FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Benjamín Benčík

On $\mathcal{P}\mathfrak{lon}\mathcal{K}$ SNARK

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Pavel Hubáček, Ph.D.
Study programme: Artificial Intelligence

Prague 2024

I would like to thank my advisor, Pavel Hubáček, for great support and also Tomáš Krňák for his valuable suggestions.

Title: On PlonK SNARK

Author: Benjamín Benčík

Institute: Computer Science Institute of Charles University

Supervisor: Mgr. Pavel Hubáček, Ph.D., Computer Science Institute of Charles University

Abstract: The thesis presents a comprehensive analysis of the PlonK zk-SNARK protocol. It delves into the core cryptographic primitives underlying Plonk and provides a detailed explanation of the protocol's execution. This in-depth exploration complements existing research on security analysis by offering a clear and accessible protocol overview. Additionally, the thesis explores optimization strategies, with focus on reducing the degree of the wire polynomials defined by the arithmetic circuit.

Keywords: cryptography, SNARK, zero-knowledge, polynomial commitment scheme

Název práce: O PlonK SNARKu

Autor: Benjamín Benčík

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Pavel Hubáček, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Bakalárska práca ponúka analýzu zk-SNARK protokolu PlonK. Zaoberá sa kryptografickymi základmi, na ktorých stojí bezpečnosť PlonKu, a poskytuje detailné vysvetlenie procedúr protokolu. Práca dopĺňa existujúci výskum zameraný na bezpečnostnú analýzu tým, že ponúka jasný a zrozumiteľný prehľad protokolu. Okrem toho práca rozoberá možné optimalizácie protokolu, s dôrazom na zníženie stupňa polynómov definovaných aritmetickým obvodom.

Klíčová slova: kryptografia, SNARK, zero-knowledge, schéma pre polynomiálne záväzky

Contents

Introduction

Zero-Knowledge proofs

In a zero-knowledge protocol, there are two parties: one prover P and at least one verifier V. The protocol allows to convince the verifier that the prover knows some secret information. The prover provides an *argument of knowledge*, which is a probabilistic proof.

More formally, the prover is an entity that has a witness $\mathtt{w}$ and aims to convince the verifier about the knowledge of $\mathtt{w}$ without revealing it. For example, say that the prover wants to convince the verifier that a graph is k-colorable. The prover knows how to color the graph validly and wants to convince the verifier about its knowledge of a valid k-coloring by providing an argument of knowledge π. The verifier is responsible for checking the validity of π. It is essential to mention that verification of the proof is a deterministic algorithm that runs strictly in polynomial time and is orders of magnitude faster than generation of the proof. The zero-knowledge property ensures that the verifier "learns nothing" about $\mathtt{w}$ from π.

Example. Consider an ordinary deck with 32 red and 32 black cards to provide another simple example. The prover picks a card. Assume it is the red 8. The prover wants to convince the verifier that he picked a red card without showing the card number. The prover can achieve this by listing all 32 black cars. Since the card he picked is not among the black cards, it must mean that his card is red. At the same time, the verifier does not get any information about the number of the card that the prover picked. Importantly, this procedure is meaningful only if the prover is honest and does not cheat by replacing cards in the standard deck.

For the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol, we will use the properties of a polynomial to guarantee that cheating is not possible except for a negligible soundness error. At this point, it might not be clear why we should consider constructing such a proof. Motivation for creating these types of protocols stems from different fields:

1. **Proving a statement on private data**
    - Proving your financial standing is often a requirement for loans, credit applications, or other financial services. However, revealing your exact account balance might be undesirable. Using zero-knowledge proof, you can demonstrate to a financial institution that your account balance meets a specific threshold without disclosing the exact amount.
    - When conducting a medical study, you may want to prove the validity of your findings without revealing any information about the individual patients involved. Using zero-knowledge proof, you can demonstrate the correctness of your statistical analysis of the data while ensuring that patient information remains completely confidential.
2. **Anonymous authorization**
    - You want to prove to some website that you have some privileges

without revealing a key or password. You can demonstrate to the website that you possess the necessary information using zero-knowledge proof.

3. **Outsourcing a computation**
   - Imagine you need to solve a complex computational problem but lack the resources. You outsource the task to a third-party service. This service provides a solution but requires payment before revealing it. The service can utilize a zero-knowledge proof, demonstrating they've computed the correct solution without revealing any details of their computation process. This allows you to pay for a verified solution confidently.
   - Suppose a company offers access to multiple versions of their AI model, each with varying capabilities and costs. You subscribe to a specific, high-performance model. The company can leverage zero-knowledge proofs to prove you are receiving the service you pay for, without revealing the parameters of the model.

In the context of the bachelor's degree specialization in artificial intelligence, zero-knowledge proofs can potentially enhance the security and privacy of AI systems. For example the training of large neural networks is often computationally expensive. As suggested above, this process could be outsourced to an untrusted party with large computational power. The remote server may then produce proof that the model was trained correctly, and this proof would be easily verifiable. Another possible application could be in distributed computation, where we require assurances that each of the participants derives the output according to the model, and input data. These challenges are addressed in the recent work *Zero-Knowledge Proof-based Practical Federated Learning on Blockchain* [1]. Despite the field's relative newness, ongoing research holds promise for further advancements in ensuring secure and verifiable AI operations.

Assuming there is sufficient motivation for this type of protocol, we proceed with the rest of the thesis. In Chapter 1, we provide a general introduction to SNARKs, including the core definitions essential for the rest of this work. Next, Chapter 2 delves into the underlying mathematical structure of our protocol, specifically focusing on elliptic curves over finite fields. This chapter also includes a detailed explanation of the KZG polynomial commitment scheme [2] and a high-level overview of $\mathcal{P}$lon$\mathcal{K}$. With these tools in hand, we then present an in-depth explanation of the $\mathcal{P}$lon$\mathcal{K}$ procedures in Chapter 3, followed by an overview of its properties and a detailed diagram in Chapter 4. Finally, Chapter 5 discusses related work on optimizing the protocol and explains our own contributions to further enhance its performance.

1 Preliminaries

1.1 Arithmetic circuits

Definition 1 (Arithmetic circuit)**.** $C(\mathbb{w}, \mathbb{x}) : \mathbb{F}_p^n \to \mathbb{F}_p$ is a direct acyclic graph with nodes representing arithmetic operations and edges flow of the variables, where $\mathbb{w}$ is the witness and $\mathbb{x}$ is public input

It is standard that any program could be compiled into a boolean circuit with gates AND, OR. It is commonly known that a boolean circuit can be easily translated into an arithmetic circuit. We can represent 0 as the neural additive element of $\mathbb{F}_p$ and one as a neutral multiplicative element of $\mathbb{F}_p$. For variables a, b operation AND becomes $a \cdot b$ and OR is $a + b - a \cdot b$.

a	b	$a \cdot c$	$a + b - a \cdot b$
0	0	0	0 - 0 = 0
1	0	0	1 - 0 = 1
0	1	0	1 - 0 = 1
1	1	1	2 - 1 = 1

Circuit size denotes $|C|$, which is the number of gates in the circuit. The following chapters will use $n = |C|$. We will simplify the explanation by considering only binary gates with operations $+, \times$. However, there exist versions of $\mathcal{P}\mathfrak{lon}\mathcal{K}$ that can work with custom gates as [3]. Leaves will be considered input nodes, and all other gates will have in-degree 2.

1.2 Argument systems

Prover aims to convince the verifier that $\exists \mathbb{w} : \mathsf{C}(\mathbb{w}, \mathbb{x}) = 0$. There are two ways to construct an argument system:

- Interactive: P and V are allowed to exchange messages
- Non-interactive: the V does not send any message to P just verifies π

We will aim toward a non-interactive argument system with procedures:

1. $\mathsf{Setup}(1^n) \to \mathsf{pp}$
2. $\mathsf{Prove}(\mathsf{pp}, \mathbb{w}, \mathbb{x}) \to \pi$
3. $\mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi) \to \mathsf{accept/reject}$

The procedure Setup takes a security parameter 1^n, which is determined by the bound on the size of the arithmetic circuit n and produces public parameters pp. The reason we use 1^n instead of n is that we want the protocol to have complexity dependent on the size of the security parameter and the standard binary representation of number n size $\log n$. Prove procedure takes the secret key sk, the witness $\mathbb{w}$, and the public input $\mathbb{x}$ to produce the argument of knowledge (proof) π. Finally, the Verify procedure takes the public key pk, public input $\mathbb{x}$, and proof π and decides whether π is valid. Before going further, it is important to formalize the properties that the argument system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ should have.

1.2.1 Correctness

Definition 2 (Perfect Completeness)**.** A proof system is *complete* if:

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi) = \mathsf{accept} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^n) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbb{w}, \mathbb{x}) \end{array}\right] = 1 \tag{1.1}$$

The correctness says that the verifier V will accept π if the prover knows the witness $\mathbb{w}$.

1.2.2 Soundness

Definition 3 (Computational Soundness)**.** A proof system $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is *computationally sound* if for all probabilistic polynomial time adversaries $\mathcal{A}$ exists negligible $\mathsf{negl}(n)$ such that:

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi) = \mathsf{reject} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^n) \\ \pi \leftarrow \mathcal{A}(\mathsf{pp}, \mathbb{x}) \end{array}\right] \geq 1 - \mathsf{negl}(n). \tag{1.2}$$

While correctness requires that an honest prover can always convince an honest verifier, computational soundness says that a dishonest prover cannot convince an honest verifier. If the prover does not know the witness $\mathbb{w}$, then there is a negligible probability that the prover can produce a proof π, which the verifier accepts. The difference between an argument and a proof is that in a proof, the soundness holds against a computationally unbounded prover. In an argument, the soundness only holds against a polynomially bounded prover.

We will require the protocol to be *knowledge-sound*, which is a stronger property. Arguments that satisfy knowledge soundness are typically referred to as arguments of knowledge, formalized in the following definition. The following notion is presented specifically for the setting where the prover must convince the prover of the knowledge of a witness $\mathbb{w}$ for which $\mathsf{C}(\mathbb{x}, \mathbb{w}) = 0$.

Definition 4 (Argument of Knowledge)**.** $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is an *argument of knowledge* for a circuit C if for every polynomial time adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, with probability of success δ such that:

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathbb{x}, \pi) = \mathsf{accept} \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(\mathsf{C}) \\ (\mathbb{x}, \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{pp}) \\ \pi \leftarrow \mathcal{A}_1(\mathsf{pp}, \mathbb{x}, \mathsf{state}) \end{array}\right] \geq \delta, \tag{1.3}$$

there is a polynomial time extractor Ext which uses $\mathcal{A}_1$ such that:

$$\Pr\left[\mathsf{C}(\mathbb{x}, \mathbb{w}) = 0 \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(\mathsf{C}) \\ (\mathbb{x}, \mathsf{state}) \leftarrow \mathcal{A}_0(\mathsf{pp}) \\ \mathbb{w} \leftarrow \mathsf{Ext}^{\mathcal{A}_1(\mathsf{pp}, \mathbb{x}, \mathsf{state})}(\mathsf{sk}, \mathbb{x}) \end{array}\right] \geq \delta - \mathsf{negl}(n). \tag{1.4}$$

The adversary $\mathcal{A}$ is split into two parts where $\mathcal{A}_0$ chooses the instance of the problem $\mathbb{x}$ and $\mathcal{A}_1$ that forges the proof π. The definition says that if an $\mathcal{A}$ exists that can forge the proof π, then there must be an algorithm Ext that uses $\mathcal{A}$ as a black box to extract a valid witness $\mathbb{w}$. Intuitively, the prover knows $\mathbb{w}$ if it can be "extracted" from the prover.

1.2.3 Zero-Knowledge

Definition 5 (Computational indistinguishability)**.** Two sequences of probability distributions $\{X_n\}_{n\in\mathbb{N}}, \{Y_n\}_{n\in\mathbb{N}}$ are *computationally indistinguishable* if, for every probabilistic poly-time distinguisher D exists a negligible function $\mathsf{negl}(n)(x)$ such that:

$$|\Pr_{x\leftarrow X_n}[\mathsf{D}(1^n, x) = 1] - \Pr_{y\leftarrow Y_n}[\mathsf{D}(1^n, y) = 1]| \leq \mathsf{negl}(n).$$

We will use this to construct the definition of honest verifier zero-knowledge specific to the problem on the arithmetic circuit.

Definition 6 (Honest Verifier Zero-Knowledge)**.** $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is *honest verifier zero-knowledge* (HVZK) for circuit C if there is an efficient simulator Sim such that $\forall \mathbb{x} \in \mathbb{F}_p$ for which $\exists \mathbb{w} : \mathsf{C}(\mathbb{x}, \mathbb{w})$ the distribution:

$$(\mathsf{pp}, \mathbb{x}, \pi) : \mathsf{pp} \leftarrow \mathsf{Setup}(1^n), \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, \mathbb{x}, \mathbb{w}),$$

is computationally indistinguishable from

$$(\mathsf{pp}, \mathbb{x}, \pi) : \mathsf{pp} \leftarrow \mathsf{Setup}(1^n), \pi \leftarrow \mathsf{Sim}(\mathsf{pp}, \mathbb{x}).$$

This definition is somewhat unintuitive. Essentially, it says that π "does not reveal" anything about $\mathbb{w}$ besides its existence. This is formalized by the existence of an efficient simulator that the verifier can use to generate π by itself. And if it is possible to generate $\mathbb{w}$ without the knowledge of π, then it must mean that π does not provide any information about $\mathbb{w}$. There exist multiple variants of the zero-knowledge property. The difference between them is explained in detail in *Proofs, Arguments, and Zero-Knowledge* [4].

It might seem that the definition of the argument of knowledge and zero-knowledge are the opposite of each other. While the first one claims that, for each prover, there needs to exist an extractor Ext that can extract the $\mathbb{w}$, the second definition claims the existence of a simulator Sim which can generate valid transcripts with proofs. Is it possible to use the extractor on the simulator in order to extract a witness $\mathbb{w}$? The key is that the extractor needs to have access to a prover and query it in order to extract the witness. It might happen that the protocol would not remain zero-knowledge after multiple queries to the prover, as will be shown in Section 2.3.2. In that particular case, the extractor might query the prover and eventually obtain the witness $\mathbb{w}$, but multiple queries would break the zero-knowledge property, so it is not possible to use the extractor.

1.3 Succinctness

The notion of succinctness captures that the proof should be *relatively small and easy to verify.* This is formalized as:

- The size of the proof (argument) $|\pi|$ is logarithmic in the circuit size n: $\mathcal{O}(\log n)$.
- The procedure Verify is poly-logarithmic in the circuit size n: $\mathcal{O}\left(\log n^k\right)$.

1.4 Interactive and Non-interactive protocols

Many protocols are described in interactive settings where the verifier challenges the prover. This is somewhat impractical for real-world use cases. In non-interactive protocol, the only message is the proof π sent by the prover. This means no interaction is required from the verifier. Under some assumptions, it is possible to transform interactive protocol into non-interactive using the Fiat-Sharmir transform [5]. This transformation needs to be handled with special care. Multiple vulnerabilities were discovered due to improper application of the heuristic [6].

1.5 zk-SNARK

$\mathcal{P}$lon$\mathcal{K}$, a member of the SNARK *(Succinct Non-Interactive Argument of Knowledge)* family of cryptographic protocols, offers several advantages. The $\mathcal{P}$lon$\mathcal{K}$ protocol can be transformed into a zero-knowledge SNARK (zk-SNARK) with minimal additional overhead, as explained in Section 2.3.2.

These protocols are particularly well-suited for scenarios where the verifier is computationally weak. Thanks to the succinctness property, the proofs are small and easily verifiable. The noninteractivity makes the protocol well-suited for practical application since no party has to wait for responses or queries. Non-interactivity is especially beneficial in multiparty settings where the same proof can be sent to multiple verifiers without requiring individual interactions with each one. For a more detailed explanation of SNARKS, the reader might refer to *Why and How zk-SNARK Works* [7].

In the following Chapter 2, we will describe the tools needed to construct the $\mathcal{P}$lon$\mathcal{K}$ protocol. Many of these techniques are commonly used also in other SNARKs. Chapter 3 gives a detailed explanation of the $\mathcal{P}$lon$\mathcal{K}$ protocol.

2 Building Blocks

In this section, we will discuss the primary building blocks to help us construct the $\mathcal{P}$lon$\mathcal{K}$ protocol. First, let us start with a very rough overview. Say there exists a problem P. The prover knows a solution $\mathbb{w}$ to P and wants to convince the verifier that he knows the solution. How will he do it without revealing the solution? Let there be a program that checks solutions to P and outputs 0 if the provided solution is valid. This program could be compiled into an arithmetic circuit C. Now, the prover aims to convince the verifier that, for some public input $\mathbb{x}$, there exists solution $\mathbb{w}$ (witness) such that $\mathsf{C}(\mathbb{w}, \mathbb{x}) = 0$. In other words, the prover wants to convince others that the program for checking P would accept his solution. This only works under the assumption that the verifier believes that the circuit C correctly checks solutions to P, so the C is public for anyone to inspect.

Example. Say the prover knows a solution to an instance of a Sudoku problem, and there is a program that checks if the Sudoku is filled validly. The public input $\mathbb{x}$ to this problem are the board's dimensions and pre-filled numbers. The prover knows how to fill the rest of the board, and this information is stored in the witness $\mathbb{w}$, which is private. To convince the verifier that $\mathsf{C}(\mathbb{w}, \mathbb{x}) = 0$, the proof should say that the circuit was executed validly. The proof is encoded in polynomials, which reveals nothing about the $\mathbb{w}$. This example is visualized in Figure 2.1. There is an instance of a sudoku problem that has public pre-filled values. The remaining values of the table compose the private witness. The program that checks Sudoku validity is public and encoded as an arithmetic circuit. The role of the prover is to convince the verifier that the public program for checking the validity of accepts his solution.

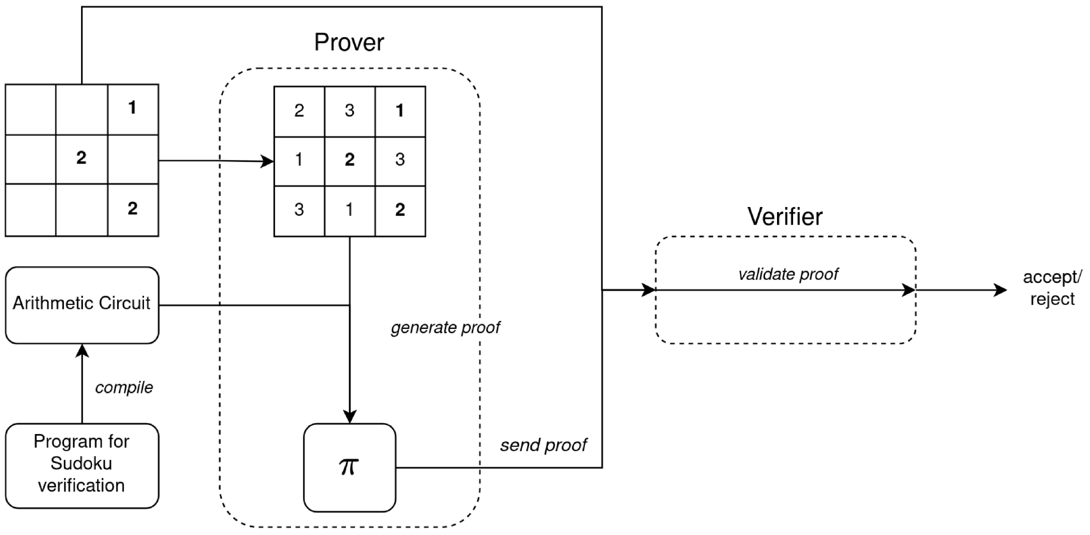


Figure 2.1 Sudoku Example

2.1 Elliptic curves

The discussed elliptic curves will be over a finite field $\mathbb{F}_p$ where each for each point $(x, y) \in \mathbb{F}_p^2$. An elliptic curve is defined as $f(x, y) : y^2 = x^3 + ax + b$

with parameters $a, b \in \mathbb{F}_p$. Points of the curve define a group $\mathbb{G}$ with a neutral element point at infinity $(0, 0)$ and $(x, -y)$ as the inverse element to (x, y). In the following chapters, we will use multiplicative notation, and exponentiation will denote repeated application of the group operation. We also denote G a generator of the group $\mathbb{G}$.

This group is especially interesting due to the hardness of the discrete logarithm problem (DLP), which is considered hard over sufficiently large $\mathbb{F}_p$. The discrete logarithm problem states can be stated as follows. Given a group generator G and a group element l, find k that solves $G^k = l$. For this reason, $|\mathbb{F}_p|$ will be viewed as a security parameter of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol. We will rely on the hardness of DLP on elliptic curves, which is standard in many modern cryptographic protocols.

2.1.1 Elliptic Curve Arithmetic

Given G^a, G^b, anyone can perform the operations listed below. However, it is very hard to extract a, b due to DLP. That allows the prover to send some encoded values to the verifier, who will be able to perform arithmetic checks without discovering the original values.

Operation	**Inputs**	**Computation**	**Result**
Addition	G^a, G^b	$G^a \cdot G^b$	G^{a+b}
Subtraction	G^a, G^b	$G^a \cdot (G^b)^{-1}$	G^{a-b}
Scalar multiplication	G^a, scalar s	$(G^a)^s$	G^{as}
Polynomial evaluation	$\{G, G^a, \dots G^{a^n}\}$ $f(x) = c_0 + c_1x + \dots c_nx^n$	$\prod_i (G^{a^i})^{c_i}$	$G^{f(a)}$

The first two operations, addition and subtraction, follow from the definition of the group, and scalar multiplication is just a generalization of the exponentiation. For polynomial evaluation, take arbitrary polynomial $f(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + ... + x^n$ of degree bound n. The evaluation of $f(x)$ at some point a can be written as

$$G^{f(a)} = G^{c_0+c_1a+c_2a^2+...+c_ka^n} = (G^{a^0})^{c_0} \cdot (G^{a^1})^{c_1} \cdot (G^{a^2})^{c_2} \cdot \dots (G^{a^k})^{c_n} = \prod_{i=0}^{n} (G^{a^i})^{c_i}$$

2.1.2 Elliptic curve pairing

Given groups $\mathbb{G}_1, \mathbb{G}_2$ and a target group $\mathbb{G}_t$ pairing e is in essence deterministic mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. To be more specific, it is a bilinear mapping that takes two encrypted elements from the source group and combines them into an element of the target group. Since this is a complex topic, we will treat the pairing as a black box; for a more detailed explanation, we refer the reader to [8]. For the purposes of this work, it will be sufficient to know that curve pairings have the following properties $\forall a, b \in \mathbb{F}_p$:

$$e(G_1^a, G_2^b) = e(G_1^b, G_2^a) = e(G_1^{ab}, G_2^1) = e(G_1^1, G_2^{ab}) = e(G_1^1, G_2^a)^b = e(G_1^1, G_2^1)^{ab}$$

In Section 2.1.1, there is no way to calculate $G^{a \times b}$ from G^a, G^b. We will use curve pairings to "emulate" this operation $e(G_1^a, G_2^b) = G_t^{ab}$. Note that the operation cannot be performed more than once because the result is in a different target group. Thus, we will be allowed to use this operation in the $\mathcal{P}$lon$\mathcal{K}$ protocol only a single time. Not all elliptic curves are pairing-friendly, and this property is rather rare.

