].

Newer research has focused on protocols which, through similar techniques to the one we just discussed, are able to work with a transparent setup process, which makes them more suitable where a trusted party is not available.

We will now analyze two of the most prominent transparent setup zkSNARKs in the current literature, the Hyrax protocol [5] and the Virgo protocol [6], both based

[24]Notice that it is not straightforward to make a transparent protocol. One might think that we could adapt the previous protocols to replace the randomness from the setup with some randomness of a hashed commitment. The issue is that as we discussed the randomness must be created *from* the commitment, and in the schemes we discussed the commitment were of the form $[p(\tau)]$ which requires to have already generated the randomness for $[\tau], \ldots, [\tau^d]$, which creates a circular dependency of commitment and randomness.

on the GKR protocol for delegated evaluation of layered arithmetic circuits.

## 6.1 Hyrax

Hyrax is a zkSNARK for layered arithmetic circuits proposed by Wahby et al. [5] with transparent setup. It is based on the GKR protocol and relies only on the Random Oracle Model and the Discrete Logarithm assumption for security. Additionally, it can be optimized for circuits with a data-parallel structure, meaning circuits with repeating blocks of a subcircuit with independent inputs, and it exposes a parameter which can be used to tune the trade off between proof size and verifier time.

Hyrax uses a novel definition of homomorphic commitment scheme which are required to be binding, hiding, and extended with the following three additional protocols

proof-of-opening : a zero-knowledge proof which allows the prover to prove that it knows a valid opening of a given commitment, without revealing the opening

proof-of-equality : a zero-knowledge proof which allows the prover to prove that two commitments $\mathsf{com}_0, \mathsf{com}_1$ open to the same value, and that the prover knows the opening

proof-of-product [25] : a zero-knowledge proof which allows the prover to prove that given commitments $\mathsf{com}_x, \mathsf{com}_y$ and $\mathsf{com}_z$, they open to values $x, y, z$ such that $z = x \cdot y$, and that the prover knows the opening

As an example, which is also the commitment scheme used by the authors in the Hyrax instantiation, the standard Pedersen commitment scheme is both homomorphic and it is shown to support these protocols.

This type of commitment scheme allows the authors to implement two improvements. First, it allows to avoid the reliance on the $q$-SBDH assumption, in favour of the weaker (meaning better) standard Discrete Logarithm assumption. Second, by implementing an additional proof-of-dot-product protocol, they can batch all the checks of the sumcheck protocol used in GKR into one check done by the prover. Additionally, since all the computation for the GKR protocol is done through the committed values and the commitment scheme is chosen to be hiding, the potential "flaw" of GKR not being zero-knowledge is automatically resolved, as the

[25] this claim can be trivially checked for the case $z = x + y$ since the commitment scheme is homomorphic.

committed values reveal no information on the original values (as they appear uniformly random) and can be simulated by a simulator.

Additionally, since this approach of sumcheck-through-commitments requires committing to the witness inputs which are needed for the final check in the GKR protocol, they improve an existing polynomial commitment scheme based on matrix representation of a polynomial to produce a polynomial commitment scheme with $\mathcal{O}(Sp)$ communication and $\mathcal{O}(Ti)$ verifier runtime where $Sp \cdot Ti = |\mathbb{w}|$, which is the source of the trade off between proof size and verifier time which mentioned before. The reason for a polynomial commitment scheme instead of the extended commitment scheme is that the authors commit to the multilinear extension $\widetilde{\mathbb{w}}$ of the polynomial which interpolates $\mathbb{w}$ over binary strings.

With all the optimizations implemented, the Hyrax protocol is a zkSNARK for layered arithmetic circuits which is secure in the Random Oracle Model under the Discrete Logarithm assumption with sizes given in Table 6.1

| | |
|---|---|
| Prover time | $\mathcal{O}(\|\mathcal{C}\|)$ |
| Proof size | $\approx (10D\log(W) + Sp)\ G$ |
| Verifier time | $\mathcal{O}(k + D\log(W) + Ti)$ |

Table 6.1: $|\mathcal{C}|$ denotes the size of the circuit, $D$ its depth and $W$ its width, $k$ is the size of the public inputs. $Sp$ and $Ti$ are tunable parameters. $G$ for a size denotes group elements in $G$ deriving from the commitment scheme

## 6.2 Virgo

In 2020 Zhang et al. [6] proposed Virgo[26], an efficient zkSNARK for layered arithmetic circuits. Similarly to Hyrax, it is based on the GKR protocol, does not require a trusted setup and results in succinct polylogaritmic proof size and verifier time. For these reasons it has been adopted in various applications.

The GKR protocol is already efficient and sound for the verification of the correct evaluation of a circuit, but it lacks some features which are required for a zkSNARK. First of all, it is interactive, however as in many of the examples we

[26]Note that this is a bit imprecise as the original authors use "Virgo" only for the C++ implementation of the proposed protocol (and not the protocol itself), which specialises the protocol to specific finite fields for efficiency on real hardware. Despite this, we will use "Virgo" for the protocol itself just to have a name to reference it instead of "the zkSNARK proposed by Zhang et al."

discussed before this can be resolved via the Fiat-Shamir transformation [28]. Then, the protocol itself is not a complete proof as it only reduce the original check to a check on the inputs of the circuit, which is not an issue in the context of delegated computation but is not compatible with the zkSNARK scheme where part of the inputs are considered part of the witness $\mathbb{w}$ hence private, which means that the verifier does not have (and must not have, for zero-knowledge) access to them or any correlated information. Lastly, at each step of the recursion in the GKR protocol, the evaluations queried by the verifier leak information on the polynomials.

The second issue is resolved through the means of what the authors define as a *zero-knowledge Verifiable Polynomial Delegation* (zkVPD) scheme. A zkVPD is similar to the definition of polynomial commitment scheme in that it allows to obtain evaluations and proof of evaluations of a committed polynomial, with two small differences: first of all, as the name suggests, it requires an additional *zero-knowledge* property which similarly to other protocols requires the existence of a simulator which produces transcripts which are (computationally) indistinguishable from real transcripts; additionally, the definition is given for an interactive protocol, though it is easy to see how it can be adapted to the definition of polynomial commitment scheme when using the Fiat-Shamir transformation, which the authors end up using for the definition of Virgo either way.

The zkVPD proposed by the authors uses a Merkle Tree for the commitment part, which, as opposed to the protocols we discussed in the trusted zkSNARK section, allows to commit to the polynomial without prior randomness. The commitment of a polynomial $f$ is done by first interpolating the coefficients of $f$ on a multiplicative subset $\mathbb{H}$ (which for a suitable finite field $\mathbb{F}_p$ can be the set of $n$-th roots of unity), masking it with a random multiple of the vanishing polynomial over $\mathbb{H}$ and then committing via Merkle Tree to the evaluations of this masked interpolating polynomial on a set $\mathbb{U} = \mathbb{L} \setminus \mathbb{H}$ where $\mathbb{L} \supset \mathbb{H}$ is a larger multiplicative subgroup. The zkVPD then allows to open to evaluations over $\mathbb{U}$ without leaking information (which would be leaked with evaluations over $\mathbb{H}$).

As for the last issue, namely the leakage of information in the evaluations of the recursive steps, the authors mask the polynomial $V_i$ of Eq. 4.2 scheme with a random multiple of the polynomial $\prod_{i=1}^{s_i} x_i(1-x_i)$, since it vanishes over the vertices of the hypercube $\{0,1\}^{s_i}$ hence does not change the result of the sumcheck, and results in a random mask elsewhere.

Putting everything together, the Virgo protocol is a zkSNARK for layered arithmetic circuits which is secure in the Random Oracle Model with sizes given in Table 6.2 (Theorem 3 of [6]).

| | |
|---|---|
| Prover time | $\mathcal{O}(\|\mathcal{C}\| + n\log(n))$ |
| Proof size | $\mathcal{O}(D\log(\|\mathcal{C}\|) + \log^2(n))$ |
| Verifier time | $\mathcal{O}(k + D\log(\|\mathcal{C}\|) + \log^2(n))$ |

Table 6.2: $|\mathcal{C}|$ denotes the size of the circuit, $D$ denotes the depth of the circuit, $n$ is the size of the total input length of the circuit and $k$ is the size of the public inputs

# Chapter 7

# ZKP libraries and DSL

While we have already discussed in depth the theoretical aspects of these protocols, we have yet to consider the availability and practicality of these in possible applications. In order to provide a complete comparison we analyse the current landscape of existing frameworks for ZKP applications.

The existing families of frameworks for ZKP applications can be divided into *libraries* and *domain specific languages* (DSL). While libraries expand existing and standardized programmin languages such as Rust or JavaScript, a *domain specific language* defines a new ad-hoc programming language dedicated solely to a specific domain, which in this case is ZKP protocols. Note that we are focusing on multi-purpose frameworks rather than application-specific frameworks, which are out of the scope of this work.

We will give e brief overview of the usage of the considered framework, as well as the list of the available ZKP backends.

## 7.1 Libraries

We start by considering libraries available for existing languages. We chose to focus on snarkjs and arkworks, which are respectively libraries for JavaScript and Rust, for two reasons: first, they are two of the leading libraries for ZKP applications; second, they are libraries for languages which sit at opposite points in portability and efficiency. On one side we have JavaScript, an interpreted language which prioritizes usability and simplicity for which a runtime exists on virtually every platform imaginable. On the other we have Rust, a compiled language with focus on rigor and memory-safety, which compiles to extremely efficient programs. Between them they cover a wide range of possible applications with different requirements, from web frontends to low level embedded systems.

