Cirrus: Performant and Accountable Distributed SNARK

Wenhao Wang
Yale Univerity, IC3
wenhao.wang@yale.edu

Fangyan Shi
Tsinghua University
sfy21@tsinghua.org.cn

Dani Vilardell
Cornell University, IC3
dv296@cornell.edu

Fan Zhang
Yale University, IC3
f.zhang@yale.edu

***Abstract*—Succinct Non-interactive Arguments of Knowledge (SNARKs) can enable efficient verification of computation in many applications. However, generating SNARK proofs for large-scale tasks, such as verifiable machine learning or virtual machines, remains computationally expensive. A promising approach is to distribute the proof generation workload across multiple workers. A practical distributed SNARK protocol should have three properties: horizontal scalability with low overhead (linear computation and logarithmic communication per worker), accountability (efficient detection of malicious workers), and a universal trusted setup independent of circuits and the number of workers. Existing protocols fail to achieve all these properties.**

In this paper, we present Cirrus, the first distributed SNARK generation protocol achieving all three desirable properties at once. Our protocol builds on HyperPlonk (EUROCRYPT'23), inheriting its universal trusted setup. It achieves linear computation complexity for both workers and the coordinator, along with low communication overhead. To achieve accountability, we introduce a highly efficient accountability protocol to localize malicious workers. Additionally, we propose a hierarchical aggregation technique to further reduce the coordinator's workload.

We implemented and evaluated Cirrus on machines with modest hardware. Our experiments show that Cirrus is highly scalable: it generates proofs for circuits with 33M gates in under 40 seconds using 32 8-core machines. Compared to the state-of-the-art accountable protocol Hekaton (CCS'24), Cirrus achieves over $7\times$ faster proof generation for PLONK-friendly circuits such as the Pedersen hash. Our accountability protocol also efficiently identifies faulty workers within just 4 seconds, making Cirrus particularly suitable for decentralized and outsourced computation scenarios.

I. Introduction

Succinct Non-interactive Arguments of Knowledge (SNARKs) [1]–[11] enable efficient, non-interactive verification of computations. The primitive has proven practical for various applications, such as privacy-preserving payments (e.g., Zerocash [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.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240668
www.ndss-symposium.org

A recent line of works [14]–[18] proposed to horizontally scale up SNARK generation with *distributed SNARK generation protocols*, where multiple *workers* collaborate on different parts of a general-purpose circuit to create a final proof. 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.

We identify three properties that a distributed SNARK generation scheme should satisfy:

1) Firstly, it should **scale horizontally** with low overhead. Specifically, each worker's computational complexity should ideally grow linearly with the size of the subcircuit it is assigned, avoiding performance degradation as more workers are added. Furthermore, each worker's communication complexity should scale sublinearly with its subtask size, as large tasks would otherwise incur prohibitive communication costs.
2) Secondly, it should achieve **accountability**, i.e., being able to efficiently detect malicious behavior and identify faulty or dishonest workers. Accountability is particularly important when proof computation is outsourced to decentralized networks that may include untrusted participants. For instance, in prover marketplaces [19]–[21], users delegate computation tasks to workers offering spare computational resources, which necessitates mechanisms to detect malfeasance and hold responsible parties accountable to maintain trust and reliability.
3) Thirdly, it should have a **universal setup**, avoiding the need for per-circuit trusted setup ceremonies. Requiring a dedicated setup each time a new application is encountered is highly inefficient, especially in proof outsourcing scenarios where workers frequently handle tasks from many different applications. The complexity and cost associated with conducting secure multi-party ceremonies for each application would severely limit scalability and usability [22]. A universal setup addresses this challenge by enabling diverse applications to leverage distributed SNARKs without additional trusted initialization.

Limitation of prior works. While prior works reduced overhead and achieved partial accountability, none achieved all three properties. Pianist [16] is the first distributed proof generation scheme based on PLONK. It achieves a robust quasilinear

prover time protocol for general circuits, but not accountability[1]. HyperPianist [18] improves the prover complexity to linear time but still lacks 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 Pianist nor Hekaton achieves linear worker time.

This work: Cirrus. We introduce Cirrus, the first distributed SNARK generation scheme that simultaneously satisfies all of the desired properties: low overhead, accountability, and universal trusted setup. Specifically, Cirrus achieves linear computational complexity for both workers and the coordinator, with communication overhead scaling sub-linearly with task size, ensuring practical efficiency for large-scale deployments. Moreover, Cirrus reuses existing universal setup parameters from HyperPlonk, which significantly simplifies its deployment. Unlike previous accountable schemes such as Hekaton, Cirrus does not require circuit-specific setups, and therefore avoiding repeated costly setup ceremonies. Furthermore, Cirrus implements an efficient accountability protocol that enables the coordinator to quickly detect malicious behavior, precisely identify offending workers, and support incentive-based systems to maintain security and trustworthiness. By simultaneously addressing computational efficiency, accountability, and setup universality, Cirrus significantly expands the potential scalability and security of SNARK applications in decentralized and outsourced computation environments.

A. Technical Overview

Our distributed scheme is built upon HyperPlonk [8], a non-distributed SNARK scheme with prover time linear in circuit size and universal trusted setup. Our adaptation of HyperPlonk to a distributed setting must preserve these properties.

We use M to denote the number of workers. A circuit C of N gates is particitioned into M sub-circuits (using approach proposed in Pianist [16]), each of size T. Additionally, M *workers* collaborate with a single *coordinator*.

Distributing HyperPlonk. To adapt HyperPlonk to a distributed setting, we need to distribute its core building blocks: multilinear KZG, the sum-check protocol (SumCheck), the zero test protocol (ZeroTest), and the permutation test protocol (PermTest). While distributing the multilinear KZG protocol is straightforward, naively distributing the SumCheck protocol using existing techniques (e.g., deVirgo [14]) introduces a critical issue. Specifically, at the conclusion of the distributed SumCheck, we must open a multivariate polynomial with constant degree $d > 1$. However, existing techniques from deVirgo [14] combined with multilinear KZG only support multivariate polynomials of degree at most 1.

We address this limitation using the following key observation: any constant-degree multivariate polynomial can be equivalently represented as a function of a constant number of multilinear (i.e., degree-1) polynomials. More formally, each polynomial $f(\boldsymbol{x})$ of constant degree can be expressed as $f(\boldsymbol{x}) = h(g_1(\boldsymbol{x}), \ldots, g_c(\boldsymbol{x}))$, where each $g_i(\boldsymbol{x})$ is multilinear. Furthermore, additions and multiplications of these degree-1 distributed polynomials produce results that match the evaluation of the original polynomial on the Boolean hypercube $\{0,1\}^n$. Consequently, we transform the original challenge into distributing the sum-check protocol for multilinear polynomials. This transformation enables us to employ distributed multilinear KZG effectively to commit to and open polynomials at the end of the distributed SumCheck. With this distributed version of SumCheck, we subsequently extend our construction to a fully distributed HyperPlonk scheme.

Optimistic accountability. When a proof is generated by workers, the coordinator verifies the proof. If this verification succeeds, no further action is required, as correctness is ensured with overwhelming probability by the soundness property. However, when verification fails, the coordinator is required to identify one or more workers at fault. A naive approach would require the coordinator to independently recompute all intermediate results and compare them with the results sent by the workers, incurring high computational costs same as generating the entire proof on its own. To address this issue, Cirrus introduces a fault localization protocol that significantly reduces computational overhead.

First, the coordinator verifies the KZG sub-proofs provided by each worker and checks their consistency with the respective KZG commitments of the witness polynomials constructed by the workers. A failed verification at this stage directly identifies a faulty worker. If all sub-proofs verify successfully, the coordinator proceeds to reconstruct the witness polynomials from each worker and evaluates them at a common random point determined at the conclusion of the distributed sum-check protocol. A mismatch in any evaluation result indicates a malicious worker. Asymptotically, our fault localization protocol requires only $O(M \log T)$ pairings and $O(MT)$ field operations. Since the field operations are lightweight in practice, our accountability protocol is substantially more efficient than the baseline approach of recomputing the proof. Our evaluation confirms its concrete efficiency, requiring only 4 seconds to localize malicious workers, even for circuits with 33 million gates distributed across 256 workers.

Reducing the coordinator's computational overhead. In the optimistic case where all workers are honest, the coordinator's computational workload scales with the number of workers, potentially becoming a performance bottleneck as the worker count grows. In the vanilla distributed HyperPlonk protocol, the coordinator's runtime complexity would be $O(M \log T)$, which originates from aggregating M vectors, each of length $O(\log T)$, in the distributed SumCheck and distributed multilinear KZG protocols. To reduce this, Cirrus introduces an efficient *hierarchical aggregation* approach. Specifically, we partition the aggregation task among $M/\log T$ designated worker provers, referred to as *leaders*, each of whom aggregates vectors from $\log T$ workers. Consequently, the coordinator's

[1]Pianist introduces a limited form of accountability designed specifically for data-parallel circuits; however, its accountability mechanism does not extend to general-purpose circuits.

computational complexity for this aggregation step is reduced significantly to $O(M)$. We note that the resulting computational overhead per leader is $O(\log^2 T)$, which remains negligible compared to the primary computation tasks, which are dominated by $O(T)$.

In summary, Cirrus is the first distributed SNARK protocol to achieve linear computational complexity, logarithmic communication per worker, efficient accountability, and a universal trusted setup. As shown in Table I, Cirrus outperforms previous state-of-the-art distributed SNARK schemes by simultaneously achieving all desirable properties. Therefore, Cirrus significantly improves both the practicality and scalability of real-world deployments of ZKP applications.

B. Implementation and Evaluation

We implement and extensively evaluate Cirrus to assess its end-to-end performance. Our implementation builds upon the HyperPlonk library [24]. To benchmark performance, we generated random PLONK circuits with corresponding random witnesses, distributing the SNARK generation process across up to 32 AWS `t3.2xlarge` machines (each equipped with 8 vCPUs and 32 GB of memory). We note that Cirrus seamlessly reuses existing universal setup parameters from HyperPlonk, simplifying its deployment in practice.

Our experiments demonstrate that Cirrus can efficiently produce proofs for circuits containing up to 33M gates in merely 39 s, using a distributed network of 32 worker machines, with 40.0 KB communication per worker and 16.6 KB proof size. In contrast, vanilla HyperPlonk can only handle circuits of size 4M gates, requiring more than 110 s on the same hardware configuration. To ground these numbers in an application, we instantiate Cirrus to prove RAM programs [25]–[29]. Distributing the RAM program CPU cycles over 256 cores, Cirrus sustains near-linear scaling and proves a trace of 2^{14} CPU steps in under 40 s, while a multi-threaded HyperPlonk baseline is substantially slower. These results clearly illustrate the horizontal scalability advantages of Cirrus.

We perform comparative evaluations with another state-of-the-art accountable distributed SNARK scheme Hekaton [17]. Our experiments highlight that for PLONK-friendly workloads, such as Pedersen hash circuits, Cirrus significantly outperforms Hekaton, achieving a performance gain of over 7×. For other tasks, including MiMC hash and PoK of Exponent, Cirrus also achieves faster performance than Hekaton, while additionally offering the advantage of a universal trusted setup. Specifically, unlike Hekaton, our approach does not require separate trusted setups for each distinct application or varying worker configurations, substantially simplifying practical deployment and enhancing flexibility.

We evaluate the efficiency of our accountability protocol. Our results indicate that when an incorrect proof is produced, the coordinator can identify at least one faulty worker within 4 s for circuits of 33M gates and 256 workers, showing the practicality of the accountability checks.

We also evaluate the performance benefits of hierarchical aggregation. We assess the coordinator's computational overhead and confirm that, with hierarchical aggregation, the coordinator's runtime is independent of sub-circuit sizes, thereby validating this theoretical improvement. Our experiments show that hierarchical aggregation only adds minimal overhead to leader workers compared with non-leader workers.

Our Contribution

- We introduce the notion of *accountability* to distributed SNARK generation, which is particularly important in decentralized settings, such as zero-knowledge prover markets [19], where participants may behave maliciously or unreliably.
- We present Cirrus, the first accountable distributed SNARK generation protocol that simultaneously achieves all the essential properties for practical deployment, including horizontal scalability, accountability, and universal setup compatible with existing parameters.
- We implement and evaluate the performance of Cirrus. The source code is publicly accessible via this link.

II. Preliminaries

In this section, we introduce the key notation, definitions, and building blocks.

Notation. Let $\mathbb{F}$ denote a finite field of size $\Omega(2^\lambda)$, where λ is the security parameter, and let $B_\mu := \{0,1\}^\mu$ denote the μ-dimensional Boolean hypercube. We write $\mathbb{F}_\mu^{\leq d}[\boldsymbol{X}]$ for the set of μ-variate polynomials whose degree in each variable is at most d; a multivariate polynomial is *multilinear* if its degree in each variable is at most 1. Any polynomial $f \in \mathbb{F}_\mu^{\leq d}[\boldsymbol{X}]$ can be expressed as $f(\boldsymbol{x}) = h(g_1(\boldsymbol{x}), \ldots, g_c(\boldsymbol{x}))$, where h has total degree $O(d)$, is computable by an arithmetic circuit of $O(d)$ gates, and each g_i is multilinear. For $\boldsymbol{w} \in B_\mu$, let $\chi_{\boldsymbol{w}}(\boldsymbol{x}) := \prod_{i=1}^{\mu}(w_i x_i + (1-w_i)(1-x_i))$ denote the multilinear Lagrange polynomial over B_μ. For $\boldsymbol{x} \in \mathbb{F}^\mu$, define $[\boldsymbol{x}] := \sum_{i=1}^{\mu} 2^{i-1} x_i$, and let $\langle v \rangle_m$ denote the m-bit representation of an integer $v \in [0, 2^m - 1]$. Finally, we use C to denote a circuit over $\mathbb{F}$, x a public input, and w a witness. We write $\mathsf{C}(x; w) = 1$ when w satisfies C on x.

Useful tools. Here we introduce frequently used tools.

