

Cirrus: Performant and Accountable Distributed SNARK

Wenhao Wang
Yale University

Fangyan Shi
Tsinghua University

Dani VilardeLL
Cornell University

Fan Zhang
Yale University

Abstract—As Succinct Non-interactive Arguments of Knowledge (SNARKs) gain traction for large-scale applications, *distributed proof generation* is a promising technique to horizontally scale up the performance. In such protocols, the workload to generate SNARK proofs is distributed among a set of workers, potentially with the help of a coordinator. Desiderata include linear worker time (in the size of their sub-tasks), low coordination overhead, low communication complexity, and *accountability* (the coordinator can identify malicious workers). State-of-the-art schemes, however, do not achieve these properties.

In this paper, we introduced **Cirrus**, the first accountable distributed proof generation protocol with linear computation complexity for all parties. **Cirrus** is based on HyperPlonk (EUROCRYPT’23) and therefore supports a universal trusted setup. **Cirrus** is horizontally scalable: proving statements about a circuit of size $O(MT)$ takes $O(T)$ time with M workers. The per-machine communication cost of **Cirrus** is low, which is only logarithmic in the size of each sub-circuit. **Cirrus** is also accountable, and the verification overhead of the coordinator is efficient. We further devised a *load balancing* technique to make the workload of the coordinator independent of the size of each sub-circuit.

We implemented an end-to-end prototype of **Cirrus** and evaluated its performance on modestly powerful machines. Our results confirm the horizontal scalability of **Cirrus**, and the proof generation time for circuits with 2^{25} gates is roughly 40s using 32 8-core machines. We also compared **Cirrus** with Hekaton (CCS’24), and **Cirrus** is faster when proving PLONK-friendly circuits such as Pedersen hash.

1. Introduction

Succinct Non-interactive Arguments of Knowledge (SNARKs) [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] have emerged as a breakthrough in cryptographic tools, enabling efficient, non-interactive verification of computations. The primitive has proven practical for various applications, such as privacy-preserving payments (e.g., Zcash [12]), scalability solutions for decentralized consensus (e.g., zkRollups [13]), and blockchain interoperability (e.g., zkBridges [14]). However, for more ambitious applications, such as verifiable machine learning (zkML) and verifiable virtual machines (zkVM), the computational demands of proof generation remain a substantial bottleneck.

A recent line of works [14], [15], [16], [17], [18] proposed to horizontally scale up SNARK generation with *distributed SNARK generation protocols*. Distributed SNARK generation protocols allow multiple *workers* to collaborate on different parts of a general-purpose circuit, creating a final proof that is still succinct and sound. By splitting up the proof task across workers, distributed proof generation not only speeds up computation but also reduces memory usage, making it feasible to prove statements about large circuits that cannot fit in the memory of a single server.

An ideal distributed proof scheme should satisfy several properties. First, it should scale horizontally with *low overhead* (e.g., a worker’s computation should be linear in the size of its sub-task). Second, to support deployment in a decentralized setting where provers may be malicious, it should achieve *accountability*, allowing detection of malfeasance and identification of offenders. Accountability is necessary for applications such as prover markets, for instance [19], [20], [21], which aim to connect users and untrusted workers with spare computation power. Third, to support diverse applications, it should avoid per-circuit trusted setup because it is infeasible to require each application to perform its independent setup ceremony [22].

While prior works reduced overhead and achieved partial accountability, none achieved all three properties. Pisanist [16] is the first distributed proof generation scheme based on PLONK. It achieved a robust quasilinear prover time protocol for general circuits but did not support identifying malicious activity. Some following works [18] improved the prover complexity to linear time but still lacked accountability. Hekaton [17] introduced an accountable protocol with quasilinear prover time, but it relies on Mirage [23], a SNARK requiring a circuit-specific setup, making it costly to use in general-purpose circuits. Neither of them achieves linear worker time.

We present **Cirrus**, the first distributed proof generation scheme that is low overhead, accountable, and supports universal trusted setup. To be precise, **Cirrus** achieves linear time complexity on both the worker and coordinator sides, all while ensuring minimal communication overhead. **Cirrus** incorporates a mechanism that allows the coordinator to not only detect the existence of malicious behavior but also pinpoint the specific worker at fault and enable the implementation of incentive-based systems to enhance security and trustworthiness. By addressing the dual challenges of computational efficiency and accountability, **Cirrus** paves

the way for more scalable and secure SNARK applications across a broader range of decentralized systems.

1.1. Technical Overview

We start by distributing HyperPlonk [8] because HyperPlonk has a prover time linear in the circuit size (from which we achieve linear worker prover time) and supports a universal trusted setup. To distribute the SNARK generation process to different nodes, we follow the idea in [16]. We split a circuit C of N gates into M sub-circuits, where each sub-circuit has size T . Following this setting, M workers and one *coordinator* generate the proof together.

Distributing HyperPlonk. To successfully distribute HyperPlonk, we need to find a way to distribute multilinear KZG, sum-check protocol (SumCheck), zero test protocol (ZeroTest), and permutation test protocol (PermTest), since they are the essential building blocks of HyperPlonk. Distributing multilinear KZG is straightforward, yet naively distributing SumCheck using techniques in [14] would lead to the following problem: at the end of SumCheck we will have to open a multivariate polynomial of degree d , where d is a constant greater than 1, but with techniques in [14] and multilinear KZG we can only commit and open multivariate polynomials with degree at most 1. We overcome this challenge using the observation from [6]: each constant-degree multivariate polynomial can be written as a function of a constant number of multilinear polynomials. Following this observation, we can carefully separate each multivariate polynomial into multilinear polynomials, therefore making it possible to use distributed multilinear KZG to open polynomials at the end of distributed SumCheck. With distributed multivariate Sumcheck constructed, we can derive the constructions of distributed ZeroTest, distributed PermTest, and finally, distributed HyperPlonk.

Minimizing computation time of the coordinator. The coordinator’s computation time depends on the number of workers; thus, it can be a bottleneck when the number of workers is large. However, naively distributing HyperPlonk, the coordinator will end up with a computation time of $O(M \log T)$. This is because the proofs of SumCheck and multivariate KZG components are of size $O(\log T)$ for each worker prover, and the coordinator will have to add up M field or group element vectors of length $O(\log T)$. To mitigate this problem, Cirrus distributes the load of aggregating $O(M)$ vectors distributed to $M/\log T$ worker provers (called leaders), where each leader is responsible for adding the vectors of $\log T$ provers. In this way, the total runtime of this step for the coordinator is reduced to $O(M)$. We note that the computation overhead for each leader is $O(\log^2 T)$, which is dominated by $O(T)$.

Optimistic accountability. Up to this point, our protocol can only work when all workers are honest. To make our protocol accountable even assuming an arbitrary number of corrupted worker nodes, we need to enable the coordinator to tell which provers are malicious apart when the generated proof is not correct. To achieve this, we have the coordinator

verify all the computations done by the individual workers, as well as distinguish when it was the worker or the leader the one that provided the incorrect proof to the coordinator. However, verifying the computation results of each worker every time is computationally expensive. For example, the verification steps in distributed PermTest introduce a computation workload linear in the size of the entire circuit to the runtime of the coordinator. To solve this problem, we leverage the soundness property of Cirrus (i.e., the verifier has an overwhelming probability of detecting an incorrect final proof) and have the coordinator verify the final proof. If the proof is valid, the protocol is in the optimistic path and no further checks are required—by the soundness property, no worker is malicious with overwhelming probability. Otherwise, the protocol switches to the pessimistic path, and the coordinator checks the transcripts from each prover separately to identify the offenders.

1.2. Implementation and Evaluation

We implement Cirrus and evaluate its end-to-end performance. Our implementation is based on the HyperPlonk library [24]. To benchmark, we sampled random PLONK circuits with random witnesses and distributed the SNARK generation to up to 32 AWS `t3.2xlarge` machines (8 vCPUs and 32 GB memory). Our experiments demonstrate that Cirrus enables the proof of PLONK circuits containing 2^{25} gates in just 39 seconds, utilizing a network of 32 machines with modest computation power. With vanilla HyperPlonk, we can only prove circuits containing 2^{22} gates (in more than 110 seconds) using these machines; our results show that Cirrus achieves horizontal scalability. We also evaluate the computation time of the coordinator and show that with our load balancing technique, the computation time of the prover is indeed independent of the size of each sub-circuit.

Furthermore, we compared Cirrus with Hekaton, another accountable distributed SNARK scheme for general circuits. Our comparison shows that for PLONK-friendly tasks such as Pedersen hash, Cirrus is substantially faster than Hekaton by up to $3\times$. We note that for other tasks, such as SHA-256 hash, we cannot outperform Hekaton since the circuit representation of SHA-256 is far less efficient using PLONK gates than using R1CS.

Our Contribution

Here we summarize our contribution:

- We formally define *accountability* for distributed SNARK generation schemes. In an accountable distributed SNARK protocol, the coordinator can detect malicious node(s) liable for incorrect proof. This notion is particularly useful for deployment in a decentralized setting, like ZKP markets [19], where some nodes may be malicious.
- In Section 4, we introduce Cirrus, the first accountable distributed SNARK generation scheme that achieves the following properties: (1) truly (optimistically) linear prover time in the size of each sub-circuit for all workers,

TABLE 1: Comparison between Cirrus and existing distributed SNARK generation protocols. Comm. denotes the communication cost in total. \mathcal{V} Time denotes the verifier time. N is the number of gates (or constraints) in the whole circuit, and M is the number of workers. $T = N/M$ is the size of each sub-circuit. For Setup, “Circuit-specific” denotes that each different circuit requires its own trusted setup, “Universal” denotes that one set of trusted setup parameters works for circuits up to a certain size, and “Transparent” denotes that no trusted setup is required.

Scheme	Worker Time	Coordinator Time	Comm.	\mathcal{V} Time	Accountable	Setup
Cirrus (Ours)	$O(T)$	$O(M)$	$O(M \log T)$	$O(\log N)$	✓	Universal
Hekaton [17]	$O(T \log T)$	$O(S + M \log M)$	$O(M)$	$O(1)$	✓	Circuit-specific
HyperPianist [18]	$O(T)$	$O(M \log T)$	$O(M \log T)$	$O(\log N)$	✗	Universal
Pianist [16]	$O(T \log T)$	$O(M \log M)$	$O(M)$	$O(1)$	\sim^1	Universal
DeVirgo [14]	$O(T)$	N/A	$O(N)$	$O(\log^2 N)$	✗	Transparent
DIZK [15]	$O(T \log^2 T)$	N/A	$O(N)$	$O(1)$	✗	Circuit-specific

(2) truly (optimistically) linear coordinator time in the number of workers, and (3) has a universal trusted setup. Moreover, Cirrus can reuse existing setup parameters of HyperPlonk.

- In Section 5, we implement Cirrus and perform experiments to demonstrate its high performance. With our implementation, we can prove circuits of 2^{25} gates in less than 40 seconds using 32 AWS t3.2xlarge machines.

2. Related work

In this section, we review prior works on distributed proof generation schemes, focusing on asymptotical performance (prover time, verifier time, communications) and accountability. We summarize the comparison in Table 1.

DIZK. Wu et. al. [15] introduced a distributed approach to zkSNARK provers, focusing on optimizing key operations such as Fast Fourier Transforms (FFTs) and multi-scalar multiplications. Their system scales Groth16 by distributing the computation across multiple machines, and can handle much larger circuits using a cluster of machines. However, a significant limitation is that the communication cost of each machine is linear in the size of the *full circuit* (as opposed to the size of a worker’s sub-circuit) due to the distributed FFT algorithm. Another limitation of DIZK is that it uses a circuit-specific setup instead of a universal setup.

DeVirgo. Introduced in zkBridge [14], DeVirgo is a distributed variant of Virgo [7]. Authors of DeVirgo proposed a way to distribute the SumCheck protocol [25] and the FRI low-degree test across multiple machines. With n machines, the proof generation time is reduced by $1/n$. The protocol only supports data-parallel circuits (circuits with repeated sub-circuits). Despite these improvements in scalability over DIZK, DeVirgo incurs a per-worker communication cost linear in the size of the full circuit due to its reliance on the FRI low-degree test.

Pianist. [16] introduces a distributed proving algorithm for the PLONK SNARK that works for general circuits [3], aiming to reduce communication overhead in distributed proving systems. The core innovation in Pianist is the use

1. Pianist is accountable when using independent sub-circuits (e.g. multiple instances of the same circuit).

of bivariate polynomial commitments, which allows for decomposing PLONK’s global permutation check (which is responsible for ensuring the correctness of circuit wiring) into local permutation checks for each prover. Pianist achieves only partial accountability (only for data-parallel circuits) and their paper does not formally define accountability. Pianist is also the first distributed SNARK scheme to achieve constant per-node communication. Despite these advances, the prover time of each worker remains *quasi-linear* in the circuit size. In comparison, our protocol achieves linear worker prover time (in the size of the sub-circuit) and stronger accountability for general circuits (not just data-parallel circuits).

Mangrove. [26] presents a framework for dividing PLONK into segments of proofs and using folding schemes to aggregate them. While Mangrove shows promising theoretical results with estimated performance comparable to leading SNARKs, it has not been fully implemented or evaluated, especially when it comes to distributing the evaluation of the segments. Additionally, if applied to distributed proving, its techniques would require an inter-worker communication complexity that is linear in the circuit size. In comparison, Cirrus achieves an amortized communication complexity logarithmic in the size of each sub-circuit. Since Mangrove is not implemented, its concrete performance is unknown.

Hekaton. [17] proposes a “distribute-and-aggregate” framework to achieve accountable distributed SNARK generation. Specifically, Hekaton leverages memory-checking techniques, where a coordinator constructs a global memory based on the value of the shared wires among circuits. Then, the provers perform consistency checks on their memory access. In this way, the coordinator can detect malicious behavior, and therefore, Hekaton is an accountable scheme. However, like Pianist, the workers’ prover time of Hekaton is quasi-linear instead of truly linear in the size of the sub-circuit. Another drawback of Hekaton is that it requires a circuit-specific setup. Moreover, the coordinator’s work scales linearly with the global memory size, which could be a potential bottleneck when the number of shared wires among circuits is large. In comparison, the per-worker prover time of Cirrus is “truly” linear in the size of the sub-circuit, and the coordinator’s workload is independent of the number of shared wires among sub-circuits.

HyperPianist. [18] is a concurrent work with similar distributed permutation test and zero test techniques with multilinear polynomials, while they adopt a different distributed polynomial commitment scheme. However, their protocol is not accountable. There has not been a proof of completeness or soundness or a thorough performance evaluation.

