HyperPianist: Pianist with Linear-Time Prover and Logarithmic Communication Cost

Chongrong Li[⋆], Pengfei Zhu[†], Yun Li[+(✉)],
Cheng Hong[+], Wenjie Qu[‡], Jiaheng Zhang[‡]

[⋆]*Shanghai Jiao Tong University*, [†]*Tsinghua University*,
[+]*Ant Group*, [‡]*National University of Singapore*

lichongrong9@gmail.com, *zpf21@mails.tsinghua.edu.cn*, *liyun24@antgroup.com*,
vince.hc@antgroup.com, *wenjiequ@u.nus.edu*, *jhzhang@nus.edu.sg*

***Abstract*—Recent years have seen great improvements in zero-knowledge proofs (ZKPs). Among them, zero-knowledge SNARKs are notable for their compact and efficiently-verifiable proofs, but suffer from high prover costs. Wu et al. (Usenix Security 2018) proposed to distribute the proving task across multiple machines, and achieved significant improvements in proving time. However, existing distributed ZKP systems still have quasi-linear prover cost, and may incur a communication cost that is linear in circuit size.**

In this paper, we introduce HyperPianist. Inspired by the state-of-the-art distributed ZKP system Pianist (Liu et al., S&P 2024) and the multivariate proof system HyperPlonk (Chen et al., EUROCRYPT 2023), we design a distributed multivariate polynomial interactive oracle proof (PIOP) system with a linear-time prover cost and logarithmic communication cost. Unlike Pianist, HyperPianist incurs no extra overhead in prover time or communication when applied to general (non-data-parallel) circuits. To instantiate the PIOP system, we adapt two additively-homomorphic multivariate polynomial commitment schemes, multivariate KZG (Papamanthou et al., TCC 2013) and Dory (Lee et al., TCC 2021), into the distributed setting, and get HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ respectively. Both systems have linear prover complexity and logarithmic communication cost; furthermore, HyperPianist$^{\mathsf{D}}$ requires no trusted setup. We also propose HyperPianist+, incorporating an optimized lookup argument based on Lasso (Setty et al., EUROCRYPT 2024) with lower prover cost.

Experiments demonstrate HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ achieve speedups of $63.1\times$ and $40.2\times$ over HyperPlonk with 32 distributed machines. Compared to Pianist, HyperPianist$^{\mathsf{K}}$ can be $2.9\times$ and $4.6\times$ as fast and HyperPianist$^{\mathsf{D}}$ can be $2.4\times$ and $3.8\times$ as fast, on vanilla gates and custom gates respectively. With layered circuits, HyperPianist$^{\mathsf{K}}$ is up to $5.9\times$ as fast on custom gates, and HyperPianist$^{\mathsf{D}}$ achieves a $4.7\times$ speedup.

1. Introduction

Zero-knowledge proofs (ZKPs) were first introduced in the 1980s by Goldwasser et al. [1], and have since become a staple of modern cryptography. They allow a party to prove the truth of a statement without revealing any additional information. In recent years, the efficiency of ZKPs has dramatically improved, enabling various new applications in blockchains and machine learning, among others.

Zero-knowledge succinct non-interactive arguments of knowledge (SNARKs) are a notable type of ZKPs where the proof is short and fast to verify ("succinctness"). One of the most popular constructions of modern SNARKs is to first design a polynomial interactive oracle proof (PIOP) system and then instantiate it with a polynomial commitment scheme (PCS). Two of the most deployed SNARKs in the industry, Plonk [2] and Marlin [3], fall into this category. Plonk stands out with its compact proof size and fast verifier, as well as support for custom gates; it has been adopted in various blockchain-related applications such as zkRollups and zkEVM (Ethereum Virtual Machine). An extension work Plonkup [4] enhances Plonk with the lookup arguments from Plookup [5], allowing the proof system to efficiently handle non-linear functions.

However, high prover costs have inhibited the application of SNARKs to large-scale circuits, such as complex EVM execution traces and large language models. In 2018, Wu et al. proposed DIZK [6], where the proof generation process is distributed across multiple machines (called sub-provers), and it demonstrated great improvements in proving time. Nevertheless, as it is built on Groth16 [7], each sub-prover runs in quasi-linear time due to polynomial interpolation, and the communication cost among the sub-provers is linear in circuit size. Subsequently, zkBridge introduced deVirgo [8], a distributed ZKP system designed for data-parallel circuits based on multivariate polynomials. The multivariate PIOP in deVirgo has linear-time prover cost, but the multivariate PCS still requires polynomial interpolation and entails a linear communication cost among the sub-provers. A very recent work Pianist [9] built a distributed ZKP system on Plonk. To extend the univariate Plonk system to the distributed setting, Pianist utilized bivariate PIOPs, and designed a bivariate PCS based on the KZG polynomial commitment scheme [10]. Pianist is able to achieve a constant communication cost under a trusted setup, but it still

has quasi-linear prover cost for polynomial interpolation. Another recent work HEKATON [11] proposed to divide the circuit into different sub-circuits and prove each sub-circuit with MIRAGE [12], a commit-carrying version of Groth16 [7]. It allows the sub-circuits to be different with a smaller common reference string, but similarly, it requires quasi-linear prover cost. While a direct comparison is not available, the authors compared with Pianist by extrapolation, and claimed a $3\times$ speedup factor over Pianist.[1]

We note that all these systems have quasi-linear prover time caused by polynomial interpolation (either in PIOPs or PCS). As reported by HyperPlonk [13], when circuit sizes are larger than 2^{21}, the quasi-linear factor will account for a significant portion of the overall proving time. HyperPlonk [13] thus adapts Plonk's univariate PIOP system to a multivariate one; with a suitable linear-time multivariate PCS, HyperPlonk can achieve a linear-time proving cost. As the linear-time prover scales better than Plonk's quasi-linear-time prover, in practice, it can be nearly $3\times$ as fast when the circuit size is 2^{20}, and the difference becomes larger as the circuit size increases. Moreover, HyperPlonk supports more efficient high-degree custom gates than Plonk due to its multivariate PIOP system.

1.1. Our Contributions

1.1.1. HyperPianist. In this work, we propose HyperPianist[2], a distributed ZKP system featuring linear proving cost and logarithmic communication cost as well as succinct proofs. It satisfies the notion of a "fully" distributed ZKP system introduced in Pianist [9], as it supports general circuits in addition to data-parallel ones. At the core of HyperPianist, we design a distribution-friendly multivariate PIOP system based on HyperPlonk, and two distributed multivariate polynomial commitment schemes for instantiation.

Distributed Multivariate PIOP System. We observe that the constraint system of HyperPlonk can be reduced to multivariate SumCheck identities. We thus extend the distributed SumCheck protocol in deVirgo to general circuits, and adapt HyperPlonk's multivariate PIOP system to the distributed setting. This distributed PIOP system comes without extra communication cost or new constraints for non-data-parallel circuits, in contrast to Pianist, which requires a helper polynomial and two additional constraints.

In HyperPlonk, the copy constraints are reduced to a multiset check identity, which is in turn handled by the grand product check PIOP from Quarks [14]. This PIOP involves a helper polynomial that is unfriendly to distribution. Naïvely computing the polynomial would entail a linear communication cost among the sub-provers. To overcome this problem, we opt for logarithmic derivative techniques [15] to reduce the multiset check identity to a rational SumCheck statement. This reduction requires no extra communication, and the resulting PIOP has $O(\log N)$ communication cost and $O(\log N)$ proof size. We can alternatively use layered circuits to directly prove the product check relation, which also incurs no extra costs. This method has $O(\log^2 N)$ communication cost and proof size, but has a faster prover.

Distributed Multivariate PCS. We observe that both the quasi-linear prover cost and linear communication cost of deVirgo result from its FRI-based multivariate PCS, which requires univariate polynomial interpolation and witness exchange among sub-provers. We recognize that a multivariate PCS with the additively-homomorphic property is more compatible with the distributed setting, since partial results from sub-provers can be easily aggregated with sublinear communication costs. We thus design a distributed version of the multivariate KZG [16] scheme (denoted by deMKZG). Given a multilinear polynomial with $\log N$ variables and M distributed sub-provers, deMKZG has $O(1)$ commitment size, $O(\log N)$ communication cost per sub-prover, $O(\log N)$ proof size and $O(\log N)$ verifier time, but requires a trusted setup.

We also design deDory, a distributed version of Dory [17]. It is transparent, with the same asymptotic complexity as deMKZG in commitment size and communication cost, and $O(\log N+\log M)$ evaluation proof size and verifier time. In doing so, we re-organized the matrix representation of Dory in a distribution-friendly fashion to reduce unnecessary communication costs for aggregation.

We compare deMKZG and deDory with the distributed PCS from previous works in Table 1. By instantiating our PIOP system with the two commitment schemes, we get HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ respectively. We compare our distributed ZKP systems with prior works in Table 2.

1.1.2. HyperPianist+. Our second contribution is HyperPianist+, an enhancement of HyperPianist with an optimized distributed lookup argument based on Lasso [18]. Lookup arguments allow a party to prove that every element in a committed vector exists in a pre-determined table. They are especially suitable for "non-arithmetic" functions like bitwise operations, range checks and so on.

Plonkup [5] enhanced Plonk with the univariate lookup protocol Plookup [5], and subsequently, HyperPlonk+ demonstrated how to extend the univariate protocol into a multivariate one. However, this transformation entails sorting the union table of public table and witnesses, and as such the prover is not strictly linear. Our optimization builds on Lasso [18], the most efficient construction of multivariate lookup arguments so far. We note that Lasso involves a well-formation check to guarantee the correctness of certain polynomials committed during the protocol. In Lasso, this is done by offline memory checking techniques from Spartan [19]. We identify that this check can be handled more efficiently using logarithmic derivative techniques from Logup [15]. By viewing the well-formation check as a set inclusion relation, we can reduce it to a rational

1. The comparison was made by extracting data from the figures provided by Pianist. However, we found that the claimed speedup factor of HEKATON may be overestimated. Specifically, for a circuit of size 2^{25}, the prover time of Pianist on 32 machines (with 2048 vCPUs in total) was about 5s, in contrast to the "sub-10"s stated in HEKATON; as HEKATON's prover takes 11.6s on a circuit of size 2^{27} with 2048 cores, this indicates a speedup factor of roughly $2\times$.

2. Stands for "HyperPlonk vIA uNlimited dISTribution".

TABLE 1: Comparisons of distributed PCS. ("Trans.": transparent setup, N: witnesses size, M: the number of distributed machines, $T = \frac{N}{M}$: the size of witnesses each machine holds. $\mathbb{H}, \mathbb{F}, \mathbb{P}, \mathbb{G}, \mathbb{G}_T$: hash, field, pairing, group, pairing target group operations, $|\cdot|$: the size of corresponding elements.)

PCS	Polynomial Type	Trans.?	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Proof size	Communication
deVirgo [8]	Multivariate	✓	$O(T \log T)\,\mathbb{F} + O(T)\,\mathbb{H}$	$O(\log^2 N + M)\,\mathbb{H}$	$O(\log^2 N + M)\,\lvert\mathbb{H}\rvert$	$O(N)\,\lvert\mathbb{F}\rvert$
Pianist [9]	Bivariate	✗	$O(T)\,\mathbb{F} + O(T)\,\mathbb{G}$	$O(1)\,\mathbb{P} + O(\log N)\mathbb{F}$	$O(1)\,\lvert\mathbb{G}\rvert$	$O(M)\,\lvert\mathbb{G}\rvert$
deDory	Multivariate	✓	$O(T)\,\mathbb{G}$	$O(\log N + \log M)\,\mathbb{G}_T$	$O(\log N + \log M)\,\lvert\mathbb{G}_T\rvert$	$O(M \log N)\,\lvert\mathbb{G}\rvert$
deMKZG	Multivariate	✗	$O(T)\,\mathbb{G}$	$O(\log N)\,\mathbb{P}$	$O(\log N)\,\lvert\mathbb{G}\rvert$	$O(M \log N)\,\lvert\mathbb{G}\rvert$

TABLE 2: Comparisons of distributed ZKP systems ("Fully": fully distributed, other notations are same as in Table 1).

Scheme	Trans.?	Fully?	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Proof size	Communication
deVirgo [8]	✓	✗	$O(T \log T)\,\mathbb{F} + O(T)\,\mathbb{H}$	$O(\log^2 N + M)\,\mathbb{H}$	$O(\log^2 N + M)\,\lvert\mathbb{H}\rvert$	$O(N)\,\lvert\mathbb{F}\rvert$
Pianist [9]	✗	✓	$O(T \log T)\,\mathbb{F} + O(T)\,\mathbb{G}$	$O(1)\,\mathbb{P} + O(\log N)\mathbb{F}$	$O(1)\,\lvert\mathbb{G}\rvert$	$O(M)\,\lvert\mathbb{G}\rvert$
HEKATON [11]	✗	✓	$O(T \log T)\,\mathbb{F} + O(T)\,\mathbb{G}$	$O(\log M)\,\mathbb{G}$	$O(\log M)\,\lvert\mathbb{G}\rvert$	$O(M)\,\lvert\mathbb{G}\rvert$
HyperPianist$^{\mathsf{K}}$	✗	✓	$O(T)\,\mathbb{F} + O(T)\,\mathbb{G}$	$O(\log N)\,\mathbb{F} + O(\log N)\,\mathbb{P}$	$O(\log N)\,\mathbb{F} + O(\log N)\,\lvert\mathbb{G}\rvert$	$O(M \cdot \log N)\,\lvert\mathbb{F}\rvert + O(M \cdot \log N)\,\lvert\mathbb{G}\rvert$
HyperPianist$^{\mathsf{D}}$	✓	✓	$O(T)\,\mathbb{F} + O(T)\,\mathbb{G}$	$O(\log N)\,\mathbb{F} + O(\log N + \log M)\,\mathbb{G}_T$	$O(\log N)\,\lvert\mathbb{F}\rvert + O(\log N + \log M)\,\lvert\mathbb{G}_T\rvert$	$O(M \cdot \log N)\,\lvert\mathbb{F}\rvert + O(M \cdot \log N)\,\lvert\mathbb{G}_T\rvert$

SumCheck identity, and can cut 50% of the commitments and roughly 30% in SumCheck prover cost.

We note a concurrent work [20] proposing a similar optimization for Lasso. We emphasize that our construction is developed independently of theirs, and we additionally adapt it to the distributed setting to get a more functional distributed ZKP system.

1.1.3. Implementation and Evaluations. We have fully implemented our distributed ZKP system based on the Rust ark-works ecosystem, including both PIOP versions and layered circuit versions, with deMKZG and deDory. Both our systems of HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ show linear scalability in the number of distributed machines. Specifically, with 2 machines, the PIOP versions of HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ can achieve $4.5\times$ and $3.6\times$ speedups over HyperPlonk respectively, while with 32 machines the speedups are $63.1\times$ and $40.2\times$.

We compare our systems with Pianist, using vanilla circuits (of degree-2 multiplication gates) and Jellyfish circuits [21] (of degree-5 custom gates) with size $2^{22} \sim 2^{26}$. The PIOP version of HyperPianist$^{\mathsf{K}}$ presents up to $2.9\times$ and $4.6\times$ speedups over Pianist on the two types of circuits, and HyperPianist$^{\mathsf{D}}$ achieves $2.4\times$ and $3.8\times$ speedups with a transparent setup; the layered circuit versions achieve even greater improvements: HyperPianist$^{\mathsf{K}}$ and HyperPianist$^{\mathsf{D}}$ can be $5.9\times$ and $4.7\times$ as fast on custom gates, respectively. Meanwhile, the communication cost, proof size as well as verifier time of the two systems are highly practical. Our optimized lookup argument also shows an improvement of $2\times$ in prover time on $2^{20} \sim 2^{24}$ XOR gates. We additionally show that even in a general wide area network, our systems still have notable improvements over Pianist for large circuits.

1.2. Other Related Works

1.2.1. Collaborative ZKPs. A series of recent works [22], [23], [24], [25], [26] have focused on distributing the proof generation process while maintaining the privacy of the witnesses. One popular approach relies on the notation of collaborative ZKPs introduced in [22]. This approach consists of two phases: First, each server sends and receives its part of the witness in a secret-sharing form. Then, all servers execute a certain secure multi-party computation (MPC) protocol for the proof generation circuit. We stress that these works are orthogonal to ours: their emphasis is on security and privacy, while we focus on scaling proof generation with sub-provers which trust each other.

1.2.2. Lookup Arguments. Lookup arguments are extensively used in SNARKs due to their efficiency in proving non-arithmetic operations, such as range proofs and bitwise operations. A series of recent works [15], [18], [27], [28], [29], [30], [31] have focused on improving the efficiency of lookup arguments. These works can be categorized into two settings: univariate and multivariate. In the univariate setting, with a one-time expensive setup, prover complexity can be made quasi-linear to the number of queries. In the multivariate setting, Lasso [18] provides a generalized approach for proving lookup arguments on structured tables, significantly enhancing efficiency.

1.3. Organization of the Paper

Section 2 presents the preliminaries. Section 3 introduces the distributed multivariate PIOP system of HyperPianist. Section 4 presents the distributed multivariate polynomial commitment schemes. Section 5 shows the construction of our optimized lookup argument for HyperPianist+. Section 6 gives some experimental results.

2. Preliminaries

2.1. Notations

We use λ to denote the security parameter. A function $f(\lambda)$ is $\mathsf{poly}(\lambda)$ if there exists a $c \in \mathbb{N}$ such that $f(\lambda) = O(\lambda^c)$. If for all $c \in \mathbb{N}$, $f(\lambda)$ is $o(\lambda^{-c})$, then $f(\lambda)$ is in $\mathsf{negl}(\lambda)$ and is said to be negligible. A probability that is

$1-\mathsf{negl}(\lambda)$ is overwhelming. We write all groups *additively*, and assume we are given some method to sample Type III pairings at a given security level. Then we are furnished with a prime field $\mathbb{F} = \mathbb{F}_p$, three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order p, a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, and generators $G_1 \in \mathbb{G}_1$, $G_2 \in \mathbb{G}_2$ such that $e(G_1, G_2)$ generates $\mathbb{G}_T$. We generally suppress the distinction between $e(\cdot,\cdot)$ and multiplication of $\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2$ or $\mathbb{G}_T$ by elements of $\mathbb{F}$, writing all of these bilinear maps as multiplication. We will use bold letters like $\boldsymbol{v}, \boldsymbol{M}$ for vectors and matrices, and use $\langle \cdot, \cdot \rangle$ to denote the generalized inner product given by $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i=1}^{n} a_i b_i$, with signatures: $\mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$, $\mathbb{F}^n \times \mathbb{G}^n_{\{1,2,T\}} \to \mathbb{G}^n_{\{1,2,T\}}$ or $\mathbb{G}_1^n \times \mathbb{G}_2^n \to \mathbb{G}_T$.