2.1.3 Multi-scalar Multiplication

Polynomial evaluation can be performed via multi-scalar multiplication (MSM), where elements $\{G, G^a, G^{a^2}, \dots, G^{a^n}\}$ are pairwise multiplied by scalars and eventually summed. In the group $\mathbb{G}$, the scalar multiplication $(G^{a^i})^{c_i}$ is defined as c_i times repeated group operation $G^{a^i} \cdot G^{a^i} \dots G^{a^i}$. When dealing with fields in order hundreds of bits, this becomes expensive, and this operation is a major bottleneck of $\mathcal{P}$lon$\mathcal{K}$ and many SNARK protocols.

To make this operation faster, it is possible to calculate the elements using the *square and multiply* version, which can be described as:

$$(G^{a_i})^n = \begin{cases} ((G^{a_i})^2)^{n/2} & 2 \mid n \\ G^{a_i}((G^{a_i})^2)^{(n-1)/2} & 2 \nmid n \end{cases}$$

The naive way needs to perform n group operations to calculate $(G_i^a)^n$ while *square and multiply* achieves the same result in $\mathcal{O}(\log_2 n)$. The resulting complexity of MSM is $\mathcal{O}(n \times \log_2 \max_i c_i)$.

A more efficient way is to use Pipenger's algorithm [9]. It is possible to optimize the algorithm by precomputation and parallelism. Moreover, there is also an effort to create a hardware-accelerated version of MSM as described in PipeMSM [10].

2.2 Polynomial toolkit

2.2.1 Comparing polynomials

The effective comparison of polynomials is one of the building blocks of $\mathcal{P}$lon$\mathcal{K}$. We can test if two polynomials are equal just by comparing them at a uniformly random point. The field size influences the security, and this section assumes that the size of the field $|\mathbb{F}_p|$ is significantly larger than the degree bound d on the polynomials used. For the rest of this thesis, we will assume that $|\mathbb{F}_p| \gg d$.

Theorem 1. Arbitrary two non-identical polynomials $a_0 + a_1x + a_2x^2 ... a_nx^n; b_0 + b_1x + b_2x^2 ... b_mx^m$, where both $n, m \leq d \in \mathbb{N}$ intersect in no more than d points.

Intersection of polynomials

Proof. An intersection point γ such that

$$a_0 + a_1\gamma + a_2\gamma^2 \dots a_n\gamma^n = b_0 + b_1\gamma + b_2\gamma^2 \dots b_m\gamma^m,$$

is a root of the following polynomial of degree $max(n, m) \leq d$

$$a_0 - b_0 + (a_1 - b_1)x + (a_2 - b_2)x^2 \dots (a_n - b_n)x^n = 0.$$

From the Fundamental Theorem of Algebra, we know that a polynomial of degree d cannot have more than d roots. $\square$

If two polynomials are identical, then $\forall x : p(x) = q(x)$, so determining identity based on evaluation at a random point is correct. Two polynomials with different coefficients can be equal at intersection points, so this approach has a soundness error rate. Based on Theorem 1, two non-identical polynomials might intersect in almost d points, so the probability of randomly selecting one of the intersection points is $\frac{d}{\text{domain size}} = \frac{d}{|\mathbb{F}_p|}$. This probability is considered negligible, so we can compare two polynomials just by a single evaluation.

A multivariate version of the previous observation could be proved by the Schwartz-Zippel lemma [11].

Lemma 1. *For domain $D \in \mathbb{F}_p$ and a polynomial $f(x_1, x_2, \dots x_n) \in \mathbb{F}_p[x_1, \dots, x_n]$ with degree bound d and independently uniformly at random selected $(r_1, r_2 \dots r_n) \in D^n$ it holds:*

$$\Pr[f(r_1, r_2 \dots r_n) = 0] \leq \frac{d}{|D|}.$$

2.2.2 Evaluation domain

To create the evaluation domain, we use the primitive n-th root of unity $\omega \in \mathbb{F}_p$ for which it holds that $\omega^n = 1$ and $\forall 1 \leq i < n : \omega^i \neq 1$. The evaluation domain H will be constructed as

$$H = \{1, \omega, \omega^2 \dots \omega^{n-1}\}.$$

This domain will be especially useful because it allows for a sparse representation of certain polynomials. This results in numerous benefits, such as faster polynomial evaluation.

Definition 7 (Vanishing polynomial)**.** Denote $Z_H(x)$ the vanishing polynomial for the domain H such that $\forall x \in H : Z_H(x) = 0$ and deg $deg(Z_H) = |H|$.

Since the vanishing polynomial has roots as the elements of H, it could be written as:

$$Z_H(x) = (x - 1)(x - \omega)(x - \omega^2) \dots (x - \omega^{n-1}).$$

However, the domain H also allows for a sparse representation of this polynomial $Z_H(x) = x^n - 1$. To observe that this holds, we verify

$$\forall x \in H : x^n - 1 = (\omega^i)^n - 1 = (\omega^n)^i - 1 = 1^i - 1 = 0.$$

2.2.3 Lagrange basis

In the $\mathcal{P}$lon$\mathcal{K}$ paper, interpolation is performed using the Lagrange basis.

Definition 8 (Lagrange basis)**.** We denote by L_i the ith polynomial of the Lagrange basis over H as:

$$L_i(x) = \begin{cases} 1 & x = \omega^i \\ 0 & \text{otherwise} \end{cases}$$

Given a vector $v = [v_0, v_1, \ldots, v_{n-1}]$, we interpolate it over the domain H via the polynomial:

$$v(x) = \sum_{i=0}^{n-1} v_i L_i(x).$$

The Lagrange basis can be expressed as:

$$L_i(x) = \prod_{k \in H, k \neq \omega^i} \frac{x - k}{\omega^i - k}.$$

The properties of this polynomial are easy to observe. When $x \in H \setminus \{\omega^i\}$ is plugged into $L_i(x)$, then there is exactly one fraction for which $k = x$, which makes the product 0. If $x = \omega^i$, the consequence is even more trivial because all of the fractions would result in $\frac{\omega^i - k}{\omega^i - k} = 1$, and since $k \neq \omega^i$, $L_i(\omega^i)$ results in 1.

Due to the choice of the evaluation domain, the Lagrange basis also has a sparse representation as presented in [12].

$$L_i(x) = \frac{b_i Z_H(x)}{x - \omega^i},$$

$$b_i = \prod_{k \in H, k \neq \omega^i} \frac{1}{x_i - x_k}.$$

Notice that each b_i depends only on the elements of the domain H. So, all of the values b_i could be precomputed.

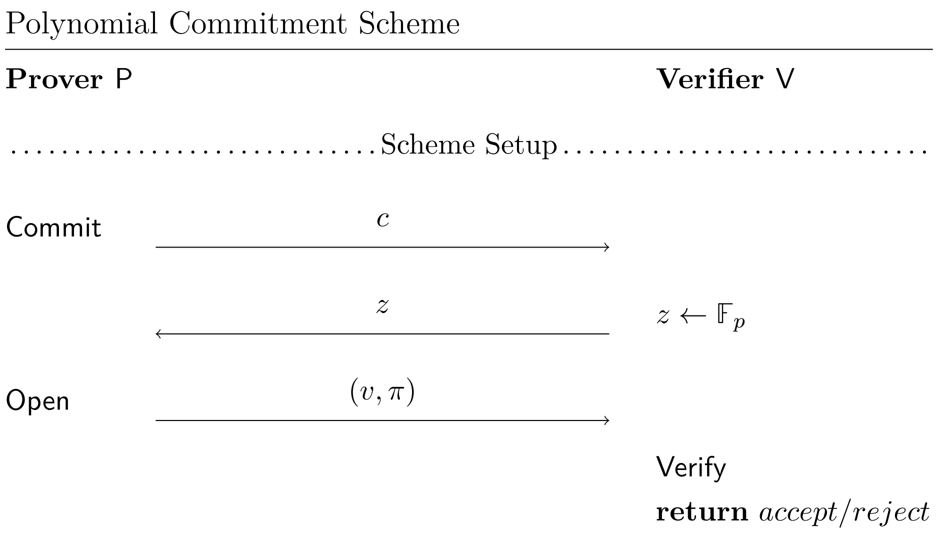
2.3 Polynomial Commitment Scheme

A commitment scheme is a cryptographic primitive that makes it possible to commit to a piece of information, ensuring that it remains unchanged throughout the verification process. One motivation for this mechanism is to prevent the prover from generating proofs depending on the verifier's queries. In the context of $\mathcal{P}$lon$\mathcal{K}$, we will rely on the polynomial commitment scheme, where the prover knows the coefficients of some polynomial $f(x)$ and wants to prove the evaluation of f at a point given by the verifier.

A standard polynomial commitment scheme has the following procedures:

1. $\mathsf{Setup}(1^{\mathsf{n}}) \to \mathsf{pp}$: generate public parameters.
2. $\mathsf{Commit}(f, \mathsf{pp}) \to c$: calculate commitment c which binds P to the polynomial $f(x)$.

3. $\mathsf{Open}(z, \mathsf{pp}) \to (v, \pi)$: given the challenge z, P returns a pair (v, π), where π is the proof that $v = f(z)$.
4. $\mathsf{Verify}(\mathsf{pp}, c, z, v, \pi) \to \mathsf{accept/reject}$: V checks validity of the claim $f(z) = v$.


Polynomial Commitment Scheme
Prover P
Verifier V
Scheme Setup
Commit
c
z
$z \leftarrow \mathbb{F}_p$
Open
(v, π)
Verify
return *accept/reject*

We will require that *Polynomial Commitment Scheme* is *binding* and *hiding.* Loosely speaking, binding means that the commitment binds the prover to a specific polynomial. Hiding property suggests that the prover does not discover the coefficients of the polynomial $f(x)$. We will define it more formally.

Definition 9. Evaluation binding guarantees that any efficient prover cannot generate a convincing π for $v = f(z)$ and π' for $v' = f(z)$.

Computational hiding property says that for polynomial $f(x)$ and its commitment c and any efficient $\mathcal{A}$, the polynomial $f(x)$ is computationally indistinguishable from a randomly chosen polynomial. We will achieve the hiding of the scheme in Section 2.3.2 with a stronger property — zero-knowledge.

The next notion we formalize ensures that if the verifier accepts the proof, the prover needs to have knowledge of the committed polynomial.

Definition 10. Extractable scheme guarantees that for all efficient provers P that takes as input public parameters, degree bound d and outputs commitment c $\exists$ efficient extractor algorithm Ext that has the same input as P and produces polynomial $f(x)$ of bound d explaining all of P answers to evaluation queries.

The extractability of a polynomial commitment is stronger than binding as it was with soundness and knowledge soundness in Chapter 1. It guarantees that for any efficient prover capable of passing all of the checks, the prover must actually “know” a polynomial p of the claimed degree that explains its answers to all evaluation queries.

2.3.1 KZG Polynomial commitment scheme

The KZG polynomial commitment scheme, based on the pairing (elliptic curve) group, was introduced in [2]. It has constant proof size and requires a trusted setup. There are other ways to design polynomial commitment schemes, such as [13] based on hash functions. We will explain KZG since it is used in the $\mathcal{P}$lon$\mathcal{K}$ protocol. This section is heavily inspired by the work of Justin Thaler in *Proofs, Arguments, and Zero-Knowledge* [4]. We will try to explain an imperfect, non-extractable KZG polynomial commitment and provide ideas of the proofs, but for the proper security analysis of KZG, we refer the reader to the Section 15.2 *Proofs, Arguments, and Zero-Knowledge* [4].

The public parameters pp of the KZG is the *structured reference string* SRS consisting of evaluations $\{G, G^{\tau}, G^{\tau^2}, ..., G^{\tau^{d-1}}, G^{\tau^d}\}$, where d is degree bound of committed polynomial and τ is uniformly randomly chosen $\tau \leftarrow \mathbb{F}_p$. The setup of the protocol needs to be trusted because if the prover discovers the value τ, it will allow him to forge proofs and, thus, prove any evaluation. So naturally, the setup either needs to be computed by the verifier or via a secure multiparty computation. Below, we list all public information for the KZG scheme:

1. $\mathbb{G}_1, \mathbb{G}_2$ are pairing friendly groups both under the field $\mathbb{F}_p$ for p prime
2. G_1, G_2 are generators of $\mathbb{G}_1, \mathbb{G}_2$
3. e: is a symmetric bilinear map meaning in $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}'$ the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are equal.
4. d is the upper bound on the polynomial degree
5. SRS evaluations $\{G, G^{\tau}, ..., G^{\tau^d}\}$

Before we proceed to describe the procedures, we need to introduce an observation that allows us to effectively transform equality checks into divisibility checks. The polynomial we get from the division will be used as the proof of evaluation.

Lemma 2. *For any degree d univariate polynomial $f \in \mathbb{F}_p[x]$ the assertion $f(z) = v$ is equivalent to checking if there exists a polynomial $w(x) \in \mathbb{F}_p[x]$ of degree at most $d-1$ such that: $w(x) = \frac{f(x)-v}{x-z}$.*

Proof. In the forward directions, we have a polynomial $f(x)$, and we know that $f(z) = v$. We can construct $\widetilde{f}(x) = f(x) - v$ which is 0 at z. If $\widetilde{f}(z) = 0$ then z is the root of $\widetilde{f}(x)$ so $\widetilde{f}(x)$ could be also written as $\widetilde{f}(x) = (x-z)w(x)$. Now we just reorder the expression:

$$\widetilde{f}(x) = (x-z)w(x),$$

$$w(x) = \frac{\widetilde{f}(x)}{x-z},$$

$$w(x) = \frac{f(x)-v}{x-z}.$$

In the opposite direction, we have polynomial $w(x) = \frac{f(x)-v}{x-z}$ and want to show that $f(z) = v$. If we have $w(x)$ it means that $\widetilde{f}(x)$ is divisible by $x-z$, so z is the root of $\widetilde{f}(x)$ which implies $f(z) = v$. $\square$

Commitment

To commit to $f(x)$ over $\mathbb{F}_p$ P sends $c \in \mathbb{G}_1$ which he claims is $f(\tau)$ encoded as elliptic curve point $G_1^{f(\tau)}$. We use the notation $G_1^{f(\tau)} = [f]_1$. However, above, we have said that the prover should not be able to discover τ under any circumstance, so how does he compute $f(\tau)$? The trick is that he does not have to know τ to compute the commitment. Recall from Section 2.1.1 that function evaluation $f(\tau)$ only requires to know the coefficients of $f(x)$ and $\{G, G^{\tau} \dots G^{\tau^d}\}$ which is provided as the `SRS` public key. This way, the prover is able to compute $G^{f(\tau)}$ without knowing τ.

Opening

To open for challenge z, the prover computes the opening proof polynomial:

$$w(x) = \frac{f(x) - v}{x - z}.$$

By Lemma 2, we know that proving the existence of $w(x)$ is equivalent to $f(z) = v$. Assuming P knows $f(x)$, he is able to calculate $G^{w(\tau)} = [w_1]$ and evaluate $f(z) = v$. Finally, the prover sends $(v, [w]_1)$ to the verifier.

Verification

The verifier knows $c, v, g^{w(\omega)}$ provided by the prover, z generated by himself and the public `SRS`. The only check that needs to be done by V is:

$$e(cG_1^{-v}, G_2) \stackrel{?}{=} e(G_1^{w(\omega)}, G_2^{\omega} G_2^{-z}).$$

Note that from the whole `SRS`, only (G_1, G_2, G_2^{ω}) are needed for the evaluation, which is why in many works, `SRS` is referred to as a prover key and (g, g^{ω}) as a verification key.

To prove correctness:

$$e(G_1^{w(\omega)}, G_2^{\omega} G_2^{-z}) = e(G_1^{\frac{f(\omega)-v}{\omega-z}}, G_2^{\omega-z})$$

$$= e(G_1, G_2)^{f(\omega)-v} = e(G_1^{f(\omega)-v} G_1, G_2) = e(cG_1^{-v}, G_2).$$

The last step of this expansion is true if and only if $c = g^{f(\omega)}$. For an honest prover who knows $f(x)$, the verification procedure would accept with probability 1, which shows the correctness of the protocol. The complete scheme is summarized below.

KZG polynomial commitment scheme

Prover P		Verifer V
Commit	$\xrightarrow{[f]_1}$	
	$\xleftarrow{z}$	$z \leftarrow \mathbb{F}_p$
$f(z) = v$ $w(x) = (f(x) - v)/(x - z)$		
Open	$\xrightarrow{([w]_1, v)}$	
		Verify $e(cG_1^{-v}, G_2) \stackrel{?}{=} e(G_1^{w(\omega)}, G_2^{\omega} G_2^{-z})$

Why does this commitment scheme need to use pairing groups? Notice that the verifier is not able to calculate "multiplication in exponent" $G^{w(\tau)\times(\omega-z)}$. As suggested in the Section 2.1.1, it is possible to solve this by using pairing.

2.3.2 KZG analysis

Hiding

The verifier gets the commitment $c = G_1^{f(\tau)}$, which is one evaluation of the commitment polynomial, and the real value is infeasible to extract due to DLP. The prove also sends tuple $(v, G^{w(\tau)})$ where $w(\tau)$ cannot be obtained from the proof of opening also because of DLP. However, the evaluation $v = f(z)$ is sent in plaintext form. This means that the verifier discovers some information about the committed $f(x)$. The KZG commitment scheme could be potentially computationally hiding if executed just once. If we do not limit the number of openings, the verifier might eventually get d evaluation of the committed polynomial $f(x)$, which would allow him to reconstruct the polynomial by interpolation.

We cannot rely on the KZG polynomial commitment scheme to sufficiently hide the committed polynomial $f(x)$. Therefore, we will use blinding in Section 2.3.2 to achieve zero-knowledge for specific cases.

Binding

Binding property is a little harder to establish. First, we introduce the cryptographic assumption SDH.

Definition 11. Strong Diffie Hellman Assumption SDH assumes that, given pp, P has no efficient way of computing a pair $(z, G^{\frac{1}{\tau-z}})$.

This assumption is stronger than DLP itself. We will show that this assumption needs to hold. Otherwise, P would be able to give two different openings for the same challenge, which breaks the evaluation binding property.

Lemma 3. *Assuming the Strong Diffie-Hellman assumption holds,* P *can provide only one valid opening* v *for any challenge* z *in the KZG scheme.*

Proof. For a contradiction, we can say that P is able to prove two different openings $v \neq v'$ of committed $f(x)$ at a challenge z. This would require calculating two different proofs of opening $w(\omega), w'(\omega)$:

$$w(\tau) = \tfrac{f(\tau)-v}{\tau-z} \quad w'(\tau) = \tfrac{f(\tau)-v'}{\tau-z}$$

$$\begin{aligned} w(\tau) - w'(\tau) &= \frac{v'-v}{\tau-z} \\ \frac{1}{\tau-z} &= \frac{w(\tau)-w'(\tau)}{v'-v} \\ G^{\frac{1}{\tau-z}} &= \frac{1}{v'-v} G^{w(\tau)-w'(\tau)} \end{aligned}$$

We get a contradiction because we can use such a prover to break the SDH assumption. $\square$

Extractable scheme

To make KZG extractable, we need to modify it. The `SRS` will be twice as long containing pairs

$$\{(G_1, G_1^{\psi}), (G_1^{\tau}, G_1^{\tau\psi}), (G_1^{\tau^2}, G_1^{\tau^2\psi}) \ldots (G_1^{\tau^d}, G_1^{\tau^d\psi})\} \text{ for } \psi \leftarrow \mathbb{F}_p, \tau \leftarrow \mathbb{F}_p$$

Definition 12 (Power Knowledge of Exponent assumption)**.** For any efficient $\mathcal{A}$ given access to `SRS`, whenever $\mathcal{A}$ outputs group elements $a, b \in \mathbb{G}$ such that $a = b^{\psi}$ then $\mathcal{A}$ needs to know the coefficients that explain $a = \prod_{i=0}^{d} = G^{c_i\tau^i}$ and $b = \prod_{i=0}^{d} = G^{c_i\tau^i\psi^i}$.

Now, the commitment will be a pair $c = (G_1^{f(\tau)}, G_1^{f(\tau\psi)})$. The calculation of the witness polynomial remains the same, and the verifier now needs to perform two checks:

$$e(G_1^{f(\tau)} G_1^{-v}, G_2) \stackrel{?}{=} e(G_1^{w(\tau)}, G_2^{\tau} G_2^{-z}) \tag{2.1}$$

$$e(G_1^{f(\tau)}, G_2^{\psi}) \stackrel{?}{=} e(G_1^{f(\tau\psi)}, G_2) \tag{2.2}$$

The correctness of Equation (2.1) holds based on the correctness of the previous KZG explanation and the correctness of Equation (2.2) holds because of:

$$\begin{array}{cc} G_1^{f(\tau)} & G_1^{f(\tau\psi)} \\ \prod_i (G_1^{\tau^i})^{c_i} & \prod_i (G_1^{\tau^i\psi})^{c_i} \\ \prod_i (G_1^{\tau^i})^{c_i} & \prod_i (G_1^{\tau^i})^{c_i\psi} \\ [f]_1 & [f]_1^{\psi} \end{array}$$

Very intuitively, this variant of the KZG polynomial commitments scheme remains binding because the modification scheme only extends to one more verification check, but the first one already binds the commitment to a specific polynomial. To prove the extractability of the scheme, we rely on the second verification check, which simplifies to $g^{q(\psi\omega)} = g^{\omega} g^{\psi}$. From the PKoE, it needs to follow that $\mathcal{A}$ has knowledge of the coefficients and, therefore, knows the polynomial $f(x)$. More details about the proof can be found in Proofs, Arguments, and Zero-Knowledge [4].