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

Lemma 2 (Schwartz-Zippel Lemma)**.** *Let $g : \mathbb{F}^\ell \to \mathbb{F}$ be a nonzero ℓ-variate polynomial of degree at most d. Then for all $\emptyset \neq S \subseteq \mathbb{F}$, $\Pr_{\boldsymbol{x} \leftarrow S^\ell}[g(\boldsymbol{x}) = 0] \leq \frac{d}{|S|}$.*

Model and assumptions. Participants in Cirrus consist of a coordinator $\mathcal{C}$, $M = 2^\xi$ workers $\mathcal{P}_1, \ldots, \mathcal{P}_M$, and a verifier $\mathcal{V}$. We assume point-to-point communication channels between $\mathcal{C}$ and workers, and between $\mathcal{C}$ and $\mathcal{V}$. The verifier only communicates with the coordinator. We assume static corruptions, i.e., an adversary may corrupt any subset $S \subseteq [M]$ of workers and/or the coordinator $\mathcal{C}$ before the protocol

Scheme	Worker Time	Coordinator Time	Comm.	$\mathcal{V}$ Time	Accountable	Setup
Cirrus (Ours)	$\boldsymbol{O(T)}$	$\boldsymbol{O(M)}$	$\boldsymbol{O(M \log T)}$	$\boldsymbol{O(\log N)}$	✓	**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]	$O(T \log T)$	$O(M \log M)$	$O(M)$	$O(1)$	✗	Universal
DeVirgo [14]	$O(T)$	N/A	$O(N)$	$O(\log^2 N)$	✗	Transparant
DIZK [15]	$O(T \log^2 T)$	N/A	$O(N)$	$O(1)$	✗	Circuit-specific

TABLE I: 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.

starts; corrupted parties can arbitrarily collude. We require the accountability property to hold when $\mathcal{C}$ is *honest* (i.e., not corrupted); we require the soundness and knowledge soundness properties to hold even if $\mathcal{C}$ is corrupted. Following existing works [16]–[18], we assume all workers and the coordinator have access to the witness.

Definition 1 (Accountable Distributed SNARK)**.** *An accountable distributed SNARK for a circuit* C *with* M *workers consists of PPT sub-protocols:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$*: Generate public parameters* pp*.*
- $\mathsf{Partition}(\mathsf{C}, M) \to (\mathsf{C}_1, \ldots, \mathsf{C}_M)$*: Split the circuit* C *into* M *sub-circuits covering all gates.*
- $\mathsf{KeyGen}(\mathsf{C}, \mathsf{pp}, M) \to (\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_M, \mathsf{vk})$*: Generate the coordinator proving key* pk*, per-worker proving keys* $\mathsf{pk}_1, \ldots, \mathsf{pk}_M$*, and verification key* vk*.*
- $\mathsf{SplitInst}(\mathsf{pp}, \mathsf{C}, x, w) \to ((\mathsf{pp}_i, x_i, w_i)_{i\in[M]})$*: From* $(\mathsf{pp}, \mathsf{C}, x, w)$ *derive per-worker inputs where* x_i *is the public input relevant to* C_i *and* w_i *the corresponding witnesses.*
- $\mathsf{Prove}_i(\mathsf{pk}_i, \mathsf{C}_i, x_i, w_i), \mathsf{Prove}(\mathsf{pk}, \mathsf{C}, x, w) \to \pi$*: Workers interact with the coordinator and other workers to prove that the circuit computation is satisfied by* w*, and output a proof* π *which is sent to* $\mathcal{V}$*. Let* tr *denote the communication transcript of the coordinator during the proof generation.*
- $\mathsf{CheckAndAccuse}(\mathsf{pk}, \mathsf{C}, x, w, \mathsf{tr}) \to S^* \subseteq [M] \cup \{\perp\}$*: Given the communication transcript* tr *during proof generation and the circuit witnesses, the coordinator interacts with workers and outputs a subset of malicious workers* S^**, or* $\perp$ *if no malicious worker is detected.*
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to b \in \{0, 1\}$*: The verifier outputs whether or not it accepts the proof* π*.*

They satisfy the following properties:

Completeness. *If* $\mathsf{C}(x; w) = 1$ *and all parties are honest, then* $\Pr\left[\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1 \wedge \mathsf{CheckAndAccuse}(\mathsf{pk}, \mathsf{C}, x, w, \mathsf{tr}) = \perp\right] = 1$*. Namely, valid proofs can be generated for satisfying witnesses when all parties are honest without false accusations.*

Knowledge soundness. *For any PPT adversary* $\mathcal{A}$ *that statically corrupts any subset* $S \subseteq [M]$ *and interacts with honest* $\mathcal{C}$ *to produce* π *such that* $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$*, there exists a PPT extractor* $\mathcal{E}$ *with oracle access to* $\mathcal{P}_1, \ldots, \mathcal{P}_M, \mathcal{C}$ *that outputs* $w^\star$ *with* $\Pr\left[\mathsf{C}(x; w^\star) = 1\right] \geq 1 - \mathsf{negl}(\lambda)$*.*

Accountability. *Let* $S \subseteq [M]$ *be a non-empty set of corrupted workers. Consider any execution with honest* $\mathcal{C}$ *that ends with* $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 0$*,* $\mathsf{CheckAndAccuse}(\mathsf{pk}, x, w, \mathsf{tr})$ *outputs a non-empty subset of* S *except with probability* $\mathsf{negl}(\lambda)$*. This also implies that no honest parties will be falsely accused.*

Zero-knowledge (Optional). *If ZK is claimed, then if the coordinator and all workers are honest, there exists a PPT simulator* Sim *such that for any* C, x, w *such that* $\mathsf{C}(x; w) = 1$*, the distribution of* $\mathsf{Sim}(\mathsf{pp}, \mathsf{C}, x)$ *is indistinguishable from a proof produced by the coordinator and the workers.*

III. 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.

A. 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 the polynomials necessary for the protocol.

Defining polynomials. Before running the distributed SNARK, we assume that a circuit 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 $\{t^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_\xi}$, 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 $\nu^{(\boldsymbol{b})}$ be the length of the binary index to represent public inputs for $\mathcal{P}_{\boldsymbol{b}}$, i.e. $2^{\nu^{(\boldsymbol{b})}} = \ell_x^{(\boldsymbol{b})}$. Let μ be the length of the binary index to represent the gates for all workers, i.e. $2^\mu = |\mathsf{C}|/M$. Let $\boldsymbol{s}^{(\boldsymbol{b})}$ represent the gate selection vector for $\mathcal{P}_{\boldsymbol{b}}$. Note that there is a wiring permutation between different circuit segments. Let $\boldsymbol{s}^{(\boldsymbol{b})}$, $\sigma^{(\boldsymbol{b})} : B_{\mu+2} \to B_{\mu+2}$ and $\rho^{(\boldsymbol{b})} : B_{\mu+2} \to B_\xi$ be the mappings of the circuit wiring for $\mathcal{P}_{\boldsymbol{b}}$, such that $\{(\sigma^{(\boldsymbol{b})}(\boldsymbol{c}), \rho^{(\boldsymbol{b})}(\boldsymbol{c})) : \boldsymbol{c} \in B_\mu, \boldsymbol{b} \in B_\xi\} = B_{\mu+2} \times B_\xi$. Specifically, the permutations indicate that $t_i^{(\boldsymbol{b})} = t_{\sigma^{(\boldsymbol{b})}(\langle i\rangle)}^{\rho^{(\boldsymbol{b})}(\langle i\rangle)}$ for all $i \in [2^\mu], \boldsymbol{b} \in B_\xi$.

A worker prover $\mathcal{P}_{\boldsymbol{b}}$ interpolates the following polynomials:

- Two multilinear polynomials $S_{\mathsf{add}}^{(\boldsymbol{b})}, S_{\mathsf{mult}}^{(\boldsymbol{b})} \in \mathbb{F}_\mu^{\le 1}[\boldsymbol{X}]$ such that $\forall i \in [N^{(\boldsymbol{b})}]$ $S_{\mathsf{add}}^{(\boldsymbol{b})}(\langle i\rangle_\mu) = s_i^{(\boldsymbol{b})}$ and $S_{\mathsf{mult}}^{(\boldsymbol{b})}(\langle i\rangle_\mu) = 1 - s_i^{(\boldsymbol{b})}$.
- A multilinear polynomial $F^{(\boldsymbol{b})} \in \mathbb{F}_{\mu+2}^{\le 1}$ such that

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

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

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 $f(\boldsymbol{x}, \boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\xi} f^{(\boldsymbol{b})}(\boldsymbol{x})\chi_{\boldsymbol{b}}(\boldsymbol{y}) \in \mathbb{F}_{\mu+\xi}^{\le 1}[\boldsymbol{X}]$, the PCS protocol is described as follows:

- KeyGen: Generate $\mathsf{crs} = (g, (g^{\tau_i})_{i\in[\mu+\xi]}, (U_{\boldsymbol{c},\boldsymbol{b}} := g^{\chi_{\boldsymbol{c}}(\tau_1,\ldots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})})_{\boldsymbol{c}\in B_\mu, \boldsymbol{b}\in B_\xi})$ where $\tau_1, \ldots, \tau_{\mu+\xi}$ are secrets. Note that we can derive $V_{\boldsymbol{b}} := g^{\chi_{\boldsymbol{b}}(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$ with crs, which will be useful to $\mathcal{C}$, since $\sum_{\boldsymbol{c}\in B_\mu} \chi_{\boldsymbol{c}} \equiv 1$.
- $\mathsf{Commit}(f, \mathsf{crs})$: each $\mathcal{P}_{\boldsymbol{b}}$ computes $\mathsf{com}_{\boldsymbol{b}} := \prod_{\boldsymbol{c}\in B_\mu} U_{\boldsymbol{c},\boldsymbol{b}}^{f^{(\boldsymbol{b})}(\boldsymbol{c})}$ and sends $\mathsf{com}_{\boldsymbol{b}}$ to $\mathcal{C}$. Then $\mathcal{C}$ computes $\mathsf{com} := \prod_{\boldsymbol{b}\in B_\xi} \mathsf{com}_{\boldsymbol{b}}$.
- $\mathsf{Open}(f, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathsf{crs})$:
  1) Each prover $\mathcal{P}_{\boldsymbol{b}}$ computes $z^{(\boldsymbol{b})} := f^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ and $f^{(\boldsymbol{b})}(\boldsymbol{x}) - f^{(\boldsymbol{b})}(\boldsymbol{\alpha}) := \sum_{i\in[\mu]} q_i^{(\boldsymbol{b})}(\boldsymbol{x})(x_i - \alpha_i)$. Then it computes $\pi_i^{(\boldsymbol{b})} = g^{q_i^{(\boldsymbol{b})}(\tau_1,\ldots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$ with crs, and sends $\boldsymbol{\pi}^{(\boldsymbol{b})}$ and $z^{(\boldsymbol{b})}$ to $\mathcal{C}$.
  2) $\mathcal{C}$ computes $z = f(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{\boldsymbol{b}\in B_\xi} z^{(\boldsymbol{b})} \cdot \chi_{\boldsymbol{b}}(\boldsymbol{\beta})$. $\mathcal{C}$ decomposes
  $$f(\boldsymbol{\alpha}, \boldsymbol{y}) - f(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{j\in[\xi]} q_j(\boldsymbol{y})(y_j - \beta_j).$$
  Then it computes $\pi_{\mu+j} = g^{q_j(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$ with $V_{\boldsymbol{b}}$. $\mathcal{C}$ also computes $\pi_i = \prod_{\boldsymbol{b}\in B_\xi} \pi_i^{(\boldsymbol{b})}$ for all $i \in [\mu]$.
  3) Then $\mathcal{C}$ sends $\boldsymbol{\pi} := (\pi_1, \ldots, \pi_{\mu+\xi})$ and z to $\mathcal{V}$.
- $\mathsf{Verify}(\mathsf{com}, \boldsymbol{\pi}, \boldsymbol{\alpha}, \boldsymbol{\beta}, z, \mathsf{crs})$: The verifier checks if $e(\mathsf{com}/g^z, g) = \prod_{i\in[\mu]} e(\pi_i, g^{\tau_i - \alpha_i}) \cdot \prod_{j\in[\xi]} e(\pi_{\mu+j}, g^{\tau_{\mu+j} - \beta_j})$.

Here we summarize the complexity of the distributed multilinear KZG PCS. The worker time is $O(2^\mu)$, and the coordinator time is $O(2^\xi \cdot \mu)$. All provers will communicate $O(\mu + \xi)$ group elements. The verifier time is $O(\mu + \xi)$.

Proposition 1. *The distributed multilinear KZG PCS is complete and sound.*

Proof. If $\boldsymbol{\pi}$ is honestly generated, we have

$$\begin{aligned} &\prod_{i\in[\mu]} e(\pi_i, g^{\tau_i - \alpha_i}) \cdot \prod_{j\in[\xi]} e(\pi_{\mu+j}, g^{\tau_{\mu+j} - \beta_j}) \\ =&e(g^{\sum_{i\in[\mu]}\sum_{\boldsymbol{b}\in B_\xi} q_i^{(\boldsymbol{b})}(\tau_1,\ldots,\tau_\mu)\cdot(\tau_i - \alpha_i)}, g) \\ &\cdot e(g^{\sum_{j\in[\xi]} q_j(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})\cdot(\tau_{\mu+j} - \beta_j)}, g) \\ =&e(g^{f(\boldsymbol{\tau}) - f(\boldsymbol{\alpha},\boldsymbol{\beta})}, g) \end{aligned}$$

Therefore, the protocol is complete. We kindly refer to exising work [30] for a full proof of the soundness and knowledge soundness of the multilinear KZG protocol. □

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

1) For each $i \in [\mu]$:
   a) Let $\boldsymbol{\alpha}_{i-1} = (\alpha_1, \ldots, \alpha_{i-1})$
   b) Each $\mathcal{P}_{\boldsymbol{b}}$ computes and sends to $\mathcal{C}$ the polynomial $r_i^{(\boldsymbol{b})}(x) := \sum_{\boldsymbol{w}\in B_{\mu-i}} f^{(\boldsymbol{b})}(\boldsymbol{\alpha}_{i-1}, x, \boldsymbol{w})$.

c) Then $\mathcal{C}$ adds

$$r_i := \sum_{\boldsymbol{b}\in B_\xi} r_i^{(\boldsymbol{b})} = \sum_{\boldsymbol{b}\in B_\xi, \boldsymbol{w}\in B_{\mu-i}} f^{(\boldsymbol{b})}(\boldsymbol{\alpha}_{i-1}, x, \boldsymbol{w}) = \sum_{\boldsymbol{w}\in B_{\mu-i}} \sum_{\boldsymbol{b}\in B_\xi} f(\boldsymbol{\alpha}_{i-1}, x, \boldsymbol{w}, \boldsymbol{b})$$

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 $\mathcal{C}$ a random $\alpha_i \leftarrow \mathbb{F}$. Then $\mathcal{V}$ sets $v := r_i(\alpha_i)$.

2) Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_\mu)$.

3) $\mathcal{C}$ receives $g_i^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ for $i \in [c]$ from each $\mathcal{P}_{\boldsymbol{b}}$. $\mathcal{C}$ first constructs

$$\tilde{g}_i(\boldsymbol{y}) := g(\boldsymbol{\alpha}, \boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\mu} g_i(\boldsymbol{\alpha}, \boldsymbol{b})\chi_{\boldsymbol{b}}(\boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\mu} g_i^{(\boldsymbol{b})}(\boldsymbol{\alpha})\chi_{\boldsymbol{b}}(\boldsymbol{y}).$$

Then $\mathcal{C}$ has $\tilde{f}(\boldsymbol{y}) = f(\boldsymbol{\alpha}, \boldsymbol{y}) = h\left(\tilde{g}_1(\boldsymbol{y}), \ldots, \tilde{g}_c(\boldsymbol{y})\right)$.

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

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

We summarize the complexity of the distributed multivariate SumCheck Poly-IOP. The worker time is $O(2^\mu \cdot d \log^2 d)$, and the coordinator time is $O(2^\xi \cdot d \log^2 d + \mu \cdot 2^\xi \cdot d)$. All provers will communicate $O((\mu + \xi) \cdot d)$ field elements. The verifier time is $O((\mu + \xi) \cdot d)$.

Proposition 2. *The distributed multivariate* SumCheck *protocol is complete and sound.*

Proof. If all workers and the coordinator are honest, in step 1 we have

$$r_i(0) + r_i(1) = \sum_{\boldsymbol{w}\in B_{\mu-i+1}} \sum_{\boldsymbol{b}\in B_\xi} f(\boldsymbol{\alpha}_{i-1}, \boldsymbol{w}, \boldsymbol{b}) = v_i,$$

so the verifier check will pass. In step 4, by the completeness of the SumCheck protocol, the verifier check will pass. Therefore, the protocol is complete. In step 3 we have

$$\forall \boldsymbol{y} \in B_\xi, \quad \tilde{f}(\boldsymbol{y}) = h(\tilde{g}_1(\boldsymbol{y}), \ldots, \tilde{g}_c(\boldsymbol{y})) \equiv \sum_{\boldsymbol{b}\in B_\xi} h(g_1^{(\boldsymbol{b})}(\boldsymbol{y}), \ldots, g_c^{(\boldsymbol{b})}(\boldsymbol{y})) \cdot \chi_{\boldsymbol{b}}(\boldsymbol{y}) = f(\boldsymbol{\alpha}, \boldsymbol{y}).$$

Therefore, we have the target v at the end of step 1 is consistent with the target at the beginning of step 4. Following the soundness of the SumCheck protocol, the distributed multivariate SumCheck protocol is sound. □

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

1) $\mathcal{V}$ samples and sends to $\mathcal{C}$ two random vectors $\boldsymbol{r} \leftarrow \mathbb{F}^\mu$ and $\boldsymbol{r}_0 \leftarrow \mathbb{F}^\xi$.
2) $\mathcal{C}$ sends $\boldsymbol{r}$ to each $\mathcal{P}_{\boldsymbol{b}}$. 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 $\mathcal{P}_{\boldsymbol{b}}$ sets $\tilde{f}^{(\boldsymbol{b})}(\boldsymbol{x}) := f^{(\boldsymbol{b})}(\boldsymbol{x}) \cdot \chi_{\boldsymbol{r}}(\boldsymbol{x}) \cdot \chi_{\boldsymbol{r}_0}(\boldsymbol{b})$.
4) The provers and $\mathcal{V}$ run distributed multivariate SumCheck protocol on $\tilde{f}^{(\boldsymbol{b})}(\boldsymbol{x})$ with target value 0.

To improve the efficiency of the protocol, the verifier has oracle access to $f(x, y) := \sum_{\boldsymbol{b}\in B_\xi} f^{(\boldsymbol{b})}(\boldsymbol{x}) \cdot \chi_{\boldsymbol{b}}(\boldsymbol{y})$ and knows $\chi_{\boldsymbol{r}}(\boldsymbol{x}) \cdot \chi_{\boldsymbol{r}_0}(\boldsymbol{y})$. In Step 4 of SumCheck, $\mathcal{P}_0$ will evaluate $\chi_{\boldsymbol{r}_0}(\boldsymbol{y})$ 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 $\chi_{\boldsymbol{r}}(\boldsymbol{x}) \cdot \chi_{\boldsymbol{r}_0}(\boldsymbol{y})$ at a random point in $O(\mu + \xi)$ time, the protocol is succinct.

Proposition 3. *The distributed multivariate* ZeroTest *Poly-IOP presented above is complete and sound.*

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

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

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

We define the multivariate polynomials $s, s_\mu^{(\boldsymbol{b})}, s_\xi^{(\boldsymbol{b})} \in \mathbb{F}_\mu^{\le 1}[\boldsymbol{X}]$ where $s(\boldsymbol{x}) := [\boldsymbol{x}]$, $s_\mu^{(\boldsymbol{b})}(\boldsymbol{x}) := [\sigma^{(\boldsymbol{b})}(\boldsymbol{x})]$ and $s_\xi^{(\boldsymbol{b})}(\boldsymbol{x}) := [\rho^{(\boldsymbol{b})}(\boldsymbol{x})]$. 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 $f_1^{(\boldsymbol{b})}(\boldsymbol{x}) := f^{(\boldsymbol{b})}(\boldsymbol{x}) + \gamma_\mu \cdot s(\boldsymbol{x}) + \gamma_\xi \cdot [\boldsymbol{b}] + \delta$ and $f_2^{(\boldsymbol{b})}(\boldsymbol{x}) := f^{(\boldsymbol{b})}(\boldsymbol{x}) + \gamma_\mu \cdot s_\mu^{(\boldsymbol{b})}(\boldsymbol{x}) + \gamma_\xi \cdot s_\xi^{(\boldsymbol{b})}(\boldsymbol{x}) + \delta$. Each prover $\mathcal{P}_{\boldsymbol{b}}$ builds a multilinear polynomial $z^{(\boldsymbol{b})} \in \mathbb{F}_{\mu+1}^{\le 1}[\boldsymbol{X}]$ such that for all $\boldsymbol{x} \in B_\mu$

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

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

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

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

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

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

Proposition 4. *The distributed* PermTest *protocol is complete and sound.*

Proof. The completeness and soundness of PermTest directly follow the completeness and soundness of ZeroTest. □

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

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

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

  $\mathcal{V}$ checks that $G^{(\boldsymbol{b})}(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in B_\mu$ and $\boldsymbol{b} \in B_\xi$ with distributed multivariate ZeroTest.
- **Wiring Constraint**: $\mathcal{V}$ verifies that $F^{(\boldsymbol{b})}(\boldsymbol{x}) = F^{(\rho^{(\boldsymbol{b})}(\boldsymbol{x}))}(\sigma^{(\boldsymbol{b})}(\boldsymbol{x}))$ for all $\boldsymbol{x} \in B_{\mu+2}$ and $\boldsymbol{b} \in B_\xi$ with distributed multivariate PermTest.

B. Efficient Accountability Protocol

Up to this point, our protocol does not yet ensure accountability. A straightforward but naive approach to accountability would require the coordinator to verify all intermediate results by independently recomputing every step of each worker's computation. However, this naive solution incurs prohibitive computational and storage overhead, making it impractical for large-scale circuits.

We observe that it is unnecessary for the coordinator to verify all intermediate computations immediately. Instead, the coordinator can optimistically defer these checks until the final aggregated proof is available, running only the verifier's check at the end. The soundness property of the distributed SNARK protocol ensures that any incorrect proof will indicate at least one malicious worker. If the verification fails, the coordinator can then retrospectively examine all communications and intermediate computations performed by each worker to pinpoint the malicious parties. We refer to this deferred verification approach as an *optimistic check*.

One remaining challenge is to significantly reduce the computational cost of the optimistic check in cases when malicious behavior is detected. Our key insight is that accountability can be efficiently enforced by splitting the verification into two clearly defined stages:

Stage 1: Verifying polynomial openings. In the first stage, the coordinator checks the correctness of separate polynomial opening proofs submitted by each worker. Specifically, recall that in the distributed multilinear KZG PCS protocol, each worker prover $\mathcal{P}_{\boldsymbol{b}}$ sends an opening proof $\boldsymbol{\pi}^{(\boldsymbol{b})}$ to the coordinator. The coordinator can verify these openings by checking the pairing equation:

$$e(\mathsf{com} f^{(\boldsymbol{b})}, g) \cdot e(g^{-z^{(\boldsymbol{b})}}, V_{\boldsymbol{b}}) \stackrel{?}{=} \prod_{i \in [\mu]} e(\pi_i^{(\boldsymbol{b})}, g^{\tau_i - \alpha_i}).$$

If this check fails, the coordinator immediately identifies the corresponding prover as malicious.

Stage 2: Verifying witness polynomials. If all polynomial opening proofs from workers verify correctly but the final proof still fails, we prove in Proposition 5 that with overwhelming probability at least one worker has committed to an incorrect witness polynomial $F^{(\boldsymbol{b})}$. Therefore, the coordinator must pinpoint the worker who constructed this incorrect polynomial.

To achieve this efficiently, we leverage the fact that all parties, including the coordinator, already perform a plaintext evaluation of the entire circuit at the start of the distributed proof generation. Therefore, the coordinator can store evaluations of each worker's witness polynomial $F^{(\boldsymbol{b})}$ over B_μ for all $\boldsymbol{b} \in B_\xi$ at no additional computational overhead.

Recall that in step 3 of the distributed SumCheck protocol, each worker sends the evaluation $F^{(\boldsymbol{b})}(\boldsymbol{r})$ at a random point $\boldsymbol{r}$ chosen by the coordinator. Therefore, the coordinator only needs to record the randomness $\boldsymbol{r}$ and locally compute $F^{(\boldsymbol{b})}(\boldsymbol{r})$ for each worker, comparing it with the evaluation reported by the worker. Any discrepancy indicates a malicious worker.

Overall, our two-stage accountability protocol efficiently detects malicious behavior without requiring the coordinator to recompute the entire distributed proof. Furthermore, this accountability protocol does not require the coordinator to communicate with workers. Here we analyze in detail the cost of the protocol for the coordinator. In the first stage, the coordinator verifies $O(M)$ polynomial opening proofs, each of length μ, resulting in a total overhead of $O(M \log T)$ pairing operations. In the second stage, the coordinator evaluates $O(M)$ multilinear polynomials, each with μ variables, incurring $O(MT) = O(N)$ field operations. In Section IV, we empirically demonstrate that this accountability protocol introduces minimal overhead and performs efficiently.

Our efficient accountability protocol is detailed as follows:

1) The coordinator ($\mathcal{C}$) first verifies the final distributed polynomial opening. If this verification passes, no further action is required. If it fails, the coordinator proceeds to identify malicious workers through a two-stage verification process:
2) **Stage 1: Verifying polynomial openings.** For each prover $\mathcal{P}_{\boldsymbol{b}}$, let $\boldsymbol{\pi}^{(\boldsymbol{b})}$ denote the polynomial opening proof submitted during the distributed polynomial commitment opening protocol. The coordinator verifies: $e(\mathsf{com} f^{(\boldsymbol{b})}, g) \cdot e(g^{-z^{(\boldsymbol{b})}}, V_{\boldsymbol{b}}) = \prod_{i\in[\mu]} e(\pi_i^{(\boldsymbol{b})}, g^{\tau_i-\alpha_i})$. Any prover $\mathcal{P}_{\boldsymbol{b}}$ whose verification fails at this step is immediately identified as malicious.
3) **Stage 2: Verifying witness polynomial evaluations.** If all polynomial openings verify correctly in Stage 1, the coordinator checks for discrepancies in witness polynomial evaluations. Let $F^{(\boldsymbol{b})}$ denote the witness polynomial for the subcircuit indexed by $\boldsymbol{b}$, and let $\boldsymbol{r}$ be the randomness used in step 3 of the distributed SumCheck protocol. Each prover $\mathcal{P}_{\boldsymbol{b}}$ previously sent the evaluation $F^{(\boldsymbol{b})}(\boldsymbol{r})$ to the coordinator. The coordinator independently evaluates each polynomial $F^{(\boldsymbol{b})}$ at point $\boldsymbol{r}$ using the correct witness data it holds and compares the results. Any discrepancy identifies the worker $\mathcal{P}_{\boldsymbol{b}}$ as malicious.

The correctness of the above protocol is shown in the following proposition.

Proposition 5. *If the final proof sent to the verifier cannot verify, in the accountability protocol described above, an honest coordinator can identify at least one malicious worker with overwhelming probability.*

Proof. Assume the coordinator is honest and the final aggregated proof sent to the verifier fails to verify. Let the set of workers be $\{\mathcal{P}_{\boldsymbol{b}}\}_{\boldsymbol{b}\in B_\xi}$.