For $n \in \mathbb{N}$, let $[n]$ be the set $\{0, 1, \ldots, n-1\}$. Vector, matrix and tensor indices will begin at 0. For any two vectors $\boldsymbol{v}_1, \boldsymbol{v}_2$, we denote their concatenation by $(\boldsymbol{v}_1 || \boldsymbol{v}_2)$. For $\boldsymbol{x}$, we use notation $\boldsymbol{x}[i:k]$ to denote the slices of vector $\boldsymbol{x}$, namely $\boldsymbol{x}[i:k] = (x_i, \ldots, x_{k-1})$. We use $\otimes$ to denote the tensor product. For any vector $\boldsymbol{v}$ of even length we will denote the left and right halves of $\boldsymbol{v}$ by $\boldsymbol{v}_L$ and $\boldsymbol{v}_R$. The natural injection mapping an integer to its binary representation is denoted as $\boldsymbol{bin}(\cdot)$. We write $\leftarrow_{\$} S$ for uniformly random samples of a set S. A relation is a set of pairs $(\mathbb{x}, \mathbb{w})$. An indexed relation is a set of triples $(\mathbb{i}, \mathbb{x}, \mathbb{w})$.

2.2. Security Model and Distribution-Friendliness

Unlike the MPC security model in Collaborative ZKP, we consider the traditional ZKP security model involving only a prover and a verifier. We assume that all sub-provers are controlled by **a single party**, and they coordinate to function as the sole prover in the traditional ZKP setting. Thus, the witness can be distributed among sub-provers in plain-text rather than a secret-sharing form.

Intuitively, we say a protocol is "distribution-friendly" if its communication cost is asymptotically no greater than the proof size. It's a high-level description of the property that captures the typical setting where each sub-prover locally generates some partial results of proof and then sends them to the master prover for aggregation.

2.3. SNARKs

We adopt the definition of SNARKs in HyperPlonk [13].

Definition 1 (Interactive Argument of Knowledge [13])**.** *An interactive protocol $\Pi = (\mathsf{Setup}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ between a prover $\mathcal{P}$ and verifier $\mathcal{V}$ is an argument of knowledge for an indexed relation $\mathcal{R}$ with knowledge error $\delta : \mathbb{N} \to [0,1]$ if the following properties hold, where given an index $\mathbb{i}$, common input $\mathbb{x}$ and prover witness $\mathbb{w}$, the deterministic indexer outputs $(\mathsf{vk}, \mathsf{pk}) \leftarrow \mathcal{I}(\mathbb{i})$ and the output of the verifier is denoted by the random variable $\langle \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathsf{vk}, \mathbb{x}) \rangle$:*

- ***Perfect Completeness:*** *for all $(\mathbb{i}, \mathbb{x}, \mathbb{w}) \in \mathcal{R}$,*
$$\Pr\left[\langle \mathcal{P}(\mathsf{pk}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathsf{vk}, \mathbb{x}) \rangle = 1 \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \end{array}\right] = 1.$$
- ***δ-Knowledge Soundness:*** *There exists a polynomial $\mathsf{poly}(\cdot)$ and a PPT oracle machine $\mathcal{E}$ called the extractor such that given oracle access to any pair of PPT adversarial prover algorithm $(\mathcal{A}_1, \mathcal{A}_2)$, the following holds:*
$$\Pr\left[\begin{array}{c} \langle \mathcal{A}_2(\mathbb{i}, \mathbb{x}, \mathsf{st}), \mathcal{V}(\mathsf{vk}, \mathbb{x}) \rangle = 1 \\ \wedge \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \\ \mathbb{w} \leftarrow \mathcal{E}^{\mathcal{A}_1, \mathcal{A}_2}(\mathsf{pp}, \mathbb{i}, \mathbb{x}) \end{array}\right] \leq \delta(|\mathbb{i}| + |\mathbb{x}|).$$
An interactive protocol is knowledge sound if the knowledge error δ is negligible in λ.
- ***Public coin:*** *An interactive protocol is public-coin if $\mathcal{V}$'s messages are chosen uniformly at random.*

If an interactive argument of knowledge protocol is public-coin, then it can be made non-interactive by the Fiat-Shamir transformation [32]. If the scheme further satisfies the following property:

- ***Succinctness****: The proof size is $|\pi| = \mathsf{poly}(\lambda, \log|\mathcal{C}|)$ and the verification time is $\mathsf{poly}(\lambda, |\mathbb{x}|, \log|\mathcal{C}|)$,*

then it is a Succinct Non-interactive Argument of Knowledge (SNARK).

2.4. Polynomial Interactive Oracle Proof

Definition 2 (Public-coin Polynomial Interactive Oracle Proof [13])**.** *A public-coin polynomial interactive oracle proof (PIOP) is a public-coin interactive proof for a polynomial oracle relation $\mathcal{R} = (\mathbb{i}, \mathbb{x}; \mathbb{w})$, where $\mathbb{i}$ and $\mathbb{x}$ can contain oracles to n-variate polynomials over some field $\mathbb{F}$. These oracles can be queried at arbitrary points in $\mathbb{F}^n$ to evaluate the polynomial at these points. The actual polynomials corresponding to the oracles are contained in pk and $\mathbb{w}$, respectively. We denote an oracle to a polynomial f by $[[f]]$. In each round, $\mathcal{P}$ sends multivariate polynomial oracles, and $\mathcal{V}$ replies with a random challenge.*

2.5. Multilinear Extension

We define the set $\mathcal{F}_n^{(\leq d)}$ to be all n-variate polynomials $f : \mathbb{F}^n \to \mathbb{F}$ where the degree is at most d in each variable. In particular, an n-variate polynomial f is said to be *multilinear* if $f \in \mathcal{F}_n^{(\leq 1)}$. For any $f : \{0,1\}^n \to \mathbb{F}$, there is a unique multilinear polynomial $\tilde{f} : \mathbb{F}^n \to \mathbb{F}$ such that $\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^n$. The polynomial $\tilde{f}$ is called the *multilinear extension* (MLE) of f, and can be expressed as $\tilde{f}(\boldsymbol{X}) = \sum_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{X})$, where $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{X}) := \prod_{i=1}^{n} (\boldsymbol{x}_i \boldsymbol{X}_i + (1 - \boldsymbol{x}_i)(1 - \boldsymbol{X}_i))$.

2.6. Polynomial Commitment Scheme

Definition 3 (Commitment Scheme [33])**.** *A commitment scheme for some space of messages $\mathcal{X}$ is a tuple of three protocols $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$ where*

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{F})$ *generates public parameters* pp.
- $(com, \pi) \leftarrow \mathsf{Commit}(\mathsf{pp}; x)$ *takes as input $x \in \mathcal{X}$; produces a commitment com and an opening hint π.*
- $b \leftarrow \mathsf{Open}(\mathsf{pp}; com, x, \pi)$*: verifies the opening of commitment com to x with the opening hint; outputs $b \in \{0,1\}$.*

Definition 4. *A commitment scheme should satisfy the binding property, meaning that for all PPT adversaries $\mathcal{A}$,*

$$\Pr\left[\begin{matrix} b_0 = b_1 \neq 0 \\ \wedge \\ x_0 \neq x_1 \end{matrix} \middle| \begin{matrix} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (com, x_0, x_1, \pi_0, \pi_1) \leftarrow \mathcal{A}(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, com, x_0, \pi_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, com, x_1, \pi_1) \end{matrix}\right] \leq \mathsf{negl}(\lambda).$$

One of our key building blocks is the Dory [17] commitment scheme, which makes use of the Pedersen and AFGHO commitments. For messages $\mathcal{X} = \mathbb{F}^n$ and any $i \in \{1, 2, T\}$, the Pedersen commitment scheme is defined as:

Definition 5 (Pedersen Commitment [34])**.**

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (g \leftarrow_\$ \mathbb{G}_i^n, h \leftarrow_\$ \mathbb{G}_i)$
- $(com, \pi) \leftarrow \mathsf{Commit}(\mathsf{pp}; x) = \{r \leftarrow_\$ \mathbb{F}; (\langle x, g\rangle) + rh, r)\}$
- $\mathsf{Open}(\mathsf{pp}; com, x, \pi)$: *Check* $\langle x, g\rangle + \pi \cdot h = com$.

AFGHO commitment is a structure-preserving commitment to group elements. In this case we have $\mathcal{X} = \mathbb{G}_i^n$ for $i \in \{1, 2\}$, and we have:

Definition 6 (AFGHO Commitment [35])**.**

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (g \leftarrow_\$ \mathbb{G}_{3-i}^n, H_1 \leftarrow_\$ \mathbb{G}_1, H_2 \leftarrow_\$ \mathbb{G}_2)$*;*
- $(com, \pi) \leftarrow \mathsf{Commit}(\mathsf{pp}; x) = \{r \leftarrow_\$ \mathbb{F}; (\langle x, g\rangle) + r \cdot e(H_1, H_2), r)\}$*;*
- $\mathsf{Open}(\mathsf{pp}; com, x, \pi)$*: Check* $\langle x, g\rangle + \pi \cdot e(H_1, H_2) = com$.

Let $(\mathsf{Gen}_\mathbb{F}, \mathsf{Commit}_\mathbb{F}, \mathsf{Open}_\mathbb{F})$ be a commitment scheme for $\mathbb{F}$ with public parameters $\mathsf{pp}_\mathbb{F}$. The polynomial commitment scheme for multilinear polynomials is defined as:

Definition 7 (Polynomial Commitment Scheme [17])**.** *A tuple of protocols* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ *is a polynomial commitment scheme for* n*-variate multilinear polynomials if*

- $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$ *is a commitment scheme for* n*-variate multilinear polynomials* f*, and*
- Eval *is an interactive argument of knowledge for:* $\mathcal{R}_{\mathsf{Eval}}(\mathsf{pp}, \mathsf{pp}_\mathbb{F}) = \{((com_f, \boldsymbol{x}, com_v); (f, \pi_f, v, \pi_v))\}$ *s.t.* $\boldsymbol{x} \in \mathbb{F}^n$*,* $f \in \mathcal{F}_n^{(\leq 1)}$*,* $f(\boldsymbol{x}) = v$*,* $\mathsf{Open}(\mathsf{pp}; com_f, f, \pi_f) = 1$ *and* $\mathsf{Open}_\mathbb{F}(\mathsf{pp}_\mathbb{F}; com_v, v, \pi_v) = 1$.

3. Distributed Multivariate PIOP System

In this section, we introduce the distributed multivariate PIOP system of HyperPianist(+). We give an overview of the PIOP system in Figure 1. At a high level, HyperPianist follows HyperPlonk to decompose the circuit constraints into gate identities and wiring identities (i.e., copy constraints). It differs from HyperPlonk at the Multiset Check PIOP, which avoids linear communication costs in the distributed setting. In this section, we will present the PIOP system in a bottom-up fashion, starting from the distributed SumCheck PIOP.

3.1. Starting Point: Distributed SumCheck PIOP

Definition 8 (SumCheck Relation [13])**.** *The relation* $\mathcal{R}_{Sum}$ *is the set of all tuples* $(\mathbb{x}; \mathbb{w}) = ((v, [[f]]); f)$ *where* $f \in \mathcal{F}_n^{(\leq d)}$ *and* $\sum_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x}) = v$.

We first review the well-known multivariate SumCheck PIOP in the non-distributed setting. Given an n-variate multilinear polynomial $f \in \mathcal{F}_n^{(\leq 1)}$ and its oracle $[[f]]$, the prover

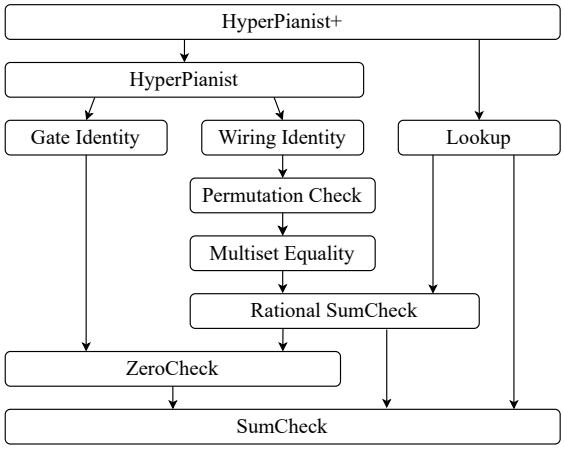

Figure 1: Overview of our multivariate PIOP system.

$\mathcal{P}$ and the verifier $\mathcal{V}$ engage in n rounds of interactions. In each round k, $\mathcal{P}$ sends a univariate polynomial

$$f_k(X_k) := \sum_{\boldsymbol{x} \in \{0,1\}^{n-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}). \quad (1)$$

$\mathcal{V}$ checks $f_k(0) + f_k(1) = f_{k-1}(r_{k-1})$ to assure the correctness of the previous round, and then sends back the next challenge r_k. In the last round, $\mathcal{V}$ checks the final claim using an oracle call to $[[f]]$.

In the distributed setting, our starting point is the distributed SumCheck in deVirgo [8]. The main focus of deVirgo is to aggregate multiple SumCheck instances for data-parallel circuits. We generalize it, and distribute a single SumCheck instance across multiple machines, assuming that each machine initially holds part of the witnesses.

Without loss of generality, suppose we have $M = 2^m$ distributed machines acting as sub-provers. For an n-variate polynomial f, each sub-prover holds 2^{n-m} witnesses. We assume that for the i-th sub-prover P_i, its 2^{n-m} witnesses are indexed by $(x_1, \ldots, x_{n-m}, \boldsymbol{bin}(i))$ where x_1, $\ldots, x_{n-m} \in \{0, 1\}$, and $\boldsymbol{bin}(i)$ is the binary representation of the value i. In other words, each sub-prover P_i is allocated with a fixed binary suffix $\boldsymbol{bin}(i)$ in its indices. Following deVirgo [8], we define $f^{(i)}(\boldsymbol{x}) := f(\boldsymbol{x}, \boldsymbol{bin}(i))$, and have

$$\begin{aligned}\sum_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x}) &= \sum_{\boldsymbol{x} \in \{0,1\}^{n-m}} \sum_{\boldsymbol{bin}(i) \in \{0,1\}^m} f(\boldsymbol{x}, \boldsymbol{bin}(i)) \\ &= \sum_{i \in [M]} \sum_{\boldsymbol{x} \in \{0,1\}^{n-m}} f^{(i)}(\boldsymbol{x}).\end{aligned}$$

Given this construction, it follows that in each round k where $1 \leq k \leq n - m$, each sub-prover $\mathcal{P}_i$ is able to locally compute a univariate polynomial $f_k^{(i)}(X_k) := \sum_{\boldsymbol{x} \in \{0,1\}^{n-m-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}, \boldsymbol{bin}(i))$, and

$$\begin{aligned}\sum_{i \in [M]} f_k^{(i)}(X_k) &= \sum_{\boldsymbol{bin(i)} \in \{0,1\}^m} f_k^{(i)}(X_k) \\ &= \sum_{\boldsymbol{x} \in \{0,1\}^{n-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}).\end{aligned}$$

The RHS is exactly the univariate polynomial $f_k(X_k)$ de-

PROTOCOL 3.1. *Distributed SumCheck PIOP (for Multilinear Polynomials)*

$\mathcal{P}_0$ claims $((v, [[f]]); f) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$ where $f \in \mathcal{F}_n^{(\leq 1)}$. $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $i \in [M]$. Let $f_0 := v$.

- In the k-th round where $1 \leq k \leq n-m$:
  - Each $\mathcal{P}_i$ sends a local univariate polynomial $f_k^{(i)}(X_k) := \sum_{\boldsymbol{x} \in \{0,1\}^{n-m-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_k(X_k) = \sum_{i \in [M]} f_k^{(i)}(X_k)$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks if $f_{k-1} = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random $r_k \in \mathbb{F}$ to $\mathcal{P}_0$, and sets $f_k := f_k(r_k)$.
  - $\mathcal{P}_0$ transmits r_k to the other $\mathcal{P}_i$. Each $\mathcal{P}_i$ then updates the local evaluations of $f^{(i)}(\boldsymbol{x})$.
- Each $\mathcal{P}_i$ sends $f^{(i)}(r_1, \cdots, r_{n-m})$ to $\mathcal{P}_0$. $\mathcal{P}_0$ constructs the polynomial $f'(\boldsymbol{x}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{x})$.
- $\mathcal{P}_0$ and $\mathcal{V}$ run the SumCheck PIOP to check $f_{n-m} = \sum_{\boldsymbol{x} \in \{0,1\}^m} f'(\boldsymbol{x})$.

fined in Equation (1) in the regular SumCheck PIOP. Thus, by letting a master prover, say, $\mathcal{P}_0$, collect and aggregate all $f_k^{(i)}(X_k)$ from the sub-provers, it can recover $f_k(X_k)$, and then interact with $\mathcal{V}$ as in the regular SumCheck PIOP. After receiving the random challenge r_k from $\mathcal{V}$ in each round, $\mathcal{P}_0$ needs to transmit it to the other sub-provers.

After the $(n-m)$-th round, the polynomial to be checked is reduced to a m-varaite polynomial denoted by $f'(\boldsymbol{x}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{x})$ where $\boldsymbol{x} \in \{0,1\}^m$. At this point, the left SumCheck computations can not be distributed, and need to be performed solely by the master prover. Specifically, all sub-provers send the evaluation $f_{n-m}^{(i)}(r_1, \cdots, r_{n-m})$ to $\mathcal{P}_0$, who then constructs the polynomial $f'(\boldsymbol{x})$ by $f'(\boldsymbol{bin}(i)) = f_{n-m}^{(i)}(r_1, \cdots, r_{n-m})$ for all $i \in [M]$. In the following m rounds, $\mathcal{P}_0$ acts as the single prover in the regular SumCheck PIOP on the claim $\sum_{\boldsymbol{x} \in \{0,1\}^m} f'(\boldsymbol{x}) = f_{n-m}(r_{n-m})$. In the last round, $\mathcal{V}$ invokes an oracle call to evaluate $f(r_1, \cdots, r_n)$ and checks the final claim. We present the details in Protocol 3.1.

For SumCheck claims on high-degree polynomials $f \in \mathcal{F}_n^{(\leq d)}$, HyperPlonk has proposed an algorithm with a prover time of $O(d \log^2 d \cdot 2^n)$. In this context, $f(\boldsymbol{x}) = h(g_0(\boldsymbol{x}), \ldots, g_{c-1}(\boldsymbol{x}))$ such that h can be evaluated using $O(d)$ operations and $g_i \in \mathcal{F}_n^{(\leq 1)}, \forall i \in [c]$. This algorithm can be similarly adapted to the distributed setting, where we assume each $\mathcal{P}_i$ holds partial polynomials $g_0^{(i)}, \ldots, g_{c-1}^{(i)}$, and it knows the description of h. In the following, we will abuse the notation of $f^{(i)}$ to refer to this allocation of g and h. The detailed construction is given in Protocol A.1.

3.2. Distributed ZeroCheck PIOP

A ZeroCheck relation shows a multivariate polynomial evaluates to zero everywhere on the boolean hypercube.