SNARK aggregation schemes. A series of works [27], [28], [29] focuses on SNARK aggregation schemes, where the system uses cryptographic techniques to aggregate proofs of sub-circuits. However, these schemes can only aggregate the proofs when the sub-circuits do not have shared wires or inputs, and cannot be directly used to construct distributed SNARK generation schemes for general circuits.

Collaborative ZK-SNARKs. A recent line of work on *collaborative ZK-SNARKs* [30], [31], [32], [33], [34] addresses the privacy problem when generating proofs with witnesses from multiple parties using multi-party computation. All these schemes require preprocessing among all servers for each proof, which requires total communication that is linear in the size of the full circuit. Therefore, Collaborative ZK-SNARKs are not as efficient, though they achieve privacy, which is a non-goal for distributed proof generation schemes.

3. Preliminaries

In this section, we introduce the key notation, definitions, and foundational tools essential for the completeness and soundness of our protocol. For the interest of space, we move canonical definitions of SNARKs, polynomial interactive oracle proofs (Poly-IOPs), and polynomial commitment schemes (PCS's) to Appendix A. We also refer readers to Appendix B for details of the HyperPlonk protocol.

Notation.

- Let λ be the security parameter. If for all $c > 0$, $f(\lambda)$ is asymptotically smaller than λ^{-c} , then $f(\lambda)$ is called *negligible* and is denoted $\text{negl}(\lambda)$.
- Let \mathbb{F} be a finite field of size $\Omega(2^\lambda)$.
- Let $B_\mu := \{0, 1\}^\mu$ denote the μ -dimension boolean hypercube.
- Let $\mathbb{F}_\mu^{\leq d}[\mathbf{X}]$ denote the set of μ -variate polynomials where the degree of each variable is not greater than d . A multivariate polynomial g is *multilinear* if the degree of the polynomial in each variable is at most 1. Each element $f \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$ would satisfy $f(\mathbf{x}) = h(g_1(\mathbf{x}), \dots, g_c(\mathbf{x}))$ where h has total degree $O(d)$ and can be evaluated with an arithmetic circuit of $O(d)$ gates, and each g_i is multilinear.
- Let $\chi_w(\mathbf{x}) := \prod_{i=1}^\mu (w_i x_i + (1 - w_i)(1 - x_i))$ be a multilinear Lagrange polynomial over B_μ .
- Let $[\mathbf{x}] := \sum_{i=1}^\mu 2^{i-1} \cdot x_i$. Let $\langle v \rangle_m$ be the m -bit representation of $v \in [0, 2^m - 1]$.

Useful tools. The following tools are frequently used in the paper.

Lemma 1 (Multilinear Extension). *For a function $g : B_\mu \rightarrow \mathbb{F}$, there is a unique multilinear polynomial \tilde{g} such that $\tilde{g}(\mathbf{x}) = g(\mathbf{x})$ for all $\mathbf{x} \in B_\mu$. The function \tilde{g} is said to be the multilinear extension (MLE) of g . \tilde{g} can be expressed as*

$$\tilde{g}(\mathbf{x}) = \sum_{\mathbf{a} \in B_\mu} f(\mathbf{a}) \cdot \chi_{\mathbf{a}}(\mathbf{x}).$$

Lemma 2 (Schwartz-Zippel Lemma). *Let $g : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be a nonzero ℓ -variate polynomial of degree at most d . Then for all $\emptyset \neq S \subseteq \mathbb{F}$,*

$$\Pr_{\mathbf{x} \leftarrow S^\ell} [g(\mathbf{x}) = 0] \leq \frac{d}{|S|}.$$

3.1. Multilinear KZG PCS.

We now present the multilinear KZG PCS which is developed upon the scheme in [6], [35] for the multilinear polynomials as follows:

- **KeyGen**($1^\lambda, \mu, d$): Generate $\text{crs} = (\tau_1, \dots, \tau_\mu, (U_{\mathbf{b}} := g^{\chi_{\mathbf{b}}(\tau)}))_{\mathbf{b} \in B_\mu}$, where τ_1, \dots, τ_μ are secrets.
- **Commit**(f, crs): Suppose $f(\mathbf{x}) = \sum_{\mathbf{b} \in B_\mu} f_{\mathbf{b}} \cdot \chi_{\mathbf{b}}(\mathbf{x})$. \mathcal{P} computes $\text{com} := \prod_{\mathbf{b} \in B_\mu} U_{\mathbf{b}}^{f_{\mathbf{b}}}$.
- **Open**($f, \mathbf{a}, \text{crs}$): \mathcal{P} has $y := f(\mathbf{a})$ locally stored. Then it evaluates the multilinear degree-1 polynomials $q_i(\mathbf{x})$ such that $f(\mathbf{x}) - y = \sum_{i \in [\mu]} (x_i - a_i) q_i(\mathbf{x})$. Then it computes $\pi_i = g^{q_i(\tau)}$, and takes π as proof of the opening.
- **Verify**($\text{com}, \pi, \mathbf{a}, y, \text{crs}$): The verifier checks if $e(\text{com}/g^y, g) = \prod_{i \in [\mu]} e(\pi_i, g^{\tau_i - a_i})$.

It is shown in [6] that to commit a μ -variate multilinear polynomial f , the cost of the prover is $O(2^\mu)$ group operations. To open f , the cost of the prover is $O(2^\mu)$ group operations, and the proof size is $O(\mu)$ group elements. The cost of the verifier is $O(\mu)$ bilinear mappings and $O(\mu)$ group operations.

3.2. Accountability

Definition 1 (Accountable Collaborative Proving System). *For a circuit \mathcal{C} with M workers and one designated coordinator, an accountable distributed SNARK generation scheme is a protocol that ensures completeness, knowledge soundness, and succinctness, along with the following key property:*

- **Accountability.** *For any number of malicious worker provers who produce incorrect proofs, the system enables the coordinator to reliably and accurately identify all such malicious participants.*

In an environment where each prover except the coordinator may sabotage the distributed proving process by intentionally sending incorrect information such as wrong proofs, an *accountable* distributed SNARK generation system enables the coordinator \mathcal{P}_0 to either (1) generate a valid proof or (2) detect *all* malicious provers. In [16] the authors proposed such a scheme for circuits comprising independent sub-circuits and left the construction of accountable

distributed SNARK generation schemes that work for single circuits as an open problem.

4. Cirrus: Accountable and Efficient Distributed SNARK

In this section, we formally describe Cirrus, a distributed SNARK generation scheme with linear prover time where the coordinator can make accountable any malicious behavior. We start by giving the construction to distribute HyperPlonk for a general circuit. We first present a distributed multilinear KZG PCS, and how to construct a distributed multivariate SumCheck protocol that is compatible with our distributed PCS. With the distributed multivariate SumCheck protocol, we show how to construct the distributed HyperPlonk accordingly.

The distributed HyperPlonk protocol is complete and sound, but it still suffers from the following drawbacks: (1) the coordinator cannot find accountable the malicious prover(s) if the final proof is incorrect, and (2) the coordinator needs to perform $O(M \log T)$ group and field operations, which is not truly linear in M and T . To tackle the first challenge, we propose a verification protocol that allows the coordinator to detect any malicious prover. The core technique in the verification protocol is that the coordinator can evaluate the permutation products of each circuit segment. Note that the additional verification steps would introduce a large overhead for the coordinator if the verification is performed on the fly. We overcome this with an alternative protocol: the coordinator first verifies the final proof to check if there exist malicious nodes. The coordinator will only run the complete check if the verification fails. In this way, the optimistic runtime overhead of the coordinator can be reduced to the verifier time of our protocol. To tackle the second challenge, we delegate part of the work of the coordinator to multiple nodes to reduce the runtime of the coordinator and show how this technique can work along with the optimistic verification. This is done while keeping all the coordinator's computation time linear.

4.1. Distributedly Computable HyperPlonk

Our protocol is built on top of HyperPlonk [8], an adaptation of PLONK to the boolean hypercube, using multilinear polynomial commitments to remove the need for FFT computation and achieving linear prover time. We first define polynomials necessary for the protocol.

Defining polynomials. Before running the distributed SNARK, we assume that a circuit \mathcal{C} has been divided into $M = 2^\xi$ different sub-circuits. This can be done by just dividing the PLONK trace t of the circuit evenly into M chunks $\{t^{(b)}\}_{b \in B_\xi}$, where the gates within the same chunk are in the same circuit segment. Then each circuit segment has its own public inputs, addition and multiplication gate selectors, and wiring permutations. Let $\nu^{(b)}$ be the length of the binary index to represent public inputs for \mathcal{P}_b , i.e. $2^{\nu^{(b)}} = \ell_x^{(b)}$. Let μ be the length of the binary index to

represent the gates for all workers, i.e. $2^\mu = |\mathcal{C}|/M$. Let $s^{(b)}$ represent the gate selection vector for \mathcal{P}_b . Note that there is a wiring permutation between different circuit segments. Let $s^{(b)}, \sigma^{(b)} : B_{\mu+2} \rightarrow B_{\mu+2}$ and $\rho^{(b)} : B_{\mu+2} \rightarrow B_\xi$ be the mappings of the circuit wiring for \mathcal{P}_b , such that $\{(\sigma^{(b)}(c), \rho^{(b)}(c)) : c \in B_{\mu+2}, b \in B_\xi\} = B_{\mu+2} \times B_\xi$. Specifically, the permutations indicate that $t_i^{(b)} = t_{\sigma^{(b)}(\langle i \rangle)}^{\rho^{(b)}(\langle i \rangle)}$ for all $i \in [2^\mu], b \in B_\xi$.

A worker prover \mathcal{P}_b interpolates the following polynomials:

- Two multilinear polynomials $S_{\text{add}}^{(b)}, S_{\text{mult}}^{(b)} \in \mathbb{F}_{\mu}^{\leq 1}[\mathbf{X}]$ such that for all $i \in [N^{(b)}]$

$$\begin{cases} S_{\text{add}}^{(b)}(\langle i \rangle_\mu) = s_i^{(b)} \\ S_{\text{mult}}^{(b)}(\langle i \rangle_\mu) = 1 - s_i^{(b)} \end{cases}$$

- A multilinear polynomial $F^{(b)} \in \mathbb{F}_{\mu+2}^{\leq 1}$ such that

$$\begin{cases} F^{(b)}(0, 0, \langle i \rangle_\mu) = t_{3i+1}^{(b)} & i \in [0, N^{(b)} - 1] \\ F^{(b)}(0, 1, \langle i \rangle_\mu) = t_{3i+2}^{(b)} & i \in [0, N^{(b)} - 1] \\ F^{(b)}(1, 0, \langle i \rangle_\mu) = t_{3i+3}^{(b)} & i \in [0, N^{(b)} - 1] \\ F^{(b)}(1, \dots, 1, \langle i \rangle_\nu) = x_{i+1}^{(b)} & i \in [0, \ell_x^{(b)} - 1] \end{cases}$$

- A multilinear polynomial $I^{(b)} \in \mathbb{F}_{\nu^{(b)}}^{\leq 1}[\mathbf{X}]$ such that for all $i \in [0, \ell_x^{(b)} - 1]$ we have $I^{(b)}(\langle i \rangle_{\nu^{(b)}}) = x_{i+1}^{(b)}$.

Distributed multilinear KZG PCS. First, we introduce the distributed multilinear KZG PCS. This protocol enables us to commit and open a multilinear polynomial distributedly. For a multilinear polynomial $f(\mathbf{x}, \mathbf{y}) = \sum_{b \in B_\xi} f^{(b)}(\mathbf{x})\chi_b(\mathbf{y}) \in \mathbb{F}_{\mu+\xi}^{\leq 1}[\mathbf{X}]$, the PCS protocol is described as follows:

- **KeyGen:** Generate $\text{crs} = (g, (g^{\tau_i})_{i \in [\mu+\xi]}, (U_{c,b} := g^{\chi_c(\tau_1, \dots, \tau_\mu) \cdot \chi_b(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})})_{c \in B_\mu, b \in B_\xi})$ where $\tau_1, \dots, \tau_{\mu+\xi}$ are secrets. Note that we can derive $V_b := g^{\chi_b(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with crs , which will be useful to \mathcal{P}_0 , since $\sum_{c \in B_\mu} \chi_c \equiv 1$.
- **Commit(f, crs):** each \mathcal{P}_b computes $\text{com}_b := \prod_{c \in B_\mu} U_{c,b}^{f^{(b)}(c)}$ and sends com_b to \mathcal{P}_0 . Then \mathcal{P}_0 computes $\text{com} := \prod_{b \in B_\xi} \text{com}_b$.
- **Open($f, \alpha, \beta, \text{crs}$):**
 - 1) Each prover \mathcal{P}_b computes $z^{(b)} := f^{(b)}(\alpha)$ and $f^{(b)}(\mathbf{x}) - f^{(b)}(\alpha) := \sum_{i \in [\mu]} q_i^{(b)}(\mathbf{x})(x_i - \alpha_i)$. Then it computes $\pi_i^{(b)} = g^{q_i^{(b)}(\tau_1, \dots, \tau_\mu) \cdot \chi_b(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with crs , and sends $\pi^{(b)}$ and $z^{(b)}$ to \mathcal{P}_0 .
 - 2) \mathcal{P}_0 computes $z = f(\alpha, \beta) = \sum_{b \in B_\xi} z^{(b)} \cdot \chi_b(\beta)$. \mathcal{P}_0 decomposes

$$f(\alpha, \mathbf{y}) - f(\alpha, \beta) = \sum_{j \in [\xi]} q_j(\mathbf{y})(y_j - \beta_j).$$

Then it computes $\pi_{\mu+j} = g^{q_j(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with V_b . \mathcal{P}_0 also computes $\pi_i = \prod_{b \in B_\xi} \pi_i^{(b)}$ for all $i \in [\mu]$.

- 3) Then \mathcal{P}_0 sends $\pi := (\pi_1, \dots, \pi_{\mu+\xi})$ and z to \mathcal{V} .

- **Verify**(com, π , α , β , z , crs): The verifier checks if $e(\text{com}/g^z, g) = \prod_{i \in [\mu]} e(\pi_i, g^{\tau_i - \alpha_i}) \cdot \prod_{j \in [\xi]} e(\pi_{\mu+j}, g^{\tau_{\mu+j} - \beta_j})$.