Blinding

We have discussed the KZG commitment scheme, which is not zero-knowledge. Openings clearly reveal some information about the committed polynomial. To make $\mathcal{P}$lon$\mathcal{K}$ zero-knowledge, we will randomize the committed polynomial. The $\mathcal{P}$lon$\mathcal{K}$ protocol suggests to blind a polynomial $f(x)$ as follows:

$$\widetilde{f}(x) = Z_H(x) \times \text{some expression} + f(x)$$

Notice that the polynomials $f(x), \widetilde{f}(x)$ agree on all points of domain H because Z_H evaluates to zero over H. The checks of the $\mathcal{P}$lon$\mathcal{K}$ protocol only verify properties on the evaluation domain H, which is why it does not matter that $f(x), \widetilde{f}(x)$ might not match outside of H.

Consider that some polynomial $f(x)$ should be opened at challenge points $(z_1 \dots z_k)$ randomly chosen by the verifier. We alter the committed polynomial with uniformly randomly and independently chosen blinding scalars $(b_1, b_2, \dots, b_{k+1}) \leftarrow \mathbb{F}_p$ as

$$\widetilde{f}(x) = (b_1 + b_2 x + b_3 x^2 ... + b_{k+1} x^k) Z_H(x) + f(x)$$

Notice that the degree of the blinding polynomial depends on k, the number of openings. Does the opening of $\widetilde{f}(x)$ reveal any information about $f(x)$? Well, of course, it does, because for $\forall x \in H : f(x) = \widetilde{f}(x)$. However, $|H|$ is orders of magnitude smaller than $|\mathbb{F}_p|$, and the probability of randomly choosing an element of H is considered to be negligible. Let's examine the case when $x \neq H$.

Definition 13 (k-blinded polynomial)**.** A polynomial $\widetilde{f}(x)$ is constructed as

$$\widetilde{f}(x) = b_k(x) Z_H(x) + f(x)$$

where $b_k(x) = b_1 + b_2 x + b_3 x^2 \dots + b_{k+1} x^k + b_{k+2} x^{k+1}$ is a blinding polynomial with coefficients chosen independently and uniformly randomly and $Z_H(x)$ is polynomial vanishing on domain H.

We would like to show that proving at most k polynomial evaluation of k-blinded polynomial using the KZG polynomial commitment scheme is honest verifier zero-knowledge as captured by Definition 5. In the context of KZG, the transcript of k opening proofs for committed polynomial $f(x)$ with an evaluation proof polynomials $w_1(x), \dots w_k(x)$ will be denoted as

$$(C = G_1^{f(\tau)}, \bar{z} = \{z_1, \dots, z_k\}, \bar{v} = \{v_1, \dots, v_k\}, \bar{W} = \{G_1^{w_1(\tau)}), \dots, G_1^{w_k(\tau)})\}.$$

Definition 14 (Honest Transcript)**.** The honest transcript is $(C, \bar{z}, \bar{v}, \bar{W})$ generated from the interaction of a non-malicious prover and verifier where the prover knows the committed polynomial and the verifier accepts the provided openings.

To prove that blinding guarantees honest verifier zero-knowledge, we need to show the construction of a simulator that produces an indistinguishable transcript from an honest transcript. First, we describe a few useful observations. We will again define the zero-knowledge property specific to this case.

Definition 15 (Honest Verifier Zero-Knowledge for KZG analysis)**.** The polynomial commitment scheme $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ is honest verifier zero-knowledge (HVZK) if there is an efficient simulator Sim for any combination of a function $f(x)$ such that the distribution:

$$(C, \bar{z}, \bar{v}, \bar{W}) : \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^n) \\ C \leftarrow \mathsf{Commit}(f, \mathsf{pp}) \\ (v, W) \leftarrow \mathsf{Open}(z, \mathsf{pp}) \end{array}$$

is computationally indistinguishable from

$$(C, \bar{z}, \bar{v}, \bar{W}) \leftarrow \mathsf{Sim}(\mathsf{pp})$$

Lemma 4. *$\forall x \in \mathbb{F}_p, a \leftarrow \mathbb{F}_p$ where a is non-zero element the function $f_a(x) = x + a$ defines a random permutation of elements of $\mathbb{F}_p$.*

Proof. To show that $f_a(X)$ is a permutation we want:

$$\{0, 1, 2, \dots, p\} = \{0 + a, 1 + a, 2 + a, \dots, p + a\}$$

Assume there exist $x_1, x_2 \in \mathbb{F}_p, x_1 \neq x_2 : f_a(x_1) = f_a(x_2)$. Then $x_1 + a = x_2 + a \implies x_1 = x_2$, which is a contradiction. Therefore we can say that $f_a(x)$ is injective. Since adding two field elements of the field results in another element of the same field the f_a needs to be subjective. From there it follows that $f_a(x)$ is a permutation function determined by the randomness of a. □

Lemma 5. *For the generator G_1 of a group $\mathbb{G}_1$ and a random element $a \leftarrow \mathcal{U}_{\mathbb{F}_p}$ the group element G_1^a is a uniformly random element of the group generated by G_1.*

Proof. Since a is chosen uniformly at random from the field, and G_1^a maps each element in the field to a unique element in the group $\mathbb{G}_1$, the image of this function must also be uniformly distributed across the group. □

Lemma 6. *Given prime p and a polynomial over a finite field $\mathbb{F}_p$ of degree bound k with independently randomly chosen coefficients we can evaluate it at k independently randomly chosen values and get k independent uniformly random field elements.*

The above lemma is used in constructing k-wise independent hash functions first described by [14], therefore we do not include the proof. The intuition is that a polynomial of degree k is uniquely defined by $k+1$ points, which can be interpolated. When provided only k points, each polynomial of degree k is equally probable. Since each of the polynomials has a uniform probability, we get that any k-tuple of distinct arguments is equally likely to be mapped to any k-tuple of evaluations.

Lemma 7. *Evaluating k-blinded polynomial at values $(x_1, \dots, x_k) \in \mathbb{F}_p^k \setminus H$ gives independent uniformly randomly distributed evaluations.*

Proof. We know that k evaluations of the blinding polynomial $b(x)$ will give k independent uniformly random evaluations based on the Lemma 6. If those are added to some fixed polynomial $f(x)$, then the blinded polynomial $\widetilde{f}(x)$ produces evaluations that are independently random based on lemma Lemma 4. □

With all of the tools needed, we can finally show that commitment to a blinded polynomial $\widetilde{f}(x)$ reveals no information about the former polynomial $f(x)$.

Theorem 2 (KZG Blinding). KZG commitment scheme with k-blinded polynomial is honest verifier zero-knowledge for k openings.

The principle of this proof is to construct a simulator Sim that can produce honest-looking transcripts (C, z, v, W). More formally, the transcripts generated by the simulator Sim need to be indistinguishable from actual *honest transcript.* In this construction, we rely on the honesty of V to independently sample challenges from $\mathcal{U}_{\mathbb{F}_p}$. The zero-knowledge property could be broken by challenging P only on points from the evaluation domain H. That would undo the blinding, and the verifier would get an evaluation of $p(x)$, which clearly leaks some information. If the verifier was honest, then sampling $x \leftarrow \mathcal{U}_{\mathbb{F}_p}$ such that $x \in H$ happens only with probability $\frac{|H|}{|\mathbb{F}_p|}$ which is negligible.

Proof. First, we analyze the distribution of the honest execution.

- C: We know that evaluation of $\widetilde{f}(X)$ give independent uniformly distributed results and thanks to lemma Lemma 5 $G_1^{\widetilde{f}(X)}$ is also uniformly randomly distributed.
- z: By definition of the protocol $z \leftarrow \mathcal{U}_{\mathbb{F}_p}$.
- v: Since $z \leftarrow \mathcal{U}_{\mathbb{F}_p}$ we know that $\widetilde{f}(z)$ will be also uniformly distributed thanks to Lemma 7.
- W: Is uniquely determined by C, z, v. Let's first look at the distribution of $w(\tau) = \frac{\widetilde{f}(\tau)-v}{\tau-z}$. In the numerator, we calculate $C - v$, and we already know that both are uniformly distributed. This is divided by $\tau - z$ where τ is fixed and $z \leftarrow \mathcal{U}_{\mathbb{F}_p}$. Therefore, $w(\tau)$ is the division of two random field elements, by Lemma 5, W will also be random.

The construction of the simulator $\mathsf{Sim}(d)$ is surprisingly simple. Sim can create random polynomial $r(x) \leftarrow \mathbb{F}_p^d[X]$ by independently sampling random coefficients. By the Lemma 6 $r(x)$ can give up to d random evaluations, which are sufficient because $d \gg k$. Consequently, the Sim will just run the prescribed prover algorithm for polynomial $r(x)$, and all elements of the generated transcript will be as well uniformly randomly distributed. We know that the verification of the verifier would accept this solution because Sim can calculate the witness for $r(z) = v$ in the prescribed way. □

The simulator can generate a valid transcript indistinguishable from a honest transcript just by picking a random function. This means that the verifier does not discover anything about the committed blinded polynomial, and the only thing V can tell is if the evaluation of the committed polynomial is correct. That is exactly what we wanted to achieve.

You might be concerned about the honesty assumption about the verifier. However, in a practical scenario, the $\mathcal{P}$lon$\mathcal{K}$ protocol is run in a non-interactive way where the prover interacts with a random oracle using the Fiat-Shamir heuristic. The randomness of the challenges is guaranteed by this oracle, which is assumed to be trusted and not by a verifier that can be potentially malicious.

Batched Multivariate PCS

Say that the committer knows polynomials f_1, f_2 and wants to prove evaluations at randomly selected challenge $f_1(z) = v_1, f_2(z) = v_2$. Rather than checking each claim independently, it is possible to batch them. We can perform a batched check using uniformly randomly selected u.

$$uf_1(z) + f_2(z) \stackrel{?}{=} uv_1 + v_2 \tag{2.3}$$

Naturally, this approach is correct because if $f_1(z) = v_1, f_2(z) = v_2$, then for any u, the Equation (2.3) holds. Notice that we can think of both $uf_1(z) + f_2(z)$ and $uv_1 + v_2$ as linear functions with variable u. If two linear functions are not identical, they have at most one intersection point, so the soundness error of the batching is at most $1/\mathbb{F}_p$, which is sufficiently low. With a very high probability, it holds that:

$$f_1(z) = v_1, f_2(z) = v_2 \iff uf_1(z) + f_2(z) = uv_1 + v_2$$

This suggests that it is sufficient to verify the opening of the batched polynomial. The commitment $[f_{batch}]_1$ can be computed by the verifier as $u[f_1]_1 + [f_2]_1$. Therefore, the prover just needs to send commitments $[f_1]_1, [f_2]_1$, opening $f_{batch}(z)$ and proof to the opening. Why is this better? The prover only needs to calculate one proof of the opening for the batched polynomial f_{batch}. If we performed separate polynomial commitment schemes for $f_1(x), f_2(x)$, there would need to be two opening proofs. The additional work put on the verifier and the prover is far lower than computing and verifying the opening proof.

Batched Multivariate PCS

Commit $\xrightarrow{[f]_1, [f]_2}$

$\xleftarrow{z, u}$ $z, u \leftarrow \mathbb{F}_p$

$f_{batch}(x) = uf_1(x) + f_2(x)$
$f_{batch}(z) = uf_1(z) + f_2(z)$
compute opening proof π

Open $\xrightarrow{(f_{batch}(z), \pi)}$

$[f_{batch}] = u[f]_1 + [f]_1$
Verify opening $f_{batch}(z)$

2.3.3 KZG in the evaluation form

The standard KZG polynomial commitment scheme was described using polynomials in the evaluation form. However, one can bypass interpolation and work entirely in the evaluation representation as explained by Justin Drake [15]. In this section, we will show how it is possible to use the polynomial commitment scheme with polynomials in the evaluation form.

Sparse Lagrange Basis

As mentioned in the $\mathcal{P}$lon$\mathcal{K}$ paper, the Lagrange basis has sparse representation over the evaluation domain generated by a primitive n-th root of unity. It could be written $\forall 1 \leq i \leq |H|$ as:

$$\forall 1 \leq 1 \leq |H| : L_i(x) = \frac{b_i Z_H(x)}{x - \omega^i}.$$

From the findings in Barycentric Lagrange interpolation [12], the constant terms b_i are

$$b_i = \prod_{j \neq i} \frac{1}{\omega^i - \omega^j}.$$

Notice that b_i depend only on the domain H and thus can be precomputed in the protocol setup.

Commitment

Let $f(x)$ be a function with a degree bound n. We will consider it in evaluation form on H as $\{f(1), f(\omega), f(\omega^2) \dots f(\omega^{n-1})\}$, which means that it could be written as:

$$f(x) = \sum_{i=0}^{n-1} f(\omega^i) L_i(x).$$

Now, the commitment could be calculated as follows:

$$[f]_1 = G^{f(\tau)} = G^{\sum_{i=0}^{n-1} f(\omega^i)L_i(\tau)} = \prod_{i=0}^{n-1} (G^{L_i(\tau)})^{f(\omega^i)}$$

Notice that $G^{L_i(\tau)}$ are actually commitment to the Lagrange basis $[L_i(\tau)]_1$. Again, since the Lagrange basis is dependent just on the domain H, this commitment could be precomputed.

Evaluation proof

In the standard form, the opening proof is computed as

$$\frac{f(X) - f(z)}{x - z}.$$

As a result, the opening proof polynomial can be computed as:

$$w(\tau) = \sum_{i=0}^{n-1} \frac{f(\omega^i) - f(z)}{\omega^i - z} L_i(\tau),$$

$$[w]_1 = \prod_{i=0}^{n-1} \frac{f(\omega^i) - f(z)}{\omega^i - z} [L_i]_1,$$

$$\text{where } f(z) = \sum_{i=0}^{n-1} \frac{b_i Z_H(z)}{z - \omega^i} f(\omega^i).$$

2.4 Fiat-Shamir transform

A SNARK by definition must be non-interactive. The $\mathcal{P}$lon$\mathcal{K}$ protocol achieves this by applying the Fiat-Shamir heuristic [5] to an interactive protocol. Essentially, the prover uses a random oracle to generate the challenges instead of the verifier. Incorrect application of this transform may introduce vulnerabilities as described in Weak Fiat-Shamir Attacks [6].

In the case of $\mathcal{P}$lon$\mathcal{K}$, the random oracle is a cryptographic hash function $\mathcal{H}$, which takes variable length input and gives fixed length output. The input to the hash is the protocol transcript, which is a concatenation of:

- common preprocessed input described in setup
- public input PI
- polynomial commitments computed by the prover so far
- polynomial openings computed by the prover so far

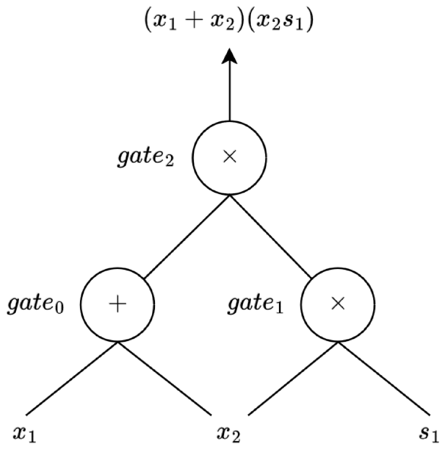
Given the protocol transcript and access to the random oracle, anyone can reconstruct the challenges. Therefore the verifier can easily recognize if the prover used challenges generated by the determined random oracle.

2.5 Arithmetization

The program on which the protocol is run is usually written in a high-level language, which is eventually compiled into an arithmetic circuit. We will not go into details, but a curious reader can look at paper [16] about Circom. As mentioned in Section 1.1, any binary circuit could be transformed into an arithmetic circuit, and in the following section, we will explain how to verify the integrity of the computation of an arithmetic circuit.

2.5.1 Gate equations

Consider a circuit for computing $(x_1 + x_2)(x_2 s_1)$ where x_1, x_2 are public inputs, and s_1 is a secret number, which only the prover knows. Since we know that there are at most binary gates in the circuit, we can think of each gate as a row of a table, where columns are in order: left input, right input, and gate output.



	w_l	w_r	w_o
$gate_0$	x_1	x_2	$x_1 + x_2$
$gate_1$	x_2	s_1	$x_2 s_1$
$gate_2$	$x_1 + x_2$	$x_2 s_1$	$(x_1 + x_2)(x_2 s_1)$
	left input	right input	gate output

Figure 2.2 Toy Circuit

The objective is to efficiently encode an arithmetic circuit into a set of polynomials. The first step is to encode each gate as an equation. Considering only gates for addition and multiplication, each gate could be written as one of the following:

$$\textit{addition:}\ \text{left input} + \text{right input} = \text{gate output} \tag{2.4}$$

$$\textit{multiplication:}\ \text{left input} \times \text{right input} = \text{gate output} \tag{2.5}$$

That is a good start, but we could create an equation that can check the validity of both gate types. First, we denote the columns of the computational table as vectors w_l, w_r, w_o. By assigning $x_1 = 2, x_2 = 1, s_1 = 3$ for the Figure 2.2 we get $w_l = [2, 1, 3], w_r = [1, 3, 3], w_o = [3, 3, 9]$. The values in the computation table represent $(\mathbb{x}, \mathbb{w})$, where $\mathbb{x}$ are values of x_1, x_2 and the rest of the circuit is private $\mathbb{w}$.

To specify the type of the gate, we use selectors: q_l left, q_r right, q_o output, q_m multiplication, q_c constant. These are vectors of size n selecting the type of gate that we want to use. For example, if we are dealing with gate i, that is a multiplication gate, then $q_{m_i} = 1$; otherwise, it is always set to 0. The selectors could be combined to obtain the equation for gate i:

$$w_{l_i} w_{r_i} q_{m_i} + w_{l_1} q_{l_i} + w_{r_i} q_{r_i} + w_{o_i} q_{o_i} + q_{c_i} + PI_i = 0. \tag{2.6}$$

Consider the example circuit and variable assignment $x_1 = 2, x_2 = 1, s_1 = 3$, and for simplicity, all unassigned selectors will have value 0. Then, it is possible to perform these operations:

- *addition:* $q_{l_i} = q_{r_i} = 1, q_{o_i} = -1$
  For the addition gate, we want to keep only the addition and output terms of the equation. The rest will cancel out, thanks to the remaining selectors being 0.
- *multiplication:* $q_{m_i} = 1, q_{o_i} = -1$
  To engage a multiplication gate, we assign the selector such that only the multiplication and output terms of the equation remain.
- *constant assignment:*
  Besides the operations of addition and multiplication, it is possible to do constant assignments (not illustrated on the example circuit). Say that we want the left input of a $gate_i$ to be equal to some constant c. To construct the equation for this gate, we can assign $q_{l_i} = 1, q_{c_i} = -c$, and the equation then simplifies to $w_{l_i} = c$, checking exactly what we want. This works also for the right input, we just set the selectors as $q_{r_i} = 1, q_{c_i} = -c$.

PI_i is public input, which is engaged only for the public input nodes (leaves of the graph). For all inner nodes, $PI_i = 0$. Notice that using the selector polynomial, some terms are canceled out so that we get to the separate gate constraints described in the beginning (2.4). Below is a table with the explicit assignment of the selector for the specific example.

	w_l	w_r	w_o	q_m	q_l	q_r	q_o	q_c
$gate_0$	2	1	3	0	1	1	-1	0
$gate_1$	1	3	3	1	0	0	-1	0
$gate_2$	3	3	9	1	0	0	-1	0

2.5.2 Transcription into polynomials

It is possible to write express computation of each gate in the circuit as Equation (2.6). But a much more elegant way is to condense all of the constraints into a single equation by interpolating the vectors.

$$\begin{aligned} a(x) &= \textstyle\sum_{i=0} L_i(x)w_{l_i} & b(x) &= \textstyle\sum_{i=0} L_i(x)w_{r_i} \\ c(x) &= \textstyle\sum_{i=0} L_i(x)w_{o_i} & q_l(x) &= \textstyle\sum_{i=0} L_i(x)q_{l_i} \\ q_{r_i}(c) &= \textstyle\sum_{i=0} L_i(x)q_{r_i} & q_m(x) &= \textstyle\sum_{i=0} L_i(x)q_{m_i} \\ q_{c_i}(x) &= \textstyle\sum_{i=0} L_i(x)q_{c_i} & PI(x) &= \textstyle\sum_{i=0} L_i(x)PI_i \end{aligned}$$

This enables to represent gate constraints for the whole circuit C as:

$$a(x)b(x)q_m(x) + a(x)q_l(x) + b(x)q_r(x) + c(x)q_o(x) + q_{c_i} + PI(x) = 0 \qquad (2.7)$$

2.6 Circuit checks

We can encode arbitrary circuits with polynomials. The following objective is to convince the verifier that the prover knows a secret witness $\mathtt{w}$, for which $\mathsf{C}(\mathtt{x}, \mathtt{w}) = 0$. The prover needs to guarantee the integrity of computing the circuit, and it turns out the verifier needs a lot of convincing since there are numerous ways to cheat. The prover needs to show the following:

1. Correct public inputs are provided (input check).
2. Gates are computed correctly (gate check).
3. Gates are wired correctly (wiring check).
4. The output of the circuit is zero (output check).

Most of the checks are descriptive, besides the wiring check. Considering the example circuit (2.2), this check ensures that outputs of $gate_0, gate_1$ are indeed provided as inputs to $gate_2$. This is also referred to as copy constraints because we want to make sure that the output of some gate is correctly copied to the input of another gate. Before proceeding to the implementation of this check, we will show how to perform zero tests on some domains.

2.6.1 Zero Test

Observe that if some polynomial $f(x)$ is zero on H, then every element of H has to be its root. The vanishing polynomial that has roots in all elements of the domain was denoted as $Z_H(x)$. That means we can factor the polynomial as $f(x) = g(x)Z_H(x)$. In other words, if a polynomial is zero on H, it has to be divisible by the vanishing polynomial $Z_H(x)$.