Definition 9 (ZeroCheck Relation [13])**.** *The relation $\mathcal{R}_{Zero}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = (([[f]]); f)$ where $f \in \mathcal{F}_n^{(\leq d)}$ and $f(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$.*

In the non-distributed setting as in HyperPlonk, the ZeroCheck relation is reduced to a SumCheck relation that $\sum_{\boldsymbol{x}} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}) f(x) = 0$ using a random challenge $\boldsymbol{r}$ from the verifier; this reduction has negligible soundness error.

In the distributed setting, given the random challenge vector $\boldsymbol{r}$, the i-th sub-prover $\mathcal{P}_i$ can construct its local witness to $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{r})$ by splitting the random challenge into $(\boldsymbol{r'}, \boldsymbol{r''}) \in \mathbb{F}^{n-m} \times \mathbb{F}^m$ and calculating the evaluation table for $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{r'}) \cdot \widetilde{eq}(\boldsymbol{bin}(i), \boldsymbol{r''})$. From here, the adaptation of ZeroCheck PIOP to the distributed setting follows naturally. The detailed construction is given in Protocol 3.2.

3.3. Distributed Multiset Check PIOP

A *multiset* is an extension of the concept of a set, where every element has a positive multiplicity. Two finite multisets are equal if they contain the same elements with the same multiplicities. A Multiset Check (MsetCheck) relation is defined as follows.

Definition 10 (Multiset Check Relation [13])**.** *For any $k \geq 1$, the relation $\mathcal{R}_{MSet}^k$ is the set of all tuples*

$$(\mathbb{x}; \mathbb{w}) = (([[f_0]], \ldots, [[f_{k-1}]], [[g_0]], \ldots, [[g_{k-1}]]); (f_0, \ldots, f_{k-1}, g_0, \ldots, g_{k-1}))$$

where $f_j, g_j \in \mathcal{F}_n^{(\leq d)}, \forall j \in [k]$ and the following two multisets of tuples are equal: $\{(f_0(\boldsymbol{x}), \ldots, f_{k-1}(\boldsymbol{x}))\}_{\boldsymbol{x} \in \{0,1\}^n} = \{(g_0(\boldsymbol{x}), \ldots, g_{k-1}(\boldsymbol{x}))\}_{\boldsymbol{x} \in \{0,1\}^n}$.

Using a random challenge γ, a k-dimension multiset check relation $\mathcal{R}_{\text{MSet}}^k$ can be reduced to a one-dimension relation $\mathcal{R}_{\text{MSet}}^1$ showing that $\{\sum_{i \in [k]} \gamma^i f_i(\boldsymbol{x})\}_{\boldsymbol{x} \in \{0,1\}^n} = \{\sum_{i \in [k]} \gamma^i g_i(\boldsymbol{x})\}_{\boldsymbol{x} \in \{0,1\}^n}$. With another random challenge β, this can be further reduced to a grand product check that

$$\prod_{\boldsymbol{x} \in \{0,1\}^n} \Big(\beta + \sum_{i \in [k]} \gamma^i f_i(\boldsymbol{x})\Big) = \prod_{\boldsymbol{x} \in \{0,1\}^n} \Big(\beta + \sum_{i \in [k]} \gamma^i g_i(\boldsymbol{x})\Big). \quad (2)$$

We formally define the grand product check relation below.

Definition 11 (Product Check Relation [13])**.** *The relation $\mathcal{R}_{Prod}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = ((s, [[f_0]], [[f_1]]); (f_0, f_1))$ where $f_0, f_1 \in \mathcal{F}_n^{(\leq d)}, f_1(\boldsymbol{b}) \neq 0, \ \forall \boldsymbol{b} \in \{0,1\}^n$ and $\prod_{\boldsymbol{x} \in \{0,1\}^n} f'(\boldsymbol{x}) = s$, where $f' = f_0/f_1$.*

HyperPlonk [13] utilizes the product check PIOP from Quarks [14], which relies on the following theorem.

Theorem 1 ([14])**.** *$P = \prod_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x})$ iff there exists $h \in \mathcal{F}_{n+1}^{(\leq 1)}$ s.t. $h(1, \ldots, 1, 0) = P$, and $\forall \boldsymbol{x} \in \{0,1\}^n$, the following hold: $h(0, \boldsymbol{x}) = f(\boldsymbol{x}), h(1, \boldsymbol{x}) = h(\boldsymbol{x}, 0) \cdot h(\boldsymbol{x}, 1)$.*

Given Theorem 1, to prove a product check relation, it suffices to show the existence of such a multilinear polynomial h. Specifically, $\mathcal{P}$ can provide an oracle purported to be such an h, and proves that (1) $h(1, \cdots, 1, 0) = P$, (2) $h(0, \boldsymbol{x}) = f(\boldsymbol{x})$, and (3) $h(1, \boldsymbol{x}) - h(\boldsymbol{x}, 0) \cdot h(\boldsymbol{x}, 1) = 0$ for all

PROTOCOL 3.2. *Distributed ZeroCheck PIOP*

$\mathcal{P}_0$ claims $(([[f]]); f) \in \mathcal{R}_{\text{Zero}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $i \in [M]$.

- $\mathcal{V}$ samples $\boldsymbol{r} \leftarrow_{\$} \mathbb{F}^n$ and sends it to $\mathcal{P}_0$. $\mathcal{P}_0$ transmits $\boldsymbol{r}$ to the other $\mathcal{P}_i$.
- Each $\mathcal{P}_i$ views $\boldsymbol{r}$ as $\boldsymbol{r} = (\boldsymbol{r}', \boldsymbol{r}'') \in \mathbb{F}^{n-m} \times \mathbb{F}^m$, and locally computes $\hat{f}^{(i)}(\boldsymbol{x}) = f^{(i)}(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}') \cdot \widetilde{eq}(\boldsymbol{bin(i)}, \boldsymbol{r}'')$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed SumCheck PIOP (Protocol 3.1) to check $((0, [[\hat{f}]]); \hat{f}) \in \mathcal{R}_{\text{Sum}}$.

PROTOCOL 3.3. *Distributed Rational SumCheck PIOP*

$\mathcal{P}_0$ claims $((v, [[p]], [[q]]); (p, q)) \in \mathcal{R}_{\text{RSum}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $p^{(i)}(\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{bin(i)})$, $q^{(i)}(\boldsymbol{x}) = q(\boldsymbol{x}, \boldsymbol{bin(i)})$, $i \in [M]$.

- Each $\mathcal{P}_i$ computes $f^{(i)}(\boldsymbol{x}) = \frac{1}{q^{(i)}(\boldsymbol{x})}$, $\forall \boldsymbol{x} \in \{0,1\}^{n-m}$.
- $\mathcal{P}_0$ sends an oracle $[[f]]$ to $\mathcal{V}$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed ZeroCheck PIOP (Protocol 3.2) to check $(([[f \cdot q - 1]]); f \cdot q - 1) \in \mathcal{R}_{\text{Zero}}$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed SumCheck PIOP (Protocol 3.1) to check $((v, [[p \cdot f]]); p \cdot f) \in \mathcal{R}_{\text{Sum}}$.

PROTOCOL 3.4. *Distributed Multiset Check PIOP*

$\mathcal{P}_0$ claims $(([[f_0]], \ldots, [[f_{k-1}]], [[g_0]], \ldots, [[g_{k-1}]]) \in \mathcal{R}^k_{\text{MSet}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $f^{(i)}_j(\boldsymbol{x})$, $g^{(i)}_j(\boldsymbol{x})$, $\forall j \in [k], i \in [M]$.

- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_{\$} \mathbb{F}$ and sends them to $\mathcal{P}_0$. $\mathcal{P}_0$ transmits them to the other $\mathcal{P}_i$.
- Each $\mathcal{P}_i$ computes $f'^{(i)}(\boldsymbol{x}) = \sum_{j \in [k]} \gamma^j f^{(i)}_j(\boldsymbol{x})$, $g'^{(i)}(\boldsymbol{x}) = \sum_{j \in [k]} \gamma^j g^{(i)}_j(\boldsymbol{x})$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed Rational SumCheck PIOP (Protocol 3.3) to check $((0, [[f' - g']], [[(f' + \beta) \cdot (g' + \beta)]]); (f' - g', (f' + \beta) \cdot (g' + \beta))) \in \mathcal{R}_{\text{RSum}}$.

$\boldsymbol{x} \in \{0,1\}^n$. Condition (1) is straightforward to verify with an oracle call, while conditions (2) and (3) can be verified using the ZeroCheck PIOP.

Problems When Distributing Product Check PIOP. Now we focus on the distributed setting. Recall that after reducing to the product check identity, each sub-prover $\mathcal{P}_i$ holds its local sub-polynomial $f^{(i)}(\boldsymbol{x})$. To apply the distributed ZeroCheck PIOP, $\mathcal{P}_i$ needs to construct its sub-polynomial $h^{(i)}(\boldsymbol{x}) := h(\boldsymbol{x}, \boldsymbol{bin}(i))$. In Quarks [14], h is constructed using the following criteria: (1) $h(1, \cdots, 1) = 0$, and (2) for all $\ell \in [0, n]$ and $\boldsymbol{x} \in \{0,1\}^{n-\ell}$, $h(1^\ell, 0, \boldsymbol{x}) = \prod_{\boldsymbol{y} \in \{0,1\}^\ell} v(\boldsymbol{x}, \boldsymbol{y})$. Unfortunately, in the distributed setting $\mathcal{P}_i$ is unable to locally construct $h^{(i)}$ from $f^{(i)}$. Due to the definition of h, $\mathcal{P}_i$ needs some necessary values from other sub-provers to construct $h^{(i)}$, which would incur a linear communication cost. We thus need to seek other approaches.

Our Solution: Rational SumCheck. Our key insight here is to use logarithmic derivatives techniques [15] to construct a PIOP for multiset check relations, where the helper polynomials are friendly to distribution. We explain this below.

Definition 12 (Logarithmic Derivatives)**.** *The logarithmic derivative of a polynomial $p(X)$ over a field $\mathbb{F}$ is the rational function $p'(X)/p(X)$. In particular, when the polynomial $p(X) = \prod_{i=1}^n (X + z_i)$, with each $z_i \in \mathbb{F}$, the logarithmic derivative of it is equal to $\frac{p'(X)}{p(X)} = \sum_{i=1}^n \frac{1}{X + z_i}$.*

Our construction relies on the following theorem.

Theorem 2 ([15])**.** *Let $(a_i)_{i=1}^n$ and $(b_i)_{i=1}^n$ be sequences of a field $\mathbb{F}$ where $|\mathbb{F}| > n$. Then $\prod_{i=1}^n (a_i + X) = \prod_{i=1}^n (b_i + X)$ in the polynomial ring $\mathbb{F}[X]$ iff $\sum_{i=1}^n \frac{1}{a_i + X} = \sum_{i=1}^n \frac{1}{b_i + X}$ in the fractional field $\mathbb{F}(X)$.*

Recall that in HyperPlonk the multiset check identity is reduced to Equation (2). Instead of the grand product check, with Theorem 2 we can reduce it to a "SumCheck relation on the fractions". We will call it a Rational SumCheck (RSumCheck) relation, defined as follows.

Definition 13 (Rational SumCheck Relation)**.** *The relation $\mathcal{R}_{RSum}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = ((v, [[p]], [[q]]); (p, q))$, where $p, q \in \mathcal{F}_n^{(\leq d)}$, $q(\boldsymbol{x}) \neq 0, \forall \boldsymbol{x} \in \{0,1\}^n$ and $\sum_{\boldsymbol{x} \in \{0,1\}^n} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} = v$.*

It is in the form of SumCheck, but the SumCheck PIOP does not apply directly to fractions. As a workaround, we can find the multilinear interpolation of the denominator part $f(\boldsymbol{x}) = 1/q(\boldsymbol{x})$. This allows us to reduce the RSumCheck identity to a regular SumCheck, along with a ZeroCheck to guarantee the well formation of the helper polynomial f.

A key feature of the our RSumCheck PIOP is that it's well suited to the distributed setting: given the partial polynomials $q^{(i)}(\boldsymbol{x})$, the corresponding $f^{(i)}(\boldsymbol{x})$ can be computed locally by the sub-provers. We present the distributed RSumCheck PIOP in Protocol 3.3.

Theorem 3. *The PIOP for $\mathcal{R}_{RSum}$ in Protocol 3.3 is perfectly complete and has knowledge error $O(dn/|\mathbb{F}|)$.*

We give the proof of this theorem in Appendix G.1.

For the Multiset check, one straightforward solution is to prove that $v_1 = \sum_{\boldsymbol{x} \in \{0,1\}^n} 1/(\beta + \sum_{j \in [k]} \gamma^j f_j(\boldsymbol{x}))$ and $v_2 = \sum_{\boldsymbol{x} \in \{0,1\}^n} 1/(\beta + \sum_{j \in [k]} \gamma^j g_j(\boldsymbol{x}))$ are equal, with two invocations of the RSumCheck PIOP. We further observe that we can batch the RSumChecks into a single check by instead showing that the difference $1/(\beta + \sum_{j \in [k]} \gamma^j f_j(\boldsymbol{x})) - 1/(\beta + \sum_{j \in [k]} \gamma^j g_j(\boldsymbol{x}))$, which is also a fraction, sums to 0 on the hypercube. This saves one

TABLE 3: The complexity of distributed PIOPs (k in MsetCheck is the length of each element in the multisets).

PIOP	$\mathcal{P}_i$ Time	$\mathcal{P}_0$ Extra Time	$\mathcal{V}$ Time	# of Queries	# of Rounds	Communication	# of Extra Oracles
SumCheck	$O(d\log^2 d\cdot 2^{n-m})$	$O\big((d\cdot n + d\log^2 d)\cdot 2^m\big)$	$O(d\cdot n)$	1	n	$O(d\cdot n)$	0
ZeroCheck	$O(d\log^2 d\cdot 2^{n-m})$	$O\big((d\cdot n + d\log^2 d)\cdot 2^m\big)$	$O(d\cdot n)$	1	$n+1$	$O(d\cdot n)$	0
RSumCheck	$O(d\log^2 d\cdot 2^{n-m})$	$O\big((d\cdot n + d\log^2 d)\cdot 2^m\big)$	$O(d\cdot n)$	4	$2n+1$	$O(d\cdot n)$	1
MsetCheck	$O(d\log^2 d\cdot 2^{n-m})$	$O\big((d\cdot n + d\log^2 d)\cdot 2^m\big)$	$O(d\cdot n)$	$2k+2$	$2n+1$	$O(d\cdot n)$	1
PermCheck	$O(d\log^2 d\cdot 2^{n-m})$	$O\big((d\cdot n + d\log^2 d)\cdot 2^m\big)$	$O(d\cdot n)$	6	$2n+1$	$O(d\cdot n)$	1

call to the RSumCheck PIOP. We present our distributed MsetCheck PIOP in Protocol 3.4.

Remark 1. *Using layered circuits to directly prove the product check relation in multiset check PIOP is also distribution-friendly. It saves two commitments after instantiation, with $O(\log^2 N)$ communication cost and proof size. We give a detailed description in Appendix E.*

3.4. Distributed Permutation Check PIOP

For two polynomials $f, g \in \mathcal{F}_n^{(\le d)}$, a permutation relation shows the evaluations of g is a predefined permutation σ of f's evaluations on the boolean hypercube. We formalize the Permutation Check (PermCheck) relation below.

Definition 14 (Permutation Check Relation [13])**.** *The indexed relation $\mathcal{R}_{Perm}$ is the set of tuples $(\mathbb{i};\mathbb{x};\mathbb{w}) = (\sigma; ([[f]],[[g]]);(f,g))$, where $f, g \in \mathcal{F}_n^{(\le d)}$, σ is a permutation $\{0,1\}^n \to \{0,1\}^n$ s.t. for all $\boldsymbol{x} \in \{0,1\}^n$, $f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x})$.*

Given a predefined permutation σ, HyperPlonk introduces two polynomials $s_{id}, s_\sigma \in \mathcal{F}_n^{(\le 1)}$ where s_{id} maps each binary vector $\boldsymbol{x} \in \{0,1\}^n$ to the corresponding integer value $[\boldsymbol{x}] = \sum_{i=1}^n x_i \cdot 2^{i-1} \in \mathbb{F}$ and s_σ maps $\boldsymbol{x} \in \{0,1\}^n$ to $[\sigma(\boldsymbol{x})]$. Then the permutation check can be reduced to a multiset check based on the observation that $f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x})$ iff the two multisets $\{([\boldsymbol{x}], f(\boldsymbol{x}))\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\{([\sigma(\boldsymbol{x})], g(\boldsymbol{x}))\}_{\boldsymbol{x}\in\{0,1\}^n}$ are equal. Thus, we can construct the distribuetd PermCheck PIOP by directly calling the distributed MsetCheck PIOP for $\mathcal{R}^2_{\text{MSet}}$ on the above two multisets, assuming each $\mathcal{P}_i$ is assigned its sub-polynomial $s_{id}^{(i)}(\boldsymbol{x}) = s_{id}(\boldsymbol{x}, \boldsymbol{bin}(i))$ and $s_\sigma^{(i)}(\boldsymbol{x}) = s_\sigma(\boldsymbol{x}, \boldsymbol{bin}(i))$.

3.5. Constraint System and Complexity Analysis

As shown in Figure 1, HyperPianist adopts the Plonkish constraint system, and we present it in Appendix B.

We present the complexity analysis of our distributed PIOPs in Table 3. For the SumCheck PIOP, using Protocol A.1, each sub-prover's workload is $O(d\log^2 d\cdot 2^{n-m})$ for polynomials $f \in \mathcal{F}_n^{(\le d)}$. The additional workload for $\mathcal{P}_0$ is $O(d\cdot 2^m)$ per round in the first $n-m$ rounds to sum 2^m degree-d univariate polynomials, and $O(d\log^2 d\cdot 2^m)$ to perform the remaining SumCheck. As to communication, each sub-prover must send a degree-d univariate polynomial, and retrieve the random challenge from $\mathcal{P}_0$ in each round. Consequently, the total communication overhead for each sub-prover is $O(d\cdot n)$ over $O(n)$ rounds. The ZeroCheck PIOP requires an additional communication round for random challenges. The RSumCheck PIOP requires an extra oracle of the helper polynomial, and an invocation of ZeroCheck and SumCheck respectively. The MsetCheck PIOP needs an additional round for random challenges, and one invocation of the RSumCheck PIOP. We emphasize that the number of rounds presented here is not optimal, as communication rounds from sub-protocols can often be parallelized without loss of soundness.

Remark 2. *The above distributed PIOPs are not zero-knowledge, but ZK properties can be achieved using the standard techniques from HyperPlonk [13, Appendix A]. We omit the details due to the page limit.*

4. Distributed Multivariate PCS

In this section, we present two distributed multivariate polynomial commitment schemes, deMKZG and deDory.

4.1. deMKZG: Distributed Multivariate KZG

We first review the original multivariate KZG scheme.