Here we summarize the complexity of the distributed multilinear KZG PCS:

- \mathcal{P}_i **Time**: $O(2^\mu)$
- \mathcal{P}_0 **Time**: $O(2^\xi \cdot \mu)$
- **Verifier Time**: $O(\mu + \xi)$
- **Communication**: $O(\mu + \xi)\mathbb{G}$

Distributed multivariate SumCheck Poly-IOP. Here we describe the distributed multivariate SumCheck Poly-IOP. Suppose each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)} \in \mathbb{F}_\mu[\mathbf{X}]$. The provers want to show to the verifier that a multivariate polynomial $f(\mathbf{x}) := h(g_1(\mathbf{x}), \dots, g_c(\mathbf{x})) \in \mathbb{F}_{\mu+\xi}[\mathbf{X}]$ satisfies $\sum_{\mathbf{x} \in B_{\mu+\xi}} f(\mathbf{x}) = v$, where each g_i is multilinear and h can be evaluated using an arithmetic circuit with $O(d)$ gates. Suppose each prover \mathcal{P}_b has access to $g_i^{(b)}$ for $i \in [c]$, where $g_i^{(b)}(\mathbf{y}) = g_i(\mathbf{y}, \mathbf{b})$. We further define $f^{(b)}(\mathbf{y}) = f(\mathbf{y}, \mathbf{b}) = h(g_1^{(b)}(\mathbf{y}), \dots, g_c^{(b)}(\mathbf{y}))$. The protocol is described as follows:

- 1) For each $i \in [\mu]$:
 - a) Let $\alpha_{i-1} = (\alpha_1, \dots, \alpha_{i-1})$
 - b) Each \mathcal{P}_b computes and sends to \mathcal{P}_0 the polynomial $r_i^{(b)}(x) := \sum_{\mathbf{w} \in B_{\mu-i}} f^{(b)}(\alpha_{i-1}, x, \mathbf{w})$.
 - c) Then \mathcal{P}_0 adds

$$\begin{aligned} r_i &:= \sum_{\mathbf{b} \in B_\xi} r_i^{(b)} = \sum_{\mathbf{b} \in B_\xi, \mathbf{w} \in B_{\mu-i}} f^{(b)}(\alpha_{i-1}, x, \mathbf{w}) \\ &= \sum_{\mathbf{w} \in B_{\mu-i}} \sum_{\mathbf{b} \in B_\xi} f(\alpha_{i-1}, x, \mathbf{w}, \mathbf{b}) \end{aligned}$$

and sends the oracle of r_i to \mathcal{V} .

- d) \mathcal{V} checks if $v = r_i(0) + r_i(1)$, and samples and sends to \mathcal{P}_0 a random $\alpha_i \leftarrow \mathbb{F}$. Then \mathcal{V} sets $v := r_i(\alpha_i)$.
- 2) Let $\alpha = (\alpha_1, \dots, \alpha_\mu)$.
- 3) \mathcal{P}_0 receives $g_i^{(b)}(\alpha)$ for $i \in [c]$ from each \mathcal{P}_b . \mathcal{P}_0 first construct

$$\begin{aligned} \tilde{g}_i(\mathbf{y}) &:= g(\alpha, \mathbf{y}) = \sum_{\mathbf{b} \in B_\mu} g_i(\alpha, \mathbf{b}) \chi_{\mathbf{b}}(\mathbf{y}) \\ &= \sum_{\mathbf{b} \in B_\mu} g_i^{(b)}(\alpha) \chi_{\mathbf{b}}(\mathbf{y}) \end{aligned}$$

Then \mathcal{P}_0 has $\tilde{f}(\mathbf{y}) = f(\alpha, \mathbf{y}) = h(\tilde{g}_1(\mathbf{y}), \dots, \tilde{g}_c(\mathbf{y}))$.

- 4) \mathcal{P}_0 and \mathcal{V} perform Multivariate SumCheck on \tilde{f} with target value v . Denote the challenge as β . Note that in the final round, the verifier queries $g_i(\alpha, \beta)$ for $i \in [c]$ and calculates $f(\alpha, \beta)$ itself.

We give an example of how to perform SumCheck with $f^{(b)} = f_1^{(b)} \cdot f_2^{(b)}$, to illustrate the sum-check protocol on $h(f_1, f_2, \dots, f_k)$. In Step 3, \mathcal{P}_b opens $f^{(b)}$, and compute $g(\mathbf{y}) := \sum_{\mathbf{b} \in B_\xi} (v_1^{(b)} \cdot \chi_{\mathbf{b}}(\mathbf{y})) \cdot \sum_{\mathbf{b} \in B_\xi} (v_2^{(b)} \cdot \chi_{\mathbf{b}}(\mathbf{y}))$. In Step 4, \mathcal{V} and the provers open $f_1(\alpha, \beta) \cdot f_2(\alpha, \beta)$ at the end.

We summarize the complexity of the distributed multivariate SumCheck Poly-IOP as follows:

- \mathcal{P}_i **Time**: $O(2^\mu \cdot d \log^2 d)$
- \mathcal{P}_0 **Time**: $O(2^\xi \cdot d \log^2 d + \mu \cdot 2^\xi \cdot d)$
- **Verifier Time**: $O((\mu + \xi) \cdot d)$
- **Communication**: $O((\mu + \xi) \cdot d)\mathbb{F}$

Distributed multivariate ZeroTest Poly-IOP. Suppose each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)}(\mathbf{x}) \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$. The provers want to show to the verifier that $f^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{b} \in B_\xi$ and $\mathbf{x} \in B_\mu$. The protocol is described as follows:

- 1) \mathcal{V} samples and sends to \mathcal{P}_0 two random vectors $\mathbf{r} \leftarrow \mathbb{F}^\mu$ and $\mathbf{r}_0 \leftarrow \mathbb{F}^\xi$.
- 2) \mathcal{P}_0 sends \mathbf{r} to each \mathcal{P}_b . Note that when we apply the Fiat-Shamir heuristic to make the argument non-interactive, communication in this step is no longer needed.
- 3) Each \mathcal{P}_b sets $\tilde{f}^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) \cdot \chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{b})$.
- 4) The provers and \mathcal{V} run distributed multivariate SumCheck protocol on $\tilde{f}^{(b)}(\mathbf{x})$ with target value 0.

To improve the efficiency of the protocol, the verifier has oracle to $f(x, y) := \sum_{\mathbf{b} \in B_\xi} f^{(b)}(\mathbf{x}) \cdot \chi_{\mathbf{b}}(\mathbf{y})$ and knows $\chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{y})$. In Step 4 of SumCheck, \mathcal{P}_0 will evaluate $\chi_{\mathbf{r}_0}(\mathbf{y})$ over the boolean hypercube B_ξ using dynamic programming techniques [6] and then add them up. Since \mathcal{V} has oracle access to f and can efficiently evaluate $\chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{y})$ at a random point in $O(\mu + \xi)$ time, the protocol is succinct.

We present the proof of soundness and completeness for the ZeroTest protocol as an example, as the proofs for the distributed protocols follow analogous arguments found in the classic protocols.

Proposition 1. *The Distributed Multivariate ZeroTest Poly-IOP presented above is complete and sound.*

Proof. Let $F(x, y) := \sum_{\mathbf{b} \in B_\xi, \mathbf{c} \in B_\mu} f^{(b)}(\mathbf{c}) \cdot \chi_{\mathbf{c}}(\mathbf{x}) \cdot \chi_{\mathbf{b}}(\mathbf{y})$. If $f^{(b)}(\mathbf{c})$ is identically zero for all $\mathbf{b} \in B_\xi$ and $\mathbf{c} \in B_\mu$, F is identically zero, therefore, $F(\mathbf{r}, \mathbf{r}_0) = 0$ and the SumCheck will always pass. On the other hand, if F is not identically zero, by Schwartz-Zippel lemma $F(\mathbf{r}, \mathbf{r}_0) = 0$ holds w.p. at most $(\mu + \xi)d/|\mathbb{F}|$, which is negligible. \square

Distributed multivariate PermTest Poly-IOP. Let $\sigma^{(b)} : B_\mu \rightarrow B_\mu$ and $\rho^{(b)} : B_\mu \rightarrow B_\xi$ be two mappings such that $\{(\sigma^{(b)}(\mathbf{c}), \rho^{(b)}(\mathbf{c})) : \mathbf{c} \in B_\mu, \mathbf{b} \in B_\xi\} = B_\mu \times B_\xi$.

Each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)} \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$. The provers want to show to the verifier that $f^{(b)}(\mathbf{c}) = f^{(\rho^{(b)}(\mathbf{c}))}(\sigma^{(b)}(\mathbf{c}))$ for all $\mathbf{b} \in B_\xi$ and $\mathbf{c} \in B_\mu$. Here we introduce the protocol in the more complicated case where $\rho^{(b)}(\mathbf{c})$ is not identically \mathbf{b} for all $\mathbf{c} \in B_\mu$.

We define the multivariate polynomials $s, s_\mu^{(b)}, s_\xi^{(b)} \in \mathbb{F}_\mu^{\leq 1}[\mathbf{X}]$ where $s(\mathbf{x}) := [\mathbf{x}]$, $s_\mu^{(b)}(\mathbf{x}) := [\sigma^{(b)}(\mathbf{x})]$ and $s_\xi^{(b)}(\mathbf{x}) := [\rho^{(b)}(\mathbf{x})]$. The protocol then goes as follows:

- 1) \mathcal{V} samples and sends to the coordinator $\gamma_\mu, \gamma_\xi, \delta \leftarrow \mathbb{F}$.
- 2) Let $f_1^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s(\mathbf{x}) + \gamma_\xi \cdot [\mathbf{b}] + \delta$ and $f_2^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s_\mu^{(b)}(\mathbf{x}) + \gamma_\xi \cdot s_\xi^{(b)}(\mathbf{x}) + \delta$. Each prover \mathcal{P}_b builds a multilinear polynomial $z^{(b)} \in \mathbb{F}_{\mu+1}^{\leq 1}[\mathbf{X}]$ such that for all $\mathbf{x} \in B_\mu$

$$\begin{cases} z^{(b)}(0, \mathbf{x}) = f_1^{(b)}(\mathbf{x})/f_2^{(b)}(\mathbf{x}) \\ z^{(b)}(1, \mathbf{x}) = z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1), \mathbf{x} \neq \mathbf{1} \\ z^{(b)}(1, \mathbf{1}) = 0 \end{cases}$$

Let $w_1^{(b)}(\mathbf{x}) := z^{(b)}(1, \mathbf{x}) - z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1)$ and $w_2^{(b)}(\mathbf{x}) := f_2^{(b)}(\mathbf{x}) \cdot z^{(b)}(0, \mathbf{x}) - f_1^{(b)}(\mathbf{x})$.

- 3) Each \mathcal{P}_b sends $z^{(b)} := z^{(b)}(1, 1, \dots, 1, 0)$ to \mathcal{P}_0 . We have $\prod_{b \in B_\mu} z^{(b)} = 1$. Then \mathcal{P}_0 interpolates a multilinear polynomial $z \in \mathbb{F}_{\xi+1}^{\leq 1}[\mathbf{X}]$ such that

$$\begin{cases} z(0, \mathbf{y}) = z^{(y)} \\ z(1, \mathbf{y}) = z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1) \end{cases}$$

Let $w_3(\mathbf{y}) := z(1, \mathbf{y}) - z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1)$ and $w_4^{(b)}(\mathbf{x}) := z(0, \mathbf{b}) - z^{(b)}(\mathbf{x}, 0) \cdot \chi_{(1, 1, \dots, 1, 0)}(\mathbf{x}, 0)$.

- 4) The provers and \mathcal{V} run distributed multivariate ZeroTest on $\{w_1^{(b)}\}$, $\{w_2^{(b)}\}$ and $\{w_4^{(b)}\}$. \mathcal{P}_0 and \mathcal{V} run multivariate ZeroTest on w_3 .

Distributed HyperPlonk Poly-IOP. We construct a distributedly computable HyperPlonk Poly-IOP as follows:

- **Input Constraint:** \mathcal{V} ensures that $F^{(b)}(1, \dots, 1, \mathbf{x}) - I^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\nu$ and $\mathbf{b} \in B_\xi$ with distributed multilinear ZeroTest.
- **Output Constraint:** \mathcal{V} queries $F^{(1, \dots, 1)}(1, 0, \langle N^{(b)} - 1 \rangle_\mu)$ and checks if it is zero.
- **Gate Constraint:** Define a multivariate polynomial

$$G^{(b)}(\mathbf{x}) := S_{\text{add}}^{(b)}(\mathbf{x})(F^{(b)}(0, 0, \mathbf{x}) + F^{(b)}(0, 1, \mathbf{x})) + S_{\text{mult}}^{(b)}(\mathbf{x})(F^{(b)}(0, 0, \mathbf{x}) \cdot F^{(b)}(0, 1, \mathbf{x}) - F^{(b)}(1, 0, \mathbf{x})).$$

\mathcal{V} checks that $G^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\mu$ and $\mathbf{b} \in B_\xi$ with distributed multivariate ZeroTest.

- **Wiring Constraint:** \mathcal{V} verifies that $F^{(b)}(\mathbf{x}) = F^{(\rho^{(b)}(\mathbf{x}))}(\sigma^{(b)}(\mathbf{x}))$ for all $\mathbf{x} \in B_{\mu+2}$ and $\mathbf{b} \in B_\xi$ with distributed multivariate PermTest.

Batch openings. The batch opening is a technique introduced in [3], [8] that reduces prover and verifier time. It is especially useful when the practical bottleneck for the prover and the verifier is polynomial opening and verification, instead of field operations. Here we introduce the distributed batch opening protocol.

Assume there are 2^ρ distributed polynomials $\{f_v^{(b)}\}_{v \in B_\rho, b \in B_\xi}$, and 2^ρ evaluation points $\{(\alpha_v, \beta_v)\}_{v \in B_\rho}$, with opening values $\{z_v\}_{v \in B_\rho}$. Then the distributed batch opening and verification protocol is described as follows:

- The verifier \mathcal{V} samples and sends to \mathcal{P}_0 $t \xleftarrow{\$} \mathbb{F}^\rho$.
- \mathcal{V} computes the target sum $s := \sum_{v \in B_\rho} \chi_v(t) \cdot z_v$.
- Each worker prover \mathcal{P}_b defines a multilinear polynomial $g^{(b)}(v, c) := \chi_v(t) \cdot f_v^{(b)}(c)$ for all $c \in B_\mu$ and $v \in B_\rho$.
- Each worker prover \mathcal{P}_b defines a multilinear polynomial $h^{(b)}(v, c) := \chi_c(\alpha_v) \cdot \chi_b(\beta_v)$ for all $c \in B_\mu$ and $v \in B_\rho$.
- The provers and the verifier runs a distributed multivariate SumCheck protocol for $\{g^{(b)} \cdot h^{(b)}\}$ with target s .