This idea test can be transformed into a standalone *Zero Test* protocol using a polynomial commitment scheme. Below is a sketched diagram of how it works. In the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol, we will use this check, but the zero checks are not run independently on each polynomial. Instead, they are batched into the quotient polynomial $t(x)$ in Section 3.4.

Zero test

$$\begin{array}{lcl}
\text{Prover } \mathsf{P} & & \text{Verifer } \mathsf{V} \\
f(x), g(x) = f(x)/Z_H(x) & \xrightarrow{[f]_1, [g]_1} & \\
 & \xleftarrow{z} & z \leftarrow \mathbb{F}_p \\
\text{evaluate } f(z), g(z) & & \\
\text{compute opening proofs } \pi_1, \pi_2 & & \\
 & \xrightarrow{f(z), g(z), \pi_1, \pi_2} & \\
 & & \text{verify KZG openings } \pi_1, \pi_2 \\
 & & Z_H(z) = z^n - 1 \\
 & & f(z) \overset{?}{=} g(z) Z_H(z)
\end{array}$$

Theorem 3. Zero-test is correct and computationally sound, assuming $\frac{d}{p}$ is negligible.

Proof. We have already discussed the properties of KZG in section 2.3.2 and the Schwartz-Zippel lemma in Section 2.2.1.

This check satisfies the correctness because if $f(x)$ is zero on H, then it can be factored into $Z_H(x)g(x)$. The verifier can check the KZG opening and compare polynomials using the Schwartz-Zippel lemma.

The zero test is also sound because if the polynomial $f(x)$ is not zero on the whole domain H, then it cannot be divided by $Z_H(x)$. As a result, the prover cannot provide $g(x)$ that would satisfy $f(z) = g(z)Z_H(z)$ with non-negligible probability, which follows from the Schwartz-Zippel lemma. □

2.6.2 Output Check

Notice that the output of the circuit is always the output of the last gate of the circuit. This means that the output of the circuit will be encoded as the last element of the witness. All the prover needs to do is show that $\mathsf{C}(\omega^{n-1}) = 0$. This can be performed in a secure way using a polynomial commitment scheme where the verifier asks to open the polynomial $c(x)$ at ω^{n-1}.

Lemma 8. *The output check is correct and sound.*

The properties of this check depend on the polynomial commitment scheme, and we have already discussed the validity of KZG in Section 2.3.2.

2.6.3 Gate Check

We have managed to encode the gate constraints by the equation Equation (2.7). Using the zero test, it is possible to verify that $\forall x \in H$:

$$a(x)b(x)q_m(x) + a(x)q_l(x) + b(x)q_r(x) + c(x)q_o(x) + q_{c_i} + PI(x) = 0$$

Lemma 9. *The gate check is correct and computationally sound*

Proof. This follows from the properties of the zero test Section 2.6.1 □

2.6.4 Input Check

We have encoded the public input is encoded using Equation (2.7). That means checking if correct public inputs were provided to the circuit can be encoded in the gate equation.

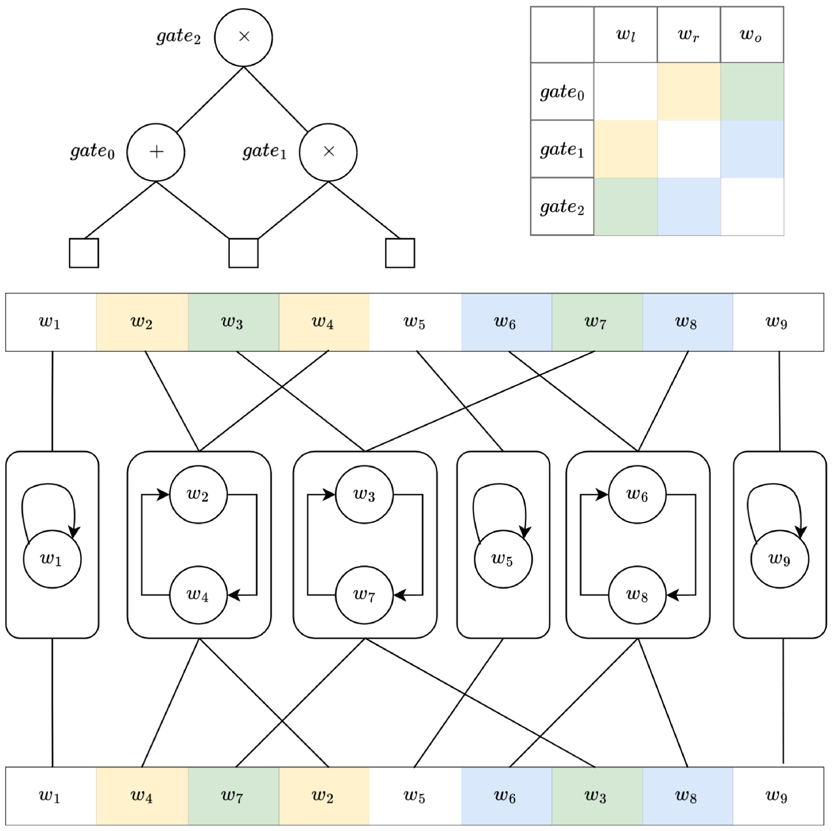
Lemma 10. *The public input check is correct and sound*

This follows from Lemma 9.

2.6.5 Wiring Check

The structure of a circuit requires some values to be identical. The edges (wires) indicate that the output of the gate at one side should be passed as the input to the other gate. To ensure that the program is computed correctly, the prover needs to show that the values were copied correctly.

This check is performed using permutations. The structure of the circuit induces some equivalence classes. We will try to show that the copy constraints are satisfied by performing a permutation on the equivalence classes. For this purpose, we will choose rotation because it changes the position of every element. The diagram below Section 2.6.5 shows how the permutation function would look for Figure 2.2.



$gate_2$
$\times$
$gate_0$
$+$
$gate_1$
$\times$
w_l
w_r
w_o
$gate_0$
$gate_1$
$gate_2$
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9
w_1 w_4 w_7 w_2 w_5 w_6 w_3 w_8 w_9

Figure 2.3 Rotation of equivalence classes

It might look complicated, but essentially, we just did a clockwise rotation on the groups of variables of the circuit. Since we have changed the variables with equal values, the circuit should work the same as before and produce the same output. If all of the gate equations were true before the shuffling, then they would succeed also after this shuffling. On the other hand, if the prover did not do the arithmetization correctly and the gate wiring is not the same as in the former circuit, this check would detect it (with very high probability). That is the core idea that will be used in the permutation check.

Lemma 11. *Consider vector $v = [v_1, v_2 \dots v_n]$, constraints $v_i = v_j, v_k = v_l \dots$ and a permutation function σ that permutes the equivalence classes determined by the constraints. All of the contains are satisfied if and only if $v = \sigma(v)$.*

Proof.

"$\Rightarrow$" If all of the contains are satisfied and ω permutes only the indices inside the permutation classes, then the values remain unchanged.

"$\Leftarrow$" If the vector remains unchanged after performing the permutation σ, then all of the elements in the equivalence classes induced by the permutation function must be the same, and thus, the constraints hold.

$\square$

3 The PlonK protocol

With the knowledge from the previous chapter, we can get to the description of the protocol itself. $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol has a trusted setup in the form of the *structured reference string* `SRS` that can be reused for proofs on multiple circuits. The natural benefit is that the setup parameters could be used indefinitely, enabling $\mathcal{P}\mathfrak{lon}\mathcal{K}$ to be a multi party protocol. This fundamental property makes the protocol valuable and promising for application in blockchain technologies. It is essential to mention that the security is based on the hardness of the discrete logarithm problem on elliptic curves. The original paper uses KZG [2] (Kate, Zaverucha, Goldberg) commitments, but the protocol could be altered to use another polynomial commitment scheme. The KZG commitments have constant size.

protocol	proof size	public parameters	verifier time	trusted setup
Groth16	$\mathcal{O}(1)$	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(1)$	per circuit
Plonk	$\mathcal{O}(1)$	$\mathcal{O}(\lvert C\rvert)$	$\mathcal{O}(1)$	universal
Bulletproofs	$\mathcal{O}(\log\lvert C\rvert)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert C\rvert)$	no
STARK	$\mathcal{O}\left(\log^2\lvert C\rvert\right)$	$\mathcal{O}(1)$	$\mathcal{O}(\log\lvert C\rvert)$	no
DARK	$\mathcal{O}(\log\lvert C\rvert)$	$\mathcal{O}(1)$	$\mathcal{O}(\log\lvert C\rvert)$	no

Table 3.1 Comparison of protocols

The table above presented on Dan Boneh's lecture [17] compares the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol against other protocols. The big benefit is that the protocol has a constant size of the proof composed of 9 polynomial commitments and 6 polynomial openings. The public parameter `SRS` is directly proportional to the size of the proof, as we will show in Section 3.7. The trusted setup is universal in the sense that the public parameter can be used for any circuit of a given bound on the number of gates.

In addition to the trusted one-time setup, the protocol has a setup phase for computing common preprocessed input for a circuit. The rest of the protocol is split into five rounds. Finally, there is an algorithm for the verifier, but we will not get into the details as the verifier simply checks all parts of the proof. The final proof is the batched KZG polynomial commitment that proves multiple evaluations on multiple points. So the verifier algorithm does some trivial validity checks to verify that elements of the correct field and groups were provided, then reconstructs proof fragments and finally does batch verification. The simplified protocol is shown in the diagram below. In the next chapter, we will show a complete prover algorithm diagram.

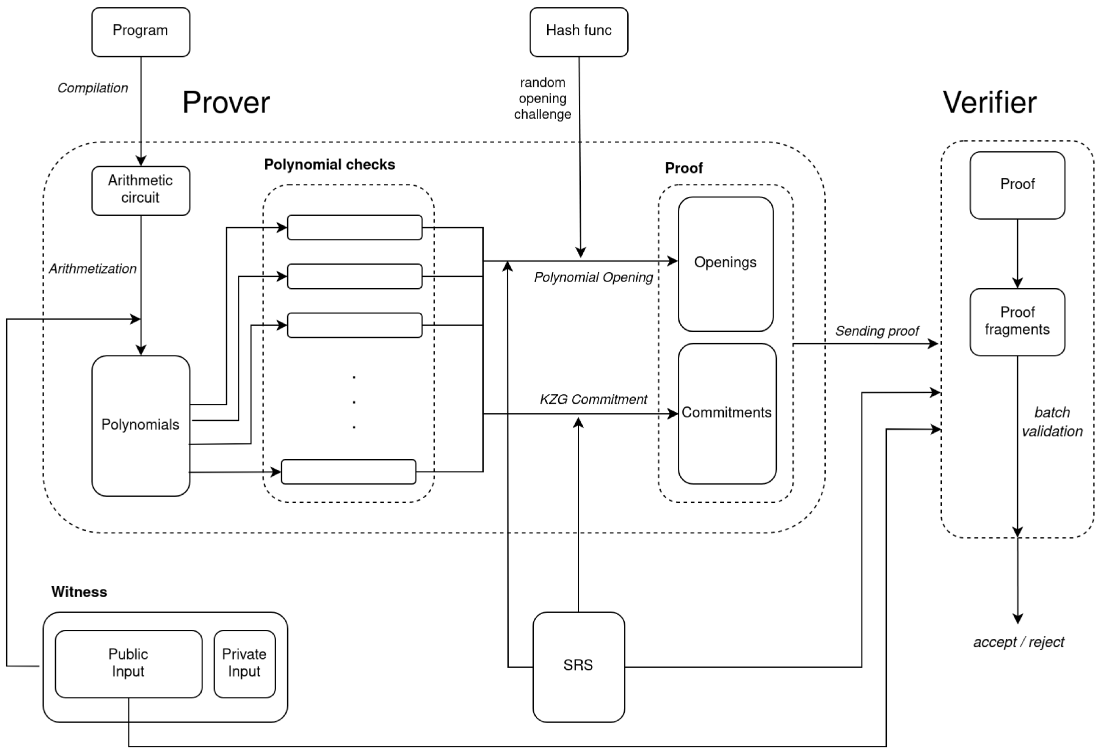


Figure 3.1 Simplified protocol

3.1 Setup algorithm

The setup algorithm describes the calculation of the common pre-processed input. It is not considered a protocol round. The only trusted part is the public key *structured reference string* `SRS` that is not tied to the structure of an arithmetic circuit but to the upper bound of the number of circuit gates. This means the preprocessed setup can be updated for new circuits while the `SRS` can be reused. As mentioned earlier, the `SRS` cannot be generated by the prover because discovering the generation key τ would allow one to forge commitments and, therefore, invalidate the whole protocol.

3.1.1 Public Information

Since the $\mathcal{P}$lon$\mathcal{K}$ protocol runs KZG, all of the parties should have access to the information needed in KZG polynomial commitment scheme

$$(\mathbb{F}_p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, G_1, G_2, G_t, \texttt{SRS}).$$

Revise that $\mathbb{F}_p$ is a prime field, $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ are groups of points on elliptic curves and e is efficiently computable group pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. Lastly G_1, G_2 are group generators of $\mathbb{G}_1, \mathbb{G}_2$. Moreover, there needs to be a KZG setup ceremony generating the *structured reference string* `SRS`, which is public.

3.1.2 Preprocessed Input

Below is summed up the common preprocessed input that is available to both parties. Each of the components will be described in detail in the following rounds.

- n number of gates in the arithmetic circuit
- $k_1, k_2 \in \mathbb{F}_p$ are needed to create cosets k_1H, k_2H such that the union $H' = H \cup k_1H \cup k_2H)$ contains $3n$ distinct elements. This will be further explained in the Section 3.3.
- permutation function: $\sigma^* : [3n] \to H'$ which is rotation of equivalence classes in the witness w as described in Section 2.6.5
- selector polynomials: $q_m(x), q_l(x), q_r(x), q_o(x), q_c(x)$ interpolated from selector vectors q_m, q_l, q_r, q_o, q_c introduced in Section 2.5

$$\begin{aligned} q_m(x) &= \textstyle\sum_{i=1}^{n} L_i(x)q_{m_i} & q_o(x) &= \textstyle\sum_{i=1}^{n} L_i(x)q_{o_i} \\ q_l(x) &= \textstyle\sum_{i=1}^{n} L_i(x)q_{l_i} & q_c(x) &= \textstyle\sum_{i=1}^{n} L_i(x)q_{c_i} \\ q_r(x) &= \textstyle\sum_{i=1}^{n} L_i(x)q_{r_i} \end{aligned}$$

- permutation polynomials: $S_{\sigma_1}, S_{\sigma_2}, S_{\sigma_3} : [n] \to H'$ interpolated from σ^*

$$\begin{aligned} S_{\sigma_1}(x) &= \textstyle\sum_{i=1}^{n} L_i(x)\sigma^*(i) \\ S_{\sigma_2}(x) &= \textstyle\sum_{i=1}^{n} L_i(x)\sigma^*(n+i) \\ S_{\sigma_3}(x) &= \textstyle\sum_{i=1}^{n} L_i(x)\sigma^*(2n+i) \end{aligned}$$

3.2 Round 1

3.2.1 Computing wire polynomials

The proof generation starts by computing the wire polynomials and committing to them. Recall that we get the witness values in the arithmetization as columns of the computational table. The domain is chosen as $H = \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$ where $\omega^n = 1$. Choosing such a domain allows for a sparse representation of the vanishing polynomial as $Z_H(x) = x^n - 1$.

By pairing this domain with the witness w, we get the wire polynomials in the evaluation domain. To make commitments, the polynomial needs to be in the coefficient form to evaluate it at the point τ. Conversion to the coefficient form can be achieved by applying the Lagrange interpolation.

Finally, they should be blinded to ensure that the commitment to the wire polynomials does not leak information about the wire polynomial. As described in Theorem 2, this construction maintains the zero-knowledge property. The prover must uniformly randomly sample 9 blinding scalars $(b_1, b_2, \ldots, b_9)$. In round 1, there are nine sampled blinding scalars, even though we only need 6. The remaining will be used in the following rounds.

Round 1

```
1 :  $(b_1, \ldots, b_9) \leftarrow \mathbb{F}_p^9$
2 :  $a(x) = (b_1 x + b_2) Z_H(x) + \sum_{i=0}^{n-1} w_i L_i(x)$
3 :  $b(x) = (b_3 x + b_4) Z_H(x) + \sum_{i=0}^{n-1} w_{n+i} L_i(x)$
4 :  $c(x) = (b_5 x + b_6) Z_H(x) + \sum_{i=0}^{n-1} w_{2n+i} L_i(x)$
5 :  return $[a]_1, [b]_1, [c]_1$
```

3.3 Round 2

This round ensures the circuit's copy constraints. We will start by showing that constraints hold for inter-vector checks and then extend them to intra-vector checks.

3.3.1 Permutation check

Say that an adversary gives you two vectors $p = [p_1, p_2 \ldots p_n], q = [q_1, q_2 \ldots q_n]$, and you need to decide if they are permutations of each other. So we check if there is a σ such that $\forall i \in [n] \exists j \in [n] : \sigma(p_i) = q_j$. You could compare elements one by one, but that is too slow. There is a smarter way to do this. A good start is to contract the vectors into a single value. So, we can perform, for example, something like a product check where we compare:

$$p_1 \cdot p_2 \ldots \cdot p_n \stackrel{?}{=} \sigma(q_1) \cdot \sigma(q_2) \ldots \cdot \sigma(q_n)$$

This approach is correct because multiplication is commutative but not sound, meaning the check can pass even if the vectors are not equal. For example, the vectors $[1, 6]$ and $[2, 3]$ have both products of element 6. However, they are not permutations of each other. What if we sample uniformly randomly an element γ and check:

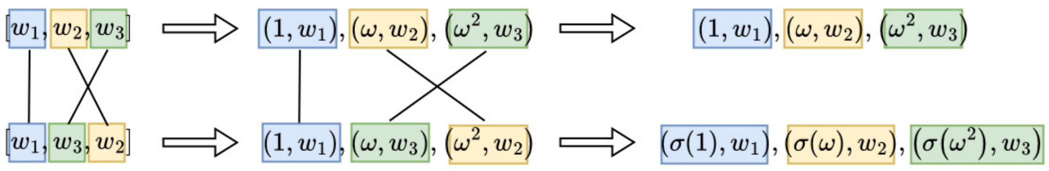
$$(p_1 + \gamma) \cdot (p_2 + \gamma) \ldots (p_n + \gamma) \stackrel{?}{=} (\sigma(q_1) + \gamma) \cdot (\sigma(q_2) + \gamma) \ldots (\sigma(q_n) + \gamma)$$
$$\prod_{i=1}^{n} p_i + \gamma \stackrel{?}{=} \prod_{i=1}^{n} \sigma(q_i) + \gamma$$

This check is also correct regarding the commutativity of the multiplication, but what about the soundness? The trick is to think of this expression in terms of polynomials where each side of the equation is a polynomial evaluated at a randomly selected point γ. Recalling the Section 2.2.1 about comparing polynomials, we already know that this check has only negligible failure probability. Therefore, we can say that the check is sound and has a high probability.

3.3.2 Intra-vector check

Now we would like to enforce a specific permutation specific permutation is enforced. How can we enforce a permutation defined as rotation on the equivalence classes? We will again have two vectors a and permutation $\sigma : H \to H$. The trick is to encode the elements as (index, value) where for the index, we will use the elements of evaluation domain H. Below, we show an example where the permutation swaps the last two elements.

$$\sigma : \sigma(1) = 1, \sigma(\omega) = \omega^2, \sigma(\omega^2) = \omega$$



$[w_1, w_2, w_3]$
$[w_1, w_3, w_2]$
$(1, w_1), (\omega, w_2), (\omega^2, w_3)$
$(1, w_1), (\omega, w_3), (\omega^2, w_2)$
$(1, w_1), (\omega, w_2), (\omega^2, w_3)$
$(\sigma(1), w_1), (\sigma(\omega), w_2), (\sigma(\omega^2), w_3)$

Figure 3.2 Enforcing specific permutation

So, we will be comparing tuples (ω^{i-1}, w_i) to ensure that the specified permutation was applied:

$$[(1, w_1), (\omega, w_2) \dots (\omega^{n-1}, w_n)] \overset{?}{=} [(\sigma(1), w_1), (\sigma(\omega), w_2) \dots (\sigma(\omega^{n-1}), w_n)]$$

The general permutation check 3.3.1 relied on the fact that we were comparing polynomials, so now we somehow need to write the tuples as polynomials. We will construct a linear combination evaluated at a random point, so (ω^{i-1}, w_1) will become $w_i + \beta\omega_{i-1}$ where β is randomly sampled.

Why can we use this encoding? To use this check safely in the protocol, we need to be sure that

$$(a, b) = (a', b') \iff a + \beta b = a' + \beta b'$$

Once again, the answer is polynomials. Notice that $a + \beta b$ is a linear polynomial with coefficients a, b evaluated at β, and so is $a' + \beta b'$ just with different coefficients. If the random linear combinations are equal, then with very high probability, the tuples will also be equal. Now, we put everything together and finalize the intra-vector check for a particular permutation.

$$(w_1 + \beta 1 + \gamma) \dots (w_n + \beta\omega^{n-1} + \gamma) \overset{?}{=} (w_1 + \beta\sigma(1) + \gamma) \dots (w_n + \beta\sigma(\omega^{n-1}) + \gamma)$$

$$\prod_{i=1}^{n} w_i + \beta\omega^{i-1} + \gamma \overset{?}{=} \prod_{i=1}^{n} w_i + \beta\sigma(\omega^{i-1}) + \gamma$$

$$1 \overset{?}{=} \frac{\prod_{i=1}^{n} w_i + \beta\omega^{i-1} + \gamma}{\prod_{i=1}^{n} w_i + \beta\sigma(\omega^{i-1}) + \gamma}$$

In summary, we have shown that the copy constraints of the arithmetic circuit could be represented as a permutation and found an effective way to ensure that these constraints are satisfied. Once again, the answer lay in converting everything to polynomials. The last obstacle in constructing the permutation polynomial is that we need to be able to check the permutation across multiple vectors.