### Circom + snarkjs

The snarkjs[27] library is a JavaScript library which can be used to create and validate proofs of R1CS constraints generated in Circom[28], which itself is a DSL for writing arithmetic circuits which compiles to R1CS constraint systems.

Since snarkjs works on Circom programs, we start by considering the Circom language. We note that Circom is of independent interest, and many other frameworks have a compatibility layer with it.

In Circom, circuits and subcircuits are defined as *templates*. A template defines internally some *signals* corresponding to the wires of the circuit, with the possibility of adding `input` and `output` keywords to specify the inputs and outputs of the circuit. Signals can then be receive assignment values through the assignment operator `<==` which automatically adds the relative constraint in the compiled result. Additionally, one can use the `<--` operator to define operations which do not automatically convert to R1CS constraint, though it should be used with care exactly for this reason.

With a valid `.circom` file defining a circuit one can use Circom to compile it into a `.r1cs` file that represents the circuit satisfiability constraint in R1CS form. Having the `.r1cs` file, one can use the snarkjs library to create a ZKP system for this constraint. The snarkjs library can be used both as a library inside a runtime such as NodeJS or a browser, as well as a standalone binary which can be installed through npm. Either way, the process then starts with creating public parameters for the protocol. The snarkjs library allows to use Groth16, Plonk or FFlonk (a variant of Plonk) as backends for the ZKP protocol, all of which require a trusted setup for the public parameters, by generating values $[1], [\tau], \ldots, [\tau^d]$. This is done through the Powers-of-tau procedure detailed in Section (5.4). An additional step for the public parameters generation is required when using Groth16 since the setup in this protocol is not universal.

The prover has then to create the extended form of the witness based on the valid inputs (see the end of the Subsection 3.2.2 for any doubts about the need of an extended witness), and can then select any of the exposed protocols to generate the proof of the satisfiability of the constraint for the created witness with the chosen ZKP backend. A verifier can then take the verification key generated with the public parameters and the proof to check its validity.

We note that snarkjs allows to export the verifier process as a Solidity smart

[27] `https://github.com/iden3/snarkjs`
[28] `https://docs.circom.io/`

contract, used for the Ethereum blockchain and EVM-compatible blockchains. This allows the development of decentralised applications involving ZKP systems through snarkjs

#### arkworks

arkworks[31] is a Rust library, or more precisely an ecosystem of libraries, which exposes interfaces for zkSNARKs as well as the necessary cryptoraphic primitives for building zkSNARKs, such as finite field and elliptic curves arithmetics and R1CS constraints.

The arkworks ecosystem exposes different zkSNARK proving systems, such as Groth16, Marlin and gm17 (the protocol inspired on Groth16 which solves the malleability through simulation extractability), as well as other non-generic proof systems that do not target R1CS or circuit contraints, such as inner pairing product arguments.

The implementation of ZKP protocol is certainly not as straightforward as the case of snarkjs, particularly because it varies slightly depending on the chosen proving system. As for the positives, it offers the security of use of a strongly typed language, as well as the compile time guarantees of memory safety that come with Rust.

Overall, writing a ZKP application in arkworks requires a deeper knowledge of the chosen proving system than a similar implementation using snarkjs, but it allows the developer to write the implementation and the constraint system in the same language of the rest of the application (with access to Rust's powerful macro system for advanced developers) instead of relying on the external language Circom.

## 7.2 Domain Specific Languages

We now analyse *domain specific languages* (DSL) for ZKP and zkSNARK applications. The advantage of these DSL is that, by being developed for a specific domain instead of being general-purpose language, usually simplify the process of integration of ZKP protocols relative to a similar implementation with a library for an existing language, reducing or eliminating the required knowledge of the involved cryptographic protocols. On the other side, these are completely new and separate languages, which comes with some downsides. First of all, they require learning a new environment, which while simplified might not be straightforward. Additionally, by the nature of being separate language, these DSL solutions re-

quire additional compatibility layers in order to be used with other systems, which are necessarily required since the specificity of the language prevents it to be used for anything other than the ZKP protocol implementation. Lastly, by being separate languages they do not come with the environment and tools which existing languages have already.

Despite the downsides, DSL for ZKP are a great option in implementing ZKP applications, given that they are build precisely for this purpose.

### ZoKrates

ZoKrates[29] is a domain specific language with a Rust-like syntax for developing ZKP applications and deploying them through Solidity verifiers, for Ethereum blockchains and EVM-compatible blockchains.

ZoKrates uses by default the arkworks ecosystem under the hood, meaning that it has access to the same ZKP backends, namely Groth16, Marlin and gm17. Users can also specify to use bellman (another Rust library for ZKP, though less developed) as different backend through command line arguments, but this restricts the choice of protocols to only Groth16.

Defining a constraint system to be used in ZoKrates requires only defining a `main` function with public and private inputs. The constraints are enforced by the use of `assert` commands which take in input a boolean result (for example obtained by checking that the result of a computation is equal to some other value) and assert it to be true.

An interesting but complicated restriction in writing a constraint in ZoKrates is that while execution is allowed to branch (with `if/else` statements), both branches are always executed, and only the "correct" branch is used for the return value. The reason behind this restriction is the internal circuit conversion, which does not allow conventional branching flow and must resort to arithmetic tricks to mimic conditional branching, such as multiplying by zero the result of the undesired branch.

In ZoKrates, many cryptographic primitives such as common hash functions are already implemented, further simplifying the development of ZoKrates programs. It also offers the same Powers-of-tau ceremony for the generation of the public parameters for the protocols.

[29] `https://zokrates.github.io/`

Lastly, while ZoKrates is mainly a DSL, it can also be used as a library for JavaScript throught the zokrates-js package. We note that the development of this library seems to have stopped two years ago with the last version being `1.1.9`.

### Noir

Similarly to ZoKrates, Noir[30] is a domain specific language with a Rust-like syntax, with a focus on deploying applications to on-chain environments. Similarly to ZoKrates, it also comes with the JavaScript+TypeScript library noir_js, with a much more active development though still in beta. This JavaScript library allows developers to deploy Noir programs on different environments such as web applications.

The main strength of Noir is its extensibility: while other libraries and languages have a fixed set of available ZKP protocols, Noir is backend-agnostic, meaning that it has support for third-party implementations of the ZKP backend. Noir defaults to its first-party backend, Barretenberg, which uses variants of Plonk, namely MegaHonk, UltraHonk and CHONK (a Client-side Highly Optimized variant). We note that the Aztec Labs company, which develops Noir and Barretenberg, is also the one which developed the original Plonk protocol.

As for the extensibility, the environment of third-party backends is mainly community-driven and not particularly rich, though there exists a curated list[31] on GitHub with references for third-party backends, as well as many more useful resources.

Finally we note that differently from ZoKrates, Noir allows deployment both through a Solidity verifier, which can be used on Ethereum or EVM-compatible blockchains, as well as through an Aztec smart contract for the Aztec Network.

A comparative summary of the analyzed frameworks is provided in Chapter 8.

[30] `https://noir-lang.org/`

[31] `https://github.com/noir-lang/awesome-noir?tab=readme-ov-file#proving-backends`

# Chapter 8

# Comparison of frameworks and protocols

We now give an in-depth comparison of the analysed protocols, focusing both on theoretical and practical aspects. The protocols which will be compared are the protocols analysed in Chapters 5 and 6, namely Groth16, Plonk, Marlin, Hyrax and Virgo.

We start from a practical point of view. We list in Table 8.1 the results discussed in Chapter 7.

Note that only some of the considered protocols, particularly only the trusted setup ones, have some form of representation in the existing libraries. This is almost true even for Noir: among the available third-party backends in the curated list[32] only one of them implements a ZKP protocol which does not require a trusted setup; it is the ProveKit backend[33] by the World Foundation which implements the Spartan zkSNARK [32], which requires no trusted setup.

There are several plausible reasons for this absence of transparent protocols in the current ecosystem. First of all, the developments on transparent zkSNARKs are more recent than the initial work on trusted setup zkSNARKs, so it is possible that the ecosystem simply had not enough time to adapt. The presence of at least one third-party backend for Noir with a zkSNARK which does not require a trusted setup could be signalling a change in this direction. The second reason is a *computational vs security* trade-off: generally speaking, trusted setup protocols usually have either a lower setup cost or lower prover runtime than transparent protocols, at the cost of a trusted environment; however, given the existance of the

[32] https://github.com/noir-lang/awesome-noir?tab=readme-ov-file#proving-backends
[33] https://github.com/worldfnd/ProveKit

| | Type | Best use case | Requires knowledge of protocols | Supported protocols | Maturity |
|---|---|---|---|---|---|
| snarkjs | Library (JS) | Web applications<br>Fast prototyping<br>On-chain integration | No | Groth16<br>Plonk<br>FFlonk | High |
| arkworks | Library (Rust) | Backend systems<br>High-performance applications<br>Custom implementations | Somewhat | Groth16<br>Marlin<br>gm17 | High |
| ZoKrates | DSL | On-chain applications<br>Quick development without deep cryptographic knowledge | No | Groth16<br>Marlin<br>gm17 | High |
| zokrates-js | Library (JS) | * | No | Groth16<br>Marlin<br>gm17 | Stale |
| Noir | DSL | On-chain applications<br>Modular systems<br>Extensible architectures | No | Plonk variants | High |
| Noir_js | Library (JS) | * | No | Plonk variants | High |
| Noir + 3rd party backend | DSL | Experimental setups<br>Advanced users needing specific backends | No | * | Medium/low |
| Noir_js + 3rd party backend | Library (JS) | * | No | * | Medium/low |

Table 8.1: Overview of the analysed libraries and DSL.