During Stage 1 each worker $\mathcal{P}_{\boldsymbol{b}}$ supplies an opening proof $\boldsymbol{\pi}^{(\boldsymbol{b})}$ for the commitment $\mathsf{com} f^{(\boldsymbol{b})}$. Then the coordinator checks

$$e(\mathsf{com} f^{(\boldsymbol{b})}, g) \cdot e(g^{-z^{(\boldsymbol{b})}}, V_{\boldsymbol{b}}) = \prod_{i\in[\mu]} e(\pi_i^{(\boldsymbol{b})}, g^{\tau_i-\alpha_i}).$$

By the soundness of the KZG PCS, an incorrect opening passes this check with only negligible probability. If the check fails, the corresponding $\mathcal{P}_{\boldsymbol{b}}$ must be malicious.

Suppose every worker passes Stage 1. Note that only the witness polynomial $F^{(\boldsymbol{b})}$ is committed by each worker instead of preprocessed. Then each commitment $\mathsf{com}_{f^{(\boldsymbol{b})}}$ corresponds to *some* degree-bounded polynomial $F^{(\boldsymbol{b})}$, and the opening value $z^{(\boldsymbol{b})}$ is consistent with that commitment. By completeness of the PermTest, SumCheck protocols, this can only happen if at least one $F^{(\boldsymbol{b})}$ is not the *correct* witness polynomial $F^{(\boldsymbol{b})}_{\text{true}}$.

Because the (incorrect) commitment was fixed *before* the challenge point $\boldsymbol{r}$ was chosen, by Schwartz-Zippel lemma

$$\Pr\left[F^{(\boldsymbol{b})}(\boldsymbol{r}) = F^{(\boldsymbol{b})}_{\text{true}}(\boldsymbol{r})\right] \le \frac{\deg(F^{(\boldsymbol{b})}_{\text{true}})}{|\mathbb{F}|},$$

which is negligible. In Stage 2 the coordinator recomputes $F^{(\boldsymbol{b})}_{\text{true}}(\boldsymbol{r})$ from the stored plaintext circuit evaluation and compares it to the worker's reported $F^{(\boldsymbol{b})}(\boldsymbol{r})$. Any discrepancy exposes $\mathcal{P}_{\boldsymbol{b}}$ as malicious.

Since the failure probability in each stage is negligible, an honest coordinator identifies at least one malicious worker with overwhelming probability. □

C. Hierarchical aggregation

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 hierarchical aggregation. The modified protocol steps are highlighted in blue.

Distributed KZG PCS with hierarchical aggregation. For a multilinear polynomial $f(\boldsymbol{x}, \boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\xi} f^{(\boldsymbol{b})}(\boldsymbol{x})\chi_{\boldsymbol{b}}(\boldsymbol{y})$, the PCS protocol is described as follows:

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

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

Then it computes $\pi_{\mu+j} = g^{q_j(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$ with $V_{\boldsymbol{b}}$.

5) $\mathcal{C}$ sets $\boldsymbol{\pi} := (\pi_1, \ldots, \pi_{\mu+\xi})$.

Distributed SumCheck with hierarchical aggregation. Using the previously presented KZG PCS with hierarchical aggregation we can build a distributed SumCheck with hierarchical aggregation as follows:

1) For each $i \in [\mu]$:
   a) For $\boldsymbol{b} \in B_\xi$, $\mathcal{P}_{\boldsymbol{b}}$ computes $r_i^{(\boldsymbol{b})}(x) := \sum_{\boldsymbol{w}\in B_{\mu-i}} f^{(\boldsymbol{b})}(\alpha_1, \ldots, \alpha_{i-1}, x, \boldsymbol{w})$. $\mathcal{P}_{\boldsymbol{b}}$ sends $r_i^{(\boldsymbol{b})}$ to both the leader of its group and $\mathcal{C}$.
   b) The leader of each group sums up the $r_i^{(\boldsymbol{b})}$ it receives, and sends the result together with all communication to the coordinator. In the case where the leader does not receive $r_i^{(\boldsymbol{b})}$ from a specific worker, it communicates with the coordinator to send a dispute. $\mathcal{C}$ sends the oracle of r_i to $\mathcal{V}$.
   c) $\mathcal{V}$ checks if $v = r_i(0) + r_i(1)$, and samples and sends a random $\alpha_i \leftarrow \mathbb{F}$ to $\mathcal{C}$. Then $\mathcal{V}$ sets $v := r_i(\alpha_i)$. $\mathcal{C}$ sends α_i to each $\mathcal{P}_{\boldsymbol{b}}$.
2) $\mathcal{C}$ receives $v^{(\boldsymbol{b})} := f^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ from each $\mathcal{P}_{\boldsymbol{b}}$. $\mathcal{C}$ defines $g(\boldsymbol{y}) := f(\boldsymbol{\alpha}, \boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\xi} v^{(\boldsymbol{b})} \cdot \chi_{\boldsymbol{b}}(\boldsymbol{y})$.
3) $\mathcal{C}$ and $\mathcal{V}$ perform SumCheck on g with target value v. $\mathcal{C}$ and the worker provers open $f(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

Dispute Control. After every invocation of the hierarchical aggregation protocols, the coordinator $\mathcal{C}$ executes the following *dispute control protocol* after identifying a malicious leader group using our accountability protocol. Its goal is to identify at least one misbehaving party (worker or leader). This dispute control is necessary, since a worker may have sent the correct value, but their leader may tamper with the communicated value sent to the coordinator.

1) **Collection of messages.** Every leader in group j has already forwarded to $\mathcal{C}$:
   - Its own partial sum, where for KZG this is $\sum_{\boldsymbol{b}\in\text{group } j} \boldsymbol{\pi}^{(\boldsymbol{b})}$, and for SumCheck it is $\sum_{\boldsymbol{b}\in\text{group } j} r_i^{(\boldsymbol{b})}$.
   - The *entire set* of messages $\{(m^{(\boldsymbol{b})})\}_{\boldsymbol{b}\in\text{group } j}$ that it received from its workers,

   where $m^{(\boldsymbol{b})}$ denotes either $(\boldsymbol{\pi}^{(\boldsymbol{b})}, z^{(\boldsymbol{b})})$ or $r_i^{(\boldsymbol{b})}$.
2) **Recognition of worker values.** For every worker $\mathcal{P}_{\boldsymbol{b}}$, $\mathcal{C}$ confirms the value that this worker sent to the coordinator in the previous protocol, denoted $\widetilde{m}^{(\boldsymbol{b})}$, using the communication sent to the coordinator.
3) **Leader consistency check.** For each group j, compare the leader's reported sum S_j with $\widetilde{S}_j$:
   - If $S_j = \widetilde{S}_j$, the leader passes the check.
   - Otherwise the leader j is marked malicious.
4) **Detection of double behaviour.** For any worker $\mathcal{P}_{\boldsymbol{b}}$ that sent *two different* signed messages—one to the leader and one directly to $\mathcal{C}$—the inconsistent pair of signatures is sufficient evidence of malicious behaviour, even if neither message was individually incorrect.
5) **Accountability check.** The leader finally runs the accountability check protocol, to further identify any malicious worker.

We summarize the accountability guarantee of the dispute control protocol in the following proposition.

Proposition 6. *The dispute control protocol satisfies that, if any worker or leader deviates from the prescribed protocol, either by sending incorrect values, omitting messages, or equivocating, the coordinator will identify at least one malicious party.*

Proof. 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 in the accountability check step.
- If the leader submits an incorrect addition, the coordinator can detect the leader's error by recomputing the additions based on the values submitted directly by the workers to the coordinator in the leader consistency check step.
- 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 in the detection of double behavior step.

Since all the possible cases are covered, the coordinator will always be able to identify the malicious node, ensuring accountability in the protocol with dispute control. □

The Cirrus distributed SNARK. Cirrus, the accountable and efficiently computable distributed SNARK, is structured as follows. Note that all verifier challenges are replaced using the Fiat-Shamir transform by the coordinator.

- **Input Constraint**: $F^{(\boldsymbol{b})}(1, \ldots, 1, \boldsymbol{x}) - I^{(\boldsymbol{b})}(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in B_\nu$ and $\boldsymbol{b} \in B_\xi$ with distributed multilinear ZeroTest.
- **Output Constraint**: Open $F^{(1,\ldots,1)}(1, 0, \langle N^{(\boldsymbol{b})} - 1\rangle_\mu)$ and show that it is 0.
- **Gate Constraint**: Define a multivariate polynomial

  $$G^{(\boldsymbol{b})}(\boldsymbol{x}) := S^{(\boldsymbol{b})}_{\mathsf{add}}(\boldsymbol{x})(F^{(\boldsymbol{b})}(0, 0, \boldsymbol{x}) + F^{(\boldsymbol{b})}(0, 1, \boldsymbol{x})) + S^{(\boldsymbol{b})}_{\mathsf{mult}}(\boldsymbol{x})(F^{(\boldsymbol{b})}(0, 0, \boldsymbol{x}) \cdot F^{(\boldsymbol{b})}(0, 1, \boldsymbol{x})) - F^{(\boldsymbol{b})}(1, 0, \boldsymbol{x}).$$

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

- **Wiring Constraint**: Check if $F^{(\boldsymbol{b})}(\boldsymbol{x}) = F^{(\rho^{(\boldsymbol{b})}(\boldsymbol{x}))}(\sigma^{(\boldsymbol{b})}(\boldsymbol{x}))$ for all $\boldsymbol{x} \in B_\mu$ and $\boldsymbol{b} \in B_\xi$ with distributed multivariate PermTest.
- **Accountability Check**: At the end the protocol, the coordinator runs the efficient accountability protocol presented in Section III-B.

Theorem 1 (Main Theorem)**.** *Cirrus is an accountable distributed SNARK (Definition 1).*

Proof. We now instantiate the interfaces in Definition 1 using the protocols from Sections III-A to III-C. This yields a concrete, accountable distributed SNARK.

$\mathsf{Setup}(1^\lambda)$.

1) Run KeyGen of distributed multilinear KZG and generate all parameters needed for creating and verifying polynomial commitments.
2) Fix a signature scheme Sig with $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vfy})$.
3) Fix a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{F}$ which will be used in the Fiat-Shamir transform.

$\mathsf{KeyGen}(\mathsf{C}, \mathsf{pp}, M)$. Let $M = 2^\xi$ and $2^\mu = |\mathsf{C}|/M$.

1) Sample $\tau_1, \ldots, \tau_{\mu+\xi} \xleftarrow{\$} \mathbb{F}$ and define $U_{\boldsymbol{c},\boldsymbol{b}} := g^{\chi_{\boldsymbol{c}}(\tau_1,\ldots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$, and $V_{\boldsymbol{b}} := g^{\chi_{\boldsymbol{b}}(\tau_{\mu+1},\ldots,\tau_{\mu+\xi})}$ for all $\boldsymbol{c} \in B_\mu, \boldsymbol{b} \in B_\xi$. Set $\mathsf{crs} := (g, \{g^{\tau_i}\}_{i\in[\mu+\xi]}, \{U_{\boldsymbol{c},\boldsymbol{b}}\}_{\boldsymbol{c},\boldsymbol{b}})$.
2) Compute the circuit-dependent wiring maps $\{\sigma^{(\boldsymbol{b})}, \rho^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_\xi}$ and selectors $\{S^{(\boldsymbol{b})}_{\mathsf{add}}, S^{(\boldsymbol{b})}_{\mathsf{mult}}\}_{\boldsymbol{b}\in B_\xi}$ as in Section III-A.
3) Each party $P \in \{\mathcal{C}\} \cup \{\mathcal{P}_{\boldsymbol{b}} : \boldsymbol{b} \in B_\xi\}$ runs Sig.KeyGen to obtain $(\mathsf{sk}_P, \mathsf{vk}_P)$.
4) Set the coordinator key $\mathsf{pk} := (\mathsf{crs}, \{V_{\boldsymbol{b}}\}_{\boldsymbol{b}\in B_\xi}, \{\sigma^{(\boldsymbol{b})}, \rho^{(\boldsymbol{b})}\}_{\boldsymbol{b}}, \{S^{(\boldsymbol{b})}_{\mathsf{add}}, S^{(\boldsymbol{b})}_{\mathsf{mult}}\}_{\boldsymbol{b}}, \mathsf{H})$.
5) For each worker $\mathcal{P}_{\boldsymbol{b}}$, set $\mathsf{pk}_{\boldsymbol{b}} := (\mathsf{crs}, \sigma^{(\boldsymbol{b})}, \rho^{(\boldsymbol{b})}, S^{(\boldsymbol{b})}_{\mathsf{add}}, S^{(\boldsymbol{b})}_{\mathsf{mult}})$.
6) The verifier key contains the CRS part needed for verification and all signature verification keys: $\mathsf{vk} := (\mathsf{crs}, \{V_{\boldsymbol{b}}\}_{\boldsymbol{b}\in B_\xi}, \{\mathsf{vk}_P\}_P, \mathsf{H})$.

$\mathsf{Partition}(\mathsf{C}, M)$.

1) Evenly split the PLONK trace t into $M = 2^\xi$ chunks $\{t^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_\xi}$.
2) For each chunk, define the sub-circuit $\mathsf{C}_{\boldsymbol{b}}$ whose trace is $t^{(\boldsymbol{b})}$ and whose wiring is given by $\sigma^{(\boldsymbol{b})}, \rho^{(\boldsymbol{b})}$.

$\mathsf{SplitInst}(\mathsf{pp}, \mathsf{C}, \boldsymbol{x}, \boldsymbol{w})$.

1) Partition the global input/witness (x, w) into per-segment pairs $(x_{\boldsymbol{b}}, w_{\boldsymbol{b}})$ consistent with $t^{(\boldsymbol{b})}$.
2) For each $\boldsymbol{b} \in B_\xi$, compute the local oracles $S^{(\boldsymbol{b})}_{\mathsf{add}}, S^{(\boldsymbol{b})}_{\mathsf{mult}}$, $F^{(\boldsymbol{b})}$, and $I^{(\boldsymbol{b})}$ as described in Section III-A.