4.1.1. Review: multivariate KZG [16]. Given a multilinear polynomial $f \in \mathcal{F}_n^{(\le 1)}$, the commitment is computed as (here we write $[a]_1 := a\cdot G_1$ and $[b]_2 := b\cdot G_2$, for some fixed generator G_1 and G_2 for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively):

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$: Samples trapdoor $\boldsymbol{t}$, and generates
$$\mathsf{pp} = \left\{[1]_1, [1]_2, \{[\widetilde{eq}(\boldsymbol{x}, \boldsymbol{t})]_1\}_{\boldsymbol{x}\in\{0,1\}^n}, \{[t_i]_2\}_{t_i\in\boldsymbol{t}}\right\}.$$
- $com_f \leftarrow \mathsf{Commit}(\mathsf{pp}; f)$: Computes $com_f = \sum_{\boldsymbol{x}} f(\boldsymbol{x})\cdot [\widetilde{eq}(\boldsymbol{x}, \boldsymbol{t})]_1$.
- $\mathsf{Open}(\mathsf{pp}; com_f, f)$: Check $com_f = \sum_{\boldsymbol{x}} f(\boldsymbol{x})\cdot[\widetilde{eq}(\boldsymbol{x}, \boldsymbol{t})]_1$.

To open f at a random point $\boldsymbol{r}$ as v, the prover $\mathcal{P}$ needs to calculate n polynomial divisions to obtain a sequence of quotient polynomials $Q_k(\boldsymbol{X_{k+1}})$ and remainder polynomials $R_k(\boldsymbol{X_{k+1}})$, where $\boldsymbol{X_{k+1}}$ denotes $(X_{k+1}, \ldots, X_n)$. Let $R_0 := f(\boldsymbol{X})$, then for $k = 1$ to n, $\mathcal{P}$ computes:

$$R_{k-1}(\boldsymbol{X_k}) := Q_k(\boldsymbol{X_{k+1}})\cdot(X_k - r_k) + R_k(\boldsymbol{X_{k+1}}). \quad (3)$$

After obtaining these polynomials, the prover computes the proof as commitments to all Q_k, and the verifier checks

$$e(com_f - [v]_1, [1]_2) = \textstyle\sum_{k=1}^n e(com_{Q_k}, [t_k - r_k]_2).$$

4.1.2. Construction of deMKZG. Recall that in the distributed setting, each sub-prover $\mathcal{P}_i$ holds a sub-polynomial $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$. Thus, the commitment scheme naturally admits distribution, with each $\mathcal{P}_i$ computing

$$com_f^{(i)} := \textstyle\sum_{\boldsymbol{x}\in\{0,1\}^{n-m}} f^{(i)}(\boldsymbol{x})\cdot[\widetilde{eq}(\boldsymbol{x}||\boldsymbol{bin(i)}, \boldsymbol{t})]_1,$$

and sending it to $\mathcal{P}_0$ for aggregation.

To generate the evaluation proof distributively, we use the observation that in Equation 3, the polynomials $Q_k(\boldsymbol{X_{k+1}})$ and $R_k(\boldsymbol{X_{k+1}})$ are multilinear. Thus they are fully determined by their evaluations over $\boldsymbol{x} \in \{0,1\}^{n-i}$. Given the evaluations of R_{i-1}, by Equation 3, the prover can compute the evaluations for Q_k and R_k as:

$$Q_k(\boldsymbol{x}) := R_{k-1}(1, \boldsymbol{x}) - R_{k-1}(0, \boldsymbol{x}),$$
$$R_k(\boldsymbol{x}) := (1 - r_k) \cdot R_{k-1}(0, \boldsymbol{x}) + r_k \cdot R_{k-1}(1, \boldsymbol{x}).$$

Thus, each sub-prover $\mathcal{P}_i$ can construct $Q_k^{(i)}$ and $R_k^{(i)}$ from the local witnesses $f^{(i)}$, and the distribution of the evaluation protocol follows naturally. We formalize the deMKZG PCS in Protocol C.1.

Theorem 4. *Given polynomial $f(\boldsymbol{x}) \in \mathcal{F}_n^{(\leq 1)}$ and 2^m sub-prover, Protocol C.1 is a PCS satisfying completeness and knowledge soundness for arbitrary evaluation. Each $\mathcal{P}_i$ computation consists of $O(2^{n-m})$ group operations, while $\mathcal{P}_0$ needs $O(2^m)$ additional group operations. The communication cost of each $\mathcal{P}_i$ is $O(n)$. The commitment size is 1 $\mathbb{G}$ element, and the proof size is $O(n)$ $\mathbb{G}$ elements. The verification cost is $O(n)$ pairings.*

We give the proof of Theorem 4 in Appendix G.2.

4.2. deDory: Distributed Dory

We now present deDory. It is built on the Dory PCS, and unlike deMKZG, it requires no trusted setup.

4.2.1. Review: Dory PCS [17]. Dory uses the following matrix representation for multilinear polynomials:

Definition 15 (Matrix Representation of Multilinear Polynomials [17])**.** *For a multilinear polynomial $f : \mathbb{F}^n \to \mathbb{F}$, w.l.o.g., we assume n is even and let $k := n/2$. Then f can be represented as a matrix $\boldsymbol{M} = (\boldsymbol{M}_{ij})$, where $\boldsymbol{M}_{ij} = f(x_1, \ldots, x_n)$ for any $(x_1, \ldots, x_n) \in \{0,1\}^n$ and $i = \sum_{t=1}^{k} 2^{k-t} \cdot x_t, j = \sum_{t=k+1}^{n} 2^{n-t} \cdot x_t$.*

Given the above matrix representation, Dory proposes a two-tiered homomorphic commitment scheme [36] by combining the Pedersen and AFGHO commitments:

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (\Gamma_1 \leftarrow_\$ \mathbb{G}_1^m, \Gamma_2 \leftarrow_\$ \mathbb{G}_2^n)$;
- $com_{\boldsymbol{M}} \leftarrow \mathsf{Commit}(\mathsf{pp}; \boldsymbol{M}_{ij})$:

$$\boldsymbol{com_{row}} \leftarrow \left\{\mathsf{Commit}_{Ped}(\Gamma_1; \boldsymbol{M}_{ij})\right\}_{i \in [n]};$$
$$com_{\boldsymbol{M}} \leftarrow \mathsf{Commit}_{AFGHO}(\Gamma_2; \boldsymbol{com_{row}});$$

- $\mathsf{Open}(\mathsf{pp}; com_{\boldsymbol{M}}, \boldsymbol{M})$: Check $\sum_{i,j} \boldsymbol{M}_{ij} \Gamma_{1j} \Gamma_{2i} = com_{\boldsymbol{M}}$.

The evaluation proof of Dory builds on the following observation: given the matrix representation of f, the evaluation of f at a point $(r_1, \ldots, r_n) \in \mathbb{F}^n$ can be written as a form of vector-matrix-vector product:

$$f(r_1, \ldots, r_n) = (\otimes_{k \leq n/2} \boldsymbol{v_k})^T \boldsymbol{M} (\otimes_{k > n/2} \boldsymbol{v_k}), \quad (4)$$

where $\boldsymbol{v_k} = (1 - r_k, r_k)$. Dory defines the following relation to capture the vector-matrix-vector product identity:

Definition 16 (Vector-Matrix-Vector Relation [17])**.** *Let $\boldsymbol{L}, \boldsymbol{R} \in \mathbb{F}^n$ be public vectors, $\boldsymbol{M} \in \mathbb{F}^{n \times n}$ be the secret matrix, $y = \boldsymbol{L}^T \boldsymbol{M} \boldsymbol{R}$, $com_{\boldsymbol{M}}$ be the commitment to $\boldsymbol{M}$ using the two-tiered commitment, and com_y be the Pedersen commitment to y. The relation $\mathcal{R}_{VMV}$ is the set of all tuples $((\boldsymbol{L}, \boldsymbol{R}, com_{\boldsymbol{M}}, com_y); (\boldsymbol{M}, y))$.*

To prove the opening of f at the point $(r_1, \cdots, r_n) \in \mathbb{F}^n$, it suffices to prove the following relation:

$$((\otimes_{k \leq n/2} \boldsymbol{r_k}, \otimes_{k > n/2} \boldsymbol{r_k}, com_{\boldsymbol{M}}, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\text{VMV}}.$$

The general strategy to prove $\mathcal{R}_{\text{VMV}}$ is as follows. Suppose commitment to $y = \boldsymbol{L}^T \boldsymbol{M} \boldsymbol{R}$ is computed as $\mathsf{Commit}_{Ped}(\Gamma_{1,fin}; y) = com_y$. $\mathcal{P}$ can compute the vector $\boldsymbol{v} = \boldsymbol{L}^T \boldsymbol{M}$, and by construction $y = \boldsymbol{L}^T \boldsymbol{M} \boldsymbol{R} = \langle \boldsymbol{v}, \boldsymbol{R} \rangle$. Since Pedersen commitments are linearly homomorphic, we have $\langle \boldsymbol{L}, \boldsymbol{com_{row}} \rangle = \mathsf{Commit}_{Ped}(\Gamma_1, \boldsymbol{v})$ is a commitment to $\boldsymbol{v}$. So to prove $((\boldsymbol{L}, \boldsymbol{R}, com_{\boldsymbol{M}}, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\text{VMV}}$, it suffices to prove knowledge of $\boldsymbol{com_{row}} \in \mathbb{G}_1^n, \boldsymbol{v} \in \mathbb{F}^n$ s.t. $com_{\boldsymbol{M}} = \langle \boldsymbol{com_{row}}, \Gamma_2 \rangle$, $\langle \boldsymbol{L}, \boldsymbol{com_{row}} \rangle = \langle \boldsymbol{v}, \Gamma_1 \rangle$ and $com_y = \langle \boldsymbol{v}, \boldsymbol{R} \rangle \Gamma_{1,fin}$. This is further proved by two direct consistency checks and an Inner-Product Argument (IPA). We formally define the inner-product relation below.

Definition 17 (Inner-Product Relation [17])**.** *Let $\boldsymbol{s_1}, \boldsymbol{s_2} \in \mathbb{F}^n$ be public vectors. The relation $\mathcal{R}_{Inner}$ is the set of all tuples $((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2}))$, where $\boldsymbol{v_1} \in \mathbb{G}_1^n$, $\boldsymbol{v_2} \in \mathbb{G}_2^n$ are witness vectors, and $D_1 = \langle \boldsymbol{v_1}, \Gamma_2 \rangle$, $D_2 = \langle \Gamma_1, \boldsymbol{v_2} \rangle$, $E_1 = \langle \boldsymbol{v_1}, \boldsymbol{s_2} \rangle$, $E_2 = \langle \boldsymbol{s_1}, \boldsymbol{v_2} \rangle$, $C = \langle \boldsymbol{v_1}, \boldsymbol{v_2} \rangle$.*

In Dory, we have $\boldsymbol{v_1} = \boldsymbol{com_{row}}, \boldsymbol{v_2} = \boldsymbol{v} \Gamma_{2,fin}$, and two public vectors $\boldsymbol{s_1} = \boldsymbol{R}, \boldsymbol{s_2} = \boldsymbol{L}$. Here we present the intuition behind Dory's IPA reduction. To reduce the size of inner-product $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ by half, $\mathcal{P}$ first sends the cross terms $\langle \boldsymbol{u_L}, \boldsymbol{v_R} \rangle$ and $\langle \boldsymbol{u_R}, \boldsymbol{v_L} \rangle$ to $\mathcal{V}$, and then retrieves a random challenge a from $\mathcal{V}$. Given the identity

$$\begin{aligned} \langle \boldsymbol{u_L} || \boldsymbol{u_R}, \boldsymbol{v_L} || \boldsymbol{v_R} \rangle =& \langle a \boldsymbol{u_L} + \boldsymbol{u_R}, a^{-1} \boldsymbol{v_L} + \boldsymbol{v_R} \rangle \\ & - a \langle \boldsymbol{u_L}, \boldsymbol{v_R} \rangle - a^{-1} \langle \boldsymbol{u_R}, \boldsymbol{v_L} \rangle, \end{aligned} \quad (5)$$

a length-n inner-product claim can be reduced to a length-$n/2$ one by eliminating the cross terms. After $\log n$ iterations, the length will be reduced to 1. $\mathcal{V}$ must compute $\langle \boldsymbol{s_1}, \otimes_{k=0}^{n-1} (\alpha_k, 1) \rangle, \langle \boldsymbol{s_2}, \otimes_{k=0}^{n-1} (\alpha_k^{-1}, 1) \rangle$ to verify the final identity. For polynomial evaluation proof, $\boldsymbol{s_1}, \boldsymbol{s_2}$ are tensor products of n vectors of length 2 respectively. Thus we have $\langle \otimes_{k=0}^{n-1} (l_k, r_k), \otimes_{k=0}^{n-1} (a_k, 1) \rangle = \prod_{k=0}^{N-1} (l_k a_k + r_k)$, which can be computed efficiently with $O(\log n)$ operations by $\mathcal{V}$.

4.2.2. Construction of deDory. Now we adapt Dory to the distributed setting. Unlike deMKZG, achieving logarithmic communication cost for deDory is not trivial.

A Naïve Attempt. At first thought, one may think Dory is naturally suitable for distribution: the polynomial is represented as a matrix, and the partial witnesses held by the sub-provers constitute the columns of the matrix. The commitment can thus be constructed by the sub-provers computing on their respective columns and aggregating their partial results. While the original Dory commits to $\boldsymbol{M}$ as

rows, it is trivial to commit to the rows of $\boldsymbol{M}^T$ instead, such that the commitment is over the columns of $\boldsymbol{M}$.

Unfortunately, such a design would incur a communication cost of $O(2^{n/2})$ for generating an evaluation proof. To be more specific, note that the evaluation proof now needs to demonstrate that $y = \boldsymbol{R}^T\boldsymbol{M}^T\boldsymbol{L}$. The sub-provers then need to compute $\boldsymbol{v} = \boldsymbol{R}^T\boldsymbol{M}^T$. Here each sub-prover holds part of the rows of $\boldsymbol{M}^T$, and can compute a partial result $\boldsymbol{t}^{(i)}$. However, this result is of size $2^{n/2}$, and must be sent to the master sub-prover for summation: $\boldsymbol{v} = \sum_i \boldsymbol{t}^{(i)}$. The master prover also has to re-distribute $\boldsymbol{v}$, so all sub-provers hold the correct $v^{(i)}$ for the computation of $C^{(i)}$ and subsequent use in the IPA protocol. This entails $O(2^{n/2})$ communication cost between each $\mathcal{P}_i$ and $\mathcal{P}_0$.

Achieving Logarithmic Communication Cost. To avoid the problem, we need to re-organize the matrix to make it distribution-friendly. Since each sub-prover $\mathcal{P}_i$ holds the witnesses defining the sub-polynomial $f^{(i)}(\boldsymbol{x}) \in \mathcal{F}_{n-m}^{(\leq 1)}$, we can represent them in matrix representations as well, denoted by $\boldsymbol{M}^{(i)}$ with size $2^{(n-m)/2} \times 2^{(n-m)/2}$ (w.l.o.g. we assume $n-m$ is even). To enable the sub-provers to compute their partial results of $C = \langle \boldsymbol{v}, \boldsymbol{com_{row}} \rangle$ locally and independently from each other, we need to carefully combine their sub-matrices into a large matrix in a block-diagonal form. Specifically, we define a new matrix $\hat{\boldsymbol{M}}$:

$$\hat{\boldsymbol{M}} = \begin{bmatrix} \boldsymbol{M}^{(0)} & & \\ & \ddots & \\ & & \boldsymbol{M}^{(2^m-1)} \end{bmatrix},$$

where each sub-matrix $\boldsymbol{M}^{(i)}$ is placed diagonally along the new matrix, with all other entries set to zero. This arrangement enables each sub-prover to calculate their partial results locally; moreover, the computation results are the correct sub-vectors for IPA reduction. Therefore, it eliminates the $O(2^{n/2})$ communication cost in the naïve approach. Below we give a detailed explanation of deDory, and formally describe the deDory PCS in Protocol 4.1.

deDory Commitment. Since each sub-matrix $\boldsymbol{M}^{(i)}$ has size $2^{(n-m)/2} \times 2^{(n-m)/2}$, the new matrix $\hat{\boldsymbol{M}}$ is of size $2^{(n+m)/2} \times 2^{(n+m)/2}$. With the new matrix representation $\hat{\boldsymbol{M}}$, the sub-provers can commit to $\hat{\boldsymbol{M}}$ by first committing its local sub-matrix $\boldsymbol{M}^{(i)}$ and then aggregating the results together by the master prover.

deDory Evaluation Proof. As we re-arranged the structure of the matrix, we need to re-arrange the public vectors $\hat{\boldsymbol{L}}$ and $\hat{\boldsymbol{R}}$ accordingly, such that $\hat{\boldsymbol{L}}^T\hat{\boldsymbol{M}}\hat{\boldsymbol{R}} = f(\boldsymbol{r})$. To see how to construct the two vectors, first recall that each sub-prover $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x})$ satisfying

$$f(\boldsymbol{r}) = \sum_{i \in [2^m]} \widetilde{eq}(\boldsymbol{r}[n-m:n], \boldsymbol{bin(i)}) f^{(i)}(\boldsymbol{r}[0:n-m]).$$

We construct three vectors using the three parts of the evaluation vector $\boldsymbol{r}$, namely: $\boldsymbol{L} = \otimes_{r_k \in \boldsymbol{r}^{(0)}}(r_k, 1-r_k)$, $\boldsymbol{R} = \otimes_{r_k \boldsymbol{r}^{(1)}}(r_k, 1-r_k)$, $\boldsymbol{E} = \otimes_{r_k \in \boldsymbol{r}^{(2)}}(r_k, 1-r_k)$, where $\boldsymbol{r}^{(0)} = \boldsymbol{r}[0:(n-m)/2]$, $\boldsymbol{r}^{(1)} = \boldsymbol{r}[(n-m)/2 : n-m]$, $\boldsymbol{r}^{(2)} = \boldsymbol{r}[n-m:n]$.

Recall that $\boldsymbol{M}^{(i)}$ is represented as in Definition 15 for $f^{(i)}(\boldsymbol{x})$, and then we have $f^{(i)}(\boldsymbol{r}[0:n-m]) = \boldsymbol{L}^T\boldsymbol{M}^{(i)}\boldsymbol{R}$. while $\widetilde{eq}(\boldsymbol{r}^{(2)}, \boldsymbol{bin(i)})$ is equal to $\boldsymbol{E}_i$. Thus to make the vector-matrix-vector relation satisfied, we could additionally multiply $\boldsymbol{L}$ by the i-th element of the vector $\boldsymbol{E}$. We formalize the construction of $\hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$ in the following theorem.

Theorem 5. *Let* $\hat{\boldsymbol{L}} = \left[\boldsymbol{E}_0 \cdot \boldsymbol{L}, \cdots, \boldsymbol{E}_{2^m-1} \cdot \boldsymbol{L}\right] = \boldsymbol{E} \otimes \boldsymbol{L}$, $\hat{\boldsymbol{R}} = \left[\boldsymbol{R}, \cdots, \boldsymbol{R}\right] = \otimes_{k \in [2^m]}(1,1) \otimes \boldsymbol{R}$, *then* $\hat{\boldsymbol{L}}^T\hat{\boldsymbol{M}}\hat{\boldsymbol{R}} = f(\boldsymbol{r})$.