Powers-of-tau setup procedure and the fact that many of these libraries natively target decentralized environments, it is quite tempting to choose a more efficient trusted setup protocol and solve the trust requirement in multi-party computation, requiring only one of the many parties involved to be trusted. Note that this is an incomplete security argument, as the difference between a trusted and a transparent protocol can be in the model used for security: as we will discuss more in details below, some models are considered more standard and tested than others, which can give more or less reassurance on the security of the chosen protocol.

Another obstacle in choosing the correct protocol for a given application is that, as we have seen, even the trusted setup protocols are not all available in all the frameworks, and not all the frameworks have the same purpose or target. ZoKrates and Noir are mainly DSL targeting decentralized or on-chain applications, with the additional possibility of deploying on a JavaScript runtime with a JavaScript library which is more or less maintained depending on the DSL. snarkjs and arkworks, on the other side, have the primary focus of being libraries for generic purpose appli-

cations in different environments. This means for example that in implementing an application not targeting on-chain environments, such as a frontend-backend webapp with SSI authentication or a service provider providing delegated verifiable computation, the set of available zkSNARKs is restricted to the ones offered by the libraries. This could be an issue if the application requires the usage of a transparent zkSNARK for security reasons, which right now is only *kind of* possible through Noir.

We now take into consideration the theoretical aspects of each protocol. We summarize the computational costs of each protocol in Table 8.2, and the properties of each protocol in Table 8.3. We also report the empirical computational costs of the available protocols when instantiated over a matrix multiplication circuit through the frameworks in Table 8.4, as reported in [8].

| | Prover time | Proof size | Verifier time |
|---|---|---|---|
| Groth16 | $\mathcal{O}(n+m)$ | $2\ G_1 + 1\ G_2$ | $\mathcal{O}(k)$ op + 3 pair |
| Marlin | $\mathcal{O}((h+\mathsf{b}\log(h+\mathsf{b}))$ | $13\ G_1 + 8\ \mathbb{F}_p$ | $\mathcal{O}(k+\log(h))$ |
| Plonk | $\mathcal{O}(m)$ | $7\ G + 6\ \mathbb{F}_p$ | $\mathcal{O}(1)$ op + 2 pair |
| Hyrax | $\mathcal{O}(\lvert\mathcal{C}\rvert)$ | $\mathcal{O}(D\log(W)+Sp)$ | $\mathcal{O}(k+D\log(W)+Ti)$ |
| Virgo | $\mathcal{O}(\lvert\mathcal{C}\rvert+n\log(n))$ | $\mathcal{O}(D\log(\lvert\mathcal{C}\rvert)+\log^2(n))$ | $\mathcal{O}(k+D\log(\lvert\mathcal{C}\rvert)+\log^2(n))$ |

Table 8.2: Overview of computational costs of the protocols Groth16, Marlin, Plonk, Hyrax, and Virgo. For the meaning of the constants, refer to the respective sections.

| | Setup | Security assumption | Security model | Post-quantum |
|---|---|---|---|---|
| Groth16 | Trusted | $q$-SBDH | GGM | No |
| Marlin | Trusted universal | $q$-SBDH | AGM + ROM | No |
| Plonk | Trusted universal | $q$-SBDH | AGM + ROM | No |
| Hyrax | Transparent | DL | ROM | No |
| Virgo | Transparent | Hash | ROM | Plausibly |

Table 8.3: Overview of computational properties of the protocols Groth16, Marlin, Plonk, Hyrax, and Virgo.

### Setup procedure

The most evident difference between these analysed protocols is in the setup procedure. The protocols can be categorized as follows: Groth16 has the most restricting

| | Setup runtime (ms) | Prover runtime (ms) | Communication & proof size | Verifier runtime (ms) |
|---|---|---|---|---|
| Groth16/snarkjs | 3113 | 1410 | 802 B | 804 |
| Groth16/arkworks | 31 | 45 | 128 B | 2 |
| Groth16/ZoKrates | 609 | 622 | 128 B | 310 |
| Marlin/arkworks | 423 | 403 | 951 B | 25 |
| Plonk/snarkjs | 190632 | 282897 | 2 KB | 686 |
| "Plonk"/Noir | - | 6972 | 2 KB | 6037 |

Table 8.4: Empirical computational costs of the protocols Groth16, Marlin, and Plonk, when instantiated over a matrix multiplication circuit through different frameworks, as reported in [8]. The prover runtime of Noir includes the runtime of the setup procedure.

setup procedure, being trusted and not universal; Marlin and Plonk require a universal trusted setup, which allows to update the public parameters for each new circuit; Hyrax and Virgo have a transparent setup. This means that in contexts where the proof circuits change or are updated often, it might be undesirable to use Groth16 since it requires the existence of a trusted third party for each circuit update. Conversely, the other protocols require a trusted third party either just once (up to a given circuit size limit) or never at all. We keep in mind that the existance of the Powers-of-tau protocol can serve to simplify the requirement of a trusted setup for these protocols.

### Computational costs

We now compare the computational costs. Looking at the results from [8] reported in Table 8.4, we notice that the empirical costs are naturally influenced both by the properties of the protocol and the nature of the chosen framework.

While theoretically all the protocols have a prover runtime which is linear in the circuit size[34], the actual costs differ of multiple order of magnitudes. On one side, this is the understandable effect of the hidden constants of the $\mathcal{O}$ notation. On the other side, however, such differences are present even in implementation that should be theoretically identical, and differ only in the implementation language: looking at the prover runtime of Groth16 (the most represented protocol over the frameworks), and considering the arkworks implementation as the baseline, we get a $\approx \times 13$ runtime increase in ZoKrates, and a $\approx \times 31$ runtime increase in snarkjs.

[34]This argument is a bit more complicated for Groth16 and Marlin which are not based on circuit relations, rather QAP or R1CS deriving from circuit relations, although as we have discussed in a previous chapter they have size which is linear in the original circuit.

This has two important implications in our comparison.

First of all, we see that efficiency-wise the choice of framework is just as important (if not more important) as the choice of protocol, as the implementations which should be equivalent end up having wildly different runtimes and costs depending on the language or DSL. This is surprisingly true not only for the runtimes, but also for the sizes: a Groth16 proof is only 128 bytes in arkworks, while 802 bytes in snarkjs, which is a $\times 6$ increase (it should not surprise that Groth16/arkworks and Groth16/ZoKrates have the same proof size with different runtimes, as ZoKrates uses arkworks under the hood, meaning an additional runtime overhead but same proof serialization).

Secondly, it means that it becomes more difficult to make a "fair" comparison between protocols which have not been implemented in the same framework: for example, from the theory we see that a Marlin proof should larger than a Plonk proof, with twice the group elements and extra field elements; however, since Marlin is only implemented in frameworks which *do not* implement Plonk, the only points of comparison available become the arkworks implementation of Marlin and the snarkjs implementation of Plonk, resulting in a Marlin proof size of only 905 bytes, roughly $\times 2$ smaller than Plonk's proof size over snarkjs. At the same time we should keep in mind that the Plonk/snarkjs case has size overhead given by snarkjs, similarly to the Groth16/snarkjs case: if we suppose a similar $\times 6$ overhead, we get that a theoretically equivalent implementation of Plonk over arkworks *might* have a proof size of $\approx 359$ bytes, which would be a $\times 2.65$ *decrease* in comparison to Marlin and closer to what we would expect theoretically.

As for Hyrax and Virgo, we can only discuss the computational costs in theory, as they have no representation in the frameworks. From the theory we get that they also have linear or quasi-linear prover runtime, and they both have polylogarithmic proof size and verifier time. While this is still a succinct result, it is a downgrade when compared to the constant sizes of the other three protocols.

In summary, given the limited representation of protocols in the frameworks, the practical computational comparison does not strictly reflect the results from the theory. The most efficient solution both in runtime and size is Groth16/arkworks winning by at least an order of magnitude over all the other available solutions. As for the rest, we can see a trend of increasing computational costs, with arkworks being the most efficient solution available, followed by ZoKrates with an order of magnitude increase, followed by Noir, and then snarkjs losing in all comparisons. Protocol-wise, the results confirm Groth16 being the most efficient independently of the chosen framework, again by at least an order of magnitude in all the metrics, with a not so clear empirical comparison between Marlin and Plonk. We also recall

the result from [4] which shows that the prover runtime comparison of Marlin and Plonk is complex, as it heavily depends on the nature of the considered circuit, and should be done on a case-by-case basis for practical results.

### Usability

Regarding usability, we note that the protocols are not equivalent, with the main difference being that Hyrax and Virgo work only for layered circuits, rather than generic circuits. We noted that any arithmetic circuit can be given layered structure at the cost of additional gates, which in turn increases the computational costs of the protocol. However, for circuits which are naturally layered and have a parallel structure (that is, they are composed of parallel identical subcircuits), Hyrax becomes an interesting choice since it can be highly parallelized to increase its efficiency.

### Security assumptions

Finally, we discuss the security assumptions and guarantees of the considered protocols. The main distinction between these protocols is the models in which the security proofs are done, which are the Generic Group Model with the $q$-SBDH assumption for Groth16, the Algebraic Group Model (plus the Random Oracle Model necessary to guarantee security with the Fiat-Shamir transformation to make the interactive protocols non-interactive) with the $q$-SBDH assumption for Marlin and Plonk, the Random Oracle Model with the DL assumption for Hyrax and just the Random Oracle Model for Virgo. These models and assumptions have different security implications.