3.3.3 Inter-vector check

In this case, we will want to check the permutation across vectors w_l, w_r, w_o, which are columns of the computation table described in Section 2.5. Each of w_l, w_r, w_o has size n and we will concatenate them into

$$w = \{w_{l_1}, \dots, w_{l_n}, w_{r_1}, \dots, w_{r_n}, w_{o_1}, \dots, w_{o_n}\}$$

which will have size $3n$. Now that we have changed the vector size, the domain H is no longer sufficient for indexing w because it has size n. We will solve this by using $H' = H \cup (k_1 H) \cup (k_2 H)$, where $k_1, k_2 \in \mathbb{F}_p$ are chosen such that $H, k_1 H, k_2 H$ are distinct meaning $\forall p, q, r : \omega^p \neq \omega^q k_1 \neq \omega^r k_2$. The permutation function σ^* will create mapping $[1, 2, \dots, 3n] \to H'$ which means $i \to \omega^i, n + i \to k_1 \omega^i, 2n + i \to k_2 \omega^i$. This is the permutation function that we mentioned in the protocol setup, and it also implements rotation on the equivalence classes. Using the same idea as before, we can extend the check to inter-vector cases.

$$1 \stackrel{?}{=} \prod_{i=1}^{n} \frac{f(i)}{g(i)}$$

$$f(i) = (w_i + \beta\omega^i + \gamma)(w_{n+i} + k_1\beta\omega^i + \gamma)(w_{2n+i} + k_2\beta\omega^i + \gamma) \tag{3.1}$$

$$g(i) = (w_i + \beta\sigma^*(i) + \gamma)(w_{n+i} + \beta\sigma^*(n+i) + \gamma)(w_{2n+i} + \beta\sigma^*(2n+i) + \gamma) \tag{3.2}$$

This expression might be undefined when $g(i)$ is 0; however, it can be proven that this only happens with negligible probability. When it happens, the protocol is instructed to abort and repeat with other randomly sampled β, γ.

3.3.4 The Permutation Polynomial

We know how to perform the permutation check, but we want to convince the verifier that it is correct. We cannot give him the values to compute the check because he would discover w, which violates zero-knowledge, and the computation would also be too heavy for the verifier. We will solve this problem by constructing the permutation polynomial, which is defined as:

$$z(x) = \begin{cases} 1 & x = \omega^0 \\ \prod_{j=1}^{i} \frac{f(j)}{g(j)} & x = \omega^i \text{ where } i \in \{1, 2, 3, \dots, n-1\} \end{cases} \tag{3.3}$$

The essential idea of the permutation check remains the same. We are taking the product of ratios, where the numerator represents the former set and the denominator represents the permuted set. If each of these ratios is 1, the copy constraints are satisfied. How do we construct this polynomial? We will proceed as usual and perform Lagrange interpolation with the evaluation domain H:

$$\left(\frac{f(1)}{g(1)}, \frac{f(1)f(2)}{g(1)g(2)}, \frac{f(1)f(2)f(3)}{g(1)g(2)g(3)}, \dots, \prod_{i=1}^{j} \frac{f(i)}{g(i)} \right)$$

That means the (almost final) permutation polynomial could be written as:

$$z'(x) = \sum_{i=1}^{n-1} L_i(x) \prod_{j=1}^{i} \frac{f(j)}{g(j)}$$

This polynomial satisfies just the condition for $x = \omega^i$ where $i \in \{1, \dots n-1\}$ and to enforce that permutation polynomial evaluates to 1 on ω^0 we add the Lagrange basis $L_0(x)$. To finish it up, just add blinding scalars, but this time, we will need a blinding polynomial of degree 2 because later, there will be two openings of $z(x)$.

$$z(x) = (b_7x^2 + b_8x + b_9)Z_H(x) + L_0(x) + z'(x)$$

Now, the prover needs to convince the verifier that it evaluates to 1 over H. If we do not include any addition checks, the prover could interpolate the following polynomial $[(1, 1), (\omega, 1), (\omega^2, 1) \dots (\omega^{n-1}, 1)]$.

```
Round 2
1 :  $\beta = \mathcal{H}(transcript, 0), \gamma = \mathcal{H}(transcript, 1)$
2 :  compute permutation polynomial $z(x)$
3 :  return $[z]_1$
```

3.4 Round 3

The prover needs to combine checks from the previous rounds as well as convince the prover that the permutation polynomial was computed as specified by the protocol. But that is not all. We also need to handle the problem with polynomials exceeding the degree bound n.

In the last round, we computed and committed to $z(x)$; however, we did not prove that it was computed correctly. Specifically we have promised that $z(\omega) = 1$, otherwise for $x = \omega^i$ it is cumulative product $\prod_{j=1}^{i} f(i)/g(i)$. This is the same as checking:

$$(z(x) - 1)L_0(x) = 0 \tag{3.4}$$
$$z(x)\tilde{f}(x) = \tilde{g}(x)z(x\omega) \tag{3.5}$$

$$\tilde{f}(x) = (a(x) + x\beta + \gamma)(b(x) + x\beta k_1 + \gamma)(c(x) + x\beta k_2 + \gamma)$$
$$\tilde{g}(x) = (a(x) + \beta S_{\sigma_1}(x) + \gamma)(b(x) + \beta S_{\sigma_2}(x) + \gamma)(c(x) + \beta S_{\sigma_3}(x) + \gamma)$$

This is not immediately obvious. The proof is in Section 3.4.1. The quotient polynomial denoted as $t(x)$ in the paper consists of the sum of 3 expressions $t = t_1 + t_2 + t_3$:

$$t_1(x) = (a(x)q_l(x) + b(x)q_r(x) + c(x)q_o(x) + a(x)b(x)q_m(x) + PI(x) + q_{c_i})\frac{1}{Z_H(x)} \tag{3.6}$$

$$t_2(x) = (f'(x)z(x))\frac{\alpha}{Z_H(X)} - (g'(x)z(\omega x))\frac{1}{Z_H(x)} \tag{3.7}$$

$$t_3(x) = (z(x)-1)L_1(x\frac{1}{Z_H(x)} \tag{3.8}$$

$$t(x) = t_1(x) + t_2(x)\alpha + t_3(x)\alpha^2$$

The quotient polynomial might be long but comprises elements that make sense.

3.4.1 Computing quotient polynomial

3rd term

This corresponds to checking the first part of the $z(x)$ definition.

Lemma 12 (First property of permutation polynomial)**.** $\forall x \in H : (z(x) - 1)L_1(x) = 0 \implies z(\omega) = 1$

Proof. For $x \neq \omega$, the Lagrange basis evaluates to 0, and there is no constraint for $z(x)$. However for $x = \omega$ we get $z(\omega) - 1 = 0$ meaning that $z(\omega)$ indeed must be equal to 1. □

2nd term

Theorem 4 (First property of permutation polynomial). $\forall i \in [n] : z(\omega^i)f'(\omega^i) = g'(\omega^i)z(\omega^{i+1}) \implies \forall i \in [n] : z(\omega^i) = \prod_{j=1}^{i-1} \frac{f'(\omega^j)}{g'(\omega^j)}$

Lemma 13. *We will show this by induction. For the base case $i = 1$ we get:*

$$z(\omega)f(\omega) = g(\omega)z(\omega^2)$$

$$z(\omega^2) = \frac{f(\omega)}{g(\omega)}$$

We know that $z(\omega) = 1$ is already checked for that; the rest simplifies easily. For the case $i = k + 1$

$$z(\omega^{k+1})f(\omega^{k+1}) = g(\omega^{k+1})z(\omega^{k+2})$$

$$\prod_{j=1}^{k} \frac{f(\omega^j)}{g(\omega^j)} f(\omega^{k+1}) = g(\omega^{k+1})z(\omega^{k+2})$$

$$\prod_{j=1}^{k} \frac{f(\omega^j)}{g(\omega^j)} \frac{f(\omega^{k+1})}{g(\omega^{k+1})} = z(\omega^{k+2})$$

$$z(\omega^{k+2}) = \prod_{j=1}^{k+1} \frac{f(\omega^j)}{g(\omega^j)}$$

1st term

These are the gate constraints introduced in the overview (2.5). Including them in the quotient polynomial makes sure they hold for each gate.

Each check passes if the designated polynomial is zero on the evaluation domain. We want to combine to batch these checks such that $t(x) = 0 \iff t_1(x) = 0 \wedge t_2(x) = 0 \wedge t_3(x) = 0$. To achieve this, we will construct the quotient polynomial using random challenge α:

$$t(x) = t_1(x) + t_2(x)\alpha + t_3(x)\alpha^2$$

.

This technique is standard for batching checks in cryptography. The intuition on why this approach is secure is that the challenge is determined by the transcript, which contains the commitments to the polynomials that construct $t(x)$. So, the prover cannot make assumptions about the challenge before the commitment.

3.4.2 Splitting quotient polynomial

Finally, we can construct the quotient polynomial $t(x)$. However, the problem is that the polynomial degree is too big. We want our polynomials to have the maximum degree of n to be able to commit to them. While creating `SRS`, we assumed so in the setup, and the whole KZG commitment scheme relies on it. We can split $t(x)$ into $< n$ degree polynomials $t'_{lo}(x), t'_{mid}(x)$ and $t'_{hi}(x)$ of degree at most $n + 5$ such that:

$$t(x) = t'_{lo}(x) + x^n t'_{mid}(x) + x^{2n} t'_{hi}(x).$$

Why does t_{hi} have degree bound $n + 5$? The degree of $t(x)$ is determined by $t_2(x)$ where we multiply wire polynomials $a(x), b(x), c(x)$ and permutation polynomial $z(x)$. Each of the wire polynomials has degree bound $n + 1$, and the permutation polynomial has $n + 2$, which makes it $4n + 5$, but $t_2(x)$ is also divided by $Z_H(x)$ of degree n. Therefore, the degree bound of the quotient polynomial $t(x)$ is $3n + 5$. This means $t(x)$ can be written as $t(x) = c_0 + c_1 x \ldots c_{3n+5} x^{3n+5}$. Then we can split as:

$$\begin{aligned}
t'_{lo}(x) &= c_0 + c_1 x + c_2 x^2 \ldots c_{n-1} x^{n-1} \\
t'_{mid}(x) &= \frac{c_n x^n + c_{n+1} x^{n+1} \ldots c_{2n-1} x^{2n-1}}{x^n} \\
t'_{hi}(x) &= \frac{c_{2n} x^{2n} + c_{2n+1} x^{2n+1} \ldots c_{3n-5} x^{3n-5}}{x^{2n}}
\end{aligned}$$

Blinding with $b_{10}, b_{11} \in \mathbb{F}_p$ is performed as follows: $t_{lo}(x) = t'_{lo}(x) + b_{10} x^n, t_{mid}(x) = t'_{mid}(x) - b_{10} + b_{11} x^n, t_{hi}(x) = t'_{hi}(x) - b_{11}$. Now we are ready to calculate commitments with a degree at most $n + 5$: $[t_{lo}(s)]_1, [t_{mid}(s)]_1, [t_{hi}(s)]_1$.

```
Round 3
1 :  $\alpha \leftarrow \mathbb{F}_p$
2 :  $t(x) = t_1(x) + t_2(x)\alpha + t_3(x)\alpha^2$
3 :  $t(x) -> t_{lo}, t_{mid}, t_{high}$
4 :  return $[t_{lo}]_1, [t_{mid}]_1, [t_{high}]_1$
```

3.5 Round 4

In this round, the prover computes evaluation openings, denoted with a horizontal line above. The openings are performed at a random point $\mathfrak{z}$. In an interactive case, $\mathfrak{z}$ is chosen by the verifier. In non-interactive variant it is given by a hash function applied to a transcript of the prover computation. All the prover has to do is calculate and output: $\overline{a} = a(\mathfrak{z}), \overline{b} = b(\mathfrak{z}), \overline{c} = c(\mathfrak{z}), \overline{z_\omega} = z(\omega\mathfrak{z}), \overline{S}_{\sigma_1} = S_{\sigma_1}(\mathfrak{z}), \overline{S}_{\sigma_2} = S_{\sigma_2}(\mathfrak{z})$. It is as simple as that. Now, let's look at ways to minimize the number of openings to reduce protocol communication costs.

3.5.1 Linearization trick

Imagine that the prover wants to show that $h_1(x)h_2(x) - h_3(c) = 0$ over a specified domain. Then he sure needs to commit to these polynomials and send their openings at a random $\mathfrak{z}$, resulting in 3 commitments and three openings. The verifier needs to check $\forall x \in H : h_1(x)h_2(x) - h_3(c) = 0$, which simplifies to checking just at a single random point $\mathfrak{z}$.

Standard approach

$$\begin{array}{lcl}
\text{Prover } \mathsf{P} & & \text{Verifer } \mathsf{V} \\
 & & \\
f_1(x), f_2(x), f_3(x) & & \\
\text{get challenge } \mathfrak{z} & & \\
\overline{f_1}, \overline{f_2}, \overline{f_3} & & \\
 & \xrightarrow{[f_1]_1, [f_2]_1, [f_3]_1} & \\
 & \xrightarrow{\overline{f_1}, \overline{f_2}, \overline{f_3}} & \\
 & & \text{verify openings} \\
 & & \overline{f_1 f_2} + \overline{f_3} \overset{?}{=} 0
\end{array}$$

There is a better approach, sketched as the *Linearization trick*, where the prover sends 3 commitments but just two 2 openings. This minimizes both the communication load and the proof size. The trick is to construct a linearization polynomial: $l(x) = \overline{f_1} f_2(x) - f_3(x)$. As before, the prover needs to send commitments $[f_1]_1, [f_2]_1, [f_3]_1$, but the openings are $\overline{f_1} = f_1(\mathfrak{z})$ and $\overline{l} = l(\mathfrak{z}) = \overline{f_1 f_2} - \overline{f_3}$.

Why is the prover not sending commitment to the linearization polynomial $l(x)$? Simply because the verifier can calculate $[l]_1$ on his own. Recall that commitments are defined as $[f]_1 = G_1^{f(\tau)}$ elements of a group defined by points on an elliptic curve over a finite field $\mathbb{F}_p$. In the group, only one operation is defined between the group elements: addition. This means that we can add commitments but not multiply them. However, it is possible to multiply a group element by a constant so we can calculate $[l]_1 = \overline{f_1}[f_2]_1 - [f_3]_1$. This means that to calculate commitment to the linearization polynomial, there can be only the addition of polynomials but not multiplication.

So, if the prover can reconstruct the commitment to the linearization polynomial $[l]_1$, he can also check that the opening $\overline{l}$ is a correct evaluation at $\mathfrak{z}$. Since the linearization polynomial is constructed as $l(x) = \overline{f_1}f_2(x) - f_3(x)$ it means checking $\overline{l} \overset{?}{=}$ is equivalent to checking $\overline{f_1 f_2} - \overline{f_3} \overset{?}{=} 0$.

Linearization trick

Prover P | Verifer V

$f_1(x), f_2(x), f_3(x)$

get challenge $\mathfrak{z}$

$l(x) = \overline{f_1}f_2(x) + f_3(x)$

$$\xrightarrow{[f_1]_1, [f_2]_1, [f_3]_1}$$

$$\xrightarrow{\overline{l}, \overline{f_1}}$$

$[l]_1 = \overline{f_1}[f_2]_1 + [f_3]_1$

verify openings

$\overline{l} \overset{?}{=} 0$

Why is it not possible to use pairings? We have defined curve pairing as a mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$, it is possible to perform "multiplication in the exponent" as described in Section 2.1.2. In the context of KZG, the verifier performs the check in the target group as

$$e(CG_1^{-v}, G_2) \overset{?}{=} e(W, G_2^{\tau}G_2^{-z}).$$

However this operation can be performed only once because we end up with points in the target group where we cannot use the mapping e. So, a very high-level explanation for why we do not use the curve pairings in the linearization trick is that the pairing can be used only once in the protocol, and we reserve this for the verifier to do the KZG verification.

Round 4

```
1 :  $\mathfrak{z} = \mathcal{H}(transcript)$
2 :  $\overline{a} = a(\mathfrak{z}), \overline{b} = b(\mathfrak{z}), \overline{c} = c(\mathfrak{z}), \overline{z_\omega} = z_\omega(x), \overline{S}_{\sigma_1} = S_{\sigma_1}(x), \overline{S}_{\sigma_2} = S_{\sigma_2}(x)$
3 :  **return** $\overline{z_\omega}, \overline{S}_{\sigma_1}, \overline{S}_{\sigma_2}$
```

3.6 Round 5

3.6.1 Linearisation polynomial

Recall from Section 3.4 that the quotient polynomial $t(x)$ was split into 3 parts to reduce the degree. So, it must hold:

$$t_{lo} + x^n t_{mid}(x) + x^{2n} t_{hi} = \frac{t_1(x) + \alpha t_2(x) + \alpha^2 t_3(x)}{Z_H(x)} = t(x).$$

This means that over the whole evaluation domain H, it holds that:

$$0 = t_1(x) + \alpha t_2(x) + \alpha^2 t_3(x) - Z_H(x)(t_{lo} + x^n t_{mid}(x) + x^{2n} t_{hi}). \tag{3.9}$$

This is the base of how we will construct the linearisation polynomial $r(x)$. Some of the terms in $t_1(x), t_2(x), t_3(x)$ are substituted with the openings $\overline{a}, \overline{b}, \overline{c}, \overline{z_\omega}, \overline{S}_{\sigma_1}, \overline{S}_{\sigma_2}$ calculated in Section 3.5. The linearisation polynomial can be written as:

$$\begin{aligned} r(x) =& \overline{a}\overline{b}q_m(x) + \overline{a}q_l(x) + \overline{b}q_r(x) + \overline{c}q_o(x) + PI(\mathfrak{z}) + q_c(x) \\ &+ \alpha[(\overline{a} + \beta\mathfrak{z} + \gamma)(\overline{b} + \beta k_1\mathfrak{z} + \gamma)(\overline{c} + \beta k_2\mathfrak{z} + \gamma)z(x) \\ &- (\overline{a} + \beta\overline{s}_{\sigma_1} + \gamma)(\overline{b} + \beta\overline{s}_{\sigma_2} + \gamma)(\overline{c} + \beta S_{\sigma_3}(x) + \gamma)\overline{z}_\omega] \\ &+ \alpha^2[(z(x) - 1)L_0(\mathfrak{z})] \\ &- Z_H(\mathfrak{z})(t_{lo}(x) + \mathfrak{z}^n t_{mid}(x) + \mathfrak{z}^{2n} t_{hi}(x)) \end{aligned}$$

The whole polynomial corresponds to the quotient polynomial minus $Z_H(\mathfrak{z})(t_{lo}(x) + \mathfrak{z}^n t_{mid}(x) + \mathfrak{z}^{2n} t_{hi}(x))$.

- The first line represents the arithmetic gate check corresponding to the $t_1(x)$ 3.6 in the quotient polynomial $t(x)$.
- Lines 2 and 3 represent the second check of the permutation polynomial 3.5, which is described by $t_2(x)$ in $t(x)$.
- Line 4 is the first check of the permutation polynomial 3.4, which corresponds to $t_3(x)$ in $t(x)$.
- The last line is from the expression above 3.9.

Since we "derived" the linearization polynomial from the expression 3.9, it should hold that $r(x)$ is zero over the whole domain H. Notice which polynomials are evaluated (denoted with the horizontal line). As described in the previous round, the linearization polynomial can contain polynomial additions and multiplication of polynomials by a constant, but not multiplication of two polynomials. And that is exactly what we are trying to achieve here. The openings are picked so that there is a single multiplication of polynomials. The terms $Z_H(\mathfrak{z}), L_0(\mathfrak{z})$ are in fact constants. This allows us to use the linearization trick, and as a result, the prover does not send the commitment $[r]_1$ because the verifier can calculate it independently.

The big picture is that the prover is trying to prove: $0 = t_1(x) + \alpha t_2(x) + \alpha^2 t_3(x) - Z_H(x)(t_{lo} + x^n t_{mid}(x) + x^{2n} t_{hi})$. If he did it naively, he would need to send an opening to every single polynomial. However, thanks to the linearization trick, we are able to minimize the number of openings, thus also reducing the proof size.

3.6.2 Opening proof polynomial

The prover needs to send a proof of the opening for $\overline{a}, \overline{b}, \overline{c}, \overline{S}_{\sigma_1}, \overline{S}_{\sigma_2}$ at $\mathfrak{z}$ and $\overline{z_\omega}$ at $\mathfrak{z}\omega$. This is captured by the polynomials $W_\mathfrak{z}(x), W_{\mathfrak{z}\omega}(x)$.