Proof. By construction, $\hat{\boldsymbol{L}}^T\hat{\boldsymbol{M}}\hat{\boldsymbol{R}} = \sum_{i=0}^{2^m-1} \boldsymbol{E}_i \cdot \boldsymbol{L}^T\boldsymbol{M}^{(i)}\boldsymbol{R}$. Since $\boldsymbol{E}_i = \widetilde{eq}(\boldsymbol{r}^2, \boldsymbol{bin(i)})$, $\boldsymbol{L}^T\boldsymbol{M}^{(i)}\boldsymbol{R} = f^{(i)}(\boldsymbol{r}^{(0)}, \boldsymbol{r}^{(1)})$, then $\hat{\boldsymbol{L}}^T\hat{\boldsymbol{M}}\hat{\boldsymbol{R}} = \sum_{i \in [2^m]} \widetilde{eq}(\boldsymbol{r}^2, \boldsymbol{bin(i)}) \cdot f^{(i)}(\boldsymbol{r}^{(0)}, \boldsymbol{r}^{(1)}) = f(\boldsymbol{r})$. □

Let $\hat{\boldsymbol{L}}^{(i)} = \boldsymbol{E}_i \cdot \boldsymbol{L}, \hat{\boldsymbol{R}}^{(i)} = \boldsymbol{R}$ be the sub-vectors of $\hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$ held by sub-prover $\mathcal{P}_i$. At this point, each $\mathcal{P}_i$ is able to locally run the IPA reduction process with public vectors $\hat{\boldsymbol{L}}^{(i)}, \hat{\boldsymbol{R}}^{(i)}$ and witness $\boldsymbol{M^{(i)}}$ in $(n-m)/2$ rounds. After that, $\mathcal{P}_0$ aggregates all the reduced results from the sub-provers, and performs the last m rounds of IPA reduction and the final verification with $\mathcal{V}$ as in the regular Dory.

Though this reorganization facilitates distribution, the resulting product $\hat{\boldsymbol{R}}\hat{\boldsymbol{L}}^T$ no longer spans $\mathbb{F}^{(n+m)/2 \times (n+m)/2}$ over choices of $\boldsymbol{r}$, which is required for the knowledge extractor to work. This issue could be resolved by running another vector-matrix-vector proof with the same matrix $\hat{\boldsymbol{M}}$ and setting $\tilde{\boldsymbol{L}} = (1, u, \ldots, u^{(n+m)/2-1})$ and $\tilde{\boldsymbol{R}} = (1, u^{(n+m)/2}, \ldots, u^{((n+m)/2-1)((n+m)/2)})$ where u is the random challenge. The sub-vectors $\tilde{\boldsymbol{L}}^{(i)}$ and $\tilde{\boldsymbol{R}}^{(i)}$ can be computed by $\mathcal{P}_i$ using u locally. Notably, in HyperPianist, only a single polynomial needs to be opened once using the batching techniques. Additionally, these two vector-matrix-vector proofs could be further batched into one using the techniques from Dory [17].

We present the full distributed IPA protocol in Protocol 4.2 (and the distributed IPA reduction protocol deDory-Reduce in Protocol D.1, the non-distributed IPA reduction protocol Dory-Reduce protocol in Protocol D.2). Based on this, the distributed vector-matrix-vector proof naturally follows by adding several consistency checks, which can be directly performed by $\mathcal{P}_0$ and $\mathcal{V}$. Our approach reduces the communication cost from $O(2^{n/2})$ to $O(n)$, with only a modest increase in proof size from $O(n)$ to $O(n+m)$ as the matrix size expands from $2^{n/2}$ to $2^{(n+m)/2}$.

Theorem 6. *Given a polynomial* $f \in \mathcal{F}_n^{(\leq 1)}$ *and* 2^m *sub-provers, Protocol 4.1 is a PCS satisfying completeness and knowledge soundness for random evaluation. Each* $\mathcal{P}_i$ *needs* $O(2^{n-m})$ *group operations, while* $\mathcal{P}_0$ *needs* $O(2^m)$ *additional group operations. The communication cost of each* $\mathcal{P}_i$ *is* $O(n)$. *The commitment size is* 1 $\mathbb{G}_T$ *element. The proof size is* $O(n+m)$ $\mathbb{G}_T$ *elements. The verification cost is* $O(n+m)$ $\mathbb{G}_T$ *operations plus* $O(1)$ *pairing.*

The proof of Theorem 6 can be found in Appendix G.3.

PROTOCOL 4.1. deDory *Polynomial Commitment Scheme*

Given $f \in \mathcal{F}_n^{(\leq 1)}$, $\mathcal{P}_i$ holds a sub-polynomial $f^{(i)} \in \mathcal{F}_{n-m}^{(\leq 1)}$ s.t. $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $\forall i \in [M]$.

- deDory.Gen(1^λ): Sample $\mathsf{pp} = (\Gamma_1 \in \mathbb{G}_1^N, \Gamma_2 \in \mathbb{G}_2^N)$.
- deDory.Commit($\mathsf{pp}; f$):
  - $\mathcal{P}_i$ obtains the matrix representation of its sub-polynomial $\boldsymbol{M}^{(i)}$, computes $\boldsymbol{com}_{\boldsymbol{row}}^{(i)} := \mathsf{Commit}_{Ped}(\Gamma_1^{(i)}; \boldsymbol{M}^{(i)})$, $com_{\boldsymbol{M}}^{(i)} := \mathsf{Commit}_{AFGHO}(\Gamma_2^{(i)}; \boldsymbol{com}_{\boldsymbol{row}}^{(i)})$, and sends $com_{\boldsymbol{M}}^{(i)}$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ computes and outputs commitment $com_{\boldsymbol{M}} := \sum_{i\in[M]} com_{\boldsymbol{M}}^{(i)}$.
- deDory.Open($\mathsf{pp}; com_{\boldsymbol{M}}, f$): $\mathcal{V}$ retrieves f from all $\mathcal{P}_i$ and checks $com_{\boldsymbol{M}} = \mathsf{deDory.Commit}(pp; f)$.
- deDory.Eval_RE($\mathsf{pp}; f, \boldsymbol{L}, \boldsymbol{R}$):
  - Each $\mathcal{P}_i$ locally computes $\boldsymbol{L}^{(i)}$ and $\boldsymbol{R}^{(i)}$ corresponding to $\boldsymbol{L}$ and $\boldsymbol{R}$.
  - $\mathcal{P}_0$ computes the commitment to the purported value $y := \boldsymbol{L}\boldsymbol{M}\boldsymbol{R}^T$ as com_y, and sends it to $\mathcal{V}$.
  - Each $\mathcal{P}_i$ computes $\boldsymbol{v}^{(i)} := {\boldsymbol{L}^{(i)}}^T \boldsymbol{M}^{(i)}$, $y^{(i)} := \langle \boldsymbol{v}^{(i)}, \boldsymbol{R}^{(i)} \rangle$, $C^{(i)} := e(\langle \boldsymbol{v}^{(i)}, \boldsymbol{com}_{\boldsymbol{row}}^{(i)} \rangle, \Gamma_{2,fin})$, $E_2^{(i)} := y^{(i)}\Gamma_{2,fin}$, $E_1^{(i)} := \langle \boldsymbol{L}^{(i)}, \boldsymbol{com}_{\boldsymbol{row}}^{(i)} \rangle$, $D_2^{(i)} := e(\langle \Gamma_1^{(i)}, \boldsymbol{v}^{(i)} \rangle, \Gamma_{2,fin})$, and sends $C^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ aggregates C, D_2, E_1, E_2 using the corresponding $C^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}$, and sends them to $\mathcal{V}$.
  - $\mathcal{V}$ checks that $E_2 = y\Gamma_{2,fin}$, $com_y = y\Gamma_{1,fin}$, and $e(E_1, \Gamma_{2,fin}) = D_2$.
  - $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run deDory-IPA($\boldsymbol{L}, \boldsymbol{R}, C, com_{\boldsymbol{M}}, D_2, E_1, E_2$).
- deDory.Eval($\mathsf{pp}; f, \boldsymbol{r}$) :
  - $\mathcal{V}$ samples a random challenge $u \leftarrow_\$ \mathbb{F}$ and sends to $\mathcal{P}_0$. Then $\mathcal{P}_0$ transmits u to the other $\mathcal{P}_i$.
  - $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run deDory.Eval_RE($\mathsf{pp}; f, \hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$) and deDory.Eval_RE($\mathsf{pp}; f, \tilde{\boldsymbol{L}}, \tilde{\boldsymbol{R}}$), where $\hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$ is specifies in Theorem 5, and $\tilde{\boldsymbol{L}} = (1, u, \ldots, u^{(n+m)/2-1})$, $\tilde{\boldsymbol{R}} = (1, u^{(n+m)/2}, \ldots, u^{((n+m)/2-1)((n+m)/2)})$.

PROTOCOL 4.2. deDory-IPA($\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2$)

$\mathcal{P}_i$ holds witness $\boldsymbol{v}_1^{(i)}, \boldsymbol{v}_2^{(i)}$ w.r.t. $\boldsymbol{v}_1, \boldsymbol{v}_2$ s.t. $((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2})) \in \mathcal{R}_{\text{Inner}}$.
For all $j \in [n]$, all $\mathcal{P}_i$ compute $\Gamma_{1,j+1} := (\Gamma_{1,j})_L, \Gamma_{2,j+1} := (\Gamma_{2,j})_L, \Delta_{1L,i} := \langle (\Gamma_{1,i})_L, \Gamma_{2,i+1} \rangle, \Delta_{2L,i} := \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_L \rangle$ $\Delta_{1R,i} := \langle (\Gamma_{1,i})_R, \Gamma_{2,i+1} \rangle$, $\Delta_{2R,i} := \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_R \rangle$, and for all $j \in [n+1]$ compute $\chi_j := \langle \Gamma_{1,j}, \Gamma_{2,j} \rangle$.

- For $j \in [n-m]$, all $\mathcal{P}_i$ and $\mathcal{V}$ run $(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2) \leftarrow \mathsf{deDory\text{-}Reduce}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.
- Each $\mathcal{P}_i$ sends $(s_1^{(i)}, s_2^{(i)}, C^{(i)}, D_1^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}, v_1^{(i)}, v_2^{(i)})$ to $\mathcal{P}_0$.
- $\mathcal{P}_0$ aggregates C, D_1, D_2, E_1, E_2 using the corresponding $C^{(i)}, D_1^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}$, and set $\boldsymbol{v_1} := (v_1^{(i)})_{i\in[M]}$, $\boldsymbol{v_2} := (v_2^{(i)})_{i\in[M]}$, $\boldsymbol{s_1} := (s_1^{(i)})_{i\in[M]}$, $\boldsymbol{s_2} := (s_2^{(i)})_{i\in[M]}$.
- For $j = n-m, \ldots, n-1$, $\mathcal{P}_0$ and $\mathcal{V}$ run $(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2) \leftarrow \mathsf{Dory\text{-}Reduce}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.
- $\mathcal{P}_0$ sends v_1, v_2 to $\mathcal{V}$, and $\mathcal{V}$ accepts if $((s_1, s_2, C, D_1, D_2, E_1, E_2); (v_1, v_2)) \in \mathcal{R}_{\text{Inner}}$.

5. HyperPianist+: HyperPianist with Optimized Lookup Arguments

In this section, we present an optimized lookup argument and adapt it to the distributed setting. Our construction is built on Lasso, which we review below.

5.1. Review: Lookup Arguments in Lasso

A lookup relation can be interpreted as a set inclusion relation on committed vectors:

Definition 18 (Lookup Relation)**.** *The indexed relation $\mathcal{R}_{Lookup}$ is the set of tuples $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\boldsymbol{b}; ([[\tilde{a}]], [[\tilde{t}]]); (\tilde{a}, \tilde{t}))$ where $\boldsymbol{b} \in \mathbb{F}^\ell$, $\tilde{a}$ and $\tilde{t}$ are the MLEs of $\boldsymbol{a} \in \mathbb{F}^\ell$, and $\boldsymbol{T} \in \mathbb{F}^N$ respectively, s.t. for all $i \in \{1, \cdots, \ell\}$, $\boldsymbol{a}_i = \boldsymbol{T}[\boldsymbol{b}_i]$.*

Lasso is specialized for structured tables. It makes the observation that the lookup tables for many operations (like bitwise AND) can be broken down into smaller-sized sub-tables, such that for some $\boldsymbol{x} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_c)$ and some simple function g, $\boldsymbol{T}[\boldsymbol{x}] = g(\boldsymbol{T}_1[\boldsymbol{x}_1], \cdots, \boldsymbol{T}_c[\boldsymbol{x}_c])$, where $\boldsymbol{T}_i$ are the sub-tables of the table $\boldsymbol{T}$.

In Lasso, proving a committed vector $\boldsymbol{a}$ is contained in the table $\boldsymbol{T}$ is turned into proving the existence of some sparse matrix $\boldsymbol{M} \in \mathbb{F}^{\ell \times n}$ s.t. there is only one non-zero entry of value 1 in each row and $\boldsymbol{M} \cdot \boldsymbol{t} = \boldsymbol{a}$. This is reduced to a SumCheck claim that

$$\sum_{\boldsymbol{y}\in\{0,1\}^{\log n}} \widetilde{M}(\boldsymbol{r}, \boldsymbol{y}) \cdot \tilde{t}(\boldsymbol{y}) = \tilde{a}(\boldsymbol{r}), \tag{6}$$

where $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$ is a random challenge vector from $\mathcal{V}$, and $\widetilde{M}, \tilde{t}$ and $\tilde{a}$ are MLEs of $\boldsymbol{M}, t$ and a respectively.

As the table T and the matrix $\boldsymbol{M}$ may be huge, directly running the SumCheck protocol may be expensive. Lasso utilizes a key feature of the matrix $\boldsymbol{M}$: it is extremely sparse, i.e., only one entry in each row of $\boldsymbol{M}$ can be non-zero, and the non-zero entry must be 1. Given this, Lasso transforms Equation (6) into $\sum_{\boldsymbol{x}\in\{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}) \cdot \boldsymbol{T}[nz(\boldsymbol{x})] = \tilde{a}(\boldsymbol{r})$, where for each $\boldsymbol{x}$-th row of the matrix $\boldsymbol{M}$, $nz(\boldsymbol{x})$ denotes the column index corresponding to the non-zero entry in

PROTOCOL 4.3. *Lookup PIOP*

$\mathcal{P}$ claims $(\boldsymbol{b}; ([[\tilde{a}]], [[\tilde{t}]]); (\tilde{a}, \tilde{t})) \in \mathcal{R}_{\text{Lookup}}$ to $\mathcal{V}$, where $\tilde{t}$ is the MLE of a decomposable table.

- $\mathcal{P}$ sends oracles of $E_1, \cdots, E_c$ and $nz_1, \cdots, nz_c$ to $\mathcal{V}$.
- $\mathcal{V}$ picks a random $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$ to $\mathcal{P}$, and makes an oracle call to $\tilde{a}$ and obtains $\tilde{a}(\boldsymbol{r})$.
- $\mathcal{P}, \mathcal{V}$ run a SumCheck PIOP to check $\tilde{a}(\boldsymbol{r}) = \sum_{\boldsymbol{k} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{r}, \boldsymbol{k}) g(E_1(\boldsymbol{k}), \cdots, E_c(\boldsymbol{k}))$.
- For $j = 1$ to c, $\mathcal{P}, \mathcal{V}$ check $E_j(\boldsymbol{k})$ is well-formed:
  - $\mathcal{P}$ sends an oracle of $m_j(X)$ (defined in Equation (9)) to $\mathcal{V}$, and retrieves random challenges $\beta, \gamma \leftarrow_{\$} \mathbb{F}$ from $\mathcal{V}$.
  - $\mathcal{P}, \mathcal{V}$ run the Rational SumCheck PIOP to check $\sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \frac{1}{nz_j(\boldsymbol{x}) + \gamma \cdot E_j(\boldsymbol{x}) + \beta} = \sum_{\boldsymbol{x} \in \{0,1\}^{\log N}} \frac{m_j(\boldsymbol{x})}{s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}) + \beta}$.

this row, and $\boldsymbol{T}[nz(\boldsymbol{x})]$ denotes the corresponding $nz(\boldsymbol{x})$-th entry of the table $\boldsymbol{T}$. Since we assume the table is decomposable, we can write the LHS of the equation as

$$\sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}) \cdot g(\boldsymbol{T}_1[nz_1(\boldsymbol{x})], \cdots, \boldsymbol{T}_c[nz_c(\boldsymbol{x})]).$$

Let $E_j(\boldsymbol{x})$ be the MLE of $\boldsymbol{T}_j[nz_j(\boldsymbol{x})]$. Then we have

$$\sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}) \cdot g(E_1(\boldsymbol{x}), \cdots, E_c(\boldsymbol{x})) = \tilde{a}(\boldsymbol{r}). \quad (7)$$

Now $\mathcal{P}$ and $\mathcal{V}$ can engage in a new SumCheck instance on Equation (7). To this end, the prover now needs to provide oracles to new polynomials E_j. Additionally, it needs to show that they are well-formed, i.e., indeed equal to the MLE of $\boldsymbol{T}_j[nz_j(\boldsymbol{x})]$. This check is formulated as follows: given a table $\boldsymbol{T}_j$ and a series of queries into the table denoted by $nz_j(\boldsymbol{x})$, it verifies the purported values $E_j(\boldsymbol{x})$ are retrieved from the table honestly. In Lasso, this is performed using offline memory checking from Spartan [19].

5.2. Reducing Prover Cost of Lasso

We optimize the well-formation check of Lasso with the following observation: if

$$\{(nz(\boldsymbol{k}), E(\boldsymbol{k}))\}_{\boldsymbol{k} \in \{0,1\}^{\log \ell}} \subset \{(\boldsymbol{x}, T[\boldsymbol{x}])\}_{\boldsymbol{x} \in \{0,1\}^{\log N}}, \quad (8)$$

then $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})]$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$. Based on this observation, the well-formation statement can be proved more efficiently with techniques from Logup [15], [37], which rely on the logarithmic derivative technique—a generalized version of Theorem 2.

Theorem 7 ([15]). *Let $\mathbb{F}$ be a field of characteristic $p > \max(\ell, N)$. Suppose $(a_i)_{i=1}^{\ell}$, $(b_j)_{j=1}^{N}$ are sequences of field elements. Then $\{a_i\} \subset \{b_j\}$ as sets if and only if there exists a sequence $(m_j)_{j=1}^{N}$ such that $\sum_{i=1}^{\ell} \frac{1}{a_i + X} = \sum_{j=1}^{N} \frac{m_j}{b_j + X}$.*

To prove the set inclusion relation in Equation (8), we first use a random challenge to combine each tuple into one element. Then applying Theorem 7, we can get

$$\begin{aligned} &\sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \frac{1}{nz_j(\boldsymbol{x}) + \gamma \cdot E_j(\boldsymbol{x}) + \beta} \\ =& \sum_{\boldsymbol{x} \in \{0,1\}^{\log N}} \frac{m(\boldsymbol{x})}{s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}) + \beta}, \end{aligned} \quad (9)$$

where γ, β are random challenges from $\mathcal{V}$, and the polynomial $s_{id} \in \mathcal{F}_n^{(\leq 1)}$ is defined as in Section 3.4 (i.e., mapping each binary vector to the corresponding integer).