Regarding Groth16, the GGM is a stronger assumption compared to the AGM, making Groth16 theoretically weaker compared to Marlin and Plonk which require only the AGM. At the same time, there is an argument to be made against Marlin and Plonk since they require the Fiat-Shamir transformation in order to be made non-interactive, and the discussion on whether the Fiat-Shamir transformation is secure or not is a delicate argument. On one side, Goldwasser & Kalai [33] proved that the Fiat-Shamir transformation *can* be insecure in some ad-hoc protocols. At the same time, a more recent result [34] shows that a strong variant of the Fiat-Shamir (which uses not only the transcript but also the statement in generating the challenge) can provide stronger security. Finally, there is a theoretical result [35] showing the impossibility of a security proof for the Fiat-Shamir transformation as a black-box on a particular set of interactive protocols. Despite these results, the Fiat-Shamir transformation is an accepted and industry standard method and implementations of hash functions used for this transformation are under extreme scrutiny, making the previous considerations relevant only for applications where

the security is *critical*.

At the same time, the AGM is still a somewhat discussed model for security, making the transparent protocols an interesting option in respect to Marlin and Plonk. Additionally, Hyrax requires only the DL assumption, which is both well studied and weaker than the $q$-SBDH, and Virgo does not even require the DL assumption, relying only on the ROM and hash functions for security. These facts make Hyrax and Virgo theoretically more secure than Groth16, Marlin and Plonk.

We recall that Groth16 has an additional property, namely the malleability of its proofs, which can be either a feature or a security issue depending on the context. The malleability can be used to re-randomize existing proofs, giving additional unlinkability with respect to the prover. At the same time, this property can be used by an attacker to create new proofs of the same statement, giving the illusion that the attacker knows the witness. Whether this is an issue or not depends on the scenario, mainly depending on whether it is possible and makes sense to accept multiple proofs of a single statement, or if each statement is globally accepted at most once. If this were to be an issue, it can be either solved while keeping Groth16 with the additional usage of a signature scheme.

### Post-quantum safety

There is also an argument to be made about post-quantum safety of these protocols. Groth16, Marlin, Plonk, and Hyrax all rely either directly or indirectly on the Discrete Logarithm assumption, which is not post-quantum secure. This is an important consideration for contexts where *harvest now, decrypt later* (HNDL) attacks are a reasonable possibility: while the discussion on *when* and *how* quantum computers will become a reality is both technical and uncertain, critical applications with a reasonable potential interest in HNDL attacks by adversaries for the duration of the next decades should at least keep this consideration in mind. On the other side, Virgo relies only on the ROM and hash functions, which are believed to be quantum-resistant, making it an interesting alternative.

### Takeaways

- **Trusted setup** vs **transparent** protocols is the primary design trade-off. Trusted setup protocols (Groth16, Marlin, Plonk) offer significantly better performance in terms of proof size and verification cost, while transparent protocols (Hyrax, Virgo) remove trust assumptions at the cost of higher computational overhead and larger proofs.
- **For computational costs**, we see that the **most relevant factor is the choice of framework**, as equivalent implementation of a given protocol

end up having wildly different runtimes and sizes depending on the chosen framework.

- There exists a natural **trade-off between trusted setup and performance**. Groth16 achieves constant-size proofs and very efficient verification, making it particularly suitable for applications such as blockchain verification, but requires a circuit-specific trusted setup, which may be impractical in dynamic settings. Marlin and Plonk rely on universal trusted setup, which allows reuse across multiple circuits, offering a good compromise between efficiency and flexibility. Their relative performance depends on the structure of the circuit and should be evaluated case by case.

- **Transparent protocols offer stronger security guarantees but lower efficiency**. Hyrax and Virgo rely on weaker and more standard assumptions (e.g., DL or ROM) and avoid trusted setup entirely, but produce larger proofs and incur higher verification costs, making them less suitable for performance-critical applications.

- The current **ecosystem strongly favours trusted setup protocols**, impacting tools and protocol choices. Most existing libraries and DSLs primarily support Groth16 and Plonk-like systems, while support for transparent protocols is limited and often experimental, which can constrain practical choices. Even when a protocol is theoretically suitable, it may not be available in existing frameworks or may require significant implementation effort, making the **choice of library or DSL a critical factor**.

- Security considerations depend on both assumptions and models. Pairing-based protocols rely on stronger assumptions (e.g., q-SBDH) and specific models (GGM/AGM), while transparent protocols rely on more standard assumptions, though often within the Random Oracle Model. The practical impact depends on the security requirements of the application.

- Application requirements should drive protocol selection. Scenarios with strict performance constraints (e.g., on-chain verification) tend to favour succinct zkSNARKs, while applications requiring minimal trust or long-term security may benefit from transparent constructions.

# Chapter 9

# Guidelines for adopting ZKPs in different domains

We now give guidelines, as well as an operative decision table, for the choice of protocols and frameworks for the implementations of ZKP systems into real applications. We note that not all the possible scenarios are best fitted to the use of these guidelines, as some use cases like applications to *cross-chain communication* or *verifiable machine learning* have been more developed than others, with specific and optimized solutions already existing in the literature. We will discuss these solutions at the end of the chapter. As for the considered cases, instead of focusing on the specific use cases we structure the guidelines around the *characteristics* of each scenario, to make these considerations applicable to a wider range of scenarios.

## 9.1 Parameters for the analysis

We consider the following parameters which characterise each possible use case. The parameters are grouped into a first set of parameters concerning the security context, and a second set of architecture context.

1. **Security requirements:** how *critical* would a system failure be?
2. **HNDL attacks:** is the scenario susceptible to *harvest now, decrypt later* attacks?
3. **Replay attacks:** is the scenario susceptible to attacks where an adversary might reuse a valid proof, or a modified version of a valid proof?
4. **Trusted party:** is there a third party that every other involved party trusts?
5. **Open vs closed environment:** are all the parties involved known a priori and limited in number? Or is this set subject to changes in time?

6. **Network structure:** does the context naturally predispose a network of parties, which might facilitate multiparty computation protocols?
7. **Circuit renovation:** does the application need to prove and verify only a fixed set of circuits, or does the set of involved circuits change over time?
8. **Computation constraint:** are any of the parties involved in the protocol computationally constrained?
9. **Communication constraint:** is there a limit on the maximum allowed communication for the protocol?
10. **Target environment:** how and where is the protocol going to be executed?

We give some concrete examples that will help better understand these analysis criteria. Note that while the security level is possibly the most important parameter in the choice of framework and protocol, it can be only correctly assessed on the specific context of the use case (taking the first of the following examples, the criticality of a system failure varies drastically if we're talking about a frontend for a banking system versus a frontend for a social media). For this reason, the evaluation of this parameter will be marked as "depends on the context" in all the considered scenarios. We stress the importance of a correct assessment of this parameter, and that this omission is done *only* because a correct evaluation of the security requirements can only be done (and must be done) on a specific context-by-context basis.

First, we consider the case of an authentication protocol for Self-Sovereign Identity of a decentralised webapp. This scenario has an user playing the role of the prover, proving to a server playing the role of the verifier that they indeed hold the claimed identity (which could be bound, for example, to the knowledge of the preimage of a given hash image). It is characterised as in Table 9.1.

Then we consider the case of verifiable machine learning / verifiable inference, in which a service provider offers inferences on a private neural network to a customer. Here, the customer would like to verify that indeed the inference was done on the specified model (instead of, for example, a cheaper model to save on costs) and the service provider wants the model to remain private, hence the service provider plays the role of the prover and the customer plays the role of the verifier. This scenario is characterised as in Table 9.2.

Lastly, we consider the example of the deployment of some smart contract (for example via a Solidity verifier, available through most of the considered frameworks) to a permissioned private blockchain, such as a chain used to document a producer's supply chain. This scenario is characterised as in Table 9.3.

| | |
|---|---|
| **Security requirements** | Depends on the context. |
| **HNDL attacks** | The scenario is possibly susceptible to HNDL attacks, given that a decrypt could reveal the authentication secrets of the user, which might permit other types of attacks on future data. |
| **Replay attacks** | The scenario is susceptible to replay attacks, with an adversary listening to an authentication proof and trying to replay it to authenticate as the prover user. |
| **Trusted party** | We cannot assume the existence of a trusted third party. |
| **Open vs closed environment** | Not all the possible provers with identities nor all the possible verifiers are known a priori. |
| **Network structure** | Depends on the type of decentralised app. |
| **Circuit renovation** | We can assume that the verification circuit is chosen once in the development of the application and remains fixed. |
| **Computation constraint** | The prover is run by a frontend client, possibly on a mobile device, hence it must be assumed to be computationally constrained. The verifier is run by a server which can be assumed to be computationally powerful. |
| **Communication constraint** | The communication has a relatively low cost, hence does not require *extreme* efficiency as in other scenarios. |
| **Target environment** | The prover algorithm is going to be executed in a browser runtime. The verifier algorithm runtime environment is more flexible and depends on the chosen backend. |