Proposition 2. *The Distributed Batch Opening protocol previously presented is sound and accountable, if the distributed SumCheck and KZG PCS are accountable.*

Proof. The soundness of the protocol depends on the soundness of the distributed SumCheck, which we prove in Proposition 4. This means that $s = \sum_{b \in B_\xi, v \in B_\rho, c \in B_\mu} g^{(b)}(v, c) \cdot h^{(b)}(v, c)$, which is the same equality that the soundness of the Batch Opening protocol in HyperPlonk relies on. Since HyperPlonk is sound, so is ours.

Given that the protocol is sound, the coordinator \mathcal{P}_0 will be able to detect when a worker is malicious. As shown in Proposition 4, the SumCheck is accountable, and $h^{(b)}(v, c)$ can be computed efficiently by the coordinator; therefore, this protocol is also accountable. \square

4.2. Making Malicious Nodes Accountable

The distributed Poly-IOP and PCS schemes presented previously cannot enable the coordinator to find which node is malicious if the final proof is incorrect. In this part, we discuss how to make malicious nodes accountable.

Accountable distributed multivariate PermTest Poly-IOP. One challenge in constructing such systems for single circuits is that, in distributed multivariate PermTest, the values $z^{(b)}$ of each circuit segment are generated by each prover and cannot be verified by the coordinator. To deal with this challenge, we let the coordinator calculate $z^{(b)}$ after evaluating the circuit, and then each prover is required to open their polynomial commitment of $z^{(b)}$ at $(1, 1, \dots, 1, 0)$ to the coordinator during PermTest. More specifically, in PermTest, each prover \mathcal{P}_b needs to construct two polynomials $f_1^{(b)}$ and $f_2^{(b)}$ from the committed polynomial $f^{(b)}$. And $z^{(b)}$ is evaluated using $z^{(b)}(\mathbf{x}) = \prod_{v \in B_\mu} f_1^{(b)}(\mathbf{x})/f_2^{(b)}(\mathbf{x})$. If $f^{(b)} \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$, then the evaluation of $f^{(b)}$ over B_μ can be efficiently evaluated in $O(d \cdot 2^\mu)$ time. Once $f^{(b)}$ over B_μ is evaluated, $z^{(b)}$ can be evaluated in $O(2^\mu)$ time, given that $s, s_\mu^{(b)}, s_\xi^{(b)}$ has been preprocessed by the coordinator.

However, it would still be a challenge to the coordinator to evaluate and store $f^{(b)}$ over B_μ for all $\mathbf{b} \in B_\xi$, and we come up with a way to circumvent this problem. We observe that the only place where PermTest is called in the distributed SNARK protocol is during Wiring Constraint, where $F^{(b)}$, the polynomial interpolated with the transcript of each prover, is the $f^{(b)}$ in the PermTest. Recall that at the beginning of the distributed proof generation, the

coordinator and all provers have to evaluate the entire circuit in plaintext, and therefore \mathcal{P}_0 would have all evaluations of $F^{(b)}$ over B_μ for all $\mathbf{b} \in B_\xi$. Therefore, \mathcal{P}_0 can store $\{F^{(b)}(\mathbf{x}) : \mathbf{x} \in B_\mu, \mathbf{b} \in B_\xi\}$ during the circuit evaluation, and use them to calculate $z^{\mathbf{b}}$.

Once \mathcal{P}_0 has $z^{\mathbf{b}}$ locally computed, it is able to verify the $z^{(b)}(1, 1, \dots, 1, 0)$ computed by \mathcal{P}_b . To do so, prover \mathcal{P}_b will send the proof of the opening of $z^{(b)}$ at $(1, 1, \dots, 1, 0)$ to \mathcal{P}_0 , and \mathcal{P}_0 will verify the correctness of the multivariate polynomial opening. The extra work of \mathcal{P}_0 this step is $O(\mu \cdot 2^\xi)$ bilinear mapping. The extra work of each other prover is $O(\mu)$ field operation and group exponentiation. The extra amortized communication is $O(\mu)\mathbb{G}$.

We summarize the technique in the following modification of distributed multivariate PermTest protocol: We assume that \mathcal{P}_0 has $f^{(b)}(\mathbf{x}), s(\mathbf{x}), s_\mu^{(b)}(\mathbf{x}), s_\xi^{(b)}(\mathbf{x})$ locally ready for all $\mathbf{b} \in B_\xi, \mathbf{x} \in B_\mu$.

Once again we consider the multivariate polynomials $s, s_\mu^{(b)}, s_\xi^{(b)} \in \mathbb{F}_\mu^{\leq 1}[\mathbf{X}]$ where $s(\mathbf{x}) := [\mathbf{x}], s_\mu^{(b)}(\mathbf{x}) := [\sigma^{(b)}(\mathbf{x})]$ and $s_\xi^{(b)}(\mathbf{x}) := [\rho^{(b)}(\mathbf{x})]$. The modified protocol steps are highlighted in blue.

- 1) \mathcal{V} samples and sends to the coordinator $\gamma_\mu, \gamma_\xi, \delta \leftarrow \mathbb{F}$.
- 2) Let $f_1^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s(\mathbf{x}) + \gamma_\xi \cdot [\mathbf{b}] + \delta$ and $f_2^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s_\mu^{(b)}(\mathbf{x}) + \gamma_\xi \cdot s_\xi^{(b)}(\mathbf{x}) + \delta$. Each prover \mathcal{P}_b builds a multilinear polynomial $z^{(b)} \in \mathbb{F}_{\mu+1}^{\leq 1}[\mathbf{X}]$ such that for all $\mathbf{x} \in B_\mu$

$$\begin{cases} z^{(b)}(0, \mathbf{x}) = f_1^{(b)}(\mathbf{x})/f_2^{(b)}(\mathbf{x}) \\ z^{(b)}(1, \mathbf{x}) = z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1), \mathbf{x} \neq \mathbf{1} \\ z^{(b)}(1, \mathbf{1}) = 0 \end{cases}$$

Let $w_1^{(b)}(\mathbf{x}) := z^{(b)}(1, \mathbf{x}) - z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1)$ and $w_2^{(b)}(\mathbf{x}) := f_2^{(b)}(\mathbf{x}) \cdot z^{(b)}(0, \mathbf{x}) - f_1^{(b)}(\mathbf{x})$.

- 3) \mathcal{P}_0 locally evaluates

$$z^{(b)} := \prod_{\mathbf{x} \in B_\mu} f_1^{(b)}(\mathbf{x})/f_2^{(b)}(\mathbf{x})$$

for all $\mathbf{b} \in B_\xi$ in $O(|\mathcal{C}|)$ field operations. Then each prover \mathcal{P}_b sends $\text{com}_{z^{(b)}}$ to \mathcal{P}_0 , and opens $z^{(b)}(1, 1, \dots, 1, 0)$, with the target value $z^{(b)}$. If the opening fails, \mathcal{P}_b is malicious.

- 4) We have $\prod_{\mathbf{b} \in B_\mu} z^{(b)} = 1$. Then \mathcal{P}_0 interpolates a multilinear polynomial $z \in B_{\xi+1}^{\leq 1}$ such that

$$\begin{cases} z(0, \mathbf{y}) = z^{(\mathbf{y})} \\ z(1, \mathbf{y}) = z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1) \end{cases}$$

Let $w_3(\mathbf{y}) := z(1, \mathbf{y}) - z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1)$ and $w_4^{(b)}(\mathbf{x}) := z(0, \mathbf{b}) - z^{(b)}(\mathbf{x}, 0) \cdot \chi_{(1, 1, \dots, 1, 0)}(\mathbf{x}, 0)$.

- 5) The provers and \mathcal{V} run distributed multivariate ZeroTest on $\{w_1^{(b)}\}, \{w_2^{(b)}\}$ and $\{w_4^{(b)}\}$. \mathcal{P}_0 and \mathcal{V} run multivariate ZeroTest on w_3 .

Accountable distributed multivariate SumCheck and multilinear KZG PCS. Recall that in the distributed SumCheck and distributed multilinear KZG PCS protocol,

the coordinator need to receive intermediate results from each prover \mathcal{P}_b and compute the complete proof depending on these intermediate results. Specifically, the coordinator needs to process the SumCheck result of $f^{(b)}$ from each node in distributed multilinear SumCheck protocol, and needs to process the separate sub-proofs from each nodes during the polynomial opening in the distributed KZG PCS. To make the malicious nodes accountable, we note that the coordinator can verify the intermediate results submitted by the nodes.

In the distributed KZG PCS, the coordinator can detect malicious prover as follows:

- KeyGen, Commit(f, crs), Verify($\text{com}, \pi, \alpha, \beta, z, \text{crs}$): Same as the previous protocol.
- Open($f, \alpha, \beta, \text{crs}$):

- 1) Each prover \mathcal{P}_b computes $z^{(b)} := f^{(b)}(\alpha)$ and $f^{(b)}(\mathbf{x}) - f^{(b)}(\alpha) := \sum_{i \in [\mu]} q_i^{(b)}(\mathbf{x})(x_i - \alpha_i)$. Then it computes $\pi_i^{(b)} = g^{q_i^{(b)}(\tau_1, \dots, \tau_\mu) \cdot \chi_b(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with crs , and sends $\pi^{(b)}$ and $z^{(b)}$ to \mathcal{P}_0 .
- 2) \mathcal{P}_0 checks if $e(\text{com}_{f^{(b)}}, g) \cdot e(g^{-z^{(b)}}, V_b) = \prod_{i \in [\mu]} e(\pi_i^{(b)}, g^{\tau_i - \alpha_i})$. If this check fails, \mathcal{P}_b is dishonest.
- 3) \mathcal{P}_0 computes $z = f(\alpha, \beta) = \sum_{\mathbf{b} \in B_\xi} z^{(b)} \cdot \chi_b(\beta)$. \mathcal{P}_0 decomposes

$$f(\alpha, \mathbf{y}) - f(\alpha, \beta) = \sum_{j \in [\xi]} q_j(\mathbf{y})(y_j - \beta_j).$$

Then it computes $\pi_{\mu+j} = g^{q_j(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with V_b . \mathcal{P}_0 also computes $\pi_i = \prod_{\mathbf{b} \in B_\xi} \pi_i^{(b)}$ for all $i \in [\mu]$.

- 4) Then \mathcal{P}_0 sends $\pi := (\pi_1, \dots, \pi_{\mu+\xi})$ and z to \mathcal{V} .

In the distributed SumCheck protocol, the coordinator runs full verification for the intermediate results of each node. Recall that each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)} \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$, and that the provers want to show the verifier that $\sum_{\mathbf{b} \in B_\xi, \mathbf{c} \in B_\mu} f^{(b)}(\mathbf{c}) = v$. Now the coordinator has distributed target $v^{(b)}$ for each prover \mathcal{P}_b , and they satisfy $\sum_{\mathbf{b} \in B_\xi} v^{(b)} = v$. Here we assume that \mathcal{P}_0 has the commitment of $\text{com}_{f^{(b)}}$. The accountable distributed multilinear SumCheck protocol is described as follows:

- 1) For each $i \in [\mu]$:
 - a) Each \mathcal{P}_b computes and sends to \mathcal{P}_0 the polynomial $r_i^{(b)}(\mathbf{x}) := \sum_{\mathbf{w} \in B_{\mu-i}} f^{(b)}(\alpha_1, \dots, \alpha_{i-1}, x, \mathbf{w})$.
 - b) For each $\mathbf{b} \in B_\xi$, \mathcal{P}_0 checks if $v^{(b)} = r_i^{(b)}(0) + r_i^{(b)}(1)$. If the check fails, \mathcal{P}_b is malicious. Then \mathcal{P}_0 sets $v^{(b)} := r_i^{(b)}(\alpha_i)$.
- 2) \mathcal{P}_0 receives $\delta_b := f^{(b)}(\alpha)$ from each \mathcal{P}_b . \mathcal{P}_b opens $f^{(b)}$ at α , and \mathcal{P}_0 verifies the proof of the opening. If the verification fails, \mathcal{P}_b is malicious.
- 3) \mathcal{P}_0 has a multivariate polynomial $g(\mathbf{y}) := f(\alpha, \mathbf{y}) = \sum_{\mathbf{b} \in B_\xi} \delta_b \cdot \chi_b(\mathbf{y})$.
- 4) \mathcal{P}_0 and \mathcal{V} perform SumCheck on g with target value v . Note that in the final round, the verifier queries $f(\alpha, \beta)$.

Accountable distributed multivariate ZeroTest Poly-IOP. Suppose each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)}(\mathbf{x}) \in \mathbb{F}_{\mu}^{\leq d}[\mathbf{X}]$. The provers wants to show to the verifier that $f^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{b} \in B_{\xi}$ and $\mathbf{x} \in B_{\mu}$. The protocol is described as follows:

- 1) \mathcal{V} samples and sends to \mathcal{P}_0 two random vectors $\mathbf{r} \leftarrow \mathbb{F}^{\mu}$ and $\mathbf{r}_0 \leftarrow \mathbb{F}^{\xi}$.
- 2) \mathcal{P}_0 sends \mathbf{r} to each \mathcal{P}_b .
- 3) Each \mathcal{P}_b sets $\tilde{f}^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) \cdot \chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{b})$.
- 4) The provers and \mathcal{V} run accountable distributed multivariate SumCheck protocol on $\tilde{f}^{(b)}(\mathbf{x})$ with target value 0 for each prover.
- 5) The coordinator verifies the SumCheck protocol. If the verification fails, the prover constructs $\tilde{f}^{(b)}(\mathbf{x})$ for each $\mathbf{b} \in B_{\xi}$ and check if the computation of \mathcal{P}_b is correct.

We observe that the coordinator does not need to verify all the computations on the fly. Instead, our coordinator could just run the verifier's program after all worker nodes have computed all intermediate results. The soundness of the verification is based on the soundness of the verifier. If the verification fails, at least one of the worker provers is malicious, and the prover will re-run the program of each node and check all the communication to determine the malicious nodes. Following this observation, we can *defer* the check whether anyone is malicious, which is carried out by the coordinator, to the end of each accountable protocol, which is a technique we refer to as *optimistic check*.

There are two advantages of performing optimistic checks compared with performing checks in the middle of each protocol. On one hand, since the verifier check of each protocol is succinct, if no one is malicious, the protocol will have minimal overhead to the vanilla Cirrus protocol without accountability. On the other hand, if some workers are malicious since the coordinator keeps track of all the communication, it can still find out exactly who is malicious by running the checks based on the communication and therefore preserves the accountability property.

4.3. Load Balancing and Optimistic Accountability

By far, we have achieved linear prover time for each worker. However, we still have a coordinator time of $O(M \log T)$, T being the size of each sub-circuit, and M being the number of worker nodes. In this part, we discuss how to eliminate the $(\log T)$ term while maintaining the accountability property.