Thus, to prove the well-formation of E, it suffices for the prover to commit one additional polynomial $m(\boldsymbol{x})$, and invoke two Rational SumChecks to prove Equation (9). We show the complete construction in Protocol 4.3.

To distribute the Lookup PIOP, note that $m(\boldsymbol{x})$ can be calculated during the circuit evaluation, and thus it can be distributed along with other witness polynomials. The rest follows directly by replacing the PIOP checks with their distributed counterparts, so we omit the detailed construction here due to the page limit.

Complexity Analysis. We provide a rough estimation for both constructions. In Lasso, the prover commits $2c$ polynomials of size ℓ and c polynomials of size $N^{1/c}$, performs a SumCheck for Equation 7, and c instances of memory checking which incurs c polynomial commitments of size $2(N^{1/c}+\ell)$ and $2c$ ZeroCheck invocations of size $(N^{1/c}+\ell)$ via Quarks [14]. In our construction, the prover commits c polynomials of size ℓ and $N^{1/c}$, along with a SumCheck for Equation 7; the well-formation check incurs c polynomial commitments of size ℓ and $N^{1/c}$, c ZeroCheck and c SumCheck invocations of size $\ell + N^{1/c}$. Compared to Lasso, our optimization roughly reduces the commitment workload by 50% and SumCheck workload by 30%.[3]

6. Evaluation

In this section, we conduct comprehensive experiments on HyperPianist(+) to evaluate its performance.

Implementation. We fully implemented[4] HyperPianist and HyperPianist+ based on the Rust ark-works [38] ecosystem. We built the two systems on the open-source HyperPlonk, with a full realization of the two distributed PCS deMKZG and deDory. For each scheme, we implemented two versions, one with our PIOP system (denoted as $\text{HyperPianist}^{\text{K}}_{\text{PIOP}}, \text{HyperPianist}^{\text{D}}_{\text{PIOP}}$) and the other based on layered circuits (denoted as $\text{HyperPianist}^{\text{K}}_{\text{LC}}, \text{HyperPianist}^{\text{D}}_{\text{LC}}$). See also Remark 1. Our lookup argument is implemented as a modified port of Lasso from the Jolt project [39]. We noted that the Gnark library [40] used in Pianist implemented a more efficient

3. Lasso enables the use of *layered circuits* to prove the grand product in the multiset check to improve prover efficiency with $O(\log^2 N)$ proof size. A similar approach can be applied in our construction, with the detailed method presented in Appendix F.

4. Our code and all evaluation results can be found in the repository: https://github.com/AntCPLab/HyperPianist.

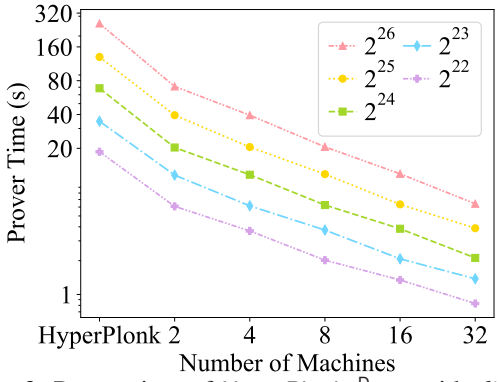

Figure 2: Prover time of HyperPianist$^{D}_{PIOP}$ with *different number of machines* on vanilla circuits.

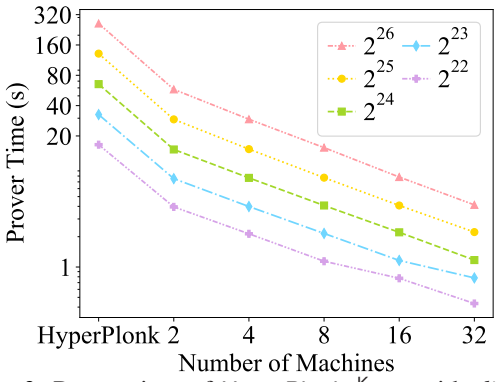

Figure 3: Prover time of HyperPianist$^{K}_{PIOP}$ with *different number of machines* on vanilla circuits.

multi-scalar-multiplication (MSM) algorithm [41] than ark-works, and as such made several optimizations to ark-works to get them on par. The curve used is BN254, the same as Pianist.

Experimental Setup. We design various types of experiments to show the following properties of HyperPianist(+):

1) **Linear scalability:** We measure the proving time of HyperPianist^{D} and HyperPianist^{K} on random *vanilla gate circuits* with different number of machines from 2 to 32, and show they have linear scalability. We also evaluate HyperPlonk on a single machine as a baseline.
2) **Linear prover workload:** We compare HyperPianist^{D} and HyperPianist^{K} with Pianist on both vanilla gates and *degree-5 custom gates* from Jellyfish Plonk [21] of size $2^{22} \sim 2^{26}$, showing they enjoy better efficiency and only a linear prover time growth in circuit size.
3) **Efficient lookup support:** As there is no implementation of lookup arguments in Pianist, we directly compare our optimized lookup argument with Lasso in the non-distributed setting using 32-bit XOR gates of size $2^{20} \sim 2^{24}$, and show that it achieves better efficiency.

Since the implementation of Pianist does not support custom gates, we implemented this feature in their code. Also, as Lasso's implementation uses layered circuits by default, we implemented a PIOP version of their protocol. We run the experiments on Alibaba ecs.r8i.8xlarge cloud servers with 32 vCPUs and 256 GB memory in a local area network. All reported results are taken as the average of multiple runs. We additionally present some evaluation results running in a (simulated) wide area network in Appendix H.

6.1. Linear Scalability

We show the overall proving time of HyperPianist^{D} and HyperPianist^{K} with different number of machines in Figure 2 and 3 (axes in logarithmic scale). In each figure HyperPlonk adopts the non-distributed version of the PCS used in HyperPianist. Both figures demonstrate a linear decrease in proving time as the number of machines increases. Specifically, for a circuit of size 2^{26}, HyperPlonk takes 261.2 s to generate a proof on a single machine. With 2 machines using HyperPianist$^{K}_{PIOP}$, it takes 58.0 s, achieving a $4.5\times$ speedup over HyperPlonk. Using 32 machines, the proving time further drops to 4.1 s, for a $63.1\times$ speedup. HyperPianist$^{D}_{PIOP}$ also shows a speedup of $3.6\times$ with 2 machines and $40.2\times$ with 32 machines on this circuit. The layered circuit versions also show linear scalability with even higher efficiency: HyperPianist$^{K}_{LC}$ achieves $5.7\times$ and $73.3\times$ speedup with 2 and 32 machines respectively, and HyperPianist$^{D}_{LC}$ achieves $4.6\times$ and $48.0\times$.

Note that our systems achieve greater efficiency gains over HyperPlonk than theoretically expected. One reason is that both our PIOP and layered circuit versions require fewer commitments than HyperPlonk, with the PIOP version requiring only one extra commitment in addition to the witnesses, and the layered circuit version requiring none. Another important factor is implementation optimizations. To ensure a fair comparison, we have already incorporated some optimizations into HyperPlonk (e.g. MSM optimizations), but certain differences remain. This does not undermine the comparison, as our focus is on linear scalability relative to HyperPlonk rather than absolute efficiency.

6.2. Comparisons with Pianist

Prover Time. Figure 4 and 5 show the overall proving time of our schemes versus Pianist, each with 8 machines, on vanilla gates and custom gates respectively. HyperPianist$^{K}_{PIOP}$ achieves speedups of up to $2.9\times$ for vanilla gates and $4.3\times$ for custom gates, while HyperPianist$^{D}_{PIOP}$ achieves up to $2.1\times$ and $3.3\times$ speedups on the two types of gates. Again, the layered circuit versions perform even better, with HyperPianist$^{K}_{LC}$ attaining up to $3.3\times$ and $5.5\times$ speedups respectively on vanilla and custom gates, and HyperPianist$^{D}_{LC}$ showing $2.6\times$ and $4.1\times$.

Our speedups grow as the size of sub-circuits each sub-prover holds increases. This is likely due to the quasi-linear prover cost of Pianist. For each sub-prover holding a sub-circuit of size 2^{24} or 2^{25}, the speedup of HyperPianist$^{K}_{LC}$ on vanilla gates and custom gates can be $3.7\times$ and $5.9\times$, and HyperPianist$^{D}_{LC}$ can be $3.0\times$ and $4.7\times$ respectively. The PIOP version HyperPianist$^{K}_{PIOP}$ comes in at $2.9\times$ and $4.6\times$, while HyperPianist$^{D}_{PIOP}$ is $2.4\times$ and $3.8\times$.

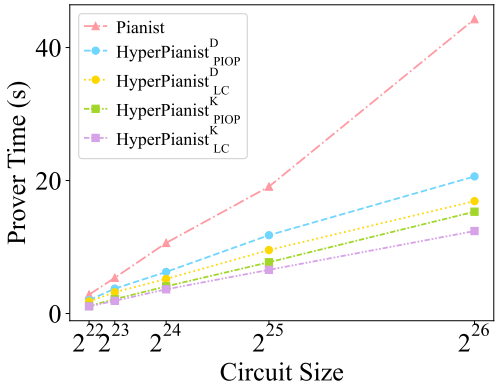

Figure 4: Comparisons of Pianist [9] and our systems on *vanilla gates* with 8 machines.

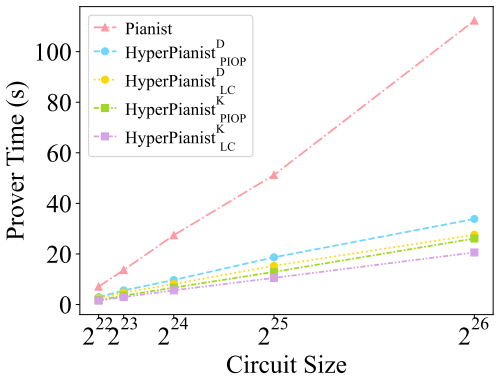

Figure 5: Comparisons of Pianist [9] and our systems on *custom gates* with 8 machines.

TABLE 4: Communication, proof size, and verifier time of our systems with 8 machines.

Scheme	HyperPianist$^{K}_{PIOP}$			HyperPianist$^{D}_{PIOP}$			HyperPianist$^{K}_{LC}$			HyperPianist$^{D}_{LC}$		
Circuit Size	2^{22}	2^{24}	2^{26}	2^{22}	2^{24}	2^{26}	2^{22}	2^{24}	2^{26}	2^{22}	2^{24}	2^{26}
$\mathcal{P}_i$ Commu. (KB)	22.2	24.3	26.3	53.7	58.7	63.7	51.7	59.9	68.6	82.9	94.0	105.6
Proof Size (KB)	10.7	11.6	12.5	45.2	48.6	52.0	40.4	46.4	52.9	74.6	83.1	92.0
$\mathcal{V}$ Time (ms)	3.7	3.7	3.9	12.0	12.8	13.8	5.1	5.2	5.7	13.4	14.8	15.6

TABLE 5: Comparisons of our lookup argument and Lasso on XOR gates (in non-distributed setting).

Scheme	Ours			Lasso [18]		
Circuit Size	2^{20}	2^{22}	2^{24}	2^{20}	2^{22}	2^{24}
$\mathcal{P}$ Time (s)	2.1	8.0	30.2	4.2	15.2	57.9
$\mathcal{V}$ Time (ms)	0.6	0.6	0.7	0.2	0.2	0.3
Proof Size (KB)	8.2	8.7	9.2	6.9	7.2	7.5

Our efficiency gain over Pianist mainly comes from: (1) fewer MSMs, and (2) avoiding FFTs. Our experiments show that the percentage of prover time taken by non-PCS parts (primarily FFTs) in Pianist grows from 50.9% to 55.4% as the circuit size grows from 2^{21} to 2^{25}. This suggests that the efficiency gains from avoiding FFTs will further increase for larger circuits. For custom gates, both advantages become more pronounced as the degree is higher.

Communication, Proof size, and Verifier Time. Tables 4 shows the communication cost between each $\mathcal{P}_i$ and $\mathcal{P}_0$, the proof size and the verifier time of our schemes running with 8 machines on vanilla gates. In Pianist, the communication cost is 2.5 KB, the proof size is 1.7 KB, and the verifier time is 3 ms for all instances. Though our schemes have (poly-)logarithmic costs in this regard, in reality, they are still reasonable and highly practical. In this regard, the PIOP schemes have much lower costs than the layered circuit schemes, since layered circuits require $O(\log^2 n)$ costs while the PIOP versions are at $O(\log n)$. Across all the test instances, HyperPianist$^{K}_{PIOP}$ incurs less than 30 KB communication per sub-prover, with a proof size of at most 12.5 KB; even HyperPianist$^{D}_{LC}$ requires at most 120 KB communication, with a proof size of less than 100 KB, and the verifier time is within 20 ms.

$\mathcal{P}_0$ **Extra Time**. For reference, we also measure the extra time on $\mathcal{P}_0$ for proof aggregation and the remaining extra computation (this part is included in the prover time reported above). For HyperPianist^{D}, this step takes less than 30 ms on vanilla gates with 8 machines. For HyperPianist^{K}, the extra work only requires less than 16 ms.

6.3. Evaluations of the Lookup Argument

Table 5 shows the evaluation results of our lookup argument and Lasso on XOR statements. Our prover can achieve up to a $2\times$ speedup over Lasso's, with comparable proof size and modestly increased verifier time. If we switch to layered circuits, both Lasso and our lookup argument show higher prover efficiency, but the verifier time and proof size notably increased. In this case, our lookup argument still demonstrates an improvement of $1.4\times$ in proving time, and $1.2\times$ in verifier time, with comparable proof size.

6.4. Comparisons with Other Related Works

Since neither HEKATON nor deVirgo was open source at the time of our experiments, we conducted comparisons based on estimations.[5]

HEKATON. As we discussed in Section 1, HEKATON offers a roughly $2\times$ speedup over Pianist in prover efficiency; we believe that our schemes should be comparable or superior to HEKATON, especially for custom gates where HEKATON has no additional advantages. Besides, our schemes achieve significantly lower communication costs with similar proof sizes. For a circuit of size 2^{38} partitioned into sub-circuits of size 2^{20}, Hekaton reports nearly 1MB communication per sub-circuit. In contrast, we estimate

5. HEKATON was open sourced after our experiments were completed; therefore, our evaluations rely on the information available prior to its release.

that HyperPianist$^{K}_{PIOP}$ and HyperPianist$^{D}_{PIOP}$ require 23.2 KB and 54.7 KB, respectively, while HyperPianist$^{K}_{LC}$ and HyperPianist$^{D}_{LC}$ require around 55.3 KB and 86.5 KB. The proof size of Hekaton is 32KB, whereas HyperPianist$^{K}_{PIOP}$ and HyperPianist$^{D}_{PIOP}$ have a proof size of 17.9 KB and 72.4 KB, respectively, while HyperPianist$^{K}_{LC}$ and HyperPianist$^{D}_{LC}$ require 100 KB and 155 KB.

deVirgo. The FRI-based distributed PCS in deVirgo offers better concrete prover efficiency. However, the high communication overhead may hinder applications in network-constrained scenarios. For example, for each sub-prover generating a proof of an EdDSA signature (comprising 2^{21} gates), the communication cost is nearly 1GB. In contrast, we estimate that HyperPianist$^{K}_{PIOP}$ and HyperPianist$^{D}_{PIOP}$ require 24.2 KB and 58.7 KB, respectively, while HyperPianist$^{K}_{LC}$ and HyperPianist$^{D}_{LC}$ require around 59.5 KB and 93.7 KB. Besides, deVirgo's proof size remains around 1.9MB across the tested instances; in contrast, for the largest instance tested in deVirgo ($\approx 2^{28}$ gates), HyperPianist$^{K}_{PIOP}$ and HyperPianist$^{D}_{PIOP}$ have a proof size of 13.4 KB and 55.4 KB, respectively, while HyperPianist$^{K}_{LC}$ and HyperPianist$^{D}_{LC}$ require 59.9 KB and 101.5 KB.

References

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, 1985, pp. 291–304.

[2] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[3] 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.

[4] L. Pearson, J. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia, "PlonKup: Reconciling PlonK with plookup," *Cryptology ePrint Archive*, 2022.

[5] A. Gabizon and Z. J. Williamson, "plookup: A simplified polynomial protocol for lookup tables," *Cryptology ePrint Archive*, 2020.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 2010, pp. 177–194.

[11] M. Rosenberg, T. Mopuri, H. Hafezi, I. Miers, and P. Mishra, "HEKATON: Horizontally-scalable zkSNARKs via proof aggregation," *Cryptology ePrint Archive*, 2024.

[12] 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.

[13] 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.

[14] S. Setty and J. Lee, "Quarks: Quadruple-efficient transparent zkSNARKs," *Cryptology ePrint Archive*, 2020.

[15] U. Haböck, "Multivariate lookups based on logarithmic derivatives," *Cryptology ePrint Archive*, 2022.

[16] C. Papamanthou, E. Shi, and R. Tamassia, "Signatures of correct computation," in *Theory of Cryptography Conference*. Springer, 2013, pp. 222–242.

[17] J. Lee, "Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments," in *Theory of Cryptography Conference*. Springer, 2021, pp. 1–34.

[18] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with Lasso," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2024, pp. 180–209.

[19] S. Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.

[20] D. Dore, "TaSSLE: Lasso for the commitment-phobic," Cryptology ePrint Archive, Paper 2024/1075, 2024, https://eprint.iacr.org/2024/1075. [Online]. Available: https://eprint.iacr.org/2024/1075

[21] E. Systems, "A Rust implementation of the PLONK ZKP system and extensions." [Online]. Available: https://github.com/EspressoSystems/jellyfish

[22] A. Ozdemir and D. Boneh, "Experimenting with collaborative zkSNARKs: Zero-knowledge proofs for distributed secrets," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4291–4308.

[23] 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.

[24] P. Dayama, A. Patra, P. Paul, N. Singh, and D. Vinayagamurthy, "How to prove any NP statement jointly? efficient distributed-prover zero-knowledge protocols," *Proceedings on Privacy Enhancing Technologies*, 2022.

[25] X. Liu, Z. Zhou, Y. Wang, J. He, B. Zhang, X. Yang, and J. Zhang, "Scalable collaborative zk-SNARK and its application to efficient proof outsourcing," *Cryptology ePrint Archive*, 2024.

[26] Y. Hu, P. Mishra, X. Wang, J. Xie, K. Yang, Y. Yu, and Y. Zhang, "DFS: Delegation-friendly zkSNARK and private delegation of provers," *Cryptology ePrint Archive*, 2025.

[27] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, "Caulk: Lookup arguments in sublinear time," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3121–3134.