$\mathsf{Prove}_{\boldsymbol{b}}(\mathsf{pk}_{\boldsymbol{b}}, \mathsf{C}_{\boldsymbol{b}}, \boldsymbol{x}_{\boldsymbol{b}}, \boldsymbol{w}_{\boldsymbol{b}})$. Worker $\mathcal{P}_{\boldsymbol{b}}$ executes:

1) *Commit*. Interpolate $F^{(\boldsymbol{b})}$ and compute $\mathsf{com}_{\boldsymbol{b}} := \prod_{\boldsymbol{c}\in B_\mu} U_{\boldsymbol{c},\boldsymbol{b}}^{F^{(\boldsymbol{b})}(\boldsymbol{c})}$. Send $(\mathsf{com}_{\boldsymbol{b}}, \mathsf{Sign}_{\mathsf{sk}_{\boldsymbol{b}}}[\mathsf{com}_{\boldsymbol{b}}])$ to $\mathcal{C}$.
2) *Distributed Poly-IOPs*. In each distributed protocol (multilinear ZeroTest, PermTest, and SumCheck):
   - Compute the required local univariates $r_i^{(\boldsymbol{b})}$ or auxiliary polynomials ($w_1^{(\boldsymbol{b})}$, $w_2^{(\boldsymbol{b})}$, etc.);
   - Send them to $\mathcal{C}$ (and to the group leader if hierarchical aggregation is used), signed with $\mathsf{sk}_{\boldsymbol{b}}$;
   - When an opening of $F^{(\boldsymbol{b})}$ at $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is requested, compute $z^{(\boldsymbol{b})} := F^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ and the witnesses $\{\pi_i^{(\boldsymbol{b})}\}_{i\in[\mu]}$ and send the signed tuple.

$\mathsf{Prove}(\mathsf{pk}, \mathsf{C}, \boldsymbol{x}, \boldsymbol{w})$. The coordinator $\mathcal{C}$:

1) *Aggregate commitments*. Collect all $\{\mathsf{com}_{\boldsymbol{b}}\}_{\boldsymbol{b}}$, check their signatures, and set $\mathsf{com} := \prod_{\boldsymbol{b}} \mathsf{com}_{\boldsymbol{b}}$. Initialize the proof π with com.
2) *Run distributed Poly-IOPs*. For each constraint family:
   - **Input** and **Gate** constraints: run distributed ZeroTest with workers as in Section III-A, aggregating $r_i := \sum_{\boldsymbol{b}} r_i^{(\boldsymbol{b})}$ (or leaders' partial sums) and forwarding them to $\mathcal{V}$. Append all communication to $\mathcal{V}$ to π instead of sending them to $\mathcal{V}$, and replace verifier challenges using the Fiat-Shamir transform with the hash function H.
   - **Wiring** constraint: run distributed PermTest over the $F^{(\boldsymbol{b})}$'s; build z from $z^{(\boldsymbol{b})}$ and check the product constraint as in Section III-A. Append all communication to $\mathcal{V}$ to π instead of sending them to $\mathcal{V}$, and replace verifier challenges using the Fiat-Shamir transform with the hash function H.
   - **Output** constraint: open $F^{(1,\ldots,1)}(1, 0, \langle N^{(\boldsymbol{b})} - 1\rangle_\mu)$, and append the opening proof to π.
3) *Openings*. Whenever the Poly-IOP requires a polynomial opening, compute $\pi_{\mu+j}$ using $V_{\boldsymbol{b}}$ and aggregate $\pi_i := \prod_{\boldsymbol{b}} \pi_i^{(\boldsymbol{b})}$, and then append $(z, \boldsymbol{\pi})$ to π.
4) *Output*. Output the proof π to the verifier.
5) *Transcript*. Log all worker messages into a transcript tr for accountability checks and dispute control.

$\mathsf{CheckAndAccuse}(\mathsf{pk}, \mathsf{C}, \boldsymbol{x}, \boldsymbol{w}, \mathsf{tr})$. Executed by $\mathcal{C}$ *only if* $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 0$: given the transcript tr and the circuit witness, run the accountability protocol in Section III-B, and the dispute control protocol in Section III-C if hierarchical aggregation is used.

$\mathsf{Verify}(\mathsf{vk}, \boldsymbol{x}, \boldsymbol{\pi})$. Given the public input x and proof π, the verifier:

1) Parses π into the global commitment com, the SumCheck univariates, the Fiat-Shamir challenges, and the KZG opening proofs.
2) Recomputes all challenges in order using the same hash function H.
3) For each sumcheck instance, checks the condition $v_{i-1} = r_i(0) + r_i(1)$ and the final evaluation consistency.
4) For each ZeroTest and PermTest, performs the stated final point checks using the derived challenges.
5) For each polynomial opening, verify the distributed KZG pairing equation using $(\mathsf{com}, z, \boldsymbol{\pi})$.
6) Output $b = 1$ iff all checks pass; otherwise output $b = 0$.

The completeness and soundness of Cirrus follows the completeness and knowledge soundness of distributed KZG

(Proposition 1), distributed SumCheck (Proposition 2), distributed ZeroTest (Proposition 3), and distributed PermTest (Proposition 4). It is shown in Lemma 2.3 in HyperPlonk [8] that sound Poly-IOPs are knowledge sound, so we have that Cirrus is knowledge sound. Existing works [31], [32] show that replacing the verifier challenges with Fiat-Shamir transform is secure for multi-round special-sound protocols and multi-round oracle proofs. The accountability of Cirrus follows Propositions 5 and 6, since we have shown the protocol for the coordinator to identify at least one malicious worker if the final proof does not successfully verify. Therefore, Cirrus is an accountable distributed SNARK as defined in Definition 1. □

ZK for Cirrus. To turn Cirrus into a zero-knowledge protocol, we apply an additive mask to every distributed SumCheck (and therefore to every ZeroTest and PermTest): for each target polynomial f we sample an independent random mask g, and the verifier sends a random scalar $\rho \in \mathbb{F}^*$ so that the protocol is run on $f + \rho g$ instead of f. Intuitively, in the simplest case where f is multilinear, a single multilinear mask g already hides all partial sums, and the verifier only needs to check $f + \rho g$ at one random point at the end. In our multivariate setting where f may have a higher per-variable degree, we choose g as a low-degree product of independent multilinear masks so that $\deg g$ matches the degree of f. At the PCS layer, commitments are blinded by adding a random polynomial mask committed under a different basis. In the ZK protocol, accountability holds unchanged, and both the worker/coordinator asymptotic costs remain identical to the non-ZK protocol. The ZK compilation of the distributed SumCheck and the distributed KZG PCS, together with the proofs, appears in Section A of the appendix.

IV. Implementation and Evaluation

A. Implementation Details

We implement an end-to-end prototype of Cirrus, including the accountability protocol. Our implementation is based on the codebase of HyperPlonk [24], and we added $5,000$+ lines of Rust code. Our code is publicly accessible via this link.

Circuit partition. An advantage of Cirrus, in addition to its high efficiency and its accountability, is that it automatically distributes workloads across workers, eliminating the need for developers to explicitly specify circuit partitions.

Universal setup. Cirrus requires only a universal setup and is fully compatible with the existing HyperPlonk setup parameters. This feature lowers the adoption barrier of Cirrus, because a setup ceremony is expensive to organize [22].

In contrast, prior schemes such as Hekaton require developers to manually partition circuits into sub-circuits, explicitly manage shared wires between partitions, and perform separate trusted setups for each partition, significantly increasing complexity and overhead.

B. Evaluation Setup

To assess the practicality of running Cirrus in a decentralized environment (e.g., by individual volunteers), we benchmark Cirrus on hardware comparable to personal computers. 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 distributed setting 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.

C. End-to-end Proof Generation Evaluation

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. We select random circuits because the overall proof generation time depends only on the number of gates and is independent of the circuit structure. Then, we run Cirrus with up to 32 machines (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} (4M) gates when it runs out of memory (32 GB). In comparison, Cirrus can support larger circuits with more machines. We stopped at 2^{25} (33M) gates with 8 AWS machines since the horizontal scalability is clear.

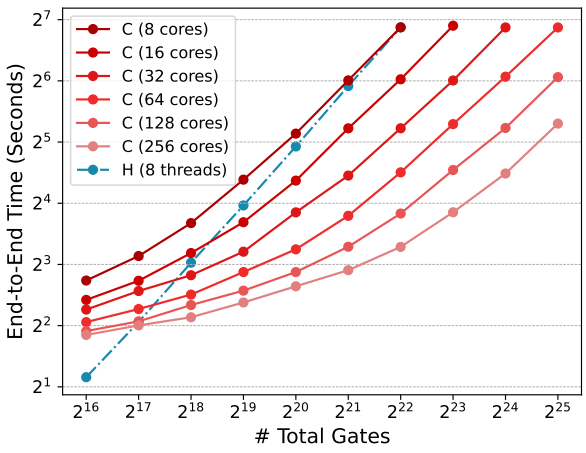


Fig. 1: End-to-end proof generation time of Cirrus when all workers are honest with different total cores and circuit sizes. C denotes Cirrus; H denotes HyperPlonk.

Example application: verifiable RAM programs. To put the numbers in context, we consider an example application of distributed proof in verifiable RAM programs, also known as zkVMs [25]–[29]. To evaluate Cirrus in this setting, we have

our circuit for RAM programs following the design of Ben-Sasson et al. [25] with 32-bit words and 16 registers; a RAM program is represented as a repetition of a single "CPU cycle" sub-circuit. We then distribute these cycles evenly across 256 cores and measure end-to-end proving time. Fig. 2 compares Cirrus running on 256 cores with a multi-threaded HyperPlonk baseline. Cirrus sustains near-linear scaling in the number of cycles and can prove a trace of 2^{14} CPU steps in under 40 seconds.

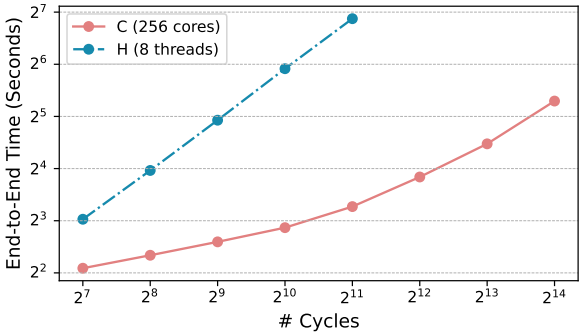


Fig. 2: Runtime of Cirrus with 256 cores compared with HyperPlonk when proving RAM programs with different numbers of cycles. C denotes Cirrus; H denotes HyperPlonk.

Communication, proof size, and verifier time. We also evaluated the communication overhead, proof size, and the verifier's runtime. The communication of each worker increases logarithmically with the sub-circuit size T. Each worker sends $(1027 \cdot \log T + 2438)$ B to the coordinator, and receives $(520 \cdot \log T + 3576)$ B from the coordinator, in $(3 \cdot \log T + 5)$ communication rounds. In our experiments, the communication cost is capped at $T = 2^{22}$, where each worker receives 25.0 KB of data and sends 15.0 KB in 71 rounds.

The proof size is $(616 \cdot \log N + 1232)$ B, which grows logarithmically with the full circuit size N. For our largest circuit, $N = 2^{25}$, this corresponds to a proof size of about 16.6 KB. The verifier time of Cirrus also grows logarithmically in N; for $N = 2^{25}$ gates, we measure a verifier time of 28 ms. In Table II, we compare worker communication cost, proof size, and verifier time with Pianist and Hekaton. Worker communication is lightweight, and both the proof size and the verifier time are succinct. Cirrus outperforms the other accountable distributed SNARK, Hekaton, in these aspects.

Protocol	Sent	Received	Rounds	π Size	$\mathcal{V}$ Time
Cirrus	15.0 KB	25.0 KB	71	16.6 KB	28 ms
Pianist [16]	2.1 KB	0.2 KB	4	2.8 KB	3.5 ms
Hekaton [17]	923 KB	0.6 KB	2	32 KB	83 ms

TABLE II: The per-worker communication cost (bytes sent to and received from the coordinator, and number of communication rounds), proof size, and verifier time for Cirrus with sub-circuit size $T = 2^{22}$ and full circuit size $N = 2^{25}$, compared with Pianist and Hekaton.

Memory usage. We record how each worker's memory usage varies with the worker's sub-circuit size in Table IIIa. 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 IIIb, which only depends on the size of the full circuit.

Sub-Circuit Size	Memory
2^{16}	383 MB
2^{17}	765 MB
2^{18}	1.4 GB
2^{19}	2.8 GB
2^{20}	5.6 GB
2^{21}	11.3 GB
2^{22}	22.6 GB

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

Full Circuit Size	Memory
2^{19}	200 MB
2^{20}	396 MB
2^{21}	799 MB
2^{22}	1.5 GB
2^{23}	3.0 GB
2^{24}	6.0 GB
2^{25}	12.1 GB

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

TABLE III: Memory usage of workers and the coordinator.

Comparison with Hekaton. From the above evaluation results, we know that the primary performance metric of interest is the proof time. To understand the performance of Cirrus further, we compare it with Hekaton, the fastest distributed SNARK system (reportedly outperforms Pianist by 3×) and the only accountable distributed SNARK known at the time of writing. We obtained the source code from the authors and evaluated Hekaton on the same machines used for our Cirrus experiments. For reference, we report relevant parameters in Table IV.

Protocol	Curve Choice	Security Target	Circuit Encoding
Cirrus	BLS12-381	128 bit	PLONK
Hekaton [17]	BLS12-381	128 bit	R1CS

TABLE IV: Parameter comparison between Cirrus and Hekaton.

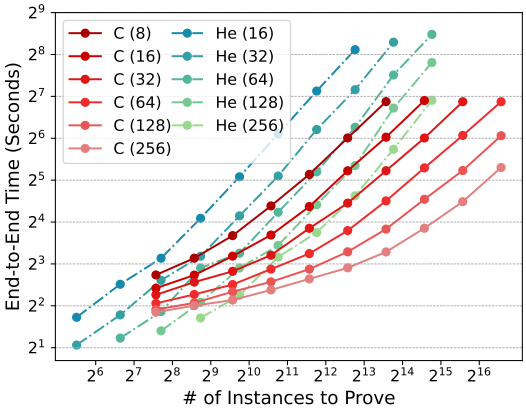
We compare the two systems across a range of tasks to show practical performance differences. We tested three tasks: Pedersen hashing [33], MiMC hashing [34], and proof of knowledge of exponent (PoK of Exp). In each case, we repeated the task multiple times in the circuit to test the scalability of both protocols when they are required to prove a large number of instances. The results are shown in Fig. 3. We highlight:

- for Pedersen hashing, Cirrus is over 7× faster;
- for MiMC hashing, Cirrus is about 2× faster;
- for PoK of Exponent, Cirrus is around 4× faster.