We note that the coordinator's work of summing up group or field elements in distributed SumCheck and distributed KZG can be distributed. However, naively distributing this step (e.g., having each node add one element) could introduce a larger round complexity. Instead, only a subset of nodes is required for computing. We demonstrate this idea with the following example. Suppose each node has A elements, and the coordinator would finally need to get the sum of all elements. We divide the M nodes into k groups and select a leader of each group. In the first round,

the leader in each group adds up all MA/k elements in its group. In the second round, the coordinator adds up k elements from the leaders. The cost of the leader of each group is $O(MA/k)$, and the cost of the coordinator is $O(k)$.

In our case, when choosing $k = \log(T)$ we end up with a $O(M)$ coordinator cost and the same worker cost as before for the leaders, as their previous cost dominates this extra computation.

In the following paragraphs, we introduce the modified protocols with load balancing and optimistic accountability. The modified protocol steps are highlighted in purple.

Optimistic accountability for distributed KZG PCS. For a multilinear polynomial $f(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{b} \in B_{\xi}} f^{(b)}(\mathbf{x}) \chi_{\mathbf{b}}(\mathbf{y})$, the PCS protocol is described as follows:

- KeyGen, Commit(f, crs), Verify($\text{com}, \pi, \alpha, \beta, z, \text{crs}$): Same as the previous protocol.
- Open($f, \alpha, \beta, \text{crs}$):
 - 1) Divide nodes into groups of size $\log T$. The coordinator randomly select one leader out of each group.
 - 2) Each prover \mathcal{P}_b computes $z^{(b)} := f^{(b)}(\alpha)$ and $f^{(b)}(\mathbf{x}) - f^{(b)}(\alpha) := \sum_{i \in [\mu]} q_i^{(b)}(\mathbf{x})(x_i - \alpha_i)$. Then it computes $\pi_i^{(b)} = g^{q_i^{(b)}(\tau_1, \dots, \tau_{\mu}) \cdot \chi_b(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ using the crs and $s^{(b)}$ the signature of $(\pi^{(b)}, z^{(b)})$. \mathcal{P}_b sends $(\pi^{(b)}, z^{(b)}, s^{(b)})$ to both leader of its group and \mathcal{P}_0 .
 - 3) The leader of each group sums up the $\pi^{(b)}$ it receives, and sends the result together with all the communication to the coordinator. In the case where the leader does not receive $\pi^{(b)}$ from a specific node, it communicates with the coordinator for result.
 - 4) \mathcal{P}_0 computes $z = f(\alpha, \beta) = \sum_{\mathbf{b} \in B_{\xi}} z^{(b)} \cdot \chi_b(\beta)$. \mathcal{P}_0 decomposes

$$f(\alpha, \mathbf{y}) - f(\alpha, \beta) = \sum_{j \in [\xi]} q_j(\mathbf{y})(y_j - \beta_j).$$

Then it computes $\pi_{\mu+j} = g^{q_j(\tau_{\mu+1}, \dots, \tau_{\mu+\xi})}$ with V_b .

- 5) \mathcal{P}_0 sets $\pi := (\pi_1, \dots, \pi_{\mu+\xi})$. \mathcal{P}_0 checks if Verify($\text{com}, \pi, \alpha, \beta, z, \text{crs}$) passes. If the check fails, \mathcal{P}_0 aborts and checks which provers are malicious. In this case, signatures are used to ensure the integrity of the messages when the leader works as a relay. Otherwise, \mathcal{P}_0 sends π and z to \mathcal{V} .

Proposition 3. *The accountable Distributed KZG PCS previously presented is sound and accountable.*

Proof. To prove soundness by contradiction, assume that a group of malicious provers can produce a proof $\pi' \neq \pi$ such that Verify($\text{com}, \pi', \alpha, \beta, z, \text{crs}$) passes. This would mean that the soundness of the classic KZG protocol is broken, because in the classic protocol, a single prover has all the information that the worker provers have. This contradiction confirms that the optimistic protocol is sound.

Regarding accountability, if the protocol fails, we can identify the malicious provers by verifying the following equality: $e(\text{com}/g^{z^{(b)}}, g) = \prod_{i \in [\mu]} e(\pi_i^{(b)}, g^{\tau_i - \alpha_i})$. Since this is the standard commitment verification for the multilinear polynomial generated by the sub-circuit that \mathcal{P}_b is

responsible for, we can assume with high probability that it is sound, and that the coordinator will be able to identify any incorrect proofs between the ones provided. If none of the verifications fail, it implies that the leader either did not correctly sum the $\pi^{(b)}$ or received $\pi'^{(b)} \neq \pi^{(b)}$. In the latter case, the leader will send the signed $(\pi^{(b)}, z^{(b)})$ to demonstrate that prover b was malicious, which can be identified using the same method as the coordinator. If the leader cannot provide such proof, then the master will assume that the malicious node is the leader itself. \square

Optimistic accountability for distributed SumCheck.

Assume we have a secure signature scheme σ , then using the previously presented accountable and distributed KZG PCS we can build an optimistic accountable distributed SumCheck as follows:

- 1) For each $i \in [\mu]$:
 - a) Let $r_i := 0$ for \mathcal{P}_0 .
 - b) For $\mathbf{b} \in B_\xi$, \mathcal{P}_b computes $r_i^{(b)}(x) := \sum_{\mathbf{w} \in B_{\mu-i}} f^{(b)}(\alpha_1, \dots, \alpha_{i-1}, x, \mathbf{w})$.
 \mathcal{P}_b sends $r_i^{(b)}$ and its signature $s^{(b)}$ to both the leader of its group and \mathcal{P}_0 .
 - c) The leader of each group sums up the $r_i^{(b)}$ it receives, and sends the result together with all communication to the coordinator. In the case where the leader does not receive $r_i^{(b)}$ from a specific worker, it communicates with the coordinator for the result.
 - d) \mathcal{P}_0 adds up the sum of $r_i^{(b)}$ from the leaders and gets r_i . \mathcal{P}_0 then checks if $v = r_i(0) + r_i(1)$. If such check does not pass, \mathcal{P}_0 aborts and checks which provers are malicious. \mathcal{P}_0 sends the oracle of r_i to \mathcal{V} .
 - e) \mathcal{V} checks if $v = r_i(0) + r_i(1)$, and samples and sends a random $\alpha_i \leftarrow \mathbb{F}$ to \mathcal{P}_0 . Then \mathcal{V} sets $v := r_i(\alpha_i)$.
 \mathcal{P}_0 sends α_i to each \mathcal{P}_b .
- 2) \mathcal{P}_0 receives $v^{(b)} := f^{(b)}(\alpha)$ from each \mathcal{P}_b . \mathcal{P}_0 has a multivariate polynomial $g(\mathbf{y}) := f(\alpha, \mathbf{y}) = \sum_{\mathbf{b} \in B_\xi} v^{(b)} \cdot \chi_{\mathbf{b}}(\mathbf{y})$.
- 3) \mathcal{P}_0 and \mathcal{V} perform SumCheck on g with target value v . In the final round, \mathcal{P}_0 and the worker provers open $f(\alpha, \beta)$. If the opening fails, \mathcal{P}_0 aborts and checks which provers are malicious. Otherwise, \mathcal{V} and \mathcal{P}_0 open $f(\alpha, \beta)$.

Proposition 4. *The accountable Distributed SumCheck protocol previously presented is sound and accountable.*

Proof. We will again prove soundness by reducing it to the classic SumCheck protocol. From the verifier's perspective, in step 1 of the protocol, they receive the univariate polynomials

$$\begin{aligned} r_i(x) &:= \sum_{\mathbf{b} \in B_\xi} \sum_{\mathbf{w} \in B_{\mu-i}} f^{(b)}(\alpha_1, \dots, \alpha_{i-1}, x, \mathbf{w}) \\ &= \sum_{\mathbf{w} \in B_{\mu-i}} \sum_{\mathbf{b} \in B_\xi} f(\alpha_1, \dots, \alpha_{i-1}, x, \mathbf{w}, \mathbf{b}). \end{aligned}$$

This represents the first μ steps of a classic SumCheck protocol for the polynomial f . In step 3, they perform the last ε steps of the SumCheck protocol by computing

the values $r_{i+\mu}(x) := \sum_{\mathbf{w} \in B_{\xi-i}} f(\alpha, x, \mathbf{w})$. Since, from the verifier's perspective, this protocol mirrors the classic SumCheck protocol, we can ensure its soundness.

Since the protocol is sound, the coordinator can detect when a malicious worker or leader has submitted incorrect proof. Accountability is ensured in these scenarios through the following mechanism:

- If a worker submits an incorrect proof to both the leader and the coordinator, the coordinator can identify the worker by recomputing all $r_i^{(b)}$ values and cross-checking them with the signed values provided during the protocol.
- If the leader submits an incorrect addition, the coordinator can detect the leader's error by recomputing the additions based on the values supplied by the workers.
- If the previous verification fails because a worker submitted inconsistent signed values to the leader and the coordinator, the leader can show that they received a different value than the one available to the coordinator by presenting the worker's signed value as evidence.
- Lastly, if both the worker and the leader engage in dishonest actions, they will be flagged as malicious. In this case, the leader will be able to show a signed value different from what the worker provided to the coordinator but will be unable to present a signed value that matches the leader's stated total.

Since all the possible cases are covered, the coordinator will always be able to identify the malicious node, ensuring accountability in the protocol. \square

Optimistic accountability for distributed ZeroTest.

Suppose each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)}(\mathbf{x}) \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$. The provers want to show to the verifier that $f^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{b} \in B_\xi$ and $\mathbf{x} \in B_\mu$. The protocol is described as follows:

- 1) \mathcal{V} samples and sends to \mathcal{P}_0 two random vectors $\mathbf{r} \leftarrow \mathbb{F}^\mu$ and $\mathbf{r}_0 \leftarrow \mathbb{F}^\xi$.
- 2) \mathcal{P}_0 sends \mathbf{r} to each \mathcal{P}_b .
- 3) Each \mathcal{P}_b sets $\tilde{f}^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) \cdot \chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{b})$.
- 4) The provers and \mathcal{V} run accountable distributed multivariate SumCheck protocol on $\tilde{f}^{(b)}(\mathbf{x})$ with target value 0 for each prover.
- 5) The coordinator verifies the SumCheck protocol. If the verification fails, \mathcal{P}_0 aborts and constructs $\tilde{f}^{(b)}(\mathbf{x})$ for each $\mathbf{b} \in B_\xi$ to check if the computation of \mathcal{P}_b is correct.

Proposition 5. *The accountable Distributed ZeroTest protocol previously presented is sound and accountable.*

Proof. Note that this protocol does not change the communication with the verifier if all parties are honest. As discussed in Proposition 1, the protocol is sound.

By the accountability of distributed multivariate SumCheck protocol (Proposition 4), this protocol is also accountable. \square

Optimistic accountability for distributed PermTest. Let $\sigma^{(b)} : B_\mu \rightarrow B_\mu$ and $\rho^{(b)} : B_\mu \rightarrow B_\xi$ be two mappings such that $\{(\sigma^{(b)}(\mathbf{c}), \rho^{(b)}(\mathbf{c})) : \mathbf{c} \in B_\mu, \mathbf{b} \in B_\xi\} = B_\mu \times B_\xi$.

Each prover \mathcal{P}_b has a multivariate polynomial $f^{(b)} \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$. The provers want to show to the verifier that $f^{(b)}(\mathbf{c}) = f^{(\rho^{(b)}(\mathbf{c}))}(\sigma^{(b)}(\mathbf{c}))$ for all $\mathbf{b} \in B_\xi$ and $\mathbf{c} \in B_\mu$. Note that $\rho^{(b)}(\mathbf{c})$ is not identically \mathbf{b} for all $\mathbf{c} \in B_\mu$.

- 1) Define the multivariate polynomials $s, s_\mu^{(b)}, s_\xi^{(b)} \in \mathbb{F}_\mu^{\leq 1}[\mathbf{X}]$ where $s(\mathbf{x}) := [\mathbf{x}]$, $s_\mu^{(b)}(\mathbf{x}) := [\sigma^{(b)}(\mathbf{x})]$ and $s_\xi^{(b)}(\mathbf{x}) := [\rho^{(b)}(\mathbf{x})]$.
- 2) \mathcal{V} samples and sends to the coordinator $\gamma_\mu, \gamma_\xi, \delta \leftarrow \mathbb{F}$.
- 3) Let $f_1^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s(\mathbf{x}) + \gamma_\xi \cdot [\mathbf{b}] + \delta$ and $f_2^{(b)}(\mathbf{x}) := f^{(b)}(\mathbf{x}) + \gamma_\mu \cdot s_\mu^{(b)}(\mathbf{x}) + \gamma_\xi \cdot s_\xi^{(b)}(\mathbf{x}) + \delta$. Each prover \mathcal{P}_b builds a multilinear polynomial $z^{(b)} \in \mathbb{F}_{\mu+1}^{\leq 1}[\mathbf{X}]$ such that for all $\mathbf{x} \in B_\mu$

$$\begin{cases} z^{(b)}(0, \mathbf{x}) = f_1^{(b)}(\mathbf{x})/f_2^{(b)}(\mathbf{x}) \\ z^{(b)}(1, \mathbf{x}) = z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1), \mathbf{x} \neq \mathbf{1} \\ z^{(b)}(1, \mathbf{1}) = 0 \end{cases}$$

Let $w_1^{(b)}(\mathbf{x}) := z^{(b)}(1, \mathbf{x}) - z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1)$ and $w_2^{(b)}(\mathbf{x}) := f_2^{(b)}(\mathbf{x}) \cdot z^{(b)}(0, \mathbf{x}) - f_1^{(b)}(\mathbf{x})$. Then each worker \mathcal{P}_b sends $z^{(b)} := z^{(b)}(1, 1, \dots, 1, 0)$ to \mathcal{P}_0 .

- 4) \mathcal{P}_0 checks if $\prod_{b \in B_\xi} z^{(b)} = 1$. If not, \mathcal{P}_0 aborts and checks which provers are malicious. Otherwise, \mathcal{P}_0 continues as follows. Then \mathcal{P}_0 interpolates a multilinear polynomial $z \in B_{\xi+1}^{\leq 1}$ such that

$$\begin{cases} z(0, \mathbf{y}) = z^{(\mathbf{y})} \\ z(1, \mathbf{y}) = z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1) \end{cases}$$

Let $w_3(\mathbf{y}) := z(1, \mathbf{y}) - z(\mathbf{y}, 0) \cdot z(\mathbf{y}, 1)$ and $w_4^{(b)}(\mathbf{x}) := z(0, \mathbf{b}) - z^{(b)}(\mathbf{x}, 0) \cdot \chi_{(1,1,\dots,1,0)}(\mathbf{x}, 0)$.