Table 9.1: Characterisation of the SSI authentication use case

## 9.2 From parameters to guidelines

We now analyse how the characterising parameters of an use case affects the choice of framework and protocol. We note that each parameter has its own implications, which in particular might contrast the implications of a different parameter. This makes it impossible to give a single global guideline on the choice of framework and protocol. In case of conflicting implications, the choice of which of these should win depends on the priority of each parameter, framework and protocol determined by the developer, which ultimately depends on the nature of the use

| | |
|---|---|
| **Security requirements** | Depends on the context. |
| **HNDL attacks** | The scenario is less susceptible to HNDL attacks, as a decrypt would reveal the weights of a by-then-obsolete model. |
| **Replay attacks** | The scenario is not susceptible to replay attacks, as each proof targets a different inference result and has no meaning on other inference results. |
| **Trusted party** | We cannot assume the existence of a trusted third party. |
| **Open vs closed environment** | Not all the verifiers are known a priori. |
| **Network structure** | The scenario does not have a network structure. |
| **Circuit renovation** | The circuit is fixed to one per model. |
| **Computation constraint** | The prover is run by a server which can be assumed to be computationally powerful. The verifier is run by a frontend client, possibly on a mobile device, hence it must be assumed to be computationally constrained. |
| **Communication constraint** | The communication has a relatively low cost. |
| **Target environment** | The prover algorithm runtime environment is flexible, and depends on the service provider architecture. The verifier algorithm is most likely going to be executed in a browser environment, or a native app. |

Table 9.2: Characterisation of verifiable inference use case

case. The earlier chapters of this work, other than supporting the claims done in the analysis section, should also serve to give a better understanding of the frameworks and protocols for an educated choice in case of conflicts.

We divide the guidelines on each of the analysis parameters, making considerations for each of these, analysing the possible “values” of each parameter.

| | |
|---|---|
| **Security requirements** | Depends on the context. |
| **HNDL attacks** | Depends on the context and the nature of the smart contract. |
| **Replay attacks** | Depends on the context and the nature of the smart contract. |
| **Trusted party** | Depends on the context, but there might be a trusted third party. |
| **Open vs closed environment** | All the provers and verifiers are known a priori. |
| **Network structure** | The scenario has a natural network structure. |
| **Circuit renovation** | The circuit is changed for each smart contract. |
| **Computation constraint** | The prover is run off-chain hence it can be assumed to be computationally powerful. The verifier is run on-chain hence it must be assumed to be extremely computationally constrained. |
| **Communication constraint** | The communication and proof end up on-chain, hence must be extremely efficient. |
| **Target environment** | The prover algorithm can be executed off-chain, the verifier algorithm must be executed on-chain hence must be available as a smart contract or solidity verifier. |

Table 9.3: Characterisation of smart contract use case

### Security requirements

The analysis of the security requirements of the use case is the most important parameter assessment. Each protocol naturally requires some security assumptions and models which under which the security holds, meaning that each protocol has an implicit security risk. This risk varies depending on the strength and the standard-ness of the assumptions and models required by the protocol. For example, Groth16 is secure under the $q$-SBDH assumption in the Generic Group Model, both of which are stronger (meaning worse) and less standard than the DL assumption and the Algebraic Group Model used by other protocols. This means that for example Marlin and Plonk are *theoretically* more secure than Groth16.

As discussed in Chapter 8, the security assumptions go from stronger (worse) to

weaker (better), and from less standard to more standard, in the order in which we analysed the protocols: Groth16, Marlin, Plonk, Hyrax, Virgo. This means that for critical infrastructure, the best option would be to go with protocols like Hyrax or Virgo, however as we noted in Chapter 8 these options are lacking in the current available frameworks. This means that the only *ready-to-use* solutions available are, from more secure to less secure, Marlin/Plonk and Groth16. For critical applications that require the most standard assumptions and models, the arkworks exposes the cryptographic tools for implementing new proof systems, and Noir allows the use of third-party backend, which allow the implementation of not-available proof systems.

We refer to Chapter 8 for a more in-depth comparison of the security assumptions and models.

### HNDL and replay attacks

We start by considering the case of an HNDL-susceptible scenario. The main risk in this scenario comes from using protocols which are assumed to be secure with the current developments, but could turn out to be secure in the future. The first possibility is a protocol which was not secure to begin with, depending on an insecure assumption. For this possibility, we repeat the discussion done for the more general *security requirements* which can mitigate the risk of an insecure assumption. The additional risk which becomes more relevant for HNDL attacks is the usage of protocols which are secure but not quantum-resistant: for critical applications with an estimated duration of decades, it is worth considering the usage of a quantum-resistant protocol. Of the analysed protocols only Virgo, which does not depend on the DL assumption, is plausibly quantum-resistant, while the other all depend on assumptions which have a quantum attacks which are more efficient than the corresponding classical attacks.

This presents a similar issue to the one encountered for the general *security requirements* assessment, that is the absence of plausibly quantum-resistant protocols in the available frameworks. The only available solution, in this case, is an implementation of a protocol like Virgo with the available tools discussed above.

The possibility of replay attacks is more easily resolved: of the considered protocols, the only one with malleable proofs (which makes it trivial to regenerate proofs) is Groth16. For an use case which is susceptible to replay attacks, it is advisable to use one of the other available protocols. If this is not possible (for example in scenarios with a heavy computation and communication constraint), this can also be resolved by also implementing an additional signature scheme for

the generated proofs.

### Trusted party, open vs closed environment, network structure, and circuit renovation

All the properties listed for this section have direct implications on the choice of type of protocol, regarding the setup process. We recall that the setup process can be organised into *trusted* (Groth16), *trusted universal* (Marlin and Plonk) and *transparent* (Hyrax and Virgo).

If the context includes the presence of a third party which is trusted by all other involved parties, the most efficient solution is to use a trusted or trusted universal protocol with the setup procedure executed by the trusted party. However, this is not likely in the case of open environments, in which parties can join at any time, making it difficult to establish a globally trusted party. As we discussed in Chapter 5, a solution which reduces the trust requirements is the execution of the setup procedure through the Powers-of-tau which most of the considered frameworks expose. This is particularly practical in closed environments without a globally trusted party, by making each party participate in the Powers-of-tau ceremony, or in network-structured environments, which naturally facilitate multiparty computation protocols.

Lastly, we recall that the issue of a trusted party (or semi-trusted network for Powers-of-tau) can be reduced to a one-time issue by using a trusted *universal* protocol, which allows to generate public parameters in a trusted environment once, and then regenerate them for each new circuit in a trustless environment.

For contexts without a globally trusted party which do not naturally allow the execution of Powers-of-tau or with a high circuit renovation frequency, it would be suggestible to use a transparent setup protocol, though we remark that as of today the ecosystem of ZKP frameworks has no mature solutions for transparent protocols.

### Computation constraint, communication constraint, and target environment

We start by considering whether or not the prover or the verifier are computationally constrained. As we have discussed in Chapter 8, given the low diversity of available protocols in the current frameworks, the most influential choice for computational aspects is the choice of framework: arkworks is the most efficient

among all the considered frameworks in all the metrics by at least one order of magnitude, followed by ZoKrates which uses arkworks under the hood, then Noir, and finally snarkjs. We also note that all the frameworks except arkworks either compile to an on-chain verifier (which might not be adapt given the context) or run as a JavaScript library, which has more overhead than a compiled Rust binary. Additionally, arkworks exposes both Groth16 and Marlin (as well as gm17) as available backends, making the efficiency of the program more fine-tunable with the choice of the backend.

As for the protocols, Groth16 is the most efficient of all, followed by Marlin and Plonk, followed by Hyrax and Virgo. We recall that Hyrax has a fine-tunable parameter to decrease or increase the verifier runtime at the cost of increasing or decreasing the proof size. This means that for a computationally constrained prover or verifier, the better choice would be one of the trusted setup protocols. We also recall from Chapter 8 that the computational comparison between Marlin and Plonk is not trivial, and is better done on a case-by-case basis.

As we have discussed in Chapter 8, all the protocols have prover runtime linear in the size of the circuit, though with slightly different constants. Vice versa, the verifier runtime varies differently: Groth16 is the most efficient of all, followed by Marlin and Plonk, followed by Hyrax and Virgo. We recall that Hyrax has a fine-tunable parameter to decrease or increase the verifier runtime at the cost of increasing or decreasing the proof size. This means that for a computationally constrained verifier, the better choice would be one of the trusted setup protocols, and for a computationally constrained prover there is less of a tradeoff between the available protocols. We also recall from Chapter 8 that the computational comparison between Marlin and Plonk is not trivial, and is better done on a case-by-case basis.

As for the frameworks, the most efficient solution is to use the arkworks Rust library, which compiles to optimizable binaries. All the other options either compile to an on-chain verifier (which might not be adapt given the context) or run as a JavaScript library, which has more overhead than a compiled Rust binary. Additionally, arkworks exposes both Groth16 and Marlin (as well as gm17) as available backends, making the efficiency of the program more fine-tunable with the choice of the backend.

We now consider whether or not the communication is constrained. Note that since the proof is part of the transcript, the communication size is impacted (mainly) by the size of the proof of the chosen protocol. We recall that in scenarios which target blockchain applications, where the proof has to be stored on-chain, communication is severely constrained.

For the choice of the protocol, the discussion for communication-constrained systems is analogous to the discussion for computationally constrained systems, given that in the available protocols the verifier runtime and the proof size are somewhat correlated (this should not surprise as the verifier has to at least read the proof, giving a lower bound to the verifier runtime based on the proof size). The most efficient solution remains Groth16 over arkworks, with an impressive proof size of 128 bytes. Equivalent results are achieved by Groth16 over ZoKrates, followed by Groth16 over snarkjs, then Marlin over arkworks, and lastly the Plonk (or variations of Plonk) implementations of snarkjs and Noir. Lastly, Hyrax and Virgo have a theoretical proof size which is polylogarithmic, instead of the constant size achieved by the other protocols, and comparable between each other.