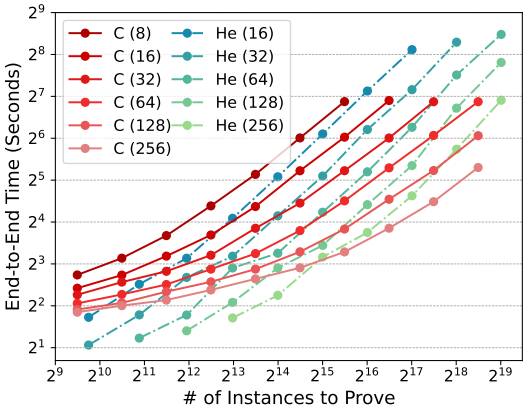
These results show that Cirrus performs well in these real-world applications, especially for PLONK-friendly tasks. We also emphasize that, unlike Hekaton, Cirrus supports a universal trusted setup and does not require any per-circuit configuration, and is therefore more flexible to deploy.

D. Evaluation of The Accountability Protocol

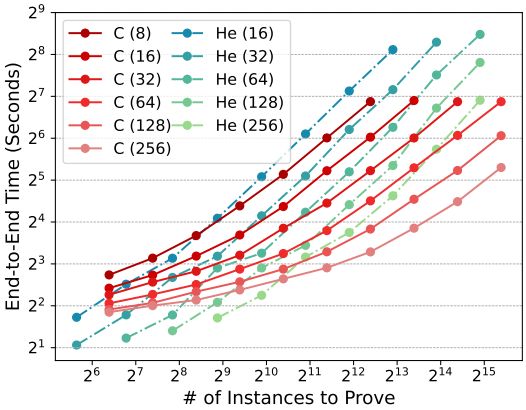
Accountability protocol runtime. A key advantage of Cirrus is the high efficiency of its accountability protocol (see Section III-B). We implement the accountability protocol and evaluate the time required by the coordinator to identify malicious worker(s) for various circuit sizes and numbers of workers. As shown in Fig. 4, the accountability check is extremely fast in practice compared with the time to generate



(a) Comparison between Cirrus and Hekaton to prove Pedersen Hashes.



(b) Comparison between Cirrus and Hekaton to prove MiMC hashes.



(c) Comparison between Cirrus and Hekaton to prove PoK of exponent tasks.

Fig. 3: Comparison between Cirrus and Hekaton on different tasks. C denotes Cirrus, and He denotes Hekaton. The total number of working cores is shown in the parentheses.

the proof. For circuits of 2^{25} total gates and the distribution of 256 workers, the accountability protocol takes under 4 seconds to run. This demonstrates that our accountability mechanism imposes minimal overhead on the coordinator compared with the time to generate the proof.

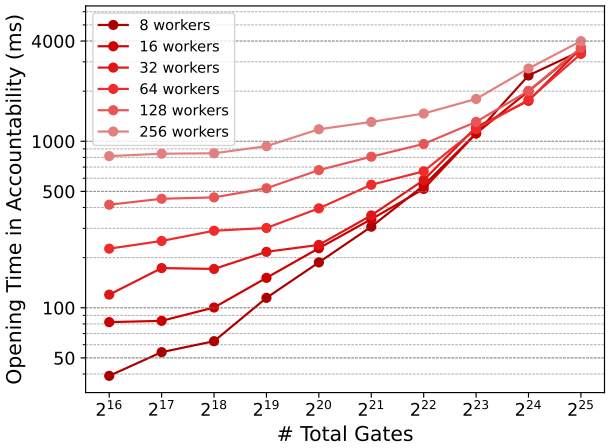


Fig. 4: The time to run the accountability protocol with different numbers of workers and full circuit sizes.

Circuit evaluation cost. Before or during proving, the coordinator needs to evaluate the circuit in plaintext to obtain the witness trace and store it in the memory, which is used later by our optimistic accountability protocol if malicious workers are present. It does not violate the security assumptions, since in Section II the coordinator is assumed to have access to the witness. We further note that it does not create a bottleneck for the coordinator. If the coordinator is also a worker, the evaluation has already been performed locally; if not, the coordinator can evaluate the circuit while waiting for messages from the workers. On our coordinator hardware (AWS `t3.2xlarge`), plaintext circuit evaluation for 2^{25} gates takes 17 seconds; the coordinator's idle time during proof generation is 39 seconds with 256 workers, so the coordinator has enough time during proof generation to complete this evaluation without increasing the wall-clock proving time.

E. Benefits of Hierarchical Aggregation

Coordinator's computation time. A feature of Cirrus is that the coordinator's computation is lightweight thanks to hierarchical aggregation (see Section III-C). We measure the coordinator's computation time in Fig. 5. Without hierarchical aggregation, the computation time of the coordinator increases as the sub-circuit size increases. With our hierarchical aggregation 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 fewer than $10,000$ workers.

Leader worker overhead in hierarchical aggregation. We evaluate the overhead of leader workers and non-leader workers

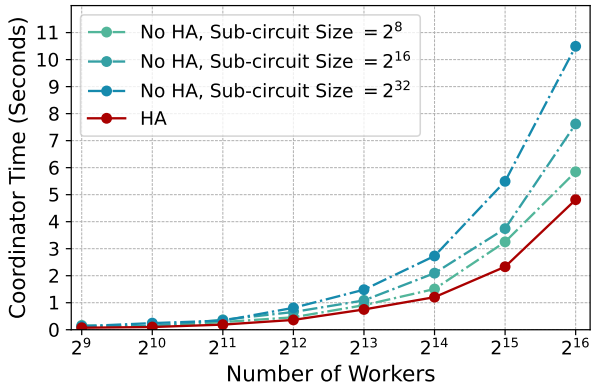


Fig. 5: Computation time of the coordinator with varying number of workers and sub-circuit sizes. In this figure, "HA" denotes the coordinator time with hierarchical aggregation, while "No HA" denotes the coordinator time without hierarchical aggregation.

in hierarchical aggregation. Each leader worker will aggregate intermediate results of $\log T$ workers, where T is the size of the sub-circuit. In our experiments, the overhead of leader workers is less than 22 ms when $T = 2^{22}$. It can be inferred that hierarchical aggregation adds little overhead to leader workers.

V. 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 I.

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 [35] 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. 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. 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 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 time (in the size of the sub-circuit) and stronger accountability for general circuits (not just data-parallel circuits).

Mangrove. Nguyen et al. [36] 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. Rosenberg et al. [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 shared wires among sub-circuits.

HyperPianist. 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 soundness or a thorough performance evaluation.

SNARK aggregation schemes. A series of works [37]–[39] 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* [40]–[43] 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.

VI. 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 our accountability protocol, the coordinator can efficiently identify any malicious prover, making Cirrus suitable for deployment in decentralized settings, e.g., in prover markets, where workers 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 subcircuit 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 the accountability protocol. Currently, identifying malicious provers in stage 2 of the accountability protocol requires a number of field operations linear in the size of the entire circuit. While the cost of the accountability protocol is acceptable in most settings, optimizing this step would make it even more scalable for larger circuits.

Acknowledgement

This project is supported in part by the Ethereum Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes.

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. Venkitasubramaniam, "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 srs," 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. ITCS '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. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and field-agnostic snarks for r1cs," in *Annual International Cryptology Conference*. Springer, 2023, pp. 193–226.

[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] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[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," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1777–1793.

[17] M. Rosenberg, T. Mopuri, H. Hafezi, I. Miers, and P. Mishra, "Hekaton: Horizontally-scalable zksnarks via proof aggregation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 929–940.

[18] C. Li, P. Zhu, Y. Li, C. Hong, W. Qu, and J. Zhang, "HyperPianist: Pianist with linear-time prover and logarithmic communication cost," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3383–3401.

[19] W. Wang, L. Zhou, A. Yaish, F. Zhang, B. Fisch, and B. Livshits, "Prooφ: A zkp market mechanism," *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/zcash-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] Espresso Systems, "Hyperplonk library," accessed: 2024-11-13. [Online]. Available: https://github.com/EspressoSystems/hyperplonk

[25] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.

[26] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Annual cryptology conference*. Springer, 2013, pp. 90–108.

[27] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vRAM: Faster verifiable ram with program-independent preprocessing,"

in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 908–925.
[28] S. Team, "ethstark documentation–version 1.1," IACR preprint archive 2021, Tech. Rep., 2021.
[29] A. Arun, S. Setty, and J. Thaler, "Jolt: Snarks for virtual machines via lookups," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2024, pp. 3–33.
[30] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying arbitrary sql queries over dynamic outsourced databases," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 863–880.
[31] T. Attema, S. Fehr, and M. Klooß, "Fiat-shamir transformation of multi-round interactive proofs," in *Theory of Cryptography Conference*. Springer, 2022, pp. 113–142.
[32] D. Wikström, "Special soundness in the random oracle model," *Cryptology ePrint Archive*, 2021.
[33] 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.
[34] 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.
[35] 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.
[36] W. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh, "Mangrove: A scalable framework for folding-based snarks," in *Annual International Cryptology Conference*. Springer, 2024, pp. 308–344.
[37] 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.
[38] M. Ambrona, M. Beunardeau, A.-L. Schmitt, and R. R. Toledo, "aPlonK: Aggregated plonk from multi-polynomial commitment schemes," in *International Workshop on Security*. Springer, 2023, pp. 195–213.
[39] 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.
[40] 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.
[41] 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.
[42] 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.
[43] X. Liu, Z. Zhou, Y. Wang, Y. Pang, J. He, B. Zhang, X. Yang, and J. Zhang, "Scalable collaborative zk-snark and its application to fully distributed proof delegation," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

APPENDIX A
ZERO-KNOWLEDGE COMPILATION FOR CIRRUS

In this section, we show a zero-knowledge (ZK) compilation for Cirrus. We first describe the ZK distributed SumCheck protocol and prove its security. Then we construct the other ZK Poly-IOPs using ZK distributed SumCheck. Finally, we show the construction of the ZK Cirrus by combining the ZK distributed KZG PCS.

ZK distributed SumCheck. Let $m := \mu + \xi$, $\boldsymbol{Z} = (\boldsymbol{x}, \boldsymbol{b}) \in \mathbb{F}^m$, and let $f(\boldsymbol{Z}) = h\big(g_1(\boldsymbol{Z}), \ldots, g_c(\boldsymbol{Z})\big)$ be the target with per-round variable degree bound d_f. Fix a masking degree $d_{\mathsf{mask}} \geq d_f$. Sample d_{mask} independent spikes $\{\boldsymbol{w}^{(t)}\}_{t=1}^{d_{\mathsf{mask}}} \xleftarrow{\$} \mathbb{F}^m$ and define $g'_t(\boldsymbol{Z}) := \chi_{\boldsymbol{w}^{(t)}}(\boldsymbol{Z})$ and $g(\boldsymbol{Z}) := \prod_{t=1}^{d_{\mathsf{mask}}} g'_t(\boldsymbol{Z}) = h'\big(g'_1(\boldsymbol{Z}), \ldots, g'_{d_{\mathsf{mask}}}(\boldsymbol{Z})\big)$, where $h'(v_1, \ldots, v_{d_{\mathsf{mask}}}) = \prod_{t=1}^{d_{\mathsf{mask}}} v_t$. Thus g has variable degree d_{mask} and each g'_t is multilinear, so g is a constant-degree composition of multilinears and is compatible with our distributed multilinear-KZG pipeline.

Let $H := \sum_{\boldsymbol{Z} \in B_m} f(\boldsymbol{Z})$ and $G := \sum_{\boldsymbol{Z} \in B_m} g(\boldsymbol{Z})$. We run SumCheck on $f + \rho g$ with target $H + \rho G$. Below we spell out the full distributed protocol, matching the two-phase structure in Section III-A.

Public blinder and scalar.

1) (*Public blinder*) $\mathcal{C}$ samples the spikes $\{\boldsymbol{w}^{(t)}\}_{t=1}^{d_{\mathsf{mask}}}$ and sends them (or a seed) to all workers. The provers run distributed Commit to each multilinear g'_t (using Section III-A) and send the commitments to $\mathcal{V}$ to bind the blinder polynomials.
2) (*Verifier's scalar*) $\mathcal{V}$ samples $\rho \xleftarrow{\$} \mathbb{F}^*$ and sends it to $\mathcal{C}$ (public-coin). This fixes the blinded target $f + \rho g$.

Phase I (rounds over $\boldsymbol{x}$; workers $\rightarrow$ coordinator). Let $v_0 := H + \rho G$ be the initial target. For each $i \in [\mu]$ do:

1) Let $\boldsymbol{\alpha}_{i-1} = (\alpha_1, \ldots, \alpha_{i-1})$ denote the past challenges. Each worker $\mathcal{P}_{\boldsymbol{b}}$ computes the usual partial-sum univariate
$$r_{i,f}^{(\boldsymbol{b})}(X) := \sum_{\boldsymbol{w} \in B_{\mu-i}} f^{(\boldsymbol{b})}(\boldsymbol{\alpha}_{i-1}, X, \boldsymbol{w})$$
and sends it to $\mathcal{C}$ (*optionally via hierarchical aggregation* (Section III-C)).
2) (*Coordinator blinding term*) Using the public spikes, $\mathcal{C}$ computes the blinder's round-i univariate $r_{i,g}(X)$ *without* worker help. Writing $L_j^{(t)}(z) := w_j^{(t)} z + (1 - w_j^{(t)})(1 - z)$, one has the closed form
$$r_{i,g}(X) = A_i \cdot B_i \cdot U_i(X), \qquad U_i(X) = \prod_{t=1}^{d_{\mathsf{mask}}} L_i^{(t)}(X),$$
where $A_i := \prod_{j<i} \prod_t L_j^{(t)}(\alpha_j)$ and $B_i := \prod_{j>i} \Big(\prod_t w_j^{(t)} + \prod_t (1 - w_j^{(t)}) \Big)$. Therefore $\deg r_{i,g} = d_{\mathsf{mask}}$ and it costs $O(d_{\mathsf{mask}})$ to evaluate.
3) (*Blinded message*) $\mathcal{C}$ aggregates the workers' contributions and adds the mask and gets $r_i(X)$. $\mathcal{C}$ sends r_i to $\mathcal{V}$ in the standard SumCheck encoding (e.g., $d + 1$ evaluations), and stores $r_i(\alpha_i)$.
4) (*Verifier check & challenge*) $\mathcal{V}$ checks $v_{i-1} \stackrel{?}{=} r_i(0) + r_i(1)$, samples $\alpha_i \xleftarrow{\$} \mathbb{F}$, sets $v_i := r_i(\alpha_i)$, and returns α_i to $\mathcal{C}$ (who forwards it to workers).