- 5) \mathcal{P}_0 runs **optimistically accountable distributed multivariate ZeroTest** on $\{w_1^{(b)}\}$, $\{w_2^{(b)}\}$ and $\{w_4^{(b)}\}$ with the worker provers. If the test fails, \mathcal{P}_0 aborts and checks which provers are malicious. Otherwise, the \mathcal{P}_0 and \mathcal{V} run **distributed multivariate ZeroTest** on $\{w_1^{(b)}\}$, $\{w_2^{(b)}\}$ and $\{w_4^{(b)}\}$. \mathcal{P}_0 and \mathcal{V} run **multivariate ZeroTest** on w_3 .

Proposition 6. *The accountable Distributed PermTest protocol previously presented is sound and accountable.*

Proof. The test W_1 ensures with high probability (w.h.p.) that

$$w_1^{(b)}(\mathbf{x}) = z^{(b)}(1, \mathbf{x}) - z^{(b)}(\mathbf{x}, 0) \cdot z^{(b)}(\mathbf{x}, 1) = 0, \forall \mathbf{x} \in B_\mu.$$

Similarly, W_2 ensures w.h.p. that

$$w_2^{(b)}(\mathbf{x}) = f_2^{(b)}(\mathbf{x}) \cdot z^{(b)}(0, \mathbf{x}) - f_1^{(b)}(\mathbf{x}) = 0, \forall \mathbf{x} \in B_\mu.$$

As long as $z^{(b)}$ is a multilinear polynomial and $z^{(b)}(\mathbf{1}) = 0$, these two constraints uniquely determine $z^{(b)}$. However, we observe that we do not need to check that $z^{(b)}(\mathbf{1}) = 0$ because we are interested in the value $z^{(b)}(1, 1, \dots, 1, 0)$

for \mathcal{P}_0 . Regardless of the value of $z^{(b)}(\mathbf{1})$, the value $z^{(b)}(1, 1, \dots, 1, 0)$ remains unaffected. Indeed, we have

$$\begin{aligned} z^{(b)}(1, 1, \dots, 1, 0) &= z^{(b)}(1, \dots, 1, 0, 0) \cdot z^{(b)}(1, \dots, 1, 0, 1) \\ &= \dots = \prod_{\mathbf{x} \in B_\mu} z^{(b)}(0, \mathbf{x}) = \prod_{\mathbf{x} \in B_\mu} \frac{f_1^{(b)}(\mathbf{x})}{f_2^{(b)}(\mathbf{x})}. \end{aligned}$$

which does not depend on the value assigned to $z^{(b)}(\mathbf{1})$. Therefore, we only need to ensure that $z^{(b)}$ is a multilinear polynomial, which is guaranteed by construction via the Polynomial Commitment Scheme (PCS). This means that as long as the W_1 and W_2 tests pass, w.h.p. the committed polynomial $z^{(b)}$ is correctly computed. Next, we need to ensure that the value $z^{(b)}$ provided by Prover $_b$ is indeed $z^{(b)}(1, 1, \dots, 1, 0)$. This is achieved by performing the W_4 ZeroTest. The soundness comes from the fact that we are using the same commitment as in the previous zero tests, which ensures that the polynomial $z^{(b)}$ was correctly computed. This demonstrates that if any of the provers Prover $_b$ is malicious, then w.h.p. at least one of the tests W_1 , W_2 , or W_4 will fail, and thus the previous optimistic protocol is sound. Due to the accountability property of ZeroTest and the fact that the coordinator has all the values needed to reconstruct all the $z^{(b)}$ values from scratch and compare them to the provided values, we can ensure that this protocol is accountable. \square

The Cirrus distributed SNARK. Cirrus, the accountable and efficiently computable distributed SNARK, is structured as follows:

- **Input Constraint:** $F^{(b)}(1, \dots, 1, \mathbf{x}) - I^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\nu$ and $\mathbf{b} \in B_\xi$ with accountable distributed multilinear ZeroTest.
- **Output Constraint:** \mathcal{V} queries $F^{(1, \dots, 1)}(1, 0, \langle N^{(b)} - 1 \rangle_\mu)$ and checks if it is zero.
- **Gate Constraint:** Define a multivariate polynomial

$$G^{(b)}(\mathbf{x}) := S_{\text{add}}^{(b)}(\mathbf{x})(F^{(b)}(0, 0, \mathbf{x}) + F^{(b)}(0, 1, \mathbf{x})) + S_{\text{mult}}^{(b)}(\mathbf{x})(F^{(b)}(0, 0, \mathbf{x}) \cdot F^{(b)}(0, 1, \mathbf{x})) - F^{(b)}(1, 0, \mathbf{x}).$$

$G^{(b)}(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\mu$ and $\mathbf{b} \in B_\xi$ with accountable distributed multivariate ZeroTest.

- **Wiring Constraint:** Check if $F^{(b)}(\mathbf{x}) = F^{(\rho^{(b)}(\mathbf{x}))}(\sigma^{(b)}(\mathbf{x}))$ for all $\mathbf{x} \in B_\mu$ and $\mathbf{b} \in B_\xi$ with accountable distributed multivariate PermTest.

5. Implementation and Evaluation

5.1. Implementation Details

We implement Cirrus with optimistic accountability and without load balancing. Our implementation is based on HyperPlonk [8] and adds 5,000+ lines of Rust.

Cirrus is developer-friendly. An advantage of Cirrus is that it can automatically distribute the workload without requiring the developers to specify how to partite the circuit.

In Hekaton, however, developers must partition the circuit into sub-circuits and carefully manage shared wires.

Universal v.s. per-circuit trusted setup. Inherited from HyperPlonk, the trusted setup in Cirrus is *universal*, which is a significant advantage in practice compared to schemes that require per-circuit trust setup (such as Hekaton) because a trustworthy setup ceremony is expensive to organize [22].

5.2. Evaluation Results

To assess the practicality of running Cirrus in a decentralized environment (e.g., by individual volunteers), we decide to benchmark Cirrus on hardware comparable to commodity PCs. In contrast, related works such as Hekaton and Pianist are evaluated on much more powerful HPC servers (e.g., with 128 cores and 512 GB memory). Our cluster consists of 32 AWS `t3.2xlarge` machines in North Virginia, each with 8 vCPUs and 32 GB of memory; the average network latency across nodes in our setup was measured at 306 microseconds.

We found that the best setting for Cirrus is one worker per core (i.e., running 8 workers per machine), and we use this configuration across all experiments.

End-to-end proof generation time. We first evaluate the end-to-end proof generation time of Cirrus. As a baseline, we run multi-threaded HyperPlonk on the same worker machine to generate proofs for random circuits of varying sizes. Then, we run Cirrus with up to 32 workers (each with 8 cores) for random circuits generated in the same way. Figure 1 shows the end-to-end proof generation time of Cirrus, as a function of total circuit size and the number of workers. The line for HyperPlonk stops at 2^{22} gates when it runs out of memory (32 GB). In comparison, Cirrus can support larger circuits with more machines. We stopped at 2^{25} with 8 AWS machines since the horizontal scalability is clear.

Coordinator’s computation time. A feature of Cirrus is that the coordinator’s computation is lightweight thanks to load balancing. We measure the coordinator’s computation time in Fig. 2. Without load balancing, the computation time of the coordinator increases as the sub-circuit size increases. With our load balancing technique, the computation time of the coordinator is independent of the size of the sub-circuits. Overall, the coordinator’s computation is lightweight and well under 1 second for less than 1,000 workers.

Memory usage. We record how each worker’s memory usage varies with the worker’s sub-circuit size in Table 2a. In Cirrus, worker memory consumption is similar to that of Pianist and Hekaton for sub-circuits of the same sizes.

The memory usage of the coordinator is reported in Table 2b. We observe that the memory usage of the coordinator only depends on the size of the full circuit in our experimental setting.

Comparison with Hekaton. To provide context for Cirrus’s performance, we compared it with Hekaton, which reportedly outperforms Pianist by $3\times$, making it the fastest

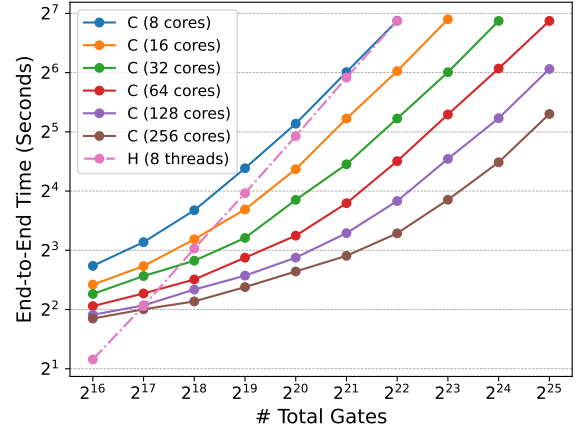


Figure 1: End-to-end proof generation time of Cirrus when all workers are honest with different total cores and circuit sizes. C denotes Cirrus; H denotes HyperPlonk.

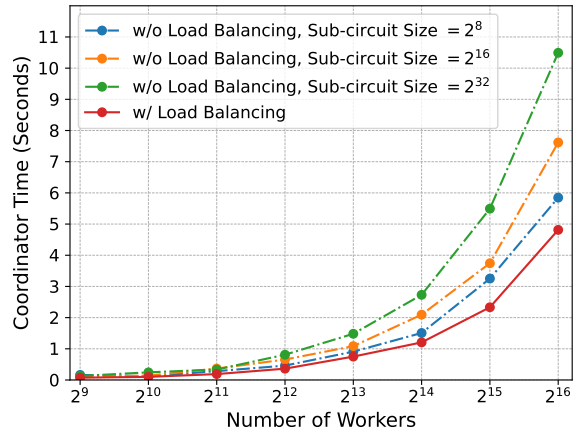


Figure 2: Computation time of the coordinator with varying number of workers and sub-circuit sizes.

distributed SNARK scheme to the best of our knowledge. We obtained the Hekaton source code from the authors and evaluated it on our worker machines.

The comparison to Hekaton is not straightforward because Hekaton works with RICS circuits instead of PLONK circuits. A given function may translate to drastically different numbers of RICS constraints and PLONK gates. We use α to denote the ratio of the number of PLONK gates to the number of RICS constraints to express a given program. For certain tasks, such as MinRoot [36], Pedersen hash function [12] and MiMC hash function [37], the number of PLONK gates can be 75% less than the number of RICS constraints [8], [38] (i.e., $\alpha = 0.25$); HyperPlonk reported that Zeke’s recursive circuit and a ZK-Rollup block of 50 private transactions have α as small as 2^{-5} . For other tasks, including SHA-256 hash, the number of PLONK gates can

TABLE 2: Memory usage of workers and the coordinator.

Sub-circuit Size	Memory	Full Circuit Size	Memory
2^{16}	383 MB	2^{19}	200 MB
2^{17}	765 MB	2^{20}	396 MB
2^{18}	1.4 GB	2^{21}	799 MB
2^{19}	2.8 GB	2^{22}	1.5 GB
2^{20}	5.6 GB	2^{23}	3.0 GB
2^{21}	11.3 GB	2^{24}	6.0 GB
2^{22}	22.6 GB	2^{25}	12.1 GB

(a) Memory usage of each worker with varying sub-circuit sizes.

(b) Memory usage of the coordinator with varying full circuit sizes.

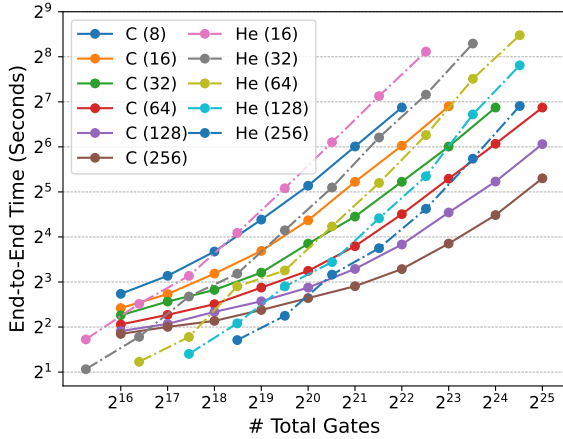


Figure 3: Cirrus and Hekaton Comparison with $\alpha = 0.25$. C denotes Cirrus, and He denotes Hekaton. The numbers in the parentheses denote the total number of working cores.

be $2.25\times$ more than the number of RICS constraints [39].

To perform a fair comparison with Hekaton, we use hash functions as the workload and report the results with $\alpha = 0.25$ (corresponding to Pedersen hash function) and $\alpha = 2.25$ (corresponding to SHA-256). We report our performance compared to Hekaton in Figs. 3 and 5. Cirrus outperforms Hekaton when proving Pedersen hashing tasks, yet underperforms Hekaton when proving SHA-256 hash tasks. For reference, we also report our performance compared to Hekaton for $\alpha = 1$ (i.e., tasks with the same number of PLONK gates and RICS constraints) in Fig. 4. We stress that Hekaton requires a *per-circuit trusted setup* while Cirrus is universal.

6. Conclusions and Future Directions

We have introduced Cirrus, the first *accountable* distributed SNARK generation scheme with linear-time worker and coordinator computation time, minimal communication overhead, and supports a universal trusted setup. By accountability, the coordinator can identify any malicious prover, making Cirrus suitable for deployment in decentralized settings, e.g., in prover markets, where workers

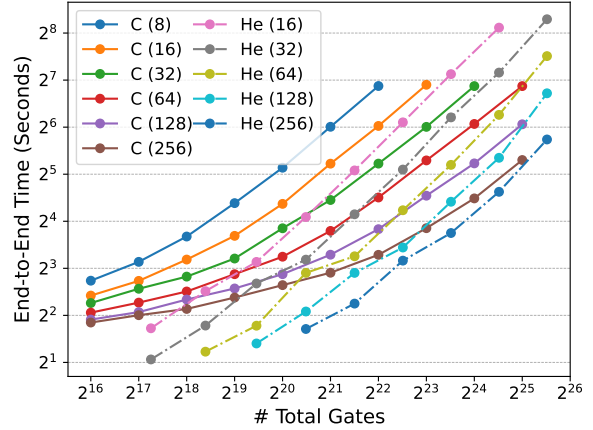


Figure 4: Cirrus and Hekaton Comparison with $\alpha = 1$. C denotes Cirrus, and He denotes Hekaton. The numbers in the parentheses denote the total number of working cores.