Given that most of the considered frameworks allow the export of some sort of smart contract verifier, either via a Solidity verifier or an Aztec contract, the restriction of an on-chain target environment has the least implications on the choice of frameworks, excluding only arkworks. As for the choice of protocol, since on-chain target environments are typically communication-constrained environments, the same considerations for the communication-constrained systems hold. While arkworks is the most efficient framework, the choice of ZoKrates results at least in equivalent proof sizes, while also being available on-chain.

In summary, there is a natural tension between the security requirements and the efficiency requirements of the system, which must be assessed on a case-by-case basis. Additionally, we remark that in the current ecosystem of ZKP frameworks the security-heavy end of the *security vs efficiency* spectrum of protocols is lacking, making some of these trade-off inaccessible without an implementation of the desired protocol with the tools offered by frameworks like arkworks and Noir.

### Decision table for adopting ZKPs

We condense the results given by the guidelines in the decision tables given in Table 9.4. As we noted, the choice of protocol is mostly influenced by the required security, the possible types of attacks, and the existence of a trusted party, while the choice of framework is mostly influenced by the computational constraints and the target environment. The decision tables are intended to be used as follows.

First, the protocol is chosen according to the requirements given by the application. Each row represents a possible evaluation of the constraint, and for each evaluation a checkmark on a given column means that the corresponding protocol satisfies the given constraint evaluation. The intersection of the results of each restriction is the list of protocols which satisfy every constraint, among which the

protocol is to be chosen. After the choice of the protocol, the framework has to be chosen in a similar fashion. In the frameworks table, an asterisk means that while the protocol is not implemented in the framework, there are the means to implement it (as a third party backend for Noir or with the tools exposed by the ecosystem in arkworks). Since not all frameworks implement all the protocols, and since some requirements are in a trade-off relation, it is possible that this procedure results in no valid solution, in which case the procedure should be repeated by relaxing the least critical constraint.

We give an example of an application of these guidelines, by applying them to the three examples of applications considered in Section 9.1.

The first example of an application was the use of ZKPs for Self Sovereign Identity authentication for some decentralised webapp. In order to further contextualise this scenario, suppose that this is the frontend of some decentralised low-traffic social network or forum. The decentralised nature of this platform allows the usage of multiparty computation to compute a trusted setup, and since the verification procedure does not change over time we can treat the "Yes, but once" case as effectively the "Yes" case, meaning no restriction on the protocol so far. While there is no risk for HNDL attacks (the most an HNDL attack could find is the old credentials, which could be simply reset), there is a potential risk for replay attacks, meaning that the simple use of Groth16 must be excluded. We could use Groth16 if the proof also included some sort of signature or some server-issued challenge, and by making sure that the server does not issue the same challenge twice we can make sure that an authentication proof cannot be replayed. The type of security required depends on the specific platform, but we can assume that a low-traffic social network of forum might not require extremely advanced security, meaning that relying on Marlin, Plonk or even Groth16 could be fine. As for the target runtime environment, we are targeting a browser frontend, with a moderate computational and communication constraint, making the best fitting choice Groth16 over snarkjs, with the addition of some server-issued one-time challenge to prevent replay attacks.

The second example covered the use of a ZKP system for verifiable inference in a service provider and customer scenario. Given the rapidly evolving nature of the inference models, as well as their gigantic sizes, this scenario is predominated by the computational constraints. The only remaining issue is the requirement of a trusted third party, which is not easily resolved. However, since this scenario naturally has a designated verifier, we can make the verifier execute the setup procedure since the verifier will trust themselves, resolving the issue of a trusted setup procedure. This gives access to Groth16 over arkworks, the most efficient solution available.

| | | Groth16 | Marlin | Plonk | Hyrax | Virgo |
|---|---|---|---|---|---|---|
| Is trusted setup possible? | No | | | | ✓ | ✓ |
| | Yes, once | | ✓ | ✓ | ✓ | ✓ |
| | Yes, in MPC | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Yes | ✓ | ✓ | ✓ | ✓ | ✓ |
| Is increased security required? | Extremely | | | | | ✓ |
| | Yes | | | | ✓ | ✓ |
| | Somewhat | | ✓ | ✓ | ✓ | ✓ |
| | No | ✓ | ✓ | ✓ | ✓ | ✓ |
| Is PQ security required? | Yes | | | | | ✓ |
| | No | ✓ | ✓ | ✓ | ✓ | ✓ |
| Are replay attacks possible? | Yes | | ✓ | ✓ | ✓ | ✓ |
| | No | ✓ | ✓ | ✓ | ✓ | ✓ |
| Computational/communication constraint | High | ✓ | | | | |
| | Mid | ✓ | ✓ | ✓ | | |
| | Low | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | snarkjs | arkworks | ZoKrates | zokrates-js | Noir | Noir-js |
|---|---|---|---|---|---|---|---|
| Target runtime | On-chain | ✓ | | ✓ | | ✓ | ✓ |
| | Browser | ✓ | | | ✓ | | ✓ |
| | Server | ✓ | ✓ | | ✓ | | ✓ |
| Computational/communication constraint | High | | ✓ | | | | |
| | Mid | | | ✓ | | | |
| | Low | | | | | ✓ | |
| | Very low | ✓ | | | ✓ | | ✓ |
| Chosen protocol | Groth16 | ✓ | ✓ | ✓ | ✓ | | |
| | Marlin | | ✓ | ✓ | ✓ | | |
| | Plonk | ✓ | * | | | ✓ | ✓ |
| | Hyrax | | * | | | * | * |
| | Virgo | | * | | | * | * |

Table 9.4: Decision table for the choice of protocol and framework. The intended way to use these decision tables is explained at the beginning of subsubsection "Decision table for adopting ZKPs"

Finally, we consider the last example of an on-chain smart contract. To further contextualise the scenario, we suppose that this smart contract is deploying some kind of verification check over some private transaction. Given the economic context, and the high possibility of HNDL attacks (which might reveal the identities behind the private transaction), we have a problematic trade-off between security and efficiency: on one side, the best fitting protocol security-wise would be a solution like Virgo, given its extremely standard security assumption and plausible post-quantum safety; on the other side, Virgo is among the most computationally heavy protocols considered, which is definitely a problem for on-chain environments. The trade-off should be evaluated by the developer, possibly considering a middle ground such as Marlin on ZoKrates, which results in moderately low computational costs (given the usage of arkworks by part of ZoKrates), with better security when compared to Groth16. Another solution, which we will explore in the next section and is used by solutions like zkBridge [36], is the use of recursive proofs, chaining a security-first protocol like Virgo with an efficiency-first protocol like Groth16 to obtain smaller proofs.

## 9.3 Considerations on recursive proofs

An interesting technique used by some applications where both computational and security constraints are present is the use of *recursive proofs*, which try to achieve the best of both worlds. We note that this is a technique rather than a protocol, hence why it was not mentioned in the protocols list, and that it is not well suited for the frameworks we have considered: it requires implementing the verification process of one zkSNARK as the arithmetic circuit of another zkSNARK, which is not straightforward in the frameworks we have considered since they expose the systems as black-boxes. This approach requires the use of lower-level libraries, as well as requiring the developer to be extremely familiar with the cryptographic tools involved, as they will have to be reimplemented from scratch for a different proof system. However, in scenarios where this is a possibility, the final result does indeed offer both extreme efficiency of a zkSNARK like Groth16 with (close to) the same security as one like Virgo.

The main idea of a recursive proofs is to treat the verification process of one zkSNARK as the circuit for the relation of another zkSNARK. This has two main uses: if we chain the same zkSNARK multiple times, we can effectively batch multiple verifications into one proof which is easier to check; if we chain different zkSNARKs with different properties, we get the verification costs of the ending proof system, with the zero-knowledge security level of the starting proof system. Of course, this does come at a cost of increased prover runtime (as we effectively have to execute two prover processes instead of one) and an increased knowledge-

soundness error, as it now suffices to break one of the two proving processes to forge a valid proof.

As we mentioned, this is a difficult technique to implement, requiring extensive knowledge of the involved cryptographic protocols as well as the use of lower level libraries which expose the cryptographic tools to implement the verification process of a zkSNARK. At the same time, this is an extremely useful technique with very interesting properties, being used by real applications: the idea of recursing the same proof system to compress multiple proofs into one proof is being used by zkRollup, and the idea of compressing a secure Virgo proof into a small and efficient Groth16 proof is being used by zkBridge.

There is an additional benefit given by recursing a secure protocol like Virgo into an efficient protocol like Groth16. In general, Groth16 requires the setup procedure to be executed *per-circuit*, which is an issue in contexts where the protocol is used to verify multiple and ever-evolving circuits. However (up to a certain size, as discussed in the theoretical part of this work), the procedure for the Virgo verification is always fixed, and only its inputs change, meaning that we combine this protocol (which has a transparent setup) with Groth16, the trusted setup has to be executed only once for the generation of the public parameters related to the Virgo verification circuit, giving Groth16 a sort of updatability property which it lacks.

## 9.4 Considerations on domain-specific solutions

As we mentioned, ZKP applications to some specific domains have been much more developed in the literature, particularly for the cases of cross-chain communication and verifiable machine learning, resulting in specialised and efficient solutions that could be preferable to custom from-scratch solutions. Some of these solutions are implemented via lower level libraries (often libsnark for C++) for increased efficiency. For completeness of the guidelines we list and analyse these domain-specific solutions for the mentioned use cases.