[28] J. Posen and A. A. Kattis, "Caulk+: Table-independent lookup arguments," *Cryptology ePrint Archive*, 2022.

[29] A. Zapico, A. Gabizon, D. Khovratovich, M. Maller, and C. Rafols, "Baloo: nearly optimal lookup arguments," *Cryptology ePrint Archive*, 2022.

[30] A. Gabizon and D. Khovratovich, "flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size," *Cryptology ePrint Archive*, 2022.

[31] L. Eagen, D. Fiore, and A. Gabizon, "cq: Cached quotients for fast lookups," *Cryptology ePrint Archive*, 2022.

[32] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.

[33] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent SNARKs from DARK compilers," 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. 677–706.

[34] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptology conference*. Springer, 1991, pp. 129–140.

[35] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo, "Structure-preserving signatures and commitments to group elements," in *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer, 2010, pp. 209–236.

[36] J. Groth, "Efficient zero-knowledge arguments from two-tiered homomorphic commitments," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 431–448.

[37] S. Papini and U. Haböck, "Improving logarithmic derivative lookups using GKR," *Cryptology ePrint Archive*, 2023.

[38] arkworks contributors, "`arkworks` zkSNARK ecosystem," 2022. [Online]. Available: https://arkworks.rs

[39] 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.

[40] Consensys, "zk-SNARK library." [Online]. Available: https://github.com/Consensys/gnark

[41] Y. El Housni and G. Botrel, "EdMSM: multi-scalar-multiplication for SNARKs and faster montgomery multiplication," *Cryptology ePrint Archive*, 2022.

[42] J. Thaler, "Time-optimal interactive proofs for circuit evaluation," in *Annual Cryptology Conference*. Springer, 2013, pp. 71–89.

Appendix A. Distributed SumCheck PIOP for High-Degree Polynomials

In HyperPlonk [13], the authors proposed an algorithm for high-degree polynomials with special structures. Consider a multivariate polynomial

$$f(\boldsymbol{X}) = h(g_0(\boldsymbol{X}), \ldots, g_{c-1}(\boldsymbol{X})) \tag{10}$$

such that $h \in \mathcal{F}_c^{(\leq d)}$ can be evaluated through an arithmetic circuit with $O(d)$ gates and $g_i \in \mathcal{F}_n^{(\leq 1)}, \forall i \in [c]$. The core idea is to compute $r_i(X)$ symbolically.

Complexity. Let 2^m be the number of sub-provers, $f \in \mathcal{F}_n^{(\leq d)}$ be the polynomial defined as Equation (10). The complexity of Protocol A.1 is as follows:

Algorithm 1 Evaluating $f_k(X)$ for each round [13]

Input: The evaluation tables for current $g_0, \ldots, g_{c-1}$, current table length ℓ.
Output: The corresponding $f_k(X)$.
1: Construct $g_{j,\boldsymbol{x}}(X) := g_j(X, \boldsymbol{x}), \forall \boldsymbol{x} \in \{0,1\}^\ell, \forall j \in [c]$.
2: Compute $f_{\boldsymbol{x}} := h(g_{0,\boldsymbol{x}}(X), \ldots, g_{c-1,\boldsymbol{x}}(X)), \forall \boldsymbol{x}\{0,1\}^\ell$ using Algorithm 2.
3: Compute $f_k(X) = \sum_{\boldsymbol{x} \in \{0,1\}^\ell} f_{\boldsymbol{x}}$.
4: **return** $f_k(X)$.

Algorithm 2 Evaluating $f_{\boldsymbol{x}}(X) = \prod_{j=1}^{d-1} g_{j,\boldsymbol{x}}(X)$ [13]

Input: $g_0, \ldots, g_{d-1}$ are linear univariate polynomials.
Output: The corresponding $f_k(X)$.
1: $t_{1,j} \leftarrow g_{j,\boldsymbol{x}}$ for all $j \in [d]$
2: **for** $i = 0$ to $\log d$ **do**
3: **for** $j = 0$ to $d/2^i - 1$ **do**
4: $t_{i+1,j}(X) \leftarrow t_{i,2j-1}(X) \cdot t_{i,2j}(X)$ ▷ Using FFT
5: **end for**
6: **end for**
7: **return** $f_{\boldsymbol{x}}(X) = t_{\log d,1}(X)$.

- The running time of each sub-prover is $O(d \log^2 d \cdot 2^{n-m})\,\mathbb{F}$ operations.
- The extra proving time for master prover is $O(d(n-m) \cdot 2^m)\,\mathbb{F}$ operations.
- The running time of the verifier is $O(d \cdot n)\,\mathbb{F}$ operations.
- The proof size is $O(d \cdot n)\,\mathbb{F}$ elements, plus an oracle corresponding to the polynomial f.
- The communication complexity for each sub-prover is $O(d \cdot n)\,\mathbb{F}$ elements.

Appendix B. Constraint System of HyperPianist

The constraint system of HyperPianist is derived from the original Plonk [2]. Here we present a basic construction with vanilla Plonk gates.

The HyperPianist constraint system is defined over a fan-in-two arithmetic circuit. The left input, the right input, and the output of each gate are encoded by multilinear polynomials $L(\boldsymbol{X})$, $R(\boldsymbol{X})$, $O(\boldsymbol{X})$ respectively. The verifier needs to check the correct computation of each gate (gate identity), and also the correct connections between the inputs and outputs of the gates as specified by the structure of the circuit (wiring identity). We assume each sub-prover holds partial witnesses to the circuit.

Gate Identity. The gate equation for a single gate is

$$\begin{aligned} 0 = {} & q_L(\boldsymbol{x}) \cdot L(\boldsymbol{x}) + q_R(\boldsymbol{x}) \cdot R(\boldsymbol{x}) \\ & + q_M(\boldsymbol{x}) \cdot L(\boldsymbol{x}) \cdot R(\boldsymbol{x}) - O(\boldsymbol{x}) \end{aligned}$$

where q_L, q_R, q_M are three selector polynomials. For an addition gate, we set $q_L(\boldsymbol{x}) = q_R(\boldsymbol{x}) = 1$, $q_M(\boldsymbol{x}) = 0$. For a multiplication gate, we set $q_L(\boldsymbol{x}) = q_R(\boldsymbol{x}) = 0$, $q_M(\boldsymbol{x}) = 1$. To prove that all gates were evaluated correctly, the sub-provers convince the verifier that the identity holds for any $\boldsymbol{x} \in \{0,1\}^n$ with the distributed ZeroCheck PIOP.

PROTOCOL A.1. *Distributed SumCheck PIOP (for High-degree Polynomials)*

$\mathcal{P}_0$ claims $((v, [[f]]); f) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$, where $f(\boldsymbol{X}) = h(g_0(\boldsymbol{X}), \ldots, g_{c-1}(\boldsymbol{X}))$. $\mathcal{P}_i$ holds $g_j^{(i)} \in \mathcal{F}_{n-m}^{(\leq 1)}$, s.t. $g_j^{(i)}(\boldsymbol{x}) = g_j(\boldsymbol{x}, \boldsymbol{bin(i)}), \forall j \in [c], i \in [M]$. Let $f_0 := v$.

- In the k-th round where $1 \leq k \leq n-m$:
  - Each $\mathcal{P}_i$ computes $f_k^{(i)}(X)$ using Algorithm 1 with tables for $g_j^{(i)}$, and table length $n-m-k$, and sends it to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_k(X) = \sum_{i \in [M]} f_k^{(k)}(X)$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{k-1}(r_{k-1}) = f_k(0) + f_k(1)$, sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}_0$, and sets $f_k = f_k(r_k)$.
  - $\mathcal{P}_0$ transmits r_k to the other $\mathcal{P}_i$. Each $\mathcal{P}_i$ updates the table corresponding to each $g_j^{(i)}$.
- After the $(n-m)$-th round, each $\mathcal{P}_i$ sends $g_j^{(i)}(\boldsymbol{r}), \forall j \in [c]$ to $\mathcal{P}_0$. $\mathcal{P}_0$ then constructs current table for $g_j, \forall j \in [c]$.
- In the k-th round where $n-m+1 \leq k \leq n$:
  - $\mathcal{P}_0$ computes $f_k(X)$ using Algorithm 1 with tables for g_j, and table length $n-m-k$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{k-1} = f_k(0) + f_k(1)$, sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}_0$, and sets $f_k := f_k(r_k)$.
  - $\mathcal{P}_0$ updates the table corresponding to each g_j.
- Finally, the verifier checks $f_n(\boldsymbol{r}) = h(g_0(\boldsymbol{r}), \ldots, g_{c-1}(\boldsymbol{r}))$ using oracle calls to $g_0, \ldots, g_{c-1}$.

Wiring Identity. To prove the gates are connected correctly, the sub-provers convince the verifier that the following identity holds using the distributed Permutation Check PIOP:

$$M(\boldsymbol{x}) = M(\sigma(\boldsymbol{x})),\ \forall \boldsymbol{x} \in \{0,1\}^{n+2},$$

where $M(0,0,\boldsymbol{x}) = L(\boldsymbol{x})$, $M(0,1,\boldsymbol{x}) = R(\boldsymbol{x})$, $M(1,0,\boldsymbol{x}) = O(\boldsymbol{x})$, and σ is a permutation specified by the circuits.

Public Input Consistency. Without loss of generality, we assume that there are 2^k public inputs to the circuit, and each public input is provided as the left input to each of the first 2^k gates. Consistency of the witnesses with the public input can then be checked as an opening at $L(\boldsymbol{r}_{pi} \| 0^{n-k})$ where $\boldsymbol{r}_{pi} \in \mathbb{F}^k$ is a random challenge.

Appendix C. Formal Construction of deMKZG PCS

We give a formal description of the deMKZG PCS in Protocol C.1.

Appendix D. Distributed Dory Reduce

We give a formal description of the IPA Reduce protocol of deDory in Protocol D.1 and the IPA reduction protocol of Dory [17] in Protocol D.2 for completeness.

Appendix E. Distributed Grand Product PIOP Using Layered Circuits

We first review the method introduced in [42] for proving $s = \prod_{\boldsymbol{z} \in \{0,1\}^n} f(\boldsymbol{z})$ with a layered circuit.

The circuit has depth n, where layer 0 is the output layer and layer n is the input layer. The final claim on layer 0 is a scalar $V_0 = s$, while the input polynomial on layer n is $V_n(\boldsymbol{z}) = f(\boldsymbol{z})$. In each j-th layer where $j \in [n]$, each gate takes inputs from two gates in the $(j+1)$-th layer. The polynomial V_j for the j-th layer is specified as $V_j(\boldsymbol{z}) = \sum_{\boldsymbol{x} \in \{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{z}) V_{j+1}(0, \boldsymbol{x}) V_{j+1}(1, \boldsymbol{x})$.

The SumCheck protocol is applied layer by layer, starting from layer 0, to check that V_{i-1} is computed correctly from V_i. At the end of each round's SumCheck, the claim from the previous layer is distilled into two claims on the evaluations of the current layer polynomial, $V_i(0, \boldsymbol{r_i})$ and $V_i(1, \boldsymbol{r_i})$. Using standard techniques, $\mathcal{V}$ can combine the two claims into one with an additional random challenge. Then the proof proceeds to the next layer. This continues until we reach the input layer, where the final claim $V_n(\boldsymbol{z}) = f(\boldsymbol{z})$ can be verified with an oracle call.

In the distributed setting, each sub-prover $\mathcal{P}_i$ holds a sub-polynomial of the input polynomial, defined as $V_n^{(i)}(\boldsymbol{z}) := f(\boldsymbol{z}, \boldsymbol{bin(i)})$. At each layer j where $j \in \{n-1, \cdots, m+1\}$, each sub-prover $\mathcal{P}_i$ locally computes its sub-polynomial $V_j^{(i)}(\boldsymbol{z}) := \sum_{\boldsymbol{x} \in \{0,1\}^{j-m}} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{z}) V_{j+1}^{(i)}(0, \boldsymbol{x}) V_{j+1}^{(i)}(1, \boldsymbol{x})$. At layer m, the sub-polynomials held by the sub-provers are reduced to scalars. Each sub-prover $\mathcal{P}_i$ sends $V_m^{(i)} = V_m(\boldsymbol{bin(i)})$ to the master prover $\mathcal{P}_0$. $\mathcal{P}_0$ reconstructs the polynomial V_m and is then able to compute $V_j(z)$ locally for $j \in \{1, 2, \cdots, m-1\}$.

Once all the V_j polynomials have been generated, the protocol proceed as follows. The first m rounds are performed by $\mathcal{P}_0$ locally. The remaining $n-m$ rounds are executed in a distributed fashion by replacing the SumCheck protocol with its distributed counterpart.

We present the full protocol of this construction in Protocol E.1. To prove equality of two grand products, we need two invocations of Protocol E.1. We present the Multiset Check protocol using layered circuits in Protocol E.2.

Theorem 8. *The PIOP for $\mathcal{R}_{Prod}$ in Protocol E.1 is perfectly complete and has knowledge error $O(n/|\mathbb{F}|)$.*

PROTOCOL C.1. *deMKZG Polynomial Commitment Scheme*

Given $f \in \mathcal{F}_n^{(\leq 1)}$, $\mathcal{P}_i$ holds a sub-polynomial $f^{(i)} \in \mathcal{F}_{n-m}^{(\leq 1)}$ s.t. $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $\forall i \in [M]$.

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$: Samples trapdoor $\boldsymbol{t}$, and generates $\mathsf{pp} = \left\{[1]_1, [1]_2, \{[\widetilde{eq}(\boldsymbol{x}, \boldsymbol{t})]_1\}_{\boldsymbol{x}\in\{0,1\}^n}, \{[t_i]_2\}_{t_i \in \boldsymbol{t}}\right\}$.
- $\mathsf{deMKZG.Commit}(\mathsf{pp}; f)$:
  - $\mathcal{P}_i$ computes $com_f^{(i)} = \sum_{\boldsymbol{x}\in\{0,1\}^{n-m}} f^{(i)}(\boldsymbol{x}) \cdot [\widetilde{eq}(\boldsymbol{x}||\boldsymbol{bin(i)}, \boldsymbol{t})]_1$.
  - $\mathcal{P}_0$ computes and outputs commitment $com_f := \sum_{i\in[M]} com_f^{(i)}$.
- $\mathsf{deMKZG.Open}(\mathsf{pp}; com_f, f)$: $\mathcal{V}$ retrieves f from all $\mathcal{P}_i$ and checks $com_f = \mathsf{deMKZG.Commit}(\mathsf{pp}; f)$.
- $\mathsf{deMKZG.Eval}(\mathsf{pp}; f, \boldsymbol{r})$:
  - In the k-th round where $1 \leq k \leq n-m$, $\mathcal{P}_i$ computes $R_k^{(i)}(\boldsymbol{x}) = (1-r_k)\cdot R_{k-1}^{(i)}(0, \boldsymbol{x}) + r_k \cdot R_{k-1}^{(i)}(1, \boldsymbol{x})$, $Q_k^{(i)}(\boldsymbol{x}) = R_{k-1}^{(i)}(1, \boldsymbol{x}) - R_{k-1}^{(i)}(0, \boldsymbol{x})$, $\forall \boldsymbol{x} \in \{0,1\}^{n-k}$, and computes the commitment for $Q_k^{(i)}$ as $com_{Q_k}^{(i)}$.
  - $\mathcal{P}_i$ sends $\left\{com_{Q_k}^{(i)}\right\}_{k=1}^{n-m}$, $Q_{n-m}^{(i)}$ and $R_{n-m}^{(i)}$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ computes $com_{Q_k} = \sum_{i\in[M]} com_{Q_k}^{(i)}, \forall k = 1, \ldots, n-m$ and construct evaluation table for Q_{n-m} and R_{n-m}.
  - In the k-th round where $n-m+1 \leq k \leq n$, $\mathcal{P}_0$ computes $R_k(\boldsymbol{x}) = (1-r_k)\cdot R_{k-1}(0, \boldsymbol{x}) + r_k \cdot R_{k-1}(1, \boldsymbol{x})$, $Q_k(\boldsymbol{x}) = R_{k-1}(1, \boldsymbol{x}) - R_{k-1}(0, \boldsymbol{x})$, $\forall \boldsymbol{x} \in \{0,1\}^{n-k}$, and computes the commitment for Q_k as com_{Q_k}.
  - $\mathcal{P}_0$ sends $\{com_{Q_k}\}_{k=1}^n$ to $\mathcal{V}$. $\mathcal{V}$ accepts if $e(com_f - [v]_1, [1]_2) = \sum_{k=1}^n e(com_{Q_k}, [t_k - r_k]_2)$.

PROTOCOL D.1. $\mathsf{deDory\text{-}Reduce}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$

$\mathcal{P}_i$ holds witness $\boldsymbol{v}_1^{(i)}, \boldsymbol{v}_2^{(i)}$ w.r.t. $\boldsymbol{v}_1, \boldsymbol{v}_2$ s.t. $((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2})) \in \mathcal{R}_{\text{Inner}}$.
All $\mathcal{P}_i$ pre-compute $\Delta_{1L} = \langle \Gamma_{1L}, \Gamma_2' \rangle$, $\Delta_{1R} = \langle \Gamma_{1R}, \Gamma_2' \rangle$, $\Delta_{2L} = \langle \Gamma_1', \Gamma_{2L} \rangle$, $\Delta_{2R} = \langle \Gamma_1', \Gamma_{2R} \rangle$, and $\chi = \langle \Gamma_1, \Gamma_2 \rangle$.