First opening proof polynomial

The opening proof polynomial is constructed by batching opening proof polynomials from the KZG polynomial commitment scheme in Section 2.3.2. We have proof polynomials:

$$\begin{aligned}
&\frac{r(x)}{x-\mathfrak{z}} \xrightarrow{\text{proves}} r(\mathfrak{z}) = 0 &\qquad& \frac{a(x)-\overline{a}}{x-\mathfrak{z}} \xrightarrow{\text{proves}} a(\mathfrak{z}) = \overline{a} \\
&\frac{b(x)-\overline{b}}{x-\mathfrak{z}} \xrightarrow{\text{proves}} b(\mathfrak{z}) = \overline{b} && \frac{c(x)-\overline{c}}{x-\mathfrak{z}} \xrightarrow{\text{proves}} c(\mathfrak{z}) = \overline{c} \\
&\frac{S_{\sigma_1}(x)-\overline{S}_{\sigma_1}}{x-\mathfrak{z}} \xrightarrow{\text{proves}} S_{\sigma_1}(\mathfrak{z}) = \overline{S}_{\sigma_1} && \frac{S_{\sigma_2}(x)-\overline{S}_{\sigma_2}}{x-\mathfrak{z}} \xrightarrow{\text{proves}} S_{\sigma_2}(\mathfrak{z}) = \overline{S}_{\sigma_2}
\end{aligned}$$

These are batched using random challenge $v \in \mathbb{F}_p$ given by $\mathcal{H}(transcript)$.

$$W_{\mathfrak{z}}(x) = \frac{1}{x-\mathfrak{z}} \begin{pmatrix} r(x) \\ + v(a(x)-\overline{a}) \\ + v^2(b(x)-\overline{b}) \\ + v^3(c(x)-\overline{c}) \\ + v^4(S_{\sigma_1}(x)-\overline{S}_{\sigma_1}) \\ + v^5(S_{\sigma_2}(x)-\overline{S}_{\sigma_2}) \end{pmatrix}$$

Second opening proof polynomial

Prover calculates $W_{\mathfrak{z}\omega}(x)$. Recall that in the quotient polynomial $t(x)$, both $z(x), z(z\omega)$ appear. Thanks to the linearization trick in (3.5), it is sufficient to compute $\overline{z_\omega}$. However, as $z(x)$ is opened at $\mathfrak{z}\omega$ instead of $\mathfrak{z}$, we need a separated opening polynomial $W_{\mathfrak{z}\omega}(x)$, which is the final polynomial that the prover needs to calculate.

$$W_{\mathfrak{z}\omega}(x) = \frac{z(x)-\overline{z}_\omega}{x-\mathfrak{z}\omega}$$

This polynomial checks that $z(\mathfrak{z}\omega) = \overline{z}_\omega$ in the same way as described for the opening polynomial $W_{\mathfrak{z}}(x)$. Now the prover can finally send the whole proof:

$$\pi = ([a]_1, [b]_1, [c]_1, [z]_1, [t_{lo}]_1, [t_{mid}]_1, [t_{hi}]_1, [W_{\mathfrak{z}}]_1, [W_{\omega\mathfrak{z}}]_1, \overline{a}, \overline{b}, \overline{c}, \overline{z_\omega}, \overline{S}_{\sigma_1}, \overline{S}_{\sigma_2})$$

```
Round 5
1 : $v = \mathcal{H}(transcript)$
2 : Compute linearisation polynomial $r(x)$
3 : Compute opening proof polynomial $W_{\mathfrak{z}}(x)$
4 : Compute opening proof polynomial $W_{\mathfrak{z}\omega}(x)$
5 : **return** $\pi$
```

3.7 Verification

The verifier has its own preprocessed input and performs multiple sanity checks on the proof. The interesting part is the batched verification of KZG. Essentially, the prover needs to validate $W_{\mathfrak{z}}(x), W_{\mathfrak{z}\omega}(x)$. The polynomials can be written in a simplified way as $W_{\mathfrak{z}}(x) = F(x)/(x - \mathfrak{z})$, $W_{\mathfrak{z}\omega}(x) = E(x)/(x - \mathfrak{z}\omega)$ and, after some rearranging, we get:

$$xW_{\mathfrak{z}}(x) = \mathfrak{z}W_{\mathfrak{z}}(x) + F(x) \tag{3.10}$$

$$xW_{\mathfrak{z}\omega}(x) = \mathfrak{z}\omega W_{\mathfrak{z}\omega}(x) + E(x) \tag{3.11}$$

Now, we can batch the expressions as in Equation (3.11).

$$x(W_{\mathfrak{z}}(x) + uW_{\mathfrak{z}\omega}(x)) = \mathfrak{z}W_{\mathfrak{z}}(x) + u\mathfrak{z}\omega W_{\mathfrak{z}\omega}(x) + F(x) + uE(x) \tag{3.12}$$

In the last steps of the verification algorithm, the verifier calculates the commitments to $[E]_1, [F]_1$. The final step checks the above identity using batched KZG-style verification with commitments and pairings.

4 Security and Efficiency

The role of the prover is to convince the verifier about the valid execution of a specified arithmetic circuit by showing that conditions form Section 2.6 are satisfied. This is done by constructing specific polynomials and proving their evaluation at uniformly randomly selected points using the KZG polynomial commitment scheme. It might not be directly clear from the diagram in Figure 4.1, but the whole protocol is basically a batched KZG for polynomials that prove valid execution of the circuit.

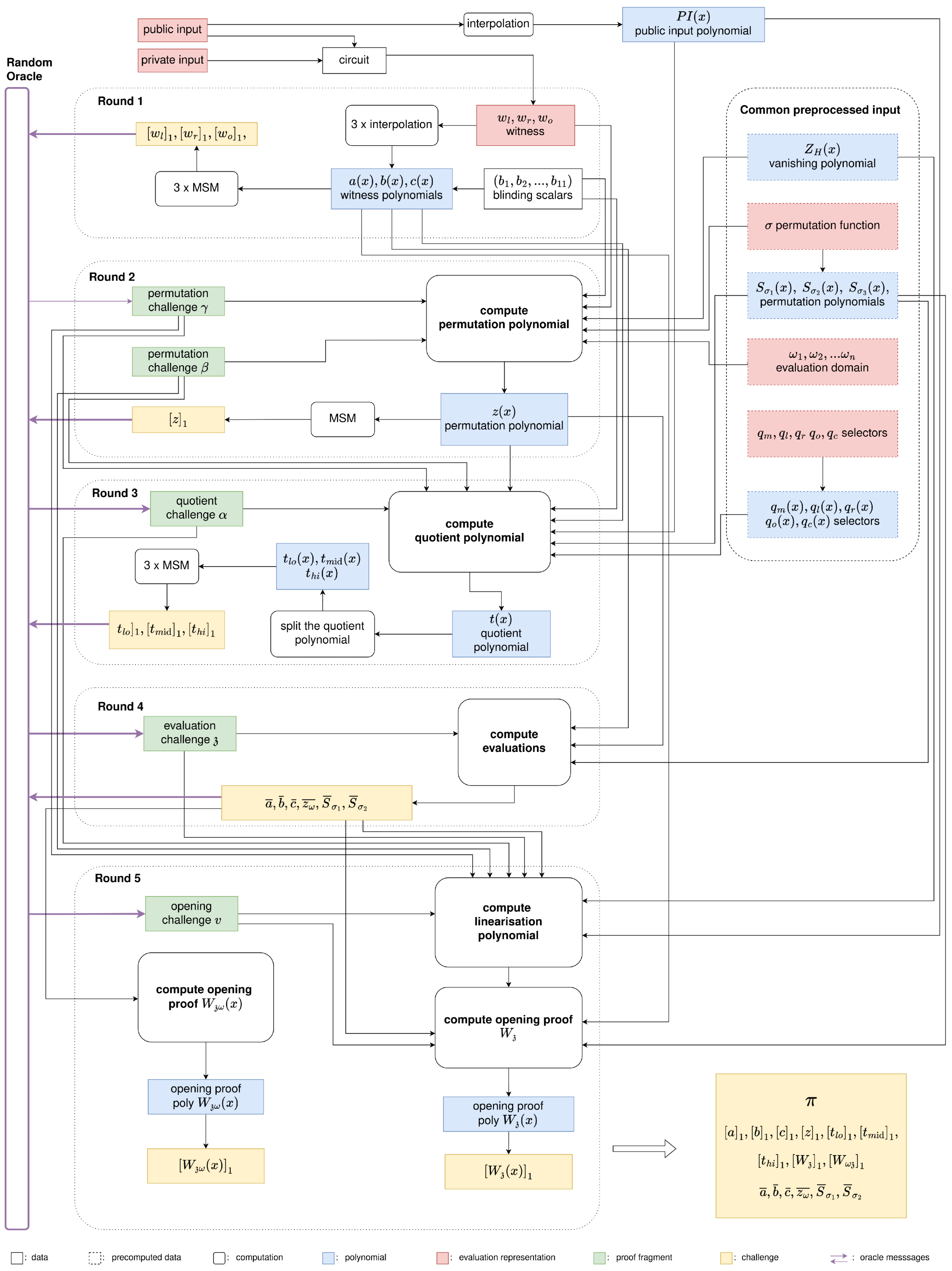
Degree bound on the polynomials constructed by the prover:

- Wire polynomials $deg(a(x)) = deg(b(x)) = deg(c(x)) = n + 1$ determined the multiplication of blinding polynomial of degree 1 and the vanishing polynomial of degree n.
- Permutation polynomial $deg(z(x)) = n + 2$ determined by multiplication of blinding polynomial of degree 2 and vanishing polynomial of degree n
- Quotient polynomial $deg(t(x))3n + 5$ described in the round 3 Section 3.4.2
- Split quotient polynomials $deg(t_{lo}(x)) = deg(t_{mid}(x)) = n, deg(t_{hi}(x)) = n + 5$ quotient polynomial is split into 3 smaller polynomials
- Linearisation polynomial $deg(r(x)) = n + 5$ as described in round 4 Section 3.5 the linearisation polynomial cannot contain polynomial multiplication. Therefore, the degree is determined by the polynomial of the highest degree, which is $t_{hi}(x)$
- Opening proof polynomial $deg(W_{\mathfrak{z}}(x)) = n + 4$ degree is determined by $t(x)$ and consequently divided by $(x - \mathfrak{z})$
- Opening proof permutation polynomial $deg(W_{\mathfrak{z}\omega}(x)) = n + 1$ degree is determined by $z(x)$ and divided by $(x - \mathfrak{z}\omega)$

4.1 Advantages and limitations

The $\mathcal{P}$lon$\mathcal{K}$ protocol can run for a general circuit of some size that is bounded by the KZG setup. The setup, in the form of common preprocessed input, is updatable, and the trusted KZG setup can be reused for any circuit of a given size bound. Moreover, the size of the proof is constant.

As stated in the article on pipeMSM [10], the calculation of the commitments seems to be a major bottleneck of $\mathcal{P}$lon$\mathcal{K}$ and other SNARK protocols. Another major bottleneck is FFT. There is a lot of effort to make the prover algorithm more effective. Possible ways for the protocol optimization are mentioned in the Section 5.1.



public input
private input
interpolation
circuit
$PI(x)$ public input polynomial
Random Oracle
Round 1
$[w_l]_1, [w_r]_1, [w_o]_1,$
3 x interpolation
w_l, w_r, w_o witness
3 x MSM
$a(x), b(x), c(x)$ witness polynomials
$(b_1, b_2, ..., b_{11})$ blinding scalars
Common preprocessed input
$Z_H(x)$ vanishing polynomial
σ permutation function
$S_{\sigma_1}(x),\ S_{\sigma_2}(x),\ S_{\sigma_3}(x),$ permutation polynomials
$\omega_1, \omega_2, ...\omega_n$ evaluation domain
$q_m, q_l, q_r\ q_o, q_c$ selectors
$q_m(x), q_l(x), q_r(x)$ $q_o(x), q_c(x)$ selectors
Round 2
permutation challenge γ
compute permutation polynomial
permutation challenge β
$[z]_1$
MSM
$z(x)$ permutation polynomial
Round 3
quotient challenge α
compute quotient polynomial
3 x MSM
$t_{lo}(x), t_{mid}(x)$ $t_{hi}(x)$
$t_{lo}]_1, [t_{mid}]_1, [t_{hi}]_1$
split the quotient polynomial
$t(x)$ quotient polynomial
Round 4
evaluation challenge $\mathfrak{z}$
compute evaluations
$\bar{a}, \bar{b}, \bar{c}, \overline{z_\omega}, \bar{S}_{\sigma_1}, \bar{S}_{\sigma_2}$
Round 5
opening challenge v
compute linearisation polynomial
compute opening proof $W_{\mathfrak{z}\omega}(x)$
compute opening proof $W_{\mathfrak{z}}$
opening proof poly $W_{\mathfrak{z}\omega}(x)$
opening proof poly $W_{\mathfrak{z}}(x)$
$[W_{\mathfrak{z}\omega}(x)]_1$
$[W_{\mathfrak{z}}(x)]_1$
π
$[a]_1, [b]_1, [c]_1, [z]_1, [t_{lo}]_1, [t_{mid}]_1,$
$[t_{hi}]_1, [W_{\mathfrak{z}}]_1, [W_{\omega\mathfrak{z}}]_1$
$\bar{a}, \bar{b}, \bar{c}, \overline{z_\omega}, \bar{S}_{\sigma_1}, \bar{S}_{\sigma_2}$
: data
precomputed data
: computation
: polynomial
: evaluation representation
: proof fragment
: challenge
: oracle messsages

Figure 4.1 Diagram of prover algorithm

4.2 Properties of the protocol

Theorem 5. The $\mathcal{P}$lon$\mathcal{K}$ protocol is a succinct, non-interactive, complete, knowledge sound, and zero-knowledge.

Succinctness: The proof π consists of 9 commitments and 6 openings, so for an arbitrarily large circuit, the proof size remains constant. The verification algorithm must perform sanity checks, evaluate 3 public polynomials, calculate 3 additional commitments, and perform a single batched KZG verification procedure. The number of operations does not change with respect to the size of the circuit, so we can conclude that the protocol is succinct.

Non-Interactivity: Non-interactivity is achieved by the Fiat-Shamir heuristic, where the prover can effectively generate challenges by accessing a random oracle $\mathcal{H}$. We will not show the correctness of the application of this heuristic.

Completeness: Correctness of $\mathcal{P}$lon$\mathcal{K}$ is dependant on many building blocks. Correctness of the checks for the arithmetic circuit was established by Lemma 10, Lemma 8, Lemma 9, Lemma 11, and we also showed the correctness of the KZG polynomial commitment scheme in Section 2.3.1.

Knowledge Soundness: We showed the soundness of each of the checks for the arithmetic circuit Lemma 10, Lemma 8, Lemma 9, Lemma 11. We did not properly show the soundness of KZG but referred to Proofs, Arguments, and Zero-Knowledge [4].

Zero-Knowledge: Since the proof contains only commitments and polynomial opening, it is sufficient to show that masking polynomial makes the KZG polynomial commitment scheme zero-knowledge, which was proven in Theorem 2.

The formal proof of the Theorem 5 follows from the above discussion.

5 Optimizations

Upon publishing, $\mathcal{P}\mathfrak{lon}\mathcal{K}$ became a popular SNARK, and variants of the protocol have been implemented in many cryptocurrency projects. This led to efforts to make the protocol more efficient. The optimizations can be done on three fronts:

- Polynomial commitment scheme
- Interactive protocol
- Recursive proof composition

5.1 Possibilities for optimization

5.1.1 Polynomial commitment scheme:

The $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol was initially described with the KZG [2] polynomial commitment scheme, but other polynomial commitment schemes like FRI [13] could be used as well. As mentioned, commitments realized by MSM place a large computational load on the prover. One of the authors of $\mathcal{P}\mathfrak{lon}\mathcal{K}$ has already written a follow-up article on the construction of efficient polynomial commitment schemes for multiple points and polynomials [18] that can be used in $\mathcal{P}\mathfrak{lon}\mathcal{K}$.

5.1.2 Interactive protocol:

This covers the rest of the protocol that is not about the polynomial commitment scheme. The possibilities for optimization range from more effective arithmetization to constructing faster polynomial checks. The major work in this area was done on the use of custom gates and the construction of lookup tables. By using more complex custom gates, it is possible to reduce the degree of the circuit, which determines the complexity of the prover. Custom gates are described in detail in *TurboPlonK* [19]. The lookup tables from *plookup* [20] enable to precompute values for inputs $\mathbb{x}$ and then prove that the witness $\mathbb{w}$ exists in that table. These two approaches are independent and can be combined, as shown in *HyperPlonK* [3].

5.1.3 Recursive proof construction:

SNARKs place most of the computational load on the prover. Since the verifier is considered computationally weak, the verification algorithm is designed to be lightweight and effective. Nevertheless, optimizing the verifier for multiple proofs is possible. The technique of recursive proof composition can aggregate several proofs into a single one and provide proof that each of the sub-proofs is valid. This approach is formalized in the *Halo* protocol [21], where the authors introduced an accumulation layer of the proof system.

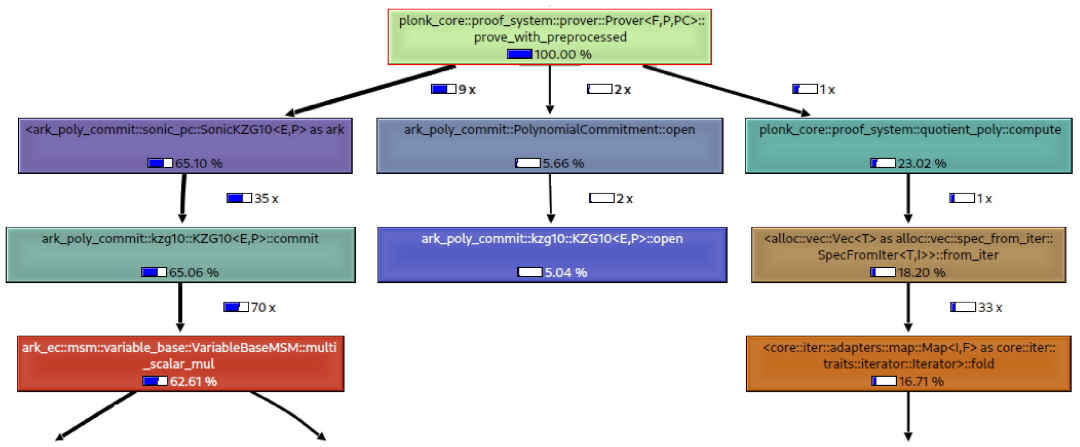

plonk_core::proof_system::prover::Prover<F,P,PC>::
prove_with_preprocessed
100.00 %
9 x
2 x
1 x
<ark_poly_commit::sonic_pc::SonicKZG10<E,P> as ark
65.10 %
ark_poly_commit::PolynomialCommitment::open
5.66 %
plonk_core::proof_system::quotient_poly::compute
23.02 %
35 x
2 x
1 x
ark_poly_commit::kzg10::KZG10<E,P>::commit
65.06 %
ark_poly_commit::kzg10::KZG10<E,P>::open
5.04 %
<alloc::vec::Vec<T> as alloc::vec::spec_from_iter::
SpecFromIter<T,I>>::from_iter
18.20 %
70 x
33 x
ark_ec::msm::variable_base::VariableBaseMSM::multi
_scalar_mul
62.61 %
<core::iter::adapters::map::Map<I,F> as core::iter::
traits::iterator::Iterator>::fold
16.71 %

Figure 5.1 Computational Load of ZK-Garage implementation

5.2 ZK-Garage PlonK

In this thesis, I use ZK-Garage [22] as a reference implementation of $\mathcal{PlonK}$. ZK-Garage is an open-source Rust implementation that uses arkworks library [23] for cryptographic primitives, arithmetic operations, and other utilities. The implementation adds the mentioned lookup tables and extends the arithmetic gates with AND, XOR, and range gates. Why did we choose Rust in the first place? The language's ownership model and strict compile-time checks help prevent common programming errors and security vulnerabilities. Additionally, Rust's performance is comparable to C++ [24], making it well-suited for computationally intensive cryptographic operations. These two factors make Rust a good choice for implementing cryptographic protocols.

Before trying out possible improvements, I analyzed demanding parts of the code. I used the popular profiling tool *Callgrind* for that task. It monitors how often each function is called and the number of CPU instructions executed within each function. This allows for identifying the most computationally demanding function and prioritizing the efforts to ensure that optimizations significantly impact overall performance.

The prover algorithm is contained in the function `prove_with_preprocessed`, which takes common preprocessed input as an argument. Figure 5.1 shows the most demanding computation of the prover in a call graph. The most computationally heavy parts of the program are the computation of commitments and the construction of the quotient polynomial $t(x)$. From the computation of the prover algorithm, 65% of the CPU instructions were spent on the commitments to the polynomials and 23% on the construction of the quotient polynomial. The function that calculates the quotient polynomial is called only once, and there are two calls to polynomial openings at $\mathfrak{z}, \mathfrak{z}\omega$. As specified in the protocol, nine polynomial commitments correspond to the calls of the `commit` function. The profiler was executed on *BenchCircuit* of size 2^6, and the relative computation load varied depending on the circuit size. Nevertheless, most of the time is spent calculating commitments and the quotient polynomial $t(x)$, so optimizing it will be the goal for the rest of this chapter. Minimizing the number of commitments would mean the checks could be encoded more efficiently with fewer polynomial. This task is complex, but there are other ways to improve the performance. For example, it is possible to reduce the size of MSM in some of the commitments.

5.3 Wire polynomials degree reduction

In this work, I aimed to give a detailed explanation of the prover algorithm in the interactive protocol and decided to think of possible improvements in this area. I led a discussion on potential improvements with Tomáš Krňák [25], who suggested looking at the construction of wire polynomials $a(x), b(x), c(x)$, which are interpolated from the witness $\mathtt{w}$.

ZK-Garage interpolates $a(x), b(x), c(x)$ using inverse Fast Fourier transform (iFFT) implemented in *arkworks*. The library calculates iFFT using a variant of the Cooley–Tukey algorithm [26], which requires the number of evaluations to be a power of two. The wire polynomial computed in round 1 also determines the degree of the polynomials computed in the following rounds. Reducing the degree of $a(x), b(x), c(x)$ naturally speeds up the computation of $[a]_1, [b]_1, [c]_1$, but the construction and commitment of the quotient polynomial should also be faster. This makes it a meaningful candidate for optimization.

I have tried to reduce the degree of $a(x), b(x), c(x)$ in two ways suggested by Tomáš Krňák. It is possible to perform the reduction by polynomial division in coefficient form and also by pairwise division in evaluation form. The two approaches are sketched in Figure 5.2.