After μ rounds, set $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_\mu)$ and note that $v_\mu = \sum_{\boldsymbol{b} \in B_\xi} \big(f(\boldsymbol{\alpha}, \boldsymbol{b}) + \rho g(\boldsymbol{\alpha}, \boldsymbol{b}) \big)$.

Phase II (coordinator builds the b-only polynomials).

1) (*Collect per-worker evaluations*) For each $j \in [c]$, each $\mathcal{P}_{\boldsymbol{b}}$ sends $g_j^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ to $\mathcal{C}$ (with distributed KZG openings if binding is required). $\mathcal{C}$ defines $\tilde{g}_j(\boldsymbol{y}) := \sum_{\boldsymbol{b} \in B_\xi} g_j^{(\boldsymbol{b})}(\boldsymbol{\alpha}) \cdot \chi_{\boldsymbol{b}}(\boldsymbol{y})$ and $\tilde{f}(\boldsymbol{y}) := h\big(\tilde{g}_1(\boldsymbol{y}), \ldots, \tilde{g}_c(\boldsymbol{y})\big)$.

2) (*Mask on the tail*) $\mathcal{C}$ also forms $\tilde{g}(\boldsymbol{y}) := g(\boldsymbol{\alpha}, \boldsymbol{y})$ directly and its sum $G_\alpha := \sum_{\boldsymbol{y} \in B_\xi} \tilde{g}(\boldsymbol{y})$, so that the phase-I target v_μ equals $\sum_{\boldsymbol{y} \in B_\xi} \big(\tilde{f}(\boldsymbol{y}) + \rho \tilde{g}(\boldsymbol{y})\big)$.

Phase III (rounds over b; coordinator $\leftrightarrow$ verifier). Run ξ more SumCheck rounds between $\mathcal{C}$ and $\mathcal{V}$ over the polynomial $\tilde{f}(\boldsymbol{y}) + \rho \tilde{g}(\boldsymbol{y})$ with initial target v_μ:

1) For $j = 1$ to ξ, $\mathcal{C}$ sends the univariate $\tilde{r}_j(Y)$ defined by summing over the remaining $\xi - j$ Boolean variables; $\mathcal{V}$ checks telescoping, samples $\beta_j \xleftarrow{\$} \mathbb{F}$, and sets $v_{\mu+j} := \tilde{r}_j(\beta_j)$.

Let $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_\xi)$ denote the final challenge.

Final openings.

1) (*Open multilinear components*) Using the distributed KZG PCS, the coordinator and workers open $\{g_j(\boldsymbol{\alpha}, \boldsymbol{\beta})\}_{j \in [c]}$ to $\mathcal{V}$ at point $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ (as in the transparent protocol).
2) (*Evaluate the mask*) $\mathcal{V}$ computes $g(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \prod_{t=1}^{d_{\mathsf{mask}}} \chi_{\boldsymbol{w}^{(t)}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ directly from the public spikes.
3) (*Consistency*) $\mathcal{V}$ checks that $f(\boldsymbol{\alpha}, \boldsymbol{\beta}) + \rho g(\boldsymbol{\alpha}, \boldsymbol{\beta})$matches the last SumCheck message If so, accept; otherwise reject and the accountability workflow in Section III-B identifies a malicious worker.

Notes. (i) Choose $d_{\mathsf{mask}} \geq \max_i \deg_{X_i} f$ so that each round's degree is fully masked. In our instantiation, $d_{\mathsf{mask}} = \deg(h)$ suffices. (ii) Binding the mask with PCS is optional; the verifier can recompute g at the final point. If desired, committing to the multilinear g'_t's via the distributed KZG (and opening them when used) gives a fully bound blinder without touching the non-multilinear product g itself.

The standard distributed SumCheck analysis applies unchanged. Completeness follows by construction; soundness is preserved since the degree bound and the final point check are unchanged. Regarding accountability, at the end of the protocol all workers still evaluates $g^{\boldsymbol{b}}(\boldsymbol{r})$, so the coordinator accountability procedure remains the same as in Section III-B. To show that this protocol is HVZK, we follow the ideas of previous works [6], [8].

Proposition 7. *Fix d and m. Let the verifier and all workers be honest, and let the blinder $g(\boldsymbol{Z}) = \prod_{t=1}^{d} \chi_{\boldsymbol{w}^{(t)}}(\boldsymbol{Z})$ with $\{\boldsymbol{w}^{(t)}\}$ chosen uniformly at random from $\mathbb{F}^m$. Then the distributed* SumCheck *on f with target H is HVZK.*

Proof. (i) Simulator commits to $g^\star$ first. Sample d spikes $\{\boldsymbol{w}^{(t)\star}\}$ uniformly, define $g^\star$, and compute the closed-form $G^\star := \sum_{\boldsymbol{Z} \in B_m} g^\star(\boldsymbol{Z})$. Send the public description of the spikes (or a commitment to $g^\star$) and $G^\star$ *before* seeing ρ.

(ii) Receive $\rho \in \mathbb{F}^*$.

(iii) Choose $f^\star$ at random. Sample $f^\star$ uniformly from degree-$\leq d$ polynomials conditioned on $\sum_{\boldsymbol{Z} \in B_m} f^\star(\boldsymbol{Z}) = H$.

(iv) Run SumCheck on $h^\star := f^\star + \rho g^\star$. At round $i \in [m]$ (with prefix $\boldsymbol{\alpha}_{<i}$) the prover sends

$$h_i^\star(X_i) := \sum_{\boldsymbol{b} \in B_{m-i}} h^\star(\boldsymbol{\alpha}_{<i}, X_i, \boldsymbol{b}) = f_i^\star(X_i) + \rho G_i^\star(X_i),$$

a univariate of degree $\leq d$. Across m rounds, the verifier learns $(md + 1)$ independent affine constraints on the coefficients of $h^\star$, due to the m degree bounds and the $(m-1)$ telescoping equalities.

Here we argue that the distribution of (iv) is statistically close in both worlds. Let $L_j^{(t)}(z) := w_j^{(t)} z + (1 - w_j^{(t)})(1 - z)$. A direct factorization gives $G_i^\star(X_i) = A_i B_i \cdot (\prod_{t=1}^{d} L_i^{(t)}(X_i))$. Thus $G_i^\star$ is a degree-d univariate in X_i, whose $(d+1)$ coefficients are polynomial functions of the spike coordinates $\{w_j^{(t)}\}$. Write $U_i(X) = \prod_{t=1}^{d} (a_t X + b_t)$ with $a_t := 2w_i^{(t)} - 1$ and $b_t := 1 - w_i^{(t)}$. The coefficient vector $\mathsf{coeff}(U_i) \in \mathbb{F}^{d+1}$ is a multivariate polynomial in $\{w_i^{(t)}\}_{t=1}^{d}$ whose Jacobian minors are (up to nonzero scalars) the Vandermonde $\prod_{t \neq t'} (w_i^{(t)} - w_i^{(t')})$ times $\prod_t a_t$. Hence, except when a single low-degree polynomial in $\{w_i^{(t)}\}$ vanishes, the differential has full row rank onto the d-dimensional affine slice determined by the round-i telescoping constraint.

Moreover, the dependence across rounds is block-triangular: $G_i^\star$ depends only on spike coordinates with indices $j \geq i$ (via the nonzero scale $A_{<i}$ and the tail factor $B_{>i}$). After reordering variables by coordinate, the global Jacobian that maps all spikes $\{\boldsymbol{w}^{(t)}\}_{t \leq d}$ to the concatenated coefficient vectors $\big(\mathsf{coeff}(G_1^\star), \ldots, \mathsf{coeff}(G_m^\star)\big)$ is upper block-triangular with diagonal blocks of the type above. By Schwartz-Zippel and a union bound over $i \in [m]$, the probability that any block loses rank is at most $O(md/|\mathbb{F}|)$.

Conditioned on the (overwhelming) "full-rank" event and since $\rho \neq 0$, each masked coefficient vector

$$C_i := \mathsf{coeff}\big(h_i^\star\big) = \mathsf{coeff}(f_i^\star) + \rho \cdot A_i B_i \cdot \mathsf{coeff}\big(U_i\big)$$

is an *affine image of independent uniforms* in $\{w_i^{(t)}\}$, and is therefore *exactly uniform* over the d-dimensional affine slice defined by the round-i relation. When the low-probability rank-deficiency event happens, the support can shrink; this yields at most $O(md/|\mathbb{F}|)$ statistical distance from uniform on that slice.

Finally, because the simulator chooses $f^\star$ uniformly subject only to the single global sum constraint, and the m masked univariates jointly impose $md+1$ linear conditions with the block-triangular independence above, the simulated and real sumcheck transcripts are statistically close, with error at most $O(md/|\mathbb{F}|)$. The final query reveals $f^\star(\boldsymbol{r}') + \rho g^\star(\boldsymbol{r}')$; $g^\star(\boldsymbol{r}')$ is determined by the public spikes, so no witness information leaks. This proves HVZK. □

Costs. Our ZK compilation of the distributed SumCheck protocol adds only small, one-time or lower-order work and does not change rounds or per-round message sizes. Concretely, the coordinator samples d spike vectors and computes G in $O(d(\mu + \xi))$ total field ops with the workers; the coordinator

and the workers forms a degree-d mask univariate per round in $O((\mu+\xi)d^2)$ field ops.

ZK distributed multivariate ZeroTest and PermTest Poly-IOP. We compile the ZeroTest and PermTest by replacing their internal sumcheck with the ZK sumcheck, using a degree bound that matches the masked instance. The asymptotic prover, coordinator, verifier, and communication costs match those of the transparent version.

ZK distributed multilinear KZG PCS. Let $f(\boldsymbol{x},\boldsymbol{y}) = \sum_{\boldsymbol{b}\in B_\xi} f^{(\boldsymbol{b})}(\boldsymbol{x})\chi_{\boldsymbol{b}}(\boldsymbol{y}) \in \mathbb{F}^{\le 1}_{\mu+\xi}[\boldsymbol{X}]$ with $\boldsymbol{x}\in B_\mu$, $\boldsymbol{y}\in B_\xi$.