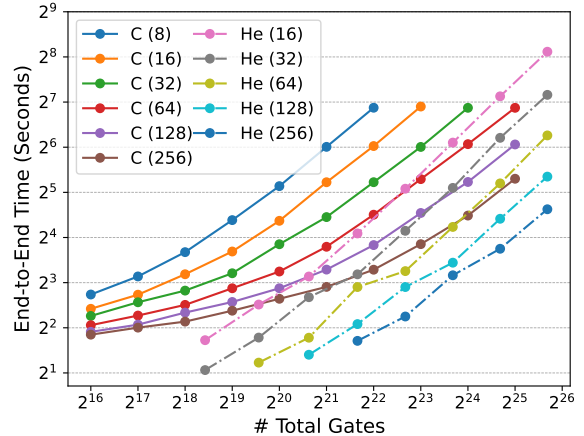


Figure 5: Cirrus and Hekaton Comparison with $\alpha = 2.25$. C denotes Cirrus, and He denotes Hekaton. The numbers in the parentheses denote the total number of working cores.

cannot be fully trusted. We formally define accountability in distributed proof generation schemes and prove that Cirrus satisfies this definition. According to our experiments, Cirrus is horizontally scalable and is concretely faster than the state-of-the-art for representative workloads.

One future direction is to further improve the communication round complexity, currently logarithmic in the size of each sub-circuit due to the distributed SumCheck. It is of both theoretical and practical interest to design an accountable distributed SNARK with a constant number of communication rounds while preserving the efficiency of Cirrus. Additionally, future work could focus on minimizing the coordinator’s overhead in holding malicious provers accountable. Currently, identifying malicious provers during the PermTest requires linear time relative to the size of

the entire circuit. Enhancing this aspect of the protocol would facilitate its deployment in scenarios with a large number of adversaries, making it more practical for broader applications.

Acknowledgments

This project was supported in part by the Ethereum Foundation ZK Grant. The authors would like to thank Josh Beal for valuable discussions during the early stages of the project, and Yiğit Kılıçoğlu for contributions to the implementation.

References

- [1] J. Groth, “On the size of pairing-based non-interactive arguments,” in *Advances in Cryptology—EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II* 35. Springer, 2016, pp. 305–326.
- [2] S. Ames, C. Hazay, Y. Ishai, and M. Venkatasubramanian, “Ligero: Lightweight sublinear arguments without a trusted setup,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2087–2104.
- [3] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge,” *Cryptology ePrint Archive*, 2019.
- [4] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, “Marlin: Preprocessing zkSNARKs with universal and updatable CRS,” in *Advances in Cryptology—EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I* 39. Springer, 2020, pp. 738–768.
- [5] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again,” in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. IITCS ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 326–349. [Online]. Available: <https://doi.org/10.1145/2090236.2090263>
- [6] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, “Libra: Succinct zero-knowledge proofs with optimal prover computation,” in *Advances in Cryptology—CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III* 39. Springer, 2019, pp. 733–764.
- [7] J. Zhang, T. Xie, Y. Zhang, and D. Song, “Transparent polynomial delegation and its applications to zero knowledge proof,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 859–876.
- [8] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, “Hyperplonk: Plonk with linear-time prover and high-degree custom gates,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 499–530.
- [9] S. Setty, “Spartan: Efficient and general-purpose zkSNARKs without trusted setup,” in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.
- [10] A. Golovnev, J. Lee, S. T. Setty, J. Thaler, and R. S. Wahby, “Brakedown: Linear-time and post-quantum snarks for r1cs,” *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1043, 2021.
- [11] T. Xie, Y. Zhang, and D. Song, “Orion: Zero knowledge proof with linear prover time,” in *Annual International Cryptology Conference*. Springer, 2022, pp. 299–328.
- [12] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox *et al.*, “Zcash protocol specification,” *GitHub: San Francisco, CA, USA*, vol. 4, no. 220, p. 32, 2016.
- [13] Ethereum Foundation, “zk-rollups: Scaling solutions for ethereum,” <https://ethereum.org/en/developers/docs/scaling/zk-rollups/>, Jul. 2024.
- [14] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, “zkbridge: Trustless cross-chain bridges made practical,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3003–3017.
- [15] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, “{DIZK}: A distributed zero knowledge proof system,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 675–692.
- [16] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang, “Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs,” *Cryptology ePrint Archive*, 2023.
- [17] M. Rosenberg, T. Mopuri, H. Hafezi, I. Miers, and P. Mishra, “Hekaton: Horizontally-scalable zkSNARKs via proof aggregation,” *Cryptology ePrint Archive*, 2024.
- [18] C. Li, Y. Li, P. Zhu, W. Qu, and J. Zhang, “Hyperpianist: Pianist with linear-time prover via fully distributed hyperplonk,” *Cryptology ePrint Archive*, 2024.
- [19] W. Wang, L. Zhou, A. Yaish, F. Zhang, B. Fisch, and B. Livshits, “Mechanism design for zk-rollup prover markets,” *arXiv preprint arXiv:2404.06495*, 2024.
- [20] “Gevulot docs,” <https://docs.gevulot.com/gevulot-docs/>, accessed: 2024-04-06.
- [21] “Ferham docs,” <https://docs.fermah.xyz/>, accessed: 2024-11-06.
- [22] S. Walters, “What is the zcash ceremony? the complete beginners guide,” accessed: 2024-11-14. [Online]. Available: <https://coinbureau.com/education/zbtc-ceremony/>
- [23] A. Kosba, D. Papadopoulos, C. Papamanthou, and D. Song, “{MIRAGE}: Succinct arguments for randomized algorithms with applications to universal {zk-SNARKs},” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2129–2146.
- [24] E. Systems, “Hyperplonk library,” accessed: 2024-11-13. [Online]. Available: <https://github.com/EspressoSystems/hyperplonk>
- [25] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, “Algebraic methods for interactive proof systems,” *Journal of the ACM (JACM)*, vol. 39, no. 4, pp. 859–868, 1992.
- [26] W. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh, “Mangrove: A scalable framework for folding-based SNARKs,” 2024, <https://eprint.iacr.org/2024/416>. [Online]. Available: <https://eprint.iacr.org/2024/416>
- [27] X. Liu, S. Gao, T. Zheng, Y. Guo, and B. Xiao, “SnarkFold: Efficient proof aggregation from incrementally verifiable computation and applications,” *Cryptology ePrint Archive, Paper 2023/1946*, 2023. [Online]. Available: <https://eprint.iacr.org/2023/1946>
- [28] M. Ambrona, M. Beunardeau, A.-L. Schmitt, and R. R. Toledo, “aPlonK : Aggregated PlonK from multi-polynomial commitment schemes,” *Cryptology ePrint Archive, Paper 2022/1352*, 2022. [Online]. Available: <https://eprint.iacr.org/2022/1352>
- [29] N. Gailly, M. Maller, and A. Nitulescu, “Snarkpack: Practical snark aggregation,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 203–229.
- [30] A. Ozdemir and D. Boneh, “Experimenting with collaborative {zk-SNARKs}: {Zero-Knowledge} proofs for distributed secrets,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4291–4308.
- [31] S. Garg, A. Goel, A. Jain, G.-V. Policharla, and S. Sekar, “{zkSaaS}: {Zero-Knowledge} {SNARKs} as a service,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4427–4444.

- [32] A. Chiesa, R. Lehmkuhl, P. Mishra, and Y. Zhang, “Eos: Efficient private delegation of {zkSNARK} provers,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6453–6469.
- [33] X. Liu, Z. Zhou, Y. Wang, B. Zhang, and X. Yang, “Scalable collaborative zk-snark: Fully distributed proof generation and malicious security,” *Cryptology ePrint Archive*, Paper 2024/143, 2024, <https://eprint.iacr.org/2024/143>. [Online]. Available: <https://eprint.iacr.org/2024/143>
- [34] X. Liu, Z. Zhou, Y. Wang, J. He, B. Zhang, X. Yang, and J. Zhang, “Scalable collaborative zk-SNARK and its application to efficient proof outsourcing,” *Cryptology ePrint Archive*, Paper 2024/940, 2024, <https://eprint.iacr.org/2024/940>. [Online]. Available: <https://eprint.iacr.org/2024/940>
- [35] C. Papamanthou, E. Shi, and R. Tamassia, “Signatures of correct computation,” in *Theory of Cryptography Conference*. Springer, 2013, pp. 222–242.
- [36] D. Khovratovich, M. Maller, and P. R. Tiwari, “Minroot: Candidate sequential function for ethereum vdf,” *Cryptology ePrint Archive*, 2022.
- [37] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, “Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 191–219.
- [38] Aztec, “Plonk benchmarks ii: Around 5x faster than groth16 on pedersen hashes,” <https://aztec.network/blog/plonk-benchmarks-ii-5x-faster-than-groth16-on-pedersen-hashes>, 2020.
- [39] M. Rezaei, “Plonk vs groth16,” <https://medium.com/@mehialiabadi/plonk-vs-groth16-50254c157196>, 2023.
- [40] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.
- [41] J. Thaler, “Time-optimal interactive proofs for circuit evaluation,” in *Annual Cryptology Conference*. Springer, 2013, pp. 71–89.
- [42] I. Corporation, “Intel® software guard extensions programming reference,” accessed: 2024-08-27. [Online]. Available: <https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf>
- [43] J.-B. Truong, W. Gallagher, T. Guo, and R. J. Walls, “Memory-efficient deep learning inference in trusted execution environments,” in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 161–167.

Appendix

1. Definitions

Succinct non-interactive argument of knowledge. We recall the definitions of SNARKs.

Definition 2 (Interactive Argument of Knowledge). *A tuple of algorithms $(\text{Setup}, \mathcal{P}, \mathcal{V})$ is an interactive argument of knowledge for relation \mathcal{R} between a prover \mathcal{P} and a verifier \mathcal{V} if it has the following completeness and knowledge soundness properties.*

- **Completeness:** For all $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$

$$\Pr[\langle \mathcal{P}(\mathbb{w}), \mathcal{V}(\mathbb{x}, \text{pp}) \rangle = 1 \mid \text{pp} \leftarrow \text{Setup}(1^\lambda, \mathcal{R})] = 1.$$

- **Knowledge Soundness and Soundness:** An interactive protocol is knowledge-sound if for any PPT adversary

$(\mathcal{A}_1, \mathcal{A}_2)$, there exists a PPT extractor algorithm \mathcal{E} with oracle access to $\mathcal{A}_1, \mathcal{A}_2$ such that this probability is $\text{negl}(\lambda)$.

$$\Pr \left[\langle \mathcal{A}_2(\mathbb{w}), \mathcal{V}(\mathbb{x}, \text{pp}) \rangle = 1 \wedge (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, \mathcal{R}) \\ \mathbb{x} \leftarrow \mathcal{A}_1(\text{pp}) \\ \mathbb{w} \leftarrow \mathcal{E}(\text{pp}, \mathbb{x}, \mathcal{R}) \end{array} \right]$$

If the statement holds when the constraint on the extractor is removed, the interactive protocol is considered sound, and the protocol is considered an interactive argument.

- **Succinctness:** An interactive argument of knowledge is considered succinct if the communication between the prover and the verifier and the running time of the verifier are both $O(\lambda, |\mathbb{x}|, \log |\mathbb{w}|)$.
- **Public-Coin:** An interactive protocol is considered to be public-coin if all of the \mathcal{V} messages can be computed as a deterministic function given a random public input.

A public coin interactive argument of knowledge can be turned into a non-interactive via Fiat-Shamir transformation [40]. The Fiat-Shamir transformation replaces the challenges of the verifier with PRF outputs of the transcript. A succinct non-interactive argument of knowledge is called a SNARK.

Polynomial interactive oracle proof. Interactive arguments of knowledge can be constructed from argument systems with oracle access to prover messages, e.g., oracle access to polynomials. The argument is then compiled using cryptographic protocols, such as polynomial commitments. Here we define *polynomial interactive oracle proof* (Poly-IOP), a specific category of such argument systems.

Definition 3 (Polynomial Interactive Oracle Proof). *A polynomial interactive oracle proof (Poly-IOP) is a public-coin interactive argument for an oracle relation \mathcal{R} . In any instance $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ of the oracle relation, the public input \mathbb{x} can contain oracles to μ -variate polynomials over a field \mathbb{F} , where μ and the degree of the polynomials are specified by the oracles. The oracles can be queried at arbitrary points in \mathbb{F}^μ to evaluate the polynomials. The polynomials specifications corresponding to the oracles are in $\text{pp} \leftarrow \text{Setup}(1^\lambda, \mathcal{R})$ and \mathbb{w} . In each round of the Poly-IOP protocol, the prover sends several polynomial oracles to the verifier, and the verifier sends random challenges to the prover.*

The knowledge soundness of a Poly-IOP protocol differs from the knowledge soundness of an ordinary interactive argument protocol in that the extractor \mathcal{E} can query the polynomial oracles at arbitrary points. It is shown in [8] that a Poly-IOP protocol is knowledge sound if it is sound.

Remark 1. *The Poly-IOP definition here should be more precisely described as polynomial interactive oracle arguments. Yet the usage of the words “proof” and “argument” are often mixed with each other, and it is more conventional to write Poly-IOPs instead of Poly-IOAs. That is why we keep the term Poly-IOP here.*

Polynomial commitment scheme. To compile a Poly-IOPs to a SNARK, a polynomial commitment scheme (PCS)

is required. On a high level, a PCS securely implements polynomial oracles and queries to such oracles. Here we define PCS as follows.

Definition 4 (Polynomial Commitment Scheme). *A polynomial commit scheme for a class of functions $\mathcal{F} \subseteq \mathbb{F}[\mathbf{X}]$ is a tuple of algorithms (KeyGen, Commit, Open, Verify), where*

- $\text{KeyGen}(1^\lambda, \mathcal{F}) \rightarrow \text{crs}$ generates the public parameter crs.
- $\text{Commit}(f, \text{crs}) \rightarrow \text{com}_f$ generates a commitment com_f of a function f .
- $\text{Open}(f, \mathbf{x}, \text{crs}) \rightarrow (y, \pi)$ outputs $y := f(\mathbf{x})$ and an opening proof π .
- $\text{Verify}(\text{com}_f, \mathbf{x}, y, \pi, \text{crs}) \rightarrow b \in \{0, 1\}$ verifies the polynomial opening given the proof, and returns whether the verification passes.