5.3.1 Problem statement

The problem with the current implementation is that wire polynomials may be at most twice as big as the size of the circuit because of the padding to the next power of two. We denote the wire vector as $w = \{w_0, w_1 \dots w_{m-1}\}$. The closest power of two to m is n. To interpolate w using iFFT, it needs to be padded to size n, which means that the size of `SRS` needs to be at least n. In the ZK-Garage, w is padded with $n - m$ zeros. We denote the interpolated polynomial as:

$$W(x) = w(x)pad_0(x),$$

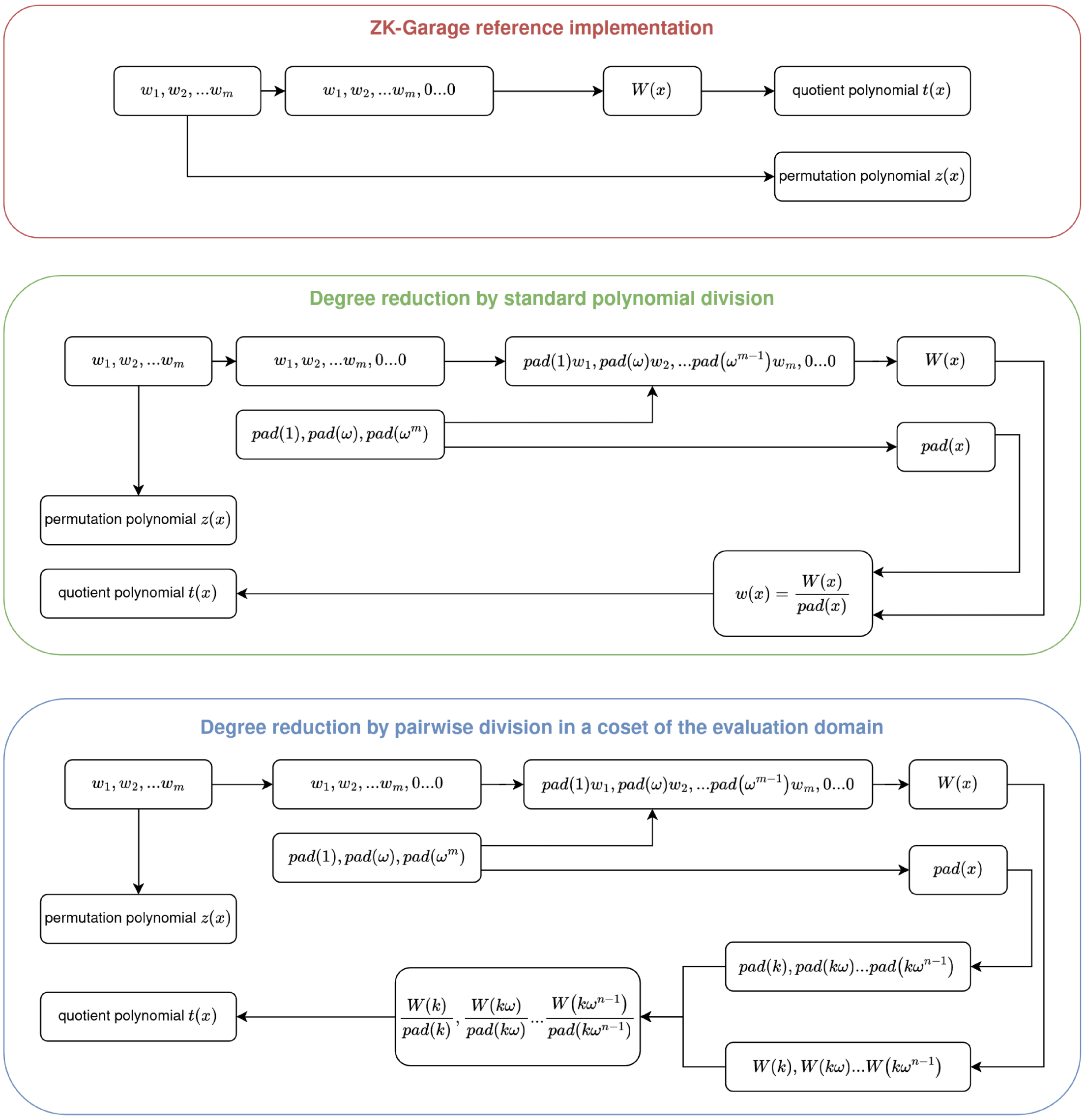
where $w(x)$ is defined by w on the evaluation domain $H = \{1, \omega, \dots \omega^{m-1}\}$ and $pad_0(x)$ is polynomial which has roots on $\{\omega^m, \omega^{m+1}, \omega^{n-1}\}$. We can write the padding polynomial as:

$$pad_0(x) = (x - \omega^m)(x - \omega^m + 1) \dots (x - \omega^{n-1}). \tag{5.1}$$

The reference implementation wire vector is padded and interpolated. There is no further degree reduction which means that $a(x), b(x), c(x)$ have degree $n - 1$. It is also worth noting that the protocol does not perform any checks on the padding domain $[\omega^m, \omega^{n-1}]$, which means we can pad the vectors with arbitrary values. As shown in the Chapter 2, the prover needs to show the following checks hold:

1. Public input check: the public input is encoded in the first l indices of the witness $\mathbb{w}$, so values on $[\omega^m, \omega^{n-1}]$ do not concern it.
2. Output check: the circuit output check verifies evaluation of the last element in the witness on ω^{m-1}, therefore $[\omega^m, \omega^{n-1}]$ are not relevant for this case.
3. Gate check: the values of the selectors $q_m(x), q_l(x), q_r(x), q_o(x), q_{(}x)$ on $[\omega^m, \omega^{n-1}]$ are zero because they do not encode any gate. Since $a(x), b(c), c(x)$ in the gate equation Equation (2.6) are always in combination with some selector the whole expression will evaluate to 0 as it should.
4. Wiring check: the permutation function the the range $[\omega^m, \omega^{n-1}]$ encodes identity. This means that the wiring check does not enforce anything on $[\omega^m, \omega^{n-1}]$.

Next, we present two optimizations that aim to use iFFT to get $W(x)$ and reduce its degree to get $w(x)$. In the following sections, each approach will be described in detail.



ZK-Garage reference implementation
$w_1, w_2, ...w_m$
$w_1, w_2, ...w_m, 0...0$
$W(x)$
quotient polynomial $t(x)$
permutation polynomial $z(x)$
Degree reduction by standard polynomial division
$w_1, w_2, ...w_m$
$w_1, w_2, ...w_m, 0...0$
$pad(1)w_1, pad(\omega)w_2, ...pad(\omega^{m-1})w_m, 0...0$
$W(x)$
$pad(1), pad(\omega), pad(\omega^m)$
$pad(x)$
permutation polynomial $z(x)$
quotient polynomial $t(x)$
$w(x) = \frac{W(x)}{pad(x)}$
Degree reduction by pairwise division in a coset of the evaluation domain
$w_1, w_2, ...w_m$
$w_1, w_2, ...w_m, 0...0$
$pad(1)w_1, pad(\omega)w_2, ...pad(\omega^{m-1})w_m, 0...0$
$W(x)$
$pad(1), pad(\omega), pad(\omega^m)$
$pad(x)$
permutation polynomial $z(x)$
$pad(k), pad(k\omega)...pad(k\omega^{n-1})$
quotient polynomial $t(x)$
$\frac{W(k)}{pad(k)}, \frac{W(k\omega)}{pad(k\omega)} ... \frac{W(k\omega^{n-1})}{pad(k\omega^{n-1})}$
$W(k), W(k\omega)...W(k\omega^{n-1})$

Figure 5.2 Degree reduction variants

5.3.2 ZK-Garage implementation

```
Reference implementation in ZK-Garage
1 : construct padding vector $[0]^{n-m}$
2 : concatenate $[w, [0]^{n-m}]$
3 : interpolate $W(x) = \mathsf{iFFT}(w, [0]^{n-m})$
4 : return $W(x)$
```

The prover algorithm receives wire vectors w_l, w_r, w_o. These are padded with zeros and interpolated by iFFT over the domain H to get polynomial $a(x), b(x), c(x)$. Padded wire vectors are further used to calculate the permutation polynomial in Section 3.3 because, according to the protocol, the permutation polynomial is computed in the evaluation form.

Time analysis: The complexity of this approach is dominated by iFFT, which has complexity $\mathcal{O}(n \log n)$.

5.3.3 Polynomial division

```
Reduction by polynomial division
1 : construct padding vector $[0]^{n-m}$
2 : evaluate $[pad_0(1), pad_0(\omega), \dots, pad_0(\omega^{m-1})]$
3 : concatenate $[w, [0]^{n-m}]$
4 : pair-wise multiplication $\tilde{w} = [w_0 pad_0(1), \dots w_{m-1} pad_0(\omega^{m-1}), 0 \dots 0]$
5 : interpolate $W(x) = \mathsf{iFFT}(\tilde{w})$
6 : reduce degree by polynomial division $w(x) = W(x)/pad_0(x)$
7 : return $w(x)$
```

The most straightforward approach when reducing the degree is to perform polynomial division by the polynomial defined by the padding values on $\{\omega^m, \dots, \omega^{n-m}\}$. Note that $[w_0, w_1 \dots w_m]$ first needs to be pair-wise multiplied by $[pad_0 \dots pad_m]$ so that it holds $W(x) = w(x)pad(x)$. In this approach, I have used the standard algorithm for long polynomial division. As a result, it is possible to retrieve $w(x)$ of degree m and use it in the following approach. The padded vector $[w_1, \dots w_{m-1}, 0 \dots 0]$ is again used in the round 2 to calculate $z(x)$.

The zero padding polynomial could be written as $pad_0(x) = \prod_{i=m}^{n-1}(x - \omega^i)$. Computing values $[pad_0(1), pad_0(\omega) \dots pad_0(\omega^{m-1})]$ requires significant number of field operations. There are $n-m$ field multiplications needed to calculate value $pad(\omega^i)$. To compute values on $[1 \dots \omega^{m-1}]$, it is needed to perform $n - m \times m = nm - m^2$ field multiplication, which introduces non-negligible overhead. Luckily there is a better approach and evaluation of $pad_0(x)$ on $[1, \dots \omega^m]$ can be calculated using recursive formula:

$$pad_0(x) = \begin{cases} \prod_{j=m}^{n-1}(1 - \omega^j) & x = 1, \\ \omega^{n-m-1} \frac{\omega^{i-1} - \omega^{m-1}}{\omega^{i-1} - \omega^{n-1}} pad_0(\omega^{i-1}) & x = \omega^i : i \in [m] \setminus \{1\} \end{cases} \tag{5.2}$$

To confirm the correctness of this formula:

$$\frac{pad_0(\omega^i)}{pad_0(\omega^{i-1})} = \prod_{j=m}^{n-1} \frac{\omega^i - \omega^j}{\omega^{i-1} - \omega^j} = \omega^{n-m} \prod_{j=m}^{n-1} \frac{\omega^{i-1} - \omega^{j-1}}{\omega^{i-1} - \omega^j}$$

$$= \omega^{n-m} \frac{\omega(\omega^{i-1} - \omega^{m-1})}{\omega^{i-1} - \omega^{n-1}} = \omega^{n-m} \frac{\omega(\omega^{i-1} - \omega^{m-1})}{\omega^{i-1} - \omega^{n-1}}$$

$$pad_0(\omega^i) = \omega^{n-m} \frac{\omega(\omega^{i-1} - \omega^{m-1})}{\omega^{i-1} - \omega^{n-1}} pad_0(\omega^{i-1})$$

Time analysis: Calculating evaluations of *pad* can be done in m iteration thanks to the recursive formula Equation (5.2) and pairwise multiplication in step 4 of the algorithm could be trivially parallelized. Naive polynomial division, which has complexity $deg(w(x)) \times deg(pad_0(x)) = n \times n - m$, which makes the total complexity $n^2 - nm + n \log n$. Since $m < n/2$ we can conclude $\mathcal{O}\left(n^2 - \frac{3}{2}n\right)$.

From the analysis, we can already conclude that this approach will be much slower in round 1 than the former implementation. However it is questionable what is the performance benefit in rounds 2 - 5. As suggested by Tomáš Krňák [25], padding the circuit with a sparse polynomial might slightly speed up the polynomial division. One possibility is to use padding polynomial $pad_{cyclic}(x)$ that is zero on every k-th element of H instead of $pad_0(x)$ which is zero on $[\omega^m \dots \omega^{n-1}]$, i.e.,

$$pad_{cyclic}(x) = x^{\frac{n}{k}} - 1. \tag{5.3}$$

The domain H is generated by the n-th root of unity, so for every $\omega^k i$ it holds that:

$$pad_{cyclic}(\omega^{ki}) = \omega^{ki\frac{n}{k}} - 1 = \omega^{ni} - 1 = 1^i - 1 = 0.$$

This variant reduces the degree of the wire polynomial to $n - \frac{n}{k}$ instead of m. Moreover it requires changing the structure of the circuit. The witness w_l, w_r, w_o selector q_m, q_l, q_r, q_o, q_c and public input PI need to be shifted in such way that every k-th index is 0, otherwise Equation (2.7) fails. We also need to change the permutation function σ^* accordingly. This would require a major change in the repository protocol. Therefore, I did not implement this alternative.

As mentioned earlier, any value could be used as padding. We can try to find a different sparse polynomial to pad with. For example, we could pad by values of the polynomial that has a root at ω^m $pad_{m0}(x - \omega^m)$. However, this polynomial has only a degree 1, which means after the division, we get a reduction of degree by 1. This is not useful, so we have to state rough conditions for the polynomial we are searching for:

1. polynomial is *relatively* sparse
2. has a *high enough* degree
3. is not zero on $[1, \omega^{m-1}]$

Condition 1. should potentially achieve better performance in the polynomial division, and condition 2. should make sure that the division is worth it. One might ask if we could use a polynomial x^k for padding. This polynomial is sparse, does not zero out on the domain H, and could have a sufficient degree. First, let us examine the case for $k = 1$. If we divide a polynomial $W(x) = c_0 + c_1x + c_2 + x^2 \dots c_nx^n$ by x we reduce the degree of each monomial as $c_0/x + c_1 + c_2 + x \dots c_nx^{n-1}$. However, the problem is that c_0 is likely not divisible by x, which invalidates this approach. The same problem arises for larger k.

In the end, I did not succeed in finding a suitable polynomial. Even if such polynomial existed, it is unclear whether there is any reasonable performance gain. I have tried to measure naive polynomial division by polynomial $pad_0(x)$ and compare the time to other sparse polynomials on $n = 2^{14}$

	pad_0	$x^{n/8}$	$x^{n/4}$	$x^{n/2}$
Time (s)	152.1	47.32	85.84	114.3
Degree	32766	8192	163681	32768

Division by all of the sparse polynomials is faster than division by the zero-pad polynomial $pad_0(x)$. However, the efficiency mainly depends on the polynomial degree. In the case of this optimization, we would like to divide by a polynomial with a high degree to reduce the degree of the wire polynomial as much as possible. While division by $x^{n/4}$ is two times faster than division by $pad_0(x)$, this variant is still pretty slow, as seen in the Section 5.4.

5.3.4 Pairwise division

```
Reduction by pairwise division in domain coset
1 :  construct padding vector $[0]^{n-m}$
2 :  interpolate padding polynomial $pad_0 = \mathsf{iFFT}([0]^{n-m})$
3 :  calculate coset evaluations $\mathsf{cFFT}(pad_0(x)) = [pad_0(k1), \dots, pad_0(k\omega^{m-1})]$
4 :  concatenate $[w, [0]^{n-m}]$
5 :  pair-wise multiplication $\tilde{w} = [w_0 pad_0(1) \dots w_{m-1} pad_0(\omega^{m-1}), 0 \dots 0]$
6 :  interpolate $W(x) = \mathsf{iFFT}(\tilde{w})$
7 :  calculate coset evaluations $\mathsf{cFFT}(W(x)) = [W(k1) \dots, W(k\omega^{m-1})]$
8 :  reduce degree by pairwise division $\hat{w} = \left[\frac{W(k)}{pad_0(k)}, \frac{W(k\omega)}{pad_0(k\omega)} \dots \frac{W(k\omega^n)}{pad_0(k\omega^n)}\right]$
9 :  interpolate $w(x) = \mathsf{ciFFT}(\hat{w})$
10 : return $w(x)$
```

With the hope of bypassing the costs introduced by the polynomial division, we introduce the final approach, where the division is performed in the evaluation of the form. It is not possible to perform this computation on the domain H because $pad_0(x)$ is zero on $[\omega^m \dots \omega^{n-1}]$, which is why the division is performed in a coset of the domain H. As in the previous approach, both multiplication by the $pad_0(x)$ in H and division by $pad_0(x)$ in the coset of H could be trivially parallelized.

It might be a bit unclear why the reduction actually works since we interpolate n values in step 6. We will not go into details about the FFT algorithm, but we provide an intuition. The vector w comes from a specific distribution determined by the circuit C, public input $\mathbb{x}$, and the witness $\mathbb{w}$, but we will assume that it is uniformly random. If we take m evaluation of $w(x)$ and interpolate them over H, then with high probability, the evaluation on $[\omega^m \dots \omega^n]$ would not be zero. To be more specific, for a uniformly random polynomial, the probability would be $\frac{1}{|\mathbb{F}_p|}^{n-m}$. This is why simply padding w with zeros and using the iFFT returns a polynomial $W(x)$ of degree $n-1$. However, if we instead divide by the evaluation of the padding polynomial in the coset of the evaluation domain, then all the n evaluations correspond to the evaluations of $w(x)$, which has degree $m-1$. As a result, we get a polynomial of the degree $m-1$ from the interpolation.

Another trick that could make this approach faster is to precompute evaluations of $pad_0(x)$ on H and the cost of H in the protocol setup. The polynomial $pad_0(x)$ depends on m which is determined by the size of w and $deg(pad_0(x))$ could be anywhere in the interval $(\frac{n}{2}, n)$. The important thing is that $deg(pad(x)) \leq m$ because otherwise, the multiplication in step 5 of the algorithm would zero out part of w. So, we have to precompute evaluation for multiple sizes of $pad_0(x)$ and then use the suitable one. Although this approach does not completely reduce the degree to m, it is sufficient to gain performance benefits. In the implementation, we decided to precompute $\log n/2$ evaluation of $pad_0(x)$. This precomputation can save time when we are interested in a SNARK for the general circuit. However, when we want to run the prover only on a single circuit, we know the m in advance and can precompute specific padding for m.

Time analysis: Evaluation of $pad_0(x)$ can be computed using the recursive formula Equation (5.2). However, this needs to be done only once in the protocol setup. Pairwise, multiplication, and division could be parallelized. The wire polynomial needs to be interpolated by iFFT, evaluated at coset by cFFT and also interpolated on the coset using coset-iFFT. The total complexity is $3n \log n = \mathcal{O}(n \log n)$.

5.4 Benchmarks

5.4.1 Implementation description

The optimization we implemented focuses solely on refining the prover algorithm in Round 1, described in Section 3.2. Instead of direct interpolation of the values from the circuit, we implement algorithms from Section 5.3.3 and Section 5.3.4. The procedures use data types created to work with polynomials over finite field implemented in the ark-works library. The decision to use ark-works was deliberate, given that it is a popular open-source library for cryptographic primitives, and the implementation of ZK-Garage already heavily relied on it.

Specifically, the polynomials are stored in a vector where the coefficient of x^i is stored at location i. In the ark-works, this structure is `DensePolynomial`, and the elements of the vector are field elements. It is possible to choose from a variety of elliptic curves, all of the measurements were conducted with `BSL 12-381`. Operations on the domain are carried out using structure `GeneralEvaluationDomain`. It supports efficient FFT operations, enabling transformation between polynomial evaluation and coefficient forms. The algorithm in the Section 5.3.4 also introduced additional optimization in the form of precomputation to minimize computational overhead during proof generation. This is achieved in the preprocessing stage, and the information is stored in `ProverKey`. Throughout the implementation, standard dynamic arrays were predominantly used for storing variables and we did not find use for specialized data structures.

5.4.2 Local benchmarks

In the benchmarks, I compared two working implementations against the reference implementation. All measurements were taken on *BenchCircuit*, which benchmarked ZK-Garage PlonK. The circuit contains only dummy constraints and can be constructed with a variable number of gates. To verify the correctness of my code, I ran it on other example circuits included in the repository. If the verifier accepts the proof produced, there is a good chance that the changes will not break the implementation. We provide a link to a public repository with the implemented optimization.

I compared the two implementations from Section 5.3.3 and Section 5.3.4 against the reference implementation. At first, I ran the benchmarks locally on a consumer-grade notebook. The major limitation was the memory capacity due to the size of `SRS`. I measured the total proving time and the time needed to construct the wire polynomials. The plot in Figure 5.3 shows that the version with a naive polynomial division is unsurprisingly much worse than the ZK-Garage implementation. Construction of the wire polynomials alone takes 3 times the construction of the proof in ZK-Garage. Therefore, I will not consider this approach in further benchmarks.

The alternative with pairwise division Section 5.3.4 introduces overhead in the construction of the wire polynomials because it introduces additional `FFT`s. The improvement raises with the size of the circuit, which follows from the fact that the increased size of the circuit leads to greater padding. And the bigger the padding is, the more we can reduce the degree of the wire polynomials, resulting in improvements in rounds 2-5. So, for circuits of bigger size, the performance benefit becomes more pronounced. However, the improvement is much lower than what we were hoping for. To get insight into the problem, I measured what was happening in rounds 2-5.

5.4.3 Remote benchmarks

The rest of the measurements were taken on servers of the Department of Applied Mathematics of the Faculty of Mathematics and Physics of Charles University. The machine has CPU type *Intel Xeon E5-2630 v3* with 16 cores and a frequency of 3200 MHz. The memory usage after the setup of the protocol for the circuit of

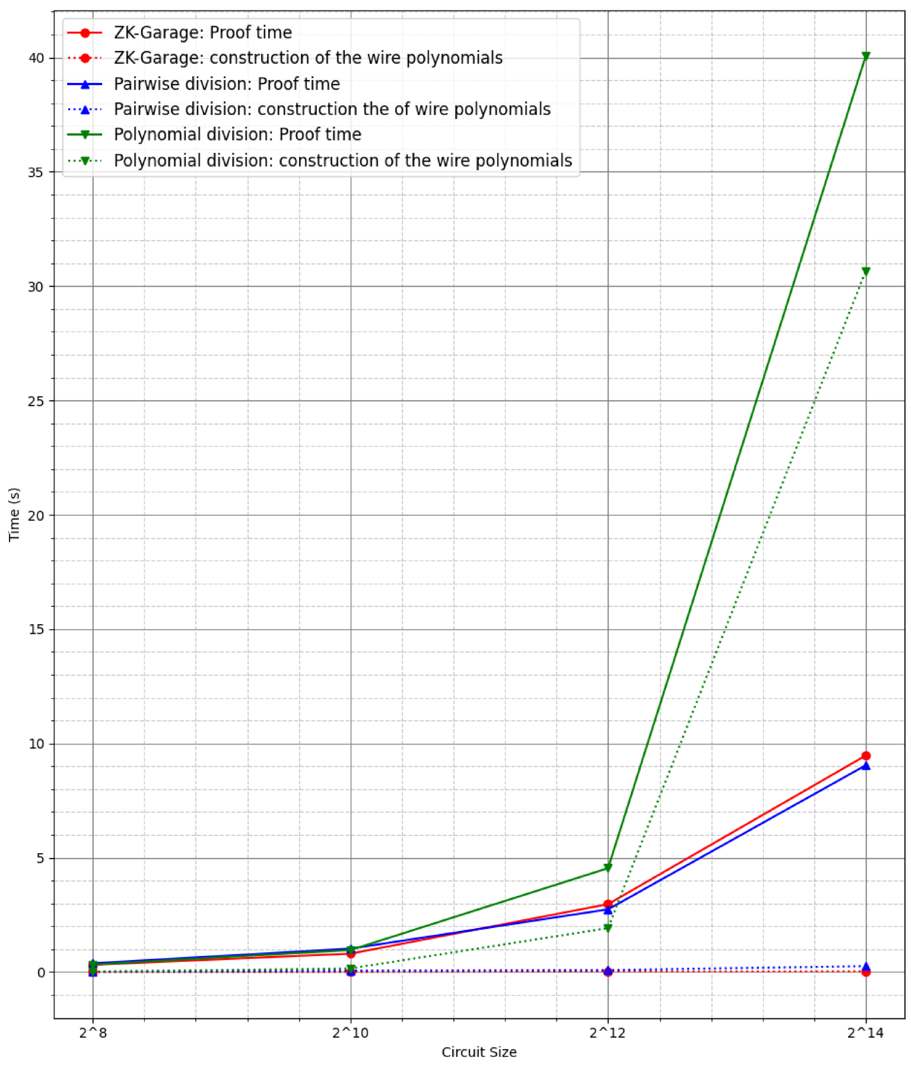


Figure 5.3 Local benchmarks on AMD Ryzen 7 5800H

size 2^{20} reached 50GB, and the whole protocol filled the entire memory of 125GB on the circuit of size 2^{21}. In addition to the proving time and the construction of wire polynomial $a(x), b(x), c(x)$, I have also measured commitments to these polynomials $[a]_1, [b]_1, [c]_1$ and also commitments to the quotient polynomial that is split into multiple parts as described in Section 3.4. Table 5.4.3 shows the exact comparison (in seconds) between the reference implementation and the degree reduction described by pairwise division in the coset from Section 5.3.4. These results are plotted on Section 5.4.3.