- $\mathsf{KeyGen}_{\mathsf{zk}}$**:** Sample independent generators $g,h\in\mathbb{G}$ with unknown discrete-log relation. Sample trapdoors $\tau_1,\dots,\tau_{\mu+\xi} \xleftarrow{\$} \mathbb{F}$. Publish $\mathsf{crs}_{\mathsf{zk}} = (g,h,\{g^{\tau_i}\}_{i\in[\mu+\xi]},\{h^{\tau_i}\}_{i\in[\mu+\xi]},\{U^{(g)}_{\boldsymbol{c},\boldsymbol{b}}\}_{\boldsymbol{c},\boldsymbol{b}},\{U^{(h)}_{\boldsymbol{c},\boldsymbol{b}}\}_{\boldsymbol{c},\boldsymbol{b}})$, where $U^{(\star)}_{\boldsymbol{c},\boldsymbol{b}} := (\star)^{\chi_{\boldsymbol{c}}(\tau_1,\dots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}$ for $\star\in\{g,h\}$, $\boldsymbol{c}\in B_\mu$, $\boldsymbol{b}\in B_\xi$. For aggregation on $\boldsymbol{y}$ we also make available $V^{(\star)}_{\boldsymbol{b}} := (\star)^{\chi_{\boldsymbol{b}}(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}$.
- $\mathsf{Commit}_{\mathsf{zk}}(f)$**:** Each worker $\mathcal{P}_{\boldsymbol{b}}$ samples an independent multilinear *mask* $r^{(\boldsymbol{b})}\in\mathbb{F}^{\le 1}_{\mu}[\boldsymbol{X}]$ and computes two partial commitments $\mathsf{com}^{(g)}_{\boldsymbol{b}} := \prod_{\boldsymbol{c}\in B_\mu}\big(U^{(g)}_{\boldsymbol{c},\boldsymbol{b}}\big)^{f^{(\boldsymbol{b})}(\boldsymbol{c})}$ and $\mathsf{com}^{(h)}_{\boldsymbol{b}} := \prod_{\boldsymbol{c}\in B_\mu}\big(U^{(h)}_{\boldsymbol{c},\boldsymbol{b}}\big)^{r^{(\boldsymbol{b})}(\boldsymbol{c})}$. It sends $\mathsf{com}_{\boldsymbol{b}} := \mathsf{com}^{(g)}_{\boldsymbol{b}}\cdot\mathsf{com}^{(h)}_{\boldsymbol{b}}$ to the coordinator. The coordinator outputs the global commitment $\mathsf{com} := \prod_{\boldsymbol{b}}\mathsf{com}_{\boldsymbol{b}}$.
- $\mathsf{Open}_{\mathsf{zk}}(f,(\boldsymbol{\alpha},\boldsymbol{\beta}))$**:**
  1) *Per-worker witnesses on $\boldsymbol{x}$:* Each $\mathcal{P}_{\boldsymbol{b}}$ computes $z^{(\boldsymbol{b})} := f^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ and $\widehat{z}^{(\boldsymbol{b})}_h := h^{r^{(\boldsymbol{b})}(\boldsymbol{\alpha})}$ (note that the scalar $r^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ itself is *not* revealed). It divides $f^{(\boldsymbol{b})}(\boldsymbol{x})-f^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ and $r^{(\boldsymbol{b})}(\boldsymbol{x})-r^{(\boldsymbol{b})}(\boldsymbol{\alpha})$ as
  $$\sum_{i\in[\mu]} q^{(\boldsymbol{b})}_i(\boldsymbol{x})\big(x_i-\alpha_i\big), \qquad \sum_{i\in[\mu]} \tilde{q}^{(\boldsymbol{b})}_i(\boldsymbol{x})\big(x_i-\alpha_i\big),$$
  and sends the *paired* witnesses $\pi^{(\boldsymbol{b})}_i := g^{q^{(\boldsymbol{b})}_i(\tau_1,\dots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}\cdot h^{\tilde{q}^{(\boldsymbol{b})}_i(\tau_1,\dots,\tau_\mu)\cdot\chi_{\boldsymbol{b}}(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}$ for all $i\in[\mu]$, together with $z^{(\boldsymbol{b})}$ and $\widehat{z}^{(\boldsymbol{b})}_h$.
  2) *Coordinator aggregation on $\boldsymbol{y}$:* The coordinator computes
  $$z := \sum_{\boldsymbol{b}} z^{(\boldsymbol{b})}\cdot\chi_{\boldsymbol{b}}(\boldsymbol{\beta}), \qquad \widehat{z}_h := \prod_{\boldsymbol{b}}\big(\widehat{z}^{(\boldsymbol{b})}_h\big)^{\chi_{\boldsymbol{b}}(\boldsymbol{\beta})}.$$
  It also expresses $f(\boldsymbol{\alpha},\boldsymbol{y})-f(\boldsymbol{\alpha},\boldsymbol{\beta}) = \sum_{j\in[\xi]} Q_j(\boldsymbol{y})(y_j-\beta_j)$ and similarly for the mask part (implicitly defined by the $\widehat{z}_h$ term), and forms for each $j\in[\xi]$ the pairwise-combined witnesses
  $$\pi_{\mu+j} := \big(g^{Q_j(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}\big)\cdot\big(h^{\widetilde{Q}_j(\tau_{\mu+1},\dots,\tau_{\mu+\xi})}\big),$$
  where Q_j and $\widetilde{Q}_j$ are obtained using $V^{(g)}_{\boldsymbol{b}}$ and $V^{(h)}_{\boldsymbol{b}}$ exactly as in the transparent protocol (with g- and h-bases, respectively). Finally it multiplies per-worker $\pi^{(\boldsymbol{b})}_i$ to get $\pi_i := \prod_{\boldsymbol{b}}\pi^{(\boldsymbol{b})}_i$ for all $i\in[\mu]$, and sends $\big(z,\widehat{z}_h,\{\pi_i\}_{i\in[\mu+\xi]}\big)$ to the verifier.
- $\mathsf{Verify}_{\mathsf{zk}}\big(\mathsf{com},\boldsymbol{\alpha},\boldsymbol{\beta};z,\widehat{z}_h,\{\pi_i\}\big)$**:** Accept iff $e(\mathsf{com}/(g^z\widehat{z}_h),g) = \prod_{i\in[\mu]} e\left(\pi_i, g^{\tau_i-\alpha_i}\right)\cdot\prod_{j\in[\xi]} e\left(\pi_{\mu+j}, g^{\tau_{\mu+j}-\beta_j}\right)$.

The commitment is binding under KZG assumptions and hiding due to the independent h-term. Accountability from Section III-B holds, since the coordinator can verify each worker's per-worker equation using the same form with that worker's $\widehat{z}^{(\boldsymbol{b})}_h$ and $\{\pi^{(\boldsymbol{b})}_i\}$.

Proposition 8. *The ZK distributed multilinear KZG PCS above is complete and sound under the standard KZG assumptions. If all workers are honest, the evaluation protocol is zero-knowledge: for a given opening point, the proof reveals no information about f beyond the claimed evaluation $z=f(\boldsymbol{\alpha},\boldsymbol{\beta})$.*

Proof. Completeness follows by multi-linearity and the same telescoping used in the transparent verifier equation. Binding follows from the binding of Pedersen-style KZG in both g- and h-bases. Zero-knowledge holds because the only value about r that ever appears is $\widehat{z}_h = h^{r(\boldsymbol{\alpha},\boldsymbol{\beta})}$, which is independent of $r(\boldsymbol{\alpha},\boldsymbol{\beta})$ under discrete-log hardness; all other terms are randomized witnesses tied only to the public commitment. □

Costs. Relative to the transparent PCS in Section III-A, the ZK variant adds a second set of witness terms and a single extra group element $\widehat{z}_h$ per opening. Therefore, the cost of the coordinator and workers remains the same up to a constant factor.

ZK Cirrus. We compile the ZK Poly-IOP into the ZK Cirrus using the distributed KZG PCS in its *hiding* variant. Then we use the PCS's ZK evaluation procedure to prove openings used inside the protocol. Note that all checks in Sections III-B and III-C work over hiding commitments. If a proof fails, the same localization identifies a faulty worker. Combining Propositions 7 and 8 and Theorem 1, we obtain the following corollary.

Corollary 1. *Composing the ZK distributed Poly-IOPs with the ZK distributed KZG PCS yields a ZK accountable distributed SNARK (Definition 1) for the same relation.*

APPENDIX B
ARTIFACT APPENDIX

This appendix describes how to obtain, build, and run Cirrus for artifact evaluation. It also documents the hardware, software, and configuration requirements, enumerates the major claims, and how to run the experiments.

A. Description & Requirements

To run our artifact, we need a set of *AWS instances* to act as the coordinator and distributed workers, and a *local machine* to interact with the AWS instances (e.g., run command efficiently, upload required files, etc.).

How to access. Our code is public on Zenodo (doi: 10.5281/zenodo.17843693).

- Source code for Cirrus (Rust), developed from HyperPlonk.
- Reproduction scripts for the major claims in the implementation section.
- Documentation: a `README.md`.

Hardware dependencies. We ran our experiments on up to 33 AWS `t3.2xlarge` instances in the same region, each with 8 vCPUs and 32 GB RAM. One instance acts as the *coordinator*, and has 500 GB storage; the other 32 instances act as *workers*, and have 50 GB storage. Each instance has a public IP address.

To run the setup necessary for our experiments and to connect to the remote AWS instances, we require one local machine with at least 64 GB RAM and 500 GB storage; if you prefer to download the setup parameters from an existing source, the local machine should have 8 GB RAM and 500 GB storage.

Software dependencies. Each instance has OS `Ubuntu Server 24.04 LTS`, and `Python 3.12` and `Rust` installed. For the local machine, please have `Python 3.12` installed, and `Rust` installed with a 64-bit Linux OS if running the setup is desired. The local machine relies on Paramiko[2] to connect to the AWS instances.

B. Artifact Installation & Configuration

Local machine setup. Install `Python` first, and we recommend `Miniconda`[3] for installation. Then install the required package to connect to the AWS instances by running:

```
pip install paramiko
```

To generate the setup parameters on your own, please install `Rust`.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
```

AWS instance setup. To have the artifact installed on the AWS instances, please first configure an EC2 launch template with the following specifications:

- OS volume type: `Ubuntu Server 24.04 LTS (HVM), SSD Volume Type`.
- Instance type: `t3.2xlarge`.
- Key pair (login): Create your ed25519 key pair, and rename the private key on the local machine to `cirrus.pem`.
- Network settings: Create a new security group with security group rule, with `All traffic` from `Anywhere` allowed. Also in the advanced network configuration, enable the `Auto assign public IP` option.
- Storage: change the storage size to 50 GiB for workers, or to 300 GiB for the coordinator.

After saving the launch template, launch 33 instances using this template, and get a list of public IPs `IP_0, IP_1, ..., IP_32`.

We run the following steps to setup the environments on each AWS instances. (1) In the local machine, place the `cirrus.pem` key file in the folder `key/` under the main folder. (2) Open the file `ec2/setup.py` and update the variable `ec2_public_ips` to the list of public IPs. (3) Run the following command on the local machine:

```
python ec2/setup.py
```

This step prepares the AWS instances with the required `Rust` and `Python` software dependencies to run our experiments.

C. Major Claims

Here are the major claims of the paper.

1) (C1): Cirrus generates proofs for circuits with 33 million gates in under 40 seconds using 32 8-core machines. This is proven by experiment (E1) whose results are illustrated in Fig. 1 in the paper.
2) (C2): The accountability protocol of Cirrus is efficient, and for circuits of 2^{25} total gates and the distribution of 256 workers, the accountability protocol takes under 4 seconds to run. This is proven by experiment (E2) whose results are illustrated in Fig. 2 in the paper.
3) (C3): The runtime of the coordinator's computation is lightweight and under 1 second for less than $10,000$ workers. This is proven by experiment (E3) whose results are illustrated in Fig. 3 in the paper.

D. Evaluation

Experiment (E1) [30 human-minutes + 4 compute-hours]. In this experiment, we benchmark the end-to-end run time of our artifact with circuit of different sizes and different number of workers. If you would like to verify the major claim (C1), please set `log_num_workers` to 8 and `log_num_vars` to 25 (note that 2^{25} is 33 million).

[Preparation] In the preparation, we get the setup files on the local machine, and upload the required setup files to the AWS instances. To prepare setup parameters, do the following on the local machine:

1) Please either (1) download them from an existing source and place them in the folder `out/` under the main artifact folder, or (2) run the following command to setup for $2^{\texttt{log\_num\_workers}}$ workers and circuit of size $2^{\texttt{log\_num\_vars}}$:

```
./scripts/run.sh setup --log_num_workers $log_num_workers --log_num_vars $log_num_vars
```

[2]https://www.paramiko.org/

[3]https://www.anaconda.com/docs/getting-started/miniconda/main

And the output files will be in the directory `out/vanilla-$log_num_vars-$log_num_workers`.

2) To upload the setup parameter for $2^{\texttt{log\_num\_workers}}$ workers and circuit of size $2^{\texttt{log\_num\_vars}}$, first open the file `ec2/transfer_setup_param.py`, and modify the following two variables:

```
log_num_workers = [$log_num_workers]
log_num_vars = [$log_num_vars]
```

Note that you can upload multiple files at the same time by including multiple `log_num_workers` and `log_num_vars` in the list. Also update the variable `ec2_public_ips` to the list of public IPs, with `IP_0` being the first item in the list. Then run the following command on the local machine:

```
python ec2/transfer_setup_param.py
```

[Execution] To benchmark the end-to-end runtime of our artifact with $2^{\texttt{log\_num\_workers}}$ workers and circuit of size $2^{\texttt{log\_num\_vars}}$, do the following on the local machine.

1) First, open the file `ec2/run_exp_single_thread.py`, and modify the following two parameters:

```
log_num_workers = [$log_num_workers]
log_num_vars = [$log_num_vars]
```

Note that you can run multiple experiments sequentially by including multiple `log_num_workers` and `log_num_vars` in the list, if the setup parameters are ready on the AWS instances. Record the private IPv4 address of the machine with `IP_0`, as `COORD_IP`. Also modify the following parameters:

```
master_ip = $IP_0
# The private IPv4 address
master_private_addr = $COORD_IP
master_listen_port = 7034
worker_ips = [
    $IP_1, ..., $IP_32
]
```

2) Then run the following command to run the experiment:

```
python ec2/run_exp_single_thread.py
```

[Results] To gather result for `log_num_workers` and `log_num_vars`, open the file

```
~/projects/cirrus/out/vanilla-$log_num_vars-$log_num_workers/master_analysis.json
```

The end-to-end runtime of the protocol is

```
json_data[1]["duration"] - json_data[0]["send_time"]
```

This is the runtime in seconds. When `log_num_workers` is 8 (i.e., 256 workers in total) and `log_num_vars` is 25 (i.e., the circuit has 33M gates), the runtime is expected to be below 40 seconds, proving the major claim (C1).

Experiment (E2) [15 human-minutes + 1 compute-hour]. In this experiment, we benchmark the accountability protocol of Cirrus with different circuit sizes and numbers of workers.
[Preparation] No extra preparation is required.

[Execution] To execute the experiment with `log_num_workers` and `log_num_vars` to benchmark the accountability protocol runtime with $2^{\texttt{log\_num\_workers}}$ for circuit of size $2^{\texttt{log\_num\_vars}}$, connect to the AWS instance of public IP address `IP_0`. Then run the following command on that AWS instance:

```
./scripts/run.sh accountability --log_num_vars $log_num_vars --num_threads 8 --log_num_workers $log_num_workers
```

[Results] The result of the experiment will be printed on the terminal, following `[INFO] total time: {t} s`. This means that the accountability protocol for $2^{\texttt{log\_num\_workers}}$ workers and circuit of size $2^{\texttt{log\_num\_vars}}$ takes `t` seconds. For `log_num_workers` not greater than 8 and `log_num_vars` not greater than 25, the total time is expected to be below 4 seconds, proving the major claim (C2).

Experiment (E3) [15 human-minutes + 45 compute-minutes]. In this experiment, we benchmark the coordinator time to demonstrate its performance.
[Preparation] No extra preparation is required.
[Execution] To execute the experiment to benchmark the coordinator time with hierarchical aggregation (HA), first connect to the AWS instance of public IP address `IP_0`. Then run the following command on that AWS instance:

```
bash ./bench_master_ha.sh
```

To benchmark the coordinator time without HA, run the following command:

```
bash ./bench_master_no_ha.sh
```

[Results] The results will be printed on the terminal like this:

```
Time elapsed for --log-num-vars=$log_num_vars and --log-num-workers=$log_num_workers: {$t} ms
```

This means that for subcircuit of size $2^{\texttt{log\_num\_vars}}$ and $2^{\texttt{log\_num\_workers}}$ workers, the coordinator time is `t` milliseconds. When `log_num_workers` is 13, the coordinator time with HA is expected to be less than 1 second, which justifies the major claim (C3).