A PCS protocol should have the following properties:

- **Completeness:** For all $f \in \mathcal{F}$ and for all \mathbf{x}

$$\Pr \left[b = 1 \left| \begin{array}{l} \text{crs} \leftarrow \text{KeyGen}(1^\lambda, \mathcal{F}) \\ \text{com}_f \leftarrow \text{Commit}(f, \text{crs}) \\ (y, \pi) \leftarrow \text{Open}(f, \mathbf{x}, \text{crs}) \\ b \leftarrow \text{Verify}(\text{com}_f, \mathbf{x}, y, \pi, \text{crs}) \end{array} \right. \right] = 1$$

- **Binding:** For all PPT adversary \mathcal{A} the following probability is $\text{negl}(\lambda)$.

$$\Pr \left[b_1 = b_2 = 1 \left| \begin{array}{l} \text{crs} \leftarrow \text{KeyGen}(1^\lambda, \mathcal{F}) \\ \text{com}_f \leftarrow \text{Commit}(f, \text{crs}) \\ \mathbf{x}_1, \mathbf{x}_2, y_1, y_2, \pi_1, \pi_2 \leftarrow \mathcal{A}(\text{crs}) \\ b_{\{1,2\}} \leftarrow \text{Verify}(\text{com}_f, \mathbf{x}_{\{1,2\}}, y_{\{1,2\}}, \pi_{\{1,2\}}, \text{crs}) \end{array} \right. \right]$$

2. HyperPlonk Poly-IOP

HyperPlonk is a Poly-IOP that features linear prover time [8]. Recall that in Plonk Poly-IOP, we need to interpolate univariate polynomials over a set of size $O(N)$. However, interpolating degree- n polynomials would incur at least $O(n \log n)$ prover time, which means that the prover time is at least $O(N \log N)$. To resolve this problem, in HyperPlonk multivariate polynomials are interpolated over the boolean hypercube $B_\mu := \{0, 1\}^\mu$. By the merits of linear time algorithms to evaluate and interpolate multilinear polynomials in the previous work [6], [41], the prover time is finally reduced to $O(N)$ in HyperPlonk Poly-IOP.

Multivariate polynomials in HyperPlonk. Let ν be the length of binary index to represent public inputs, i.e. $2^\nu = \ell_x$. Let μ be the length of binary index to represent the gates, i.e. $2^\mu = N$. We can interpret σ to another permutation $\sigma_\mu : B_{\mu+2} \rightarrow B_{\mu+2}$. Then the multivariate polynomials that characterizes the Plonk constraint (s, σ) are defined as follows:

- Two multilinear polynomials $S_{\text{add}}(\mathbf{x}), S_{\text{mult}}(\mathbf{x}) \in \mathbb{F}_{\mu+2}^{\leq 1}[\mathbf{X}]$ such that for all $i \in [N]$

$$\begin{cases} S_{\text{add}}(\langle i \rangle_\mu) = s_i \\ S_{\text{mult}}(\langle i \rangle_\mu) = 1 - s_i \end{cases}$$

- A multilinear polynomial $F \in \mathbb{F}_{\mu+2}^{\leq 1}$ such that

$$\begin{cases} F(0, 0, \langle i \rangle_\mu) = t_{3i+1} & i \in [0, N-1] \\ F(0, 1, \langle i \rangle_\mu) = t_{3i+2} & i \in [0, N-1] \\ F(1, 0, \langle i \rangle_\mu) = t_{3i+3} & i \in [0, N-1] \\ F(1, \dots, 1, \langle i \rangle_\nu) = x_{i+1} & i \in [0, \ell_x-1] \end{cases}$$

- A multilinear polynomial $I \in \mathbb{F}_{\nu}^{\leq 1}$ such that for all $i \in [0, \ell_x-1]$ we have

$$I(\langle i \rangle_\nu) = x_{i+1}.$$

Multivariate SumCheck Poly-IOP. Prover wants to show to the verifier that a multivariate polynomial $f(\mathbf{x}) := h(g_1(\mathbf{x}), \dots, g_c(\mathbf{x})) \in \mathbb{F}_\mu[\mathbf{X}]$ satisfies $\sum_{\mathbf{x} \in B_\mu} f(\mathbf{x}) = v$, where each g_i is multilinear and h can be evaluated using an arithmetic circuit with $O(d)$ gates. Before describing the final Poly-IOP protocol, recall the original SumCheck protocol in [25] that works for $f \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$:

- For each $i \in [\mu]$:
 - \mathcal{P} computes $r_i(x) := \sum_{\mathbf{b} \in B_{\mu-i}} f(\alpha_1, \dots, \alpha_{i-1}, x, \mathbf{b})$ and sends the oracle of r_i to \mathcal{V} .
 - \mathcal{V} checks if $v = r_i(0) + r_i(1)$. Then \mathcal{V} samples $\alpha_i \leftarrow \mathbb{F}$ and sets $v := r_i(\alpha_i)$. \mathcal{V} sends α_i to \mathcal{P} .
- \mathcal{V} checks if $f(\alpha) = v$.

Note that in [25] the vanilla SumCheck protocol assumes that the prover has oracle access to f . To efficiently evaluate f at the given points along the protocol for the prover, Chen et al. build upon the protocols in [6], [41] and developed a dynamic-programming-based scheme for polynomials $f \in \mathbb{F}_\mu^{\leq d}$. Specifically, the compiled SumCheck protocol without oracle access to f is described as follows:

- For all $j \in [c]$, \mathcal{P} evaluate $g_j(\mathbf{x})$ for all $\mathbf{x} \in B_\mu$ and build a table A_j to store the valuations.
- For each $i \in [\mu]$:
 - \mathcal{P} computes $r_j^{(b)}(x) := (1-x)A(0, \mathbf{b}) + xA(1, \mathbf{b})$ for all $\mathbf{b} \in B_{\mu-i}$ and for all $j \in [c]$.
 - For all $\mathbf{b} \in B_{\mu-i}$, \mathcal{P} computes $r^{(b)}(x) := h(r_1^{(b)}(x), \dots, r_c^{(b)}(x))$, where each addition gate is computed using polynomial addition, and each multiplication is computed using fast polynomial multiplication.
 - \mathcal{P} computes $r_i(x) := \sum_{\mathbf{b} \in B_{\mu-i}} r^{(b)}(x)$, and sends the oracle of r_i to \mathcal{V} .
 - \mathcal{V} checks if $v = r_i(0) + r_i(1)$. Then \mathcal{V} samples $\alpha_i \leftarrow \mathbb{F}$ and sets $v := r_i(\alpha_i)$.
 - \mathcal{V} sends α_i to \mathcal{P} .
 - \mathcal{P} sets $A_j[\mathbf{b}] = r_j^{(b)}(\alpha_i)$ for all $\mathbf{b} \in B_{\mu-i}$.
- \mathcal{V} queries $f(\alpha)$ and checks if it is equal to v .

Here we summarize the complexity of the multivariate SumCheck Poly-IOP:

- **Prover Time:** $O(2^\mu \cdot d \log^2 d)$
- **Verifier Time:** $O(\mu)$
- **Communication:** $O(\mu)\mathbb{F}$

- **Oracle Access:** The prover builds and sends oracle for $O(\mu)$ degree- d univariate polynomials and $O(1)$ multivariate polynomials in $\mathbb{F}_\mu^{\leq d}[\mathbf{X}]$. The verifier queries $O(\mu)$ degree- d univariate polynomials and $O(1)$ multivariate polynomials in $\mathbb{F}_\mu^{\leq d}[\mathbf{X}]$.

Multivariate ZeroTest Poly-IOP. The prover wants to show to the verifier that a multivariate polynomial $f \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$ satisfies $f(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\mu$. The protocol reduces ZeroTest to SumCheck, and is described as follows:

- \mathcal{V} samples and sends to \mathcal{P} a random vector $\mathbf{r} \leftarrow \mathbb{F}^\mu$.
- Let $\tilde{f}(\mathbf{x}) := f(\mathbf{x}) \cdot \chi_{\mathbf{r}}(\mathbf{x})$. \mathcal{P} and \mathcal{V} run multilinear SumCheck on \tilde{f} with target value 0.

We argue that this protocol is complete. Let $g(\mathbf{y}) := \sum_{\mathbf{x} \in B_\mu} f(\mathbf{x}) \cdot \chi_{\mathbf{y}}(\mathbf{x})$. Then it is clear that $g(\mathbf{y}) = f(\mathbf{y})$ for all $\mathbf{y} \in B_\mu$. Since g is multilinear, we have $g(\mathbf{y})$ is zero polynomial if and only if $f(\mathbf{y})$ is zero over B_μ .

Multivariate PermTest Poly-IOP. Let $f \in \mathbb{F}_\mu^{\leq d}[\mathbf{X}]$ be a multivariate polynomial. Let $\sigma : B_\mu \rightarrow B_\mu$ be a permutation over B_μ . The prover wants to show to the verifier that $f(\mathbf{x}) = f(\sigma(\mathbf{x}))$ for all $\mathbf{x} \in B_\mu$. The multivariate PermTest Poly-IOP is described as follows:

- Let $s, s_\sigma \in \mathbb{F}_\mu^{\leq 1}[\mathbf{X}]$ be two mappings, where $s(\mathbf{x}) := [\mathbf{x}]$ and $s_\sigma(\mathbf{x}) := [\sigma(\mathbf{x})]$.
- \mathcal{V} samples and sends to the prover $\gamma, \delta \leftarrow \mathbb{F}$.
- Let $f_1 := f + \gamma s + \delta$ and $f_2 := f + \gamma s_\sigma + \delta$. \mathcal{P} builds a multilinear polynomial $z \in \mathbb{F}_\mu^{\leq 1}$ such that for all $\mathbf{x} \in B_\mu$

$$\begin{cases} z(0, \mathbf{x}) = f_1(\mathbf{x})/f_2(\mathbf{x}) \\ z(1, \mathbf{x}) = z(\mathbf{x}, 0) \cdot z(\mathbf{x}, 1) \end{cases}$$

Let $w_1(\mathbf{x}) := z(1, \mathbf{x}) - z(\mathbf{x}, 0) \cdot z(\mathbf{x}, 1)$ and $w_2(\mathbf{x}) := f_2(\mathbf{x}) \cdot z(0, \mathbf{x}) - f_1(\mathbf{x})$.

- \mathcal{P} sends the oracle of z to \mathcal{V} .
- \mathcal{P} and \mathcal{V} run multilinear ZeroTest on w_1 and w_2 .
- \mathcal{V} queries $z(1, 1, \dots, 1, 0)$ and checks if it is 1.

The HyperPlonk Poly-IOP. The HyperPlonk Poly-IOP is structured as follows:

- **Input Constraint:** $F(1, \dots, 1, \mathbf{x}) - I(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\nu$ with multilinear ZeroTest.
- **Output Constraint:** \mathcal{V} queries $F(1, 0, \langle N-1 \rangle_\mu)$ and checks if it is zero.
- **Gate Constraint:** Define a multivariate polynomial

$$G(\mathbf{x}) := S_{\text{add}}(\mathbf{x})(F(0, 0, \mathbf{x}) + F(0, 1, \mathbf{x})) + S_{\text{mult}}(\mathbf{x})(F(0, 0, \mathbf{x}) \cdot F(0, 1, \mathbf{x})) - F(1, 0, \mathbf{x}).$$

$$G(\mathbf{x}) = 0$$
 for all $\mathbf{x} \in B_\mu$ with multivariate ZeroTest.
- **Wiring Constraint:** Check if $F(\mathbf{x}) = F(\sigma_\mu(\mathbf{x}))$ for all $\mathbf{x} \in B_\mu$ with multivariate PermTest.

3. ZK for Cirrus

Chen et al. [8] described an efficient ZK compiler for SumChecks, built over the previous construction by Xie et al. [6].

Zero-knowledge distributed SumCheck. The zero-knowledge distributed SumCheck protocol is structured as follows:

- 1) \mathcal{P}_0 samples two random vectors $\mathbf{r} \xleftarrow{\$} \mathbb{F}^\mu$ and $\mathbf{r}_0 \xleftarrow{\$} \mathbb{F}^\epsilon$ and defines $g^{(b)}(\mathbf{x}) = \chi_{\mathbf{r}}(\mathbf{x}) \cdot \chi_{\mathbf{r}_0}(\mathbf{b})$.
- 2) \mathcal{P}_0 sends \mathbf{r} and $\chi_{\mathbf{r}_0}(\mathbf{b})$ to \mathcal{P}_b .
- 3) \mathcal{V} sends a challenge $\rho \xleftarrow{\$} \mathbb{F}^*$ to \mathcal{P}_0 that is relayed to all \mathcal{P}_b .
- 4) The provers and the verifier now run SumCheck PIOP 4.3 over polynomial $f + \rho g$.
- 5) \mathcal{V} queries g and f at point \mathbf{r}' where $\mathbf{r}' \in \mathbb{F}^{\epsilon+\mu}$ is the vector of sumcheck's challenge. \mathcal{V} checks that $f(\mathbf{r}') + \rho g(\mathbf{r}')$ is consistent with the last message of the SumCheck.

Once we have the zero-knowledge Distributed SumCheck, we can build Cirrus the same way it was done in 4.3.

4. Application to SNARK Outsourcing

With Cirrus, we propose a novel design for outsourcing ZK-SNARKs. We focus on using trusted execution environments (TEEs) [42] to enable efficient ZK-SNARK outsourcing while keeping the secret inputs of the users private. TEEs provide a secure enclave within a processor, where sensitive data and computations can be performed in isolation from the rest of the system. TEEs ensure that even privileged users, such as system administrators, cannot access the data or computation within the enclave. This isolation, combined with the ability to verify the integrity of the execution through remote attestation, makes TEEs a powerful tool for securely outsourcing ZK-SNARK generation. By leveraging TEEs, we can achieve efficient computation while ensuring that the secret inputs remain protected throughout the outsourcing process.

The use of Trusted Execution Environments (TEEs) as a straightforward approach for generating ZK-SNARKs is likely to face considerable performance bottlenecks due to the high memory demands inherent to ZK-SNARK generation. This process involves substantial cryptographic computations—such as group exponentiations and bilinear mappings—that require significant memory resources, posing a challenge to the efficiency of TEE-based solutions. Given that most TEEs, like Intel SGX, provide relatively small secure memory enclaves (e.g., 128 MB or less of usable secure memory), such operations quickly exceed the available memory within the enclave, leading to frequent page thrashing. Page thrashing occurs when memory pages are continuously swapped between the secure enclave and untrusted memory, drastically increasing latency. [43] This is particularly problematic for ZK-SNARK generation, where

each cryptographic operation must maintain high integrity and constant memory evictions introduce significant overhead.

With Cirrus, we can use the memory-efficient property of the scheme on TEEs to build an efficient TEE-based ZK-SNARK outsourcing protocol. By combining the parallel processing capabilities of Cirrus with the secure computation environment of TEEs, our solution not only addresses the memory bottleneck but also ensures that the secret inputs remain protected during the proof generation process. This hybrid approach allows for both scalable and secure ZK-SNARK outsourcing, overcoming the memory limitations typically associated with TEEs while preserving privacy guarantees.