### 9.4.1 Cross-chain applications for blockchains

#### Overview

The cross-chain problem in blockchains refers to the issues related to developing applications that span multiple chains. Examples of such applications go from simple verification of transaction existence, data transfer and transfer/exchange of assets to more complex solutions. Most popular chains are not designed with such interoperability in mind, and require additional work in order to implement

cross-chain applications. While the implementation of on-chain conditional logic is not necessarily an issue (assuming that the chain supports smart contracts), there are still some key challenges that need to be addressed.

Atomicity, which is usually a concern for these types of operations, can be resolved with the usage of Hash Time Locked Contracts (HTLC). The main challenge for cross-chain applications is that, as a basic block for such operations, one would need to perform validity checks that require context of the whole source chain, which can be prohibitively expensive both computation-wise and communication-wise. Additionally, cross-chain applications between a permissionless and a permissioned chain must make sure not to reveal permissioned or secret data (possibly necessary for the verification) to a publicly-readable chain.

These challenges make the usage of ZKP systems and zkSNARKs the ideal solution. The current state of the art for cross-chain applications through ZKP systems further addresses the size of the involved circuits and computation time by developing proofs which operate on a Light Client check, rather than inspecting the whole chain which could be unfeasible (for example, at the time of writing, context of the whole Bitcoin chain would require around 720GB of data).

### Current development and implementations

zkBridge, proposed by Xie et al. [36], defines a bridge for generic cross-chain applications. Instead of designing a bridge for a specific chain architecture, zkBridge defines an API which can then be used by smart contracts for the specific source and target chains. This allows developers to create cross-chain applications just by specifying smart contracts in the source and target chains that interface with the bridge API. Note that this does not mean that zkBridge offers a completely chain-agnostic solution, as the actual implementation of zkBridge depends on the specific Light Client of the source chain. The specific implementation given by the authors is a bidirectional bridge between Cosmos and Ethereum.

To minimize the prover runtime, Xie et al. also develop a new zkSNARK of independent interest, deVirgo, a decentralised version of Virgo which takes advantage of the data-parallel structure of the signature circuit to obtain a $\times 30$ speed up from the original Virgo. This greatly reduces the (decentralised) prover time, at the cost of a suboptimal proof size. To address this issue, they deploy a recursive proof approach (as discussed in the previous section) by proving the deVirgo statement via Groth16, which further compress the proof size and reduce verification time.

Harmonia, proposed by Belchior et al. [37], offers a framework for cross-chain

applications similar in spirit to zkBridge (in that this is also based on creating a proof of the verification done by a Light Client) specialised to a novel Light Client (DendrETH) for Ethereum. The implementation uses Groth16 through snarkjs, offering excellent proof size and verification time at the cost of a trusted setup which the authors resolve by using the multiparty computation ceremony Powers-of-Tau [30].

### 9.4.2 Applications to Machine Learning

#### Overview

The recent developments in AI technology have caused an increasing number of services offering query access to neural networks and other types of machine learning applications.

The most common scenario, which is often referred to as Machine Learning as a Service, or MLaaS, is where a service provider, assumed to be computationally-powerful, offers access to the service for a fee to a customer, which lacks the computational power and the models to run the service locally.

In MLaaS the weights of the model which are necessary to perform the inferences are also usually privately owned by service provider, which has interest in keeping them hidden. This can happen both for economic reason, as the provider might want to protect the model which required investment to produce, but also for more ethical reasons, such as in a medical scenario where a model is produced by training on patients data which is considered sensitive and might leak from knowledge of the weights.

We note that all applications to machine learning and neural networks share some technical issues when adapting to ZKP protocols which need to be resolved to make the implementation feasible, such as the fact that almost all neural networks work in floating point arithmetic which is difficult to represent in finite field arithmetic.

The most common use cases considered by current research in ZKP applications to machine learning include are following:

- **Proof of inference:** proving that a given output was indeed obtained by running a private neural network on a known input.

  For example, in a medical application, one might want to have proof that a certain result was obtained by a specific, well tested neural network as a safety measure. In this context the zero-knowledge is necessary as knowledge

of the weights might reveal knowledge of the training data which is usually sensitive information.

As another example, in a customer-provider MLaaS scenario as discussed before, a customer might want proof that the results being given by the provider are indeed obtained from the machine that the customer is paying for, and not a "cheaper" machine which would cost less to the provider.

- **Proofs in accuracy-testing:** proving that a certain accuracy score was obtained by a specific neural network, without revealing sensitive information such as the weights of the network.
- **Proof of training:** some of the literature also explores the scenario where a customer might want to outsource the training of a new neural network to a computationally-powerful provider. This can be a simple case of outsourcing of computation if the dataset is provided by the customer, in which the zero-knowledge property is not necessarily needed, or a proper zero-knowledge scenario if the provider is also providing a private dataset, where the customer might instruct the provider to train the network only on a subset of the data defined by properties (e.g., "train on all the data related to the Rust language") for a specialised neural network. In this case, a ZKP system allows for training on specialised private datasets.

Given the natural representation of a neural network as a (possibly big) circuit and that most SNARKs are developed to work directly for circuits or some arithmetization of these, one might think easy to adapt such proofs to neural networks. In practice, though, there are two main problems in this adaptation.

First of all, most neural networks operate on floating-point arithmetic, which does not mesh well with the finite-field arithmetic on which almost all of the current literature of zero-knowledge proof is based. This issue is resolved by a process called quantization in which floating-point numbers get converted to fixed-point numbers, which can be more easily represented in finite arithmetic.

Secondly, most neural networks operate using both linear and non-linear operators, and the latter are much more difficult to represent as constraints for zero-knowledge proofs.

### Current development and implementations

The current state of the art for *ready-to-use* solutions is given by EZKL [38]. EZKL is an open source[35] audited library for generating inference proofs for neural network. It is ready-to-use meaning that it takes as input an `.onnx` representation

[35] `https://github.com/zkonduit/ezkl`

of the neural network, which can be exported directly from PyTorch or Tensorflow.

Other frameworks are being developed but are currently only theoretical proof of concepts for new methods of implementation, and are not intended as a practical solution. Most of these libraries implement proofs for simple known models such as ResNet18/50 and VGG-11/16. Developers intending to use these solutions will have to manually implement the desired models in the specific library framework.

The most relevant frameworks in the current ecosystem are the following.

- Mystique [39] is considered the base layer for recent frameworks and is usually used as a reference point for benchmarks.
- zkNN [40], which is the current state of the art for inference proofs for convolutional neural networks. It provides benchmarks and test for ready to use models and has an open source implementation[36]. The authors claim a speed-up of proof generation time of a factor of $\approx \times 41.5$ on ResNet50 when compared to Mystique, and a speed-up of $\approx \times 35.7$ when compared to EZKL for generating a proof of GPT-2, making it in theory the state of the art for efficiency, though requiring a manual implementation of the desired networks, with little to no documentation.
- zkCNN [41] defines an efficient inference proof scheme for convolutional neural networks, although the provided source code[37] does not yet implement fully the zero-knowledge property, and development seems to be abandoned. For these reasons, despite the interesting theoretical results, this might not be an option even for developers intending to manually implement their neural networks in the framework.
- ZEN [42] implements a compiler for neural networks into a ZKP scheme for inference and a ZKP scheme for accuracy testing, although the source provided for the code results in a dead link, hence it is not a suitable option.

[36] https://anonymous.4open.science/r/zknn-D209
[37] https://github.com/TAMUCrypto/zkCNN

# Chapter 10

# Conclusions and future work

In this work we analysed and compared the available solutions for implementing ZKP systems in real life applications, by studying the theoretical protocols from the literature as well as the available frameworks for implementation. While this is not the first structured comparison of ZKP protocols or ZKP frameworks to be done, existing comparisons focus either on the theoretical aspects of the protocols or on the empirical properties of the frameworks. This work offers an unified comparison taking in consideration both theoretical aspects for security and practical aspects for efficiency. The results of this comparison are then used to produce guidelines meant to help the choice of ZKP protocol and framework for a given application, based on the properties of the given use case.

We remark the main results of the analysis and comparison. First, we noted the absence of transparent setup protocols in the available frameworks: every solution implements a subset of Groth16, Marlin, Plonk, or variants of these, all of which require trusted (possibly universal) setup. Among all of the considered solutions, there exist only one ready to use solution which implements a transparent setup protocol; it is Noir with the third party backend ProveKit, which implements the transparent setup zkSNARK Spartan.

From the comparison it appears that there exist a natural trade-off between security and efficiency in the existing protocols, with the trusted setup protocols sitting in the efficiency side of the spectrum, and the transparent setup protocols sitting in the security side. At the same time, empirically the efficiency of both the runtime of the protocol as well as the communication and proof size is mainly influenced by the choice of framework rather than the protocol. Given that not all frameworks implement all the available protocols, today the choice of framework is much more relevant for computationally constrained applications. At the same time, while some DSL are created specifically with on-chain environments as the target, almost all available frameworks allow exporting the verifier into a Solidity

verifier for EVM-compatible chains, making the choice of framework less restricting in this aspect.