- Each $\mathcal{P}_i$ computes $D_{1L}^{(i)} = \langle \boldsymbol{v_{1L}^{(i)}}, \Gamma_2'^{(i)} \rangle$, $D_{1R}^{(i)} = \langle \boldsymbol{v_{1R}^{(i)}}, \Gamma_2'^{(i)} \rangle$, $D_{2L}^{(i)} = \langle \Gamma_1'^{(i)}, \boldsymbol{v_{2L}^{(i)}} \rangle$, $D_{2R}^{(i)} = \langle \Gamma_1'^{(i)}, \boldsymbol{v_{2R}^{(i)}} \rangle$, $E_{1\beta}^{(i)} = \langle \Gamma_1^{(i)}, \boldsymbol{s_2^{(i)}} \rangle$, $E_{2\beta}^{(i)} = \langle \boldsymbol{s_1^{(i)}}, \Gamma_2^{(i)} \rangle$, and sends them to $\mathcal{P}_0$.
- $\mathcal{P}_0$ reconstructs $D_{1L}, D_{1R}, D_{2L}, D_{2R}, E_{1\beta}, E_{2\beta}$ using the corresponding share, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ samples $\beta \leftarrow_\$ \mathbb{F}$ and sends it to $\mathcal{P}_0$. Then $\mathcal{P}_0$ transmits β to the other $\mathcal{P}_i$.
- Each $\mathcal{P}_i$ sets $\boldsymbol{v_1^{(i)}} \leftarrow \boldsymbol{v_1^{(i)}} + \beta \Gamma_1^{(i)}$ and $\boldsymbol{v_2^{(i)}} \leftarrow \boldsymbol{v_2^{(i)}} + \beta^{-1} \Gamma_2^{(i)}$.
- Each $\mathcal{P}_i$ computes $E_{1+}^{(i)} = \langle \boldsymbol{v_{1L}^{(i)}}, \boldsymbol{s_{2R}^{(i)}} \rangle$, $E_{1-}^{(i)} = \langle \boldsymbol{v_{1R}^{(i)}}, \boldsymbol{s_{2L}^{(i)}} \rangle$, $E_{2+}^{(i)} = \langle \boldsymbol{s_{1L}^{(i)}}, \boldsymbol{v_{2R}^{(i)}} \rangle$, $E_{2-}^{(i)} = \langle \boldsymbol{s_{1R}^{(i)}}, \boldsymbol{v_{2L}^{(i)}} \rangle$, $C_+^{(i)} = \langle \boldsymbol{v_{1L}^{(i)}}, \boldsymbol{v_{2R}^{(i)}} \rangle$, $C_-^{(i)} = \langle \boldsymbol{v_{1R}^{(i)}}, \boldsymbol{v_{2L}^{(i)}} \rangle$, and sends them to $\mathcal{P}_0$.
- $\mathcal{P}_0$ reconstructs $E_{1+}, E_{1-}, E_{2+}, E_{2-}, C_+, C_-$ using the corresponding share, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ samples $\alpha \leftarrow_\$ \mathbb{F}$ and sends it to $\mathcal{P}_0$. Then $\mathcal{P}_0$ transmits α to the other $\mathcal{P}_i$.
- Each $\mathcal{P}_i$ sets $\boldsymbol{v_1'}^{(i)} \leftarrow \alpha \boldsymbol{v_{1L}^{(i)}} + \boldsymbol{v_{1R}^{(i)}}$ and $\boldsymbol{v_2'}^{(i)} \leftarrow \alpha^{-1} \boldsymbol{v_{1L}^{(i)}} + \boldsymbol{v_{1R}^{(i)}}$.
- The verifier $\mathcal{V}$ computes $C' = C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_-$, $D_1' = \alpha D_{1L} + D_{1R} + \alpha\beta \Delta_{1L} + \beta \Delta_{1R}$, $D_2' = \alpha^{-1} D_{2L} + D_{2R} + \alpha^{-1}\beta^{-1}\Delta_{2L} + \beta^{-1}\Delta_{2R}$, $E_1' = E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}$, $E_2' = E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}$.
- Each $\mathcal{P}_i$ sets $\boldsymbol{s_1'}^{(i)} \leftarrow \alpha \boldsymbol{s_{1L}^{(i)}} + \boldsymbol{s_{1R}^{(i)}}$ and $\boldsymbol{s_2'}^{(i)} \leftarrow \alpha^{-1} \boldsymbol{s_{2L}^{(i)}} + \boldsymbol{s_{2R}^{(i)}}$.
- $\mathcal{V}$ accepts if $((\boldsymbol{s_1'}, \boldsymbol{s_2'}, C', D_1', D_2', E_1', E_2'); (\boldsymbol{v_1'}, \boldsymbol{v_2'})) \in \mathcal{R}_{\text{Inner}}$.

Appendix F. Distributed Rational SumCheck with Layered Circuits

In this section, we elaborate the Rational SumCheck protocol with layered circuits proposed in [37]. The distributed version naturally follows by replacing the PIOP protocols with their distributed counterparts.

The layered circuit to prove $v = \sum_{x\in\{0,1\}^n} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}$ is greatly akin to the layered circuit for product check, where layer 0 is the output layer and layer n is the input layer. The input polynomial in layer n is specified by $(p_n(\boldsymbol{x}), q_n(\boldsymbol{x})) = (p(\boldsymbol{x}), q(\boldsymbol{x}))$. Then in each j-th layer where $0 \leq j \leq n-1$, each gate takes inputs from two gates in the $(j+1)$-th layer. The witness polynomial $(p_j(\boldsymbol{x}), q_j(\boldsymbol{x}))$ for the j-th layer is specified by

$$\begin{aligned} p_j(\boldsymbol{z}) &= \textstyle\sum_{\boldsymbol{x}\{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{z})(p_{j+1}(0, \boldsymbol{z}) \cdot q_{j+1}(1, \boldsymbol{z}) \\ &\quad + p_{j+1}(1, \boldsymbol{z}) \cdot q_{j+1}(0, \boldsymbol{z})), \\ q_j(\boldsymbol{z}) &= \textstyle\sum_{\boldsymbol{x}\{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{z}) q_{j+1}(0, \boldsymbol{x}) q_{j+1}(1, \boldsymbol{x}). \end{aligned}$$

To prove the Rational SumCheck relation, $\mathcal{P}$ and $\mathcal{V}$ need n invocations of the SumCheck protocol. The proof starts from the layer 0, and the claim is finally reduced to evaluations of the input polynomials p and q on a random point.

PROTOCOL D.2. Dory-Reduce$(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$

$\mathcal{P}$ holds $\boldsymbol{v_1}, \boldsymbol{v_2}$ s.t. $((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2})) \in \mathcal{R}_{\text{Inner}}$.
The prover pre-compute $\Delta_{1L} = \langle \Gamma_{1L}, \Gamma'_2 \rangle$, $\Delta_{1R} = \langle \Gamma_{1R}, \Gamma'_2 \rangle$, $\Delta_{2L} = \langle \Gamma'_1, \Gamma_{2L} \rangle$, $\Delta_{2R} = \langle \Gamma'_1, \Gamma_{2R} \rangle$, and $\chi = \langle \Gamma_1, \Gamma_2 \rangle$.

- $\mathcal{P}$ computes and sends $D_{1L} = \langle \boldsymbol{v_{1L}}, \Gamma'_2 \rangle$, $D_{1R} = \langle \boldsymbol{v_{1R}}, \Gamma'_2 \rangle$, $D_{2L} = \langle \Gamma'_1, \boldsymbol{v_{2L}} \rangle$, $D_{2R} = \langle \Gamma'_1, \boldsymbol{v_{2R}} \rangle$, $E_{1\beta} = \langle \Gamma_1, \boldsymbol{s_2} \rangle$, $E_{2\beta} = \langle \boldsymbol{s_1}, \Gamma_2 \rangle$, then then retrieves random challenge $\beta \leftarrow_{\$} \mathbb{F}$ from $\mathcal{V}$.
- $\mathcal{P}$ sets $\boldsymbol{v_1} \leftarrow \boldsymbol{v_1} + \beta \Gamma_1$, and $\boldsymbol{v_2} \leftarrow \boldsymbol{v_2} + \beta^{-1} \Gamma_2$.
- $\mathcal{P}$ computes and sends $E_{1+} = \langle \boldsymbol{v_{1L}}, \boldsymbol{s_{2R}} \rangle$, $E_{1-} = \langle \boldsymbol{v_{1R}}, \boldsymbol{s_{2L}} \rangle$, $E_{2+} = \langle \boldsymbol{s_{1L}}, \boldsymbol{v_{2R}} \rangle$, $E_{2-} = \langle \boldsymbol{s_{1R}}, \boldsymbol{v_{2L}} \rangle$, $C_+ = \langle \boldsymbol{v_{1L}}, \boldsymbol{v_{2R}} \rangle$, $C_- = \langle \boldsymbol{v_{1R}}, \boldsymbol{v_{2L}} \rangle$, then retrieves random challenge $\alpha \leftarrow_{\$} \mathbb{F}$ from $\mathcal{V}$.
- $\mathcal{P}$ sets $\boldsymbol{v'_1} \leftarrow \alpha \boldsymbol{v_{1L}} + \boldsymbol{v_{1R}}$, and $\boldsymbol{v'_2} \leftarrow \alpha^{-1} \boldsymbol{v_{1L}} + \boldsymbol{v_{1R}}$.
- $\mathcal{V}$ computes $C' = C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_-$, $D'_1 = \alpha D_{1L} + D_{1R} + \alpha\beta \Delta_{1L} + \beta \Delta_{1R}$, $D'_2 = \alpha^{-1} D_{2L} + D_{2R} + \alpha^{-1}\beta^{-1} \Delta_{2L} + \beta^{-1} \Delta_{2R}$, $E'_1 = E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}$, $E'_2 = E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}$.
- $\mathcal{P}$ and $\mathcal{V}$ both set $\boldsymbol{s'_1} \leftarrow \alpha \boldsymbol{s_{1L}} + \boldsymbol{s_{1R}}$, and $\boldsymbol{s'_2} \leftarrow \alpha^{-1} \boldsymbol{s_{2L}} + \boldsymbol{s_{2R}}$.
- $\mathcal{V}$ accepts if $((\boldsymbol{s'_1}, \boldsymbol{s'_2}, C', D'_1, D'_2, E'_1, E'_2); (\boldsymbol{v'_1}, \boldsymbol{v'_2})) \in \mathcal{R}_{\text{Inner}}$.

PROTOCOL E.1. *Distributed Product Check with Layered Circuits*

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master prover. Given a multivariate polynomial $f : \mathbb{F}^n \to \mathbb{F}$, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $f^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ s.t. $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$. All sub-provers want to convince $\mathcal{V}$ that $\prod_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x}) = v$.

- Witness Generation Phase
  - Each sub-prover defines $V_n^{(i)}(\boldsymbol{x}) = f^{(i)}(\boldsymbol{x}), \forall \boldsymbol{x} \in \{0,1\}^{n-m}$.
  - For $n-1 \geq j \geq m$, each sub-prover $\mathcal{P}_i$ computes $V_j^{(i)}(\boldsymbol{x}) = V_{j+1}^{(i)}(0, \boldsymbol{x}) V_{j+1}^{(i)}(1, \boldsymbol{x}), \forall \boldsymbol{x} \in \{0,1\}^{j-m}$, and sends $V_m^{(i)}$ to the master prover $\mathcal{P}_0$.
  - The master prover $\mathcal{P}_0$ constructs $V_m(\boldsymbol{x})$ as $V_m(\boldsymbol{x}) = \sum_{i \in [0, M-1]} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{bin(i)}) V_m^{(i)}$.
  - For $m-1 \geq j \geq 1$, the master prover $\mathcal{P}_0$ computes $V_j(\boldsymbol{x}) = V_{j+1}(0, \boldsymbol{x}) V_{j+1}(1, \boldsymbol{x}), \forall \boldsymbol{x} \in \{0,1\}^j$.
- Proof Phase
  - For $0 \leq j \leq m-1$, $\mathcal{P}_0, \mathcal{V}$ run the SumCheck PIOP to check that $v_j = \sum_{x \in \{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \gamma^j) V_{j+1}(0, \boldsymbol{x}) v_{j+1}(1, \boldsymbol{x})$.
  - For $m-1 < j \leq n-1$, $\{\mathcal{P}_i\}_{i \in [0, M-1]}, \mathcal{V}$ run the distributed SumCheck PIOP (Protocol 3.1) to check that $v_j = \sum_{x \in \{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \gamma^j) V_{j+1}(0, \boldsymbol{x}) v_{j+1}(1, \boldsymbol{x})$.

Appendix G. Proof of Theorems

G.1. Proof of Theorem 3

Proof. **Completeness.** First, if the prover honestly generates f, it holds that $([[f \cdot q - 1]]; f \cdot q - 1) \in \mathcal{R}_{\text{Zero}}$, and the verifier accepts in the ZeroCheck PIOP, given that ZeroCheck PIOP is complete. Second, if $((v, [[p]], [[q]]); (p, q)) \in \mathcal{R}_{\text{RSum}}$, then v is exactly the summation of $p \cdot f$'s evaluations on the $\{0,1\}^n$, and the verifier accepts in the SumCheck PIOP, given that SumCheck PIOP is complete.

Knowledge soundness. We will show the soundness error of the protocol. For any $((v, [[p]], [[q]]); (p, q)) \notin \mathcal{R}_{\text{RSum}}$, it holds that either $q(\boldsymbol{x}) = 0$ for some $\boldsymbol{x} \in \{0,1\}^n$, or

$$\sum_{\boldsymbol{x} \in \{0,1\}^n} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} \neq v.$$

In the former situation, the probability that $\mathcal{V}$ accepts is at most equal to the probability that the ZeroCheck PIOP verifier accepts for $([[f \cdot q - 1]], f \cdot q - 1) \notin \mathcal{R}_{\text{Zero}}$, which is $O(dn/|\mathbb{F}|)$. In the later situation, the probability that $\mathcal{V}$ accepts is at most equal to the probability that the SumCheck PIOP verifier accepts for $((v, [[p \cdot f]]); p \cdot f)$, which is $O(dn/|\mathbb{F}|)$. Thus by union bound, the soundness error of the Rational SumCheck PIOP is $O(dn/|\mathbb{F}|)$. □

G.2. Proof of Theorem 4

Proof. The completeness and knowledge soundness follows directly from the original protocol. In the following we mainly focus on the efficiency of the protocol. To commit the polynomial $f(\boldsymbol{X})$, each sub-prover $\mathcal{P}_i$ need to compute $com_f^{(i)}$, which costs $O(2^{n-m})$ $\mathbb{G}_1$ operations, and the master prover sums them up using additional $O(2^m)$ $\mathbb{G}_1$ operations. To open the polynomial at $\boldsymbol{r}$, each sub-prover $\mathcal{P}_i$ needs to compute the corresponding $Q_k^{(i)}$ and $R_k^{(i)}$ for $k = 1, \ldots, n-m$. This costs $O(2^{n-m-k})$ $\mathbb{F}$ operations for each k, which adds up to $O(2^{n-m})$ $\mathbb{F}$ operations. The master prover reconstructs each com_{Q_k} using $O(2^m)$ $\mathbb{G}_1$ operations. The remaining Q_k and R_k, where $k = n - m +$

PROTOCOL E.2. *Distributed Multiset Check PIOP (Using Layered Circuits)*

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master prover. Given two multisets of tuples $\left\{(f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\left\{(g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ as defined in Definition 10, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $\left\{(f_1^{(i)}(\boldsymbol{x}), \ldots, f_k^{(i)}(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\left\{(g_1^{(i)}(\boldsymbol{x}), \ldots, g_k^{(i)}(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$. All sub-provers want to convince $\mathcal{V}$ that the two multisets are equal.

- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_{\$} \mathbb{F}$ and sends them to the master prover $\mathcal{P}_0$, who then transmits it to the other sub-provers.
- Each sub-prover $\mathcal{P}_i$ computes $f'^{(i)}(\boldsymbol{x}) := \sum_{i=1}^{k} \gamma^{i-1} f_i^{(i)}(\boldsymbol{x})$ and $g'^{(i)}(\boldsymbol{x}) := \sum_{i=1}^{k} \gamma^{i-1} g_i^{(i)}(\boldsymbol{x})$.
- $\{\mathcal{P}_i\}_{i\in[0,M-1]}, \mathcal{V}$ run the distributed Product Check PIOP (Protocol E.1) to check the relation $((1, [[f'+\beta]], [[g'+\beta]]); (f'+\beta, g'+\beta)) \in \mathcal{R}_{\text{Prod}}$.

$1, \ldots, n$, can be computed by the master prover using a total $O(2^m)\,\mathbb{F}$ operations, while the remaining com_{Q_k} can be computed using a total $O(2^m)\,\mathbb{G}_1$ operations. For the communication, in the commitment phase $\mathcal{P}_i$ sends $com_f^{(i)}$ to $\mathcal{P}_0$, and in opening phase, $\mathcal{P}_i$ sends $com_{Q_k}^{(i)}, \forall[k]$. Thus, the communication complexity per sub-prover is mainly $O(n)\,\mathbb{G}_1$ elements. The proof size is $O(n)\,\mathbb{G}_1$ elements corresponding to all com_{Q_k}, and the verifier time is mainly determined by $O(n)$ pairings. □

G.3. Proof of Theorem 6

Proof. The completeness follows the original Dory protocol. The knowledge soundness follows directly from the fact that deDory.Eval is knowledge sound for any vector-matrix-vector product relation. In the following we mainly focus on the efficiency of the protocol. To commit the polynomial $f(\boldsymbol{X})$, each sub-prover $\mathcal{P}_i$ need to compute $\boldsymbol{com}_{\boldsymbol{row}}^{(i)}$ and $com_M^{(i)}$, which costs $O(2^{n-m})\,\mathbb{G}_1$ operations and $O(2^{(n-m)/2})$ parings, and the master prover products them up using $O(2^m)\,\mathbb{G}_T$ operations. To open the polynomial at $\boldsymbol{r}$, each sub-prover needs to compute the corresponding $\boldsymbol{v}^{(i)}, y^{(i)}, C^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}$, which costs $O(2^{(n-m)/2})\,\mathbb{F}$, $\mathbb{G}_1$ operations and pairings, and the master prover reconstruct the elements using $O(2^m)\,\mathbb{G}_1$ and $\mathbb{G}_T$ operations. The cost in the IPA protocol is $O(2^{n-m})$ $\mathbb{G}_1$ and $\mathbb{G}_T$ operations for each sub-prover, and the additional cost for master prover is $O(2^m)$ $\mathbb{G}_1$ and $\mathbb{G}_T$ operations for the final rounds. For the communication, in the commitment phase $\mathcal{P}_i$ sends $com_M^{(i)}$ to $\mathcal{P}_0$, and in opening phase, $\mathcal{P}_i$ sends $C^{(i)}, D_2^{(i)}, E_1^{(i)}, E_2^{(i)}$ to $\mathcal{P}_0$ plus the $O(n)$ $\mathbb{G}_1$ and $\mathbb{G}_T$ elements sent in the IPA protocol. Thus, the communication complexity is $O(n)$ $\mathbb{G}_1$ and $\mathbb{G}_T$ elements. The proof size and verification time are mainly determined by IPA protocol, and in our case, they are both $O(n+m)$. □

Appendix H. Evaluation Results in WAN

As our ZKP systems have a logarithmic communication cost and round complexity among the distributed machines, we demonstrate that in a general wide area network (with 80 Mbps bandwidth and 10 ms delay), our computational advantage still dominates the overall proving time for large circuits, and our systems still outperform Pianist with notable speedups. Specifically, for vanilla gates and custom gates of size 2^{26}, the overall proving time of Pianist with 4 machines is 85.4 s and 224.3 s respectively, while HyperPianist$^{\mathsf{K}}_{\mathsf{PIOP}}$ takes 30.2 s and 50.5 s in this setting, achieving speedups of $2.8\times$ and $4.4\times$. HyperPianist$^{\mathsf{D}}_{\mathsf{PIOP}}$ also shows an improvement of $2.1\times$ and $3.4\times$ on the two circuits. The gaps get larger as the size of the sub-circuit held by each sub-prover increases. With 2 distributed machines on circuit size 2^{26}, the improvements of HyperPianist$^{\mathsf{K}}_{\mathsf{PIOP}}$ and HyperPianist$^{\mathsf{D}}_{\mathsf{PIOP}}$ over Pianist can be $3.0\times$ and $2.3\times$ on vanilla gates, and $4.6\times$ and $3.8\times$ on custom gates respectively.