	ZK-Garage				Reduced Degree			
Circuit size	$a(x)$ $b(x)$ $c(x)$	$[a]_1$ $[b]_1$ $[c]_1$	$[t]_1$	Proof time	$a(x)$ $b(x)$ $c(x)$	$[a]_1$ $[b]_1$ $[c]_1$	$[t]_1$	Proof time
2^{15}	0.023	0.413	0.466	7.393	0.102	0.260	0.467	7.000
2^{16}	0.039	0.834	0.856	14.325	0.183	0.491	0.883	13.882
2^{17}	0.061	1.499	1.487	27.369	0.341	0.843	1.489	26.035
2^{18}	0.133	2.894	2.954	54.144	0.692	1.642	2.972	52.479
2^{19}	0.244	5.393	5.110	106.628	1.268	3.182	5.310	103.123
2^{20}	0.499	9.166	9.166	201.599	2.615	5.555	10.817	198.667

It is again evident that the improvement in the proof time is negligible. Time for the commitment of the wire polynomial decreased significantly as it should; however, the commitment of the quotient polynomial did not change at all. In this optimization, we wanted to reduce the degree of the wire polynomial with the hope the polynomial $t(x)$ constructed from the wire polynomials would also have a smaller degree. Since commitments to the quotient polynomial take approximately the same time and the complexity of commitment is dominated MSM, it must mean that the degree of the quotient polynomial $t(x)$ does not change at all.

Going back to the ZK-Garage implementation, I found out that the quotient polynomial is computed in the evaluation form, unlike the protocol description in the original paper [27]. First, all of the required polynomials are transformed into a coset of the domain H of size 8n. This is due to a similar reason as in Section 5.3.4 to avoid division by 0. Each of the parts of the quotient polynomial is computed in the evaluation form. Finally, they are merged and interpolated through coset-iFFT. The resulting polynomial is split into 8 parts, and the prover calculates the commitment to each of them. The reason why the polynomial needs to be split into 8 parts is that the Equation (2.7) is extended with checks for custom gates, and there is also an additional polynomial proving the validity of the lookup table. As a result, the degree of $t(x)$ is determined by interpolation, and the reduction of the wire polynomial does not affect the degree of the quotient polynomial.

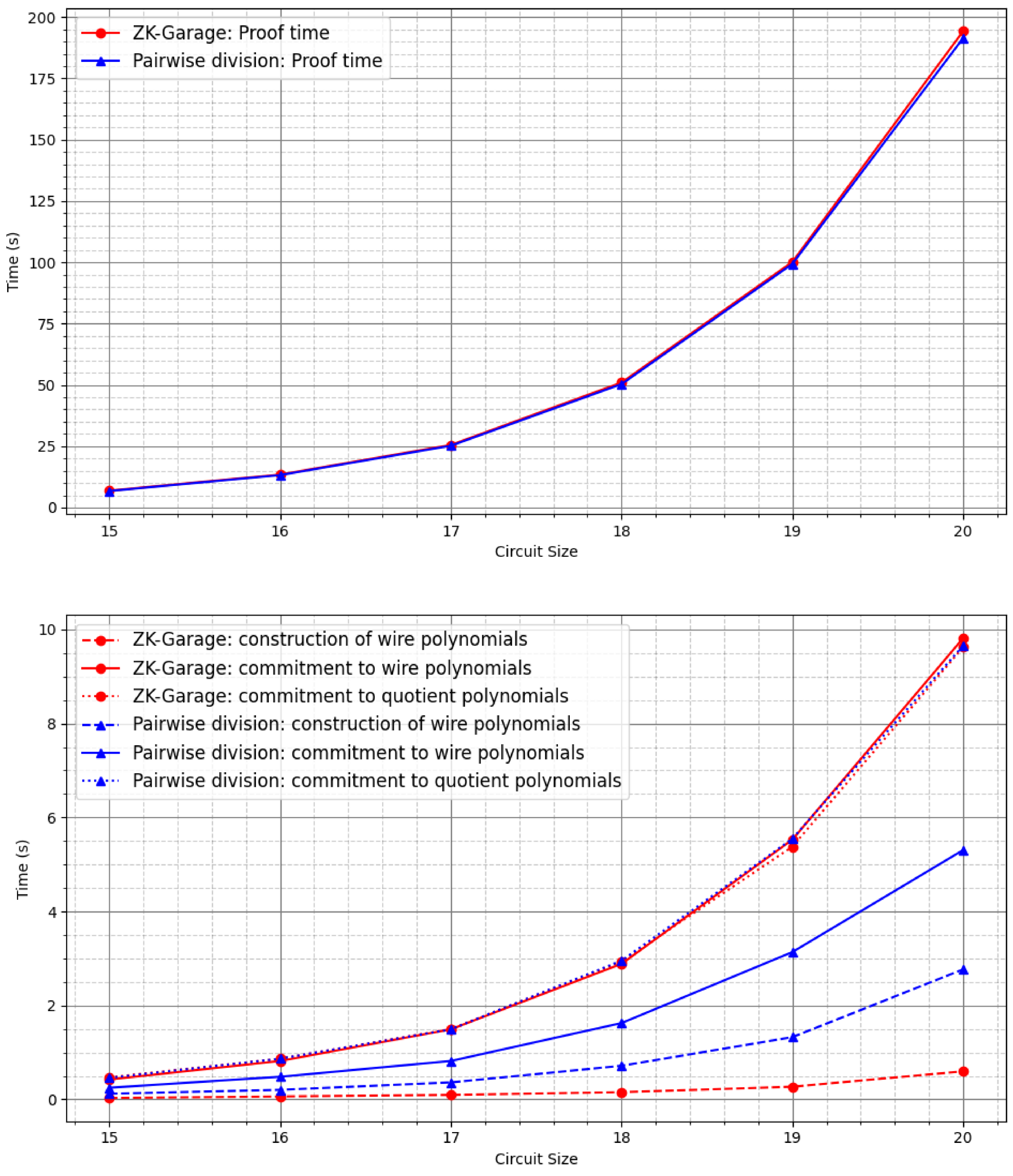


Figure 5.4 Remote benchmarks on Intel Xeon E5-2630 v3

```
ZK-Garage Round 3
1 : evaluate polynomial on coset of domain of size $8n$
2 : calculate $t_1(x)3.6, t_2(x)3.7t_3(x)3.8$ in the evaluation form
3 : calculate polynomial $t_4(x)$ for proving correctness of the lookup table
4 : merge and interpolate $t(x) = \mathsf{ciFFT}(t_1(x) + t_2(x) + t_3(x) + t_4(x))$
5 : split the result into 8 polynomials and calculate corresponding commitments
```

We can conclude that the only performance benefit of the optimization from Section 5.3.4 is in the computation of commitments $[a]_1, [b]_1, [c]_1$ and the quotient polynomial is computed from evaluations of wire polynomials. We know that it is possible to compute the commitment form a polynomial in evaluation form as explained in Section 2.3.3, so if it even needed to construct the polynomials $a(x), b(x), c(x)$? Round 3 uses the evaluation wire polynomials in the cost of the domain, which is easy to compute. The problem is that there are $8n$ evaluations instead of n. We have described how to compute a new evaluation with precomputed barycentric weights in n operation. That means calculating the evaluations on the domain of size 8n will take $8n^2$, which is slow. However, if there is a smarter way to do this that is comparable to $n \log n$, we could skip the computation of $a(x), b(x), c(x)$ together.

In conclusion, the optimization with degree reduction of the wire polynomials did not bring the desired performance benefit for this implementation of the $\mathcal{P}$lon$\mathcal{K}$ protocol due to the reason that round 3 is computed in another way than in the description of the protocol.

5.4.4 Engineering approach

Designing a more efficient variant of the $\mathcal{P}$lon$\mathcal{K}$ protocol is undoubtedly a hard task, and sometimes, an engineering approach might produce good results. Even though this was not the purpose of the work, there are many possible improvements on the software side. The most notable difference was produced by parallelized computations in round 2 and round 3. As already discussed, the construction of the wire protocol takes a significant portion of the proving time, which becomes even more pronounced for circuits of larger size. Since the quotient polynomial is computed in the evaluation form, the whole computation is trivially parallelizable. We improved the function for computing $t(x)$ by a parallelized approach to computing the evaluation of the polynomial for gate constraints, permutation constraints, and lookup table constraints. A similar approach also helped to speed up the construction of the permutation polynomial $z(x)$ in round 2. This approach was implemented using the parallel iterator in rayon [28], which is a lightweight data parallelism library. In the next measurement, we compared the approach Section 5.3.4 with parallel computation of $t(x)$ and $z(x)$. As can be seen from the results Figure 5.5, this change yields a significantly faster prover algorithm.

	permutaiton polynomial			quotient polynomial			total proof time		
Degree	zkg.	par.	imp.	zkg.	par.	imp.	zkg.	par.	imp.
2^{15}	0.23	0.05	76.52%	3.72	1.41	62.13%	6.84	4.18	38.88%
2^{16}	0.46	0.10	77.73%	7.40	2.77	62.58%	13.39	8.13	39.28%
2^{17}	0.89	0.18	79.42%	14.72	5.56	62.25%	25.55	15.16	40.66%
2^{18}	1.80	0.33	81.58%	29.56	11.64	60.63%	51.02	30.48	40.26%
2^{19}	3.55	0.66	81.44%	59.56	23.24	60.99%	100.75	59.26	41.18%
2^{20}	7.11	1.37	80.71%	120.17	47.00	60.89%	195.09	112.38	42.40%

While we have mentioned other design optimizations of the protocol, it might be also meaningful to work on an engineering solution. There have already been multiple attempts for hardware optimization of computationally expensive tasks like MSM, one of which is covered in the paper [10].

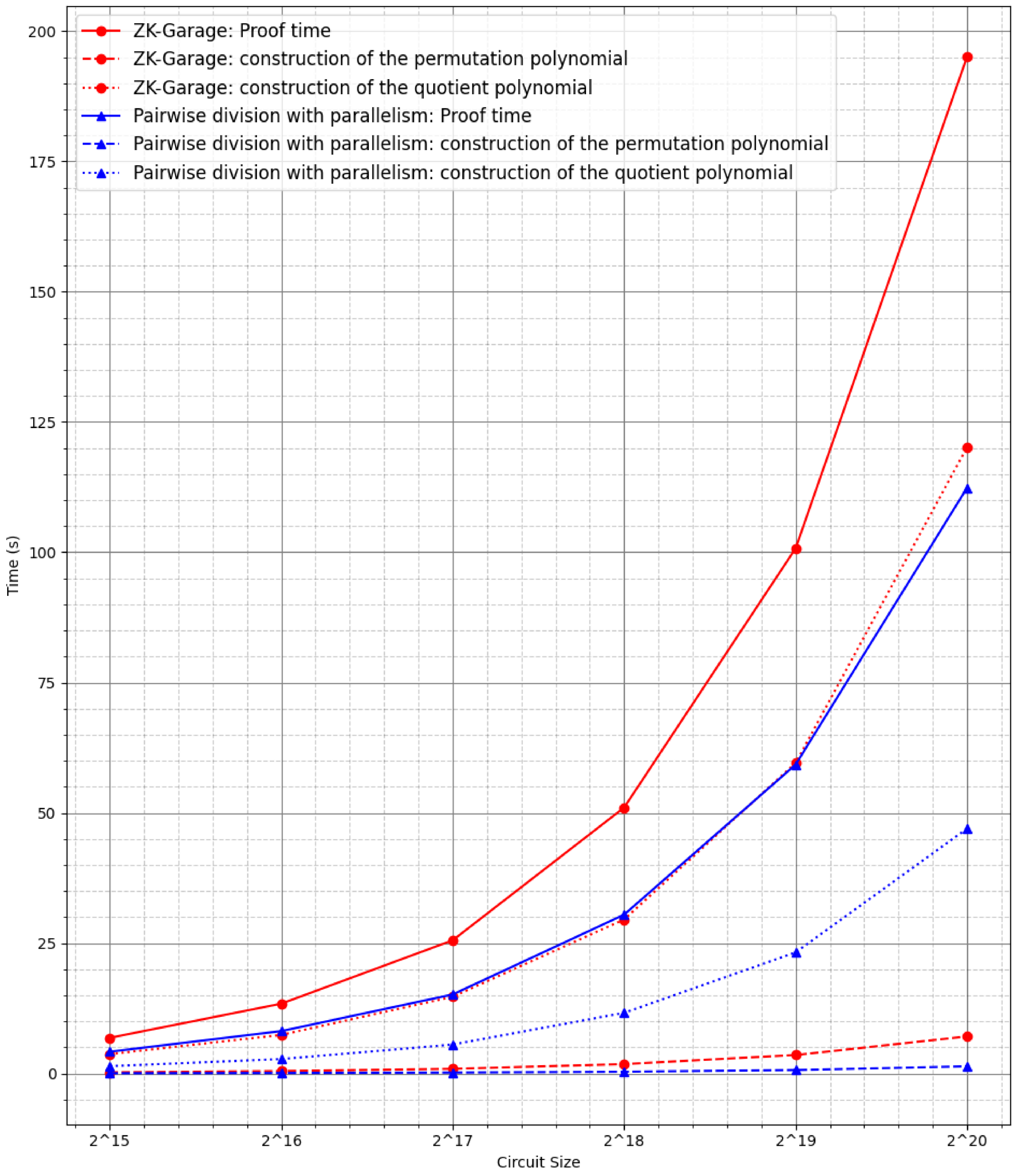


Figure 5.5 Parallel computation of the quotient polynomial

Conclusion

This work introduces SNARKs and provides a high-level overview of the core concepts underlying $\mathcal{P}\mathfrak{lon}\mathcal{K}$. The main focus is on explaining the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ protocol in detail and making it more accessible. Additionally, we explored potential optimization techniques, focusing on reducing the degree of wire polynomials. We discussed various approaches and their potential effectiveness and achieved a slight performance improvement. Ultimately, we discovered that a straightforward software optimization yielded a more notable speed-up.

Bibliography

1. Xing, Zhibo; Zhang, Zijian; Liu, Jiamou; Zhang, Ziang; Li, Meng; Zhu, Liehuang; Russello, Giovanni. *Zero-knowledge Proof Meets Machine Learning in Verifiability: A Survey.* 2023. Available from arXiv: 2310.14848 [cs.LG].
2. Kate, Aniket; Zaverucha, Gregory M.; Goldberg, Ian. Constant-Size Commitments to Polynomials and Their Applications. In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2010, vol. 6477, pp. 177–194. Lecture Notes in Computer Science. Available from doi: 10.1007/978-3-642-17373-8_11.
3. Chen, Binyi; Bünz, Benedikt; Boneh, Dan; Zhang, Zhenfei. *HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates* [Cryptology ePrint Archive, Paper 2022/1355]. 2022. Available also from: https://eprint.iacr.org/2022/1355. https://eprint.iacr.org/2022/1355.
4. Thaler, J. *Proofs, Arguments, and Zero-Knowledge.* Now Publishers, 2022. Foundations and Trends in Privacy and Security. isbn 9781638281252. Available also from: https://books.google.cz/books?id=imSazwEACAAJ.
5. Fiat, Amos; Shamir, Adi. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings.* Springer, 1986, vol. 263, pp. 186–194. Lecture Notes in Computer Science. Available from doi: 10.1007/3-540-47721-7_12.
6. Dao, Quang; Miller, Jim; Wright, Opal; Grubbs, Paul. *Weak Fiat-Shamir Attacks on Modern Proof Systems* [Cryptology ePrint Archive, Paper 2023/691]. 2023. Available also from: https://eprint.iacr.org/2023/691. https://eprint.iacr.org/2023/691.
7. Petkus, Maksym. Why and How zk-SNARK Works. *CoRR.* 2019. Available from arXiv: 1906.07221.
8. Bellés-Muñoz, Marta; Urroz, Jorge Jiménez; Silva, Javier. *Revisiting cycles of pairing-friendly elliptic curves* [Cryptology ePrint Archive, Paper 2022/1662]. 2022. Available also from: https://eprint.iacr.org/2022/1662. https://eprint.iacr.org/2022/1662.
9. Henry, Ryan; Cheriton, David R.; Ontario, Northeastern. Pippenger's Multiproduct and Multiexponentiation Algorithms (Extended Version). In: 2010. Available also from: https://api.semanticscholar.org/CorpusID:168527.
10. Xavier, Charles. F. *PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication* [Cryptology ePrint Archive, Paper 2022/999]. 2022. Available also from: https://eprint.iacr.org/2022/999. https://eprint.iacr.org/2022/999.
11. Agrawal, M.; Biswas, S. Primality and identity testing via Chinese remaindering. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039).* 1999, pp. 202–208. Available from doi: 10.1109/SFFCS.1999.814592.

12. Berrut, Jean-Paul; Trefethen, Lloyd N. Barycentric Lagrange Interpolation. *SIAM Review.* 2004, vol. 46, no. 3, pp. 501–517. Available from DOI: 10.1137/S0036144502417715.
13. Ben-Sasson, Eli; Bentov, Iddo; Horesh, Yinon; Riabzev, Michael. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In: *Electron. Colloquium Comput. Complex.* 2017. Available also from: https://api.semanticscholar.org/CorpusID:5637668.
14. Wegman, Mark N.; Carter, J.Lawrence. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences.* 1981, vol. 22, no. 3, pp. 265–279. ISSN 0022-0000. Available from DOI: https://doi.org/10.1016/0022-0000(81)90033-7.
15. Drake, Justin. *PLONK without FFTs.* 2020. Available also from: https://www.youtube.com/watch?v=ffXgxvlCBvo&t=2146s. Presented at zkSummit5.
16. Bellés-Muñoz, Marta; Isabel, Miguel; Muñoz-Tapia, Jose Luis; Rubio, Albert; Baylina, Jordi. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing.* 2023, vol. 20, no. 6, pp. 4733–4751. Available from DOI: 10.1109/TDSC.2022.3232813.
17. Boneh, Dan. *Building a SNARK (Part I).* [N.d.]. Available also from: https://zkhack.dev/whiteboard/module-two/. Table retrieved from slide 18.
18. Boneh, Dan; Drake, Justin; Fisch, Ben; Gabizon, Ariel. *Efficient polynomial commitment schemes for multiple points and polynomials* [Cryptology ePrint Archive, Paper 2020/081]. 2020. Available also from: https://eprint.iacr.org/2020/081. https://eprint.iacr.org/2020/081.
19. Ambrona, Miguel; Schmitt, Anne-Laure; Toledo, Raphael R.; Willems, Danny. *New optimization techniques for PlonK's arithmetization* [Cryptology ePrint Archive, Paper 2022/462]. 2022. Available also from: https://eprint.iacr.org/2022/462. https://eprint.iacr.org/2022/462.
20. Gabizon, Ariel; Williamson, Zachary J. *plookup: A simplified polynomial protocol for lookup tables* [Cryptology ePrint Archive, Paper 2020/315]. 2020. Available also from: https://eprint.iacr.org/2020/315. https://eprint.iacr.org/2020/315.
21. Bowe, Sean; Grigg, Jack; Hopwood, Daira. *Recursive Proof Composition without a Trusted Setup* [Cryptology ePrint Archive, Paper 2019/1021]. 2019. Available also from: https://eprint.iacr.org/2019/1021. https://eprint.iacr.org/2019/1021.
22. contributors, zk-garage. *zk-garage: PlonK Implementation by zk-Garage.* [N.d.]. Available also from: https://github.com/ZK-Garage/plonk.
23. contributors, arkworks. *arkworks: zkSNARK ecosystem.* [N.d.]. Available also from: https://arkworks.rs.
24. Ivanov, Nikolay. *Is Rust C++-fast? Benchmarking System Languages on Everyday Routines.* 2022. Available from arXiv: 2209.09127 [cs.PL].
25. Krňák, Tomáš. *Personal Conversation about PlonK acceleration.* 2024.
26. Cooley, James W.; Tukey, John W. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation.* 1965, vol. 19, pp. 297–301. ISSN 0025–5718. URL: http://cr.yp.to/bib/entries.html#1965/cooley.

27. Gabizon, Ariel; Williamson, Zachary J.; Ciobotaru, Oana. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge* [Cryptology ePrint Archive, Paper 2019/953]. 2019. Available also from: `https://eprint.iacr.org/2019/953`. `https://eprint.iacr.org/2019/953`.
28. contributors, rayon. ***rayon**: A data parallelism library for Rust.* [N.d.]. Available also from: `https://github.com/rayon-rs/rayon`.