Regarding availability of the protocols, the implemented frameworks still lack some relevant protocols present in the literature. In particular, there is an absence of transparent setup protocols, which right now are only available through manual implementations within frameworks which expose the tools to do so, like arkworks and Noir. However, these manual implementations require strong knowledge of the cryptographic tools involved, making this path not easily accessible. Additionally, all the canonically implemented protocols are not post-quantum safe. The most common implemented protocol among the frameworks is Groth16, which provides excellent efficiency at the cost of security and proof malleability.

Lastly, we noted some additional precautions which must be taken into consideration for applications which are susceptible to replay attacks or harvest-now-decrypt-later attacks, as well as post-quantum safety, which exclude some protocols.

**Future work**

The scope of this work is meant to be wide enough to cover most of the relevant protocols and frameworks, but limited and finite in order to allow a proper analysis of the subject. This means that this work has some open ends which could be covered in future work.

First of all, while this work takes in consideration five of the most relevant protocols and 4 of the most relevant frameworks, this is by no means an exhaustive search: there are plenty of existing and established protocols such as Libra, zk-STARK, Hyperplonk, or Spartan, as well as frameworks such as Gnark or libsnark, which have not been considered in this work. This was a deliberate choice but it would still be interesting to see, in an expanding work, how these solutions fare in comparison to the already analysed ones.

Additionally, this work focuses specifically on ZKP systems for arbitrary arithmetic circuits which can be used as black-boxes in available frameworks. This is meant to reflect how a developer not experienced in cryptography might use these tools, but it is a noticeable restriction over all the families of ZKP systems. For example, when focusing on specific problems rather than generic computation, it could be interesting to see a similar comparison analysing different systems, such as inner product arguments or range arguments like Bulletproofs.

Another possible direction of expansion for this work is the implementation of some of the mentioned protocols with the tools offered by arkworks and Noir: while we mentioned that currently there exist a discrepancy between the protocols proposed by the literature and the available protocols in the frameworks, some of these frameworks do offer the tools to fill this gap. Given that the main trade-off for a transparent setup comes in the form of increased computational costs, and given that the resulting efficiency is heavily influenced by the framework, it would be interesting to see how a less efficient transparent setup protocol would compare to existing solutions when implemented in an efficient framework like the arkworks ecosystem.

Lastly, this work only briefly mentions recursive proofs as an available tool for trying to obtain the best of both worlds security and computationally-wise. This aspect could certainly be expanded, adding to the comparison solutions which implement this tool, or developing a framework to compare and benchmark generic recursive proofs which can then be instantiated with various zkSNARKs.

# Bibliography


[1] 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*. 1985, pp. 291–304.

[2] Jens Groth. "On the size of pairing-based non-interactive arguments". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2016, pp. 305–326.

[3] Alessandro Chiesa et al. "Marlin: Preprocessing zkSNARKs with universal and updatable SRS". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2020, pp. 738–768.

[4] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *Cryptology ePrint Archive* (2019).

[5] Riad S Wahby et al. "Doubly-efficient zkSNARKs without trusted setup". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 926–943.

[6] Jiaheng Zhang et al. "Transparent polynomial delegation and its applications to zero knowledge proof". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 859–876.

[7] Thomas Chen et al. "A review of zk-snarks". In: *arXiv preprint arXiv:2202.06877* (2022).

[8] Nojan Sheybani et al. "Zero-knowledge proof frameworks: A systematic survey". In: *arXiv preprint arXiv:2502.07063* (2025).

[9] Ayush Nainwal, Atharva Kamble, and Nitin Awathare. "A Comparative Analysis of zk-SNARKs and zk-STARKs: Theory and Practice". In: *arXiv preprint arXiv:2512.10020* (2025).

[10] Eli Ben-Sasson et al. "Scalable, transparent, and post-quantum secure computational integrity". In: *Cryptology ePrint Archive* (2018).

[11] Mohammed El-Hajj and Bjorn Oude Roelink. "Evaluating the efficiency of zk-snark, zk-stark, and bulletproof in real-world scenarios: A benchmark study". In: *Information* 15.8 (2024), p. 463.

[12] Benedikt Bünz et al. "Bulletproofs: Short proofs for confidential transactions and more". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[13] Karim Baghery, Axel Mertens, and Mahdi Sedaghat. "Benchmarking the setup of updatable zk-SNARKs". In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2023, pp. 375–396.

[14] Mary Maller et al. "Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings". In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 2111–2128.

[15] Matteo Campanelli et al. "Lunar: a toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2021, pp. 3–33.

[16] Carla Ràfols and Arantxa Zapico. "An algebraic framework for universal and updatable SNARKs". In: *Annual International Cryptology Conference*. Springer. 2021, pp. 774–804.

[17] Jens Ernstberger et al. "zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks". In: *International Conference on Security and Cryptography for Networks*. Springer. 2024, pp. 46–72.

[18] Richard A DeMillo and Richard J Lipton. *A probabilistic remark on algebraic program testing.* Tech. rep. 1977.

[19] Jacob T Schwartz. "Fast probabilistic algorithms for verification of polynomial identities". In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 701–717.

[20] Richard Zippel. "Probabilistic algorithms for sparse polynomials". In: *International symposium on symbolic and algebraic manipulation*. Springer. 1979, pp. 216–226.

[21] Torben Pryds Pedersen. "Non-interactive and information-theoretic secure verifiable secret sharing". In: *Annual international cryptology conference*. Springer. 1991, pp. 129–140.

[22] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. "Constant-size commitments to polynomials and their applications". In: *International conference on the theory and application of cryptology and information security*. Springer. 2010, pp. 177–194.

[23] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. "The algebraic group model and its applications". In: *Annual International Cryptology Conference*. Springer. 2018, pp. 33–62.

[24] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. "Delegating computation: interactive proofs for muggles". In: *Journal of the ACM (JACM)* 62.4 (2015), pp. 1–64.

[25] Tiacheng Xie et al. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *Annual International Cryptology Conference*. Springer. 2019, pp. 733–764.

[26] Matteo Campanelli et al. "Zero-knowledge contingent payments revisited: Attacks and payments for services". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 229–243.

[27] Rosario Gennaro et al. "Quadratic span programs and succinct NIZKs without PCPs". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2013, pp. 626–645.

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

[29] Alessandro Chiesa, Michael A Forbes, and Nicholas Spooner. "A zero knowledge sumcheck and its applications". In: *arXiv preprint arXiv:1704.02086* (2017).

[30] Valeria Nikolaenko et al. "Powers-of-tau to the people: Decentralizing setup ceremonies". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2024, pp. 105–134.

[31] arkworks contributors. ***arkworks*** *zkSNARK ecosystem*. 2022. URL: `https://arkworks.rs`.

[32] Srinath Setty. "Spartan: Efficient and general-purpose zkSNARKs without trusted setup". In: *Annual International Cryptology Conference*. Springer. 2020, pp. 704–737.

[33] Shafi Goldwasser and Yael Tauman Kalai. "On the (in) security of the Fiat-Shamir paradigm". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE. 2003, pp. 102–113.

[34] David Bernhard, Olivier Pereira, and Bogdan Warinschi. "How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2012, pp. 626–643.

[35] Nir Bitansky et al. "Why "Fiat-Shamir for proofs" lacks a proof". In: *Theory of cryptography conference*. Springer. 2013, pp. 182–201.

[36] Tiancheng Xie et al. "zkbridge: Trustless cross-chain bridges made practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 3003–3017.

[37] Rafael Belchior et al. "Harmonia: Securing cross-chain applications using zero-knowledge proofs". In: *Authorea Preprints* (2023).

[38] Tobin South et al. "Verifiable evaluations of machine learning models using zkSNARKs". In: *arXiv preprint arXiv:2402.02675* (2024).

[39] Chenkai Weng et al. "Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 501–518.

[40] Tao Lu et al. "An efficient and extensible zero-knowledge proof framework for neural networks". In: *Cryptology ePrint Archive* (2024).

[41] Tianyi Liu, Xiang Xie, and Yupeng Zhang. "Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2968–2985.

[42] Boyuan Feng et al. "Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences". In: *Cryptology ePrint Archive* (2021).

# Appendix A

# Terminology for Plonk

The original presentation of the Plonk protocol [4] defines the protocol in terms of number of *gates and wires* of the circuit instead of *gates and inputs* like we did. The reason of this choice is that the circuit representation used by the authors differs slightly from our definition (based on directed acyclic graphs), which we now explain.

First of all, in our definition the inputs and outputs of the circuit are *nodes* of the graph, while in the the representation used by the authors of Plonk (which in fairness is probably more representative of what wires and gates actually are in hardware) inputs and outputs are *wires* that come from outside or go outside the circuit. Additionally, we said that the wires of the circuit are the edges of the graph, which implies that two wires coming out of one node and going to two different nodes are different wires. In the other representation, however, all the wires coming out of one node are considered to be "one-but-split" wire, which would then account only for 1 in the count of the wires and would automatically assume the same value independently of where it was inputted.

However, by corresponding the value of the "one-but-split" output wire to the "value assumed by the gate", we get a clean correspondence that resolves this issue. Indeed, in the original article the variables $z_i$ are not in correspondence of the nodes, rather in correspondence of the wires of the circuit. If we were to use this notation then, since for us a wire is an edge of the graph, then we would have to account for multiple wires coming from a common node, which we would then have to enforce to be equal. This is not necessarily an issue since the point of the permutation check is precisely to assert that "values that should be equal are actually equal", however it would add to computational cost to account for the duplicate wires.

Furthermore, this correspondence actually preserves the computational constants

as we are mapping each "stray" input wire to an input node, and each wire which comes from a gate is mapped to that gate.