HyperPianist: Pianist with Linear-Time Prover and Sub-Linear Communication Cost Under Transparent Setup

Chongrong Li[⋆], Yun Li[†], Pengfei Zhu[†],
Wenjie Qu[‡], Jiaheng Zhang[‡]

[⋆] *Beijing University of Posts and Telecommunications*,
[†] *Tsinghua University*, [‡] *National University of Singapore*

lichongrong@bupt.edu.cn, *liyunscss@gmail.com*,
zpf21@mails.tsinghua.edu.cn, *wenjiequ@u.nus.edu*, *jhzhang@nus.edu.sg*

Abstract. Zero-knowledge proofs allow one party to prove the truth of a statement without disclosing any extra information. Recent years have seen great improvements in zero-knowledge proofs. Among them, zero-knowledge SNARKs are notable for their compact and efficiently-verifiable proofs, but have relatively high prover costs. To accelerate proving, distributed zero-knowledge proof systems (Wu et al., Usenix Security 2018) are proposed: by distributing the proving process across multiple machines, such systems can achieve notable speedups in overall proving time. However, existing distributed zero-knowledge proof systems still have quasi-linear proving complexity, and they either incur a linear communication cost in circuit size among the distributed machines or achieve sub-linear communication cost with a trusted setup.

In this paper, we introduce HyperPianist, a distributed zero-knowledge proof system with a linear-time prover and sub-linear communication cost under a transparent setup. It applies to arbitrary circuits given the assumption that the witnesses are initially distributed. We first build a distributed multivariate polynomial interactive oracle proof system based on the distributed multivariate SumCheck protocol in deVirgo (Xie et al., CCS 2022) and the multivariate proof system HyperPlonk (Chen et al., Eurocrypt 2023), with a linear-time prover and incurring no extra overhead in communication or the number of constraints during distribution. To instantiate the interactive oracle proof system, we adapt a multivariate polynomial commitment scheme, Dory (Lee et al., TCC 2021), to the distributed setting, and achieve logarithmic communication cost among the distributed machines with a transparent setup. In addition, we propose HyperPianist+ as an extension of HyperPianist, by designing an optimized lookup argument based on Lasso (Setty et al., Eurocrypt 2024) and adapting it to the distributed setting.

1 Introduction

Zero-knowledge proofs (ZKPs) were first introduced in the 1980s by Goldwasser, Micali, and Rackoff, and have since become a staple of modern cryptography. In recent years, the efficiency of ZKPs has dramatically improved, enabling a multitude of new applications in blockchains and machine learning, among others.

Zero-knowledge succinct non-interactive arguments of knowledge (SNARK) are a representative 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 construct a polynomial interactive oracle proof (PIOP) system and then instantiate it with a polynomial commitment scheme (PCS). Two of the most deployed SNARKs in industries, Plonk [17] and Marlin [7], fall into this category. Plonk stands out with its compact proof size and fast verifier with a universal trusted setup, as well as support for custom gates; it has been adopted in various blockchain-related applications such as zkRollups and zkEVM (Zero-knowledge Ethereum Virtual Machine). An extension work Plonkup [31] enhances Plonk with the lookup arguments from Plookup [16], allowing the proof system to efficiently handle non-linear functions.

However, high prover costs (in computation and memory consumption) have been a main roadblock for SNARKs like Plonk to apply to large-scale circuits, such as complex EVM traces on blockchains and large language models (LLMs) in machine learning. As reported in a recent work Pianist [25], Plonk requires about 200GB of memory for a circuit with 2^{25} gates.

Wu et al. proposed a distributed zero-knowledge proof system DIZK [39] in 2018, aiming to accelerate proving by distributing the proof generation process across multiple machines (called sub-provers). As a seminal work, DIZK shows significant improvements in proving time by using distributed systems. However, considering the computational complexity, each sub-prover also needs quasi-linear time for polynomial interpolation; the communication cost among the distributed sub-provers is linear in circuit size. Subsequently, zkBridge introduced deVirgo [41], a distributed ZKP system based on multivariate polynomials. The PIOPs in deVirgo only have linear-time proving complexity, but the multivariate PCS still requires polynomial interpolation and incurs a linear communication cost among sub-provers. The most recent work Pianist [25] built a distributed ZKP system on Plonk. It utilized bivariate PIOPs for distribution, and designed a bivariate PCS based on KZG10 [23]. Armed with these, Pianist is able to achieve constant communication cost per sub-prover, but it still has quasi-linear proving cost for interpolation and requires a trusted setup.

We identify that all previous distributed ZKP systems incur quasi-linear proving time due to polynomial interpolation, involved in either PIOPs or PCS. On the other hand, another recent work HyperPlonk [6] adapts Plonk's univariate PIOPs to multivariate ones; by cooperating with a suitable multivariate PCS (that requires no polynomial interpolation), HyperPlonk can achieve linear proving complexity. As Hyperplonk's linear-time prover scales better than Plonk's quasi-linear-time prover, in practice, it can be nearly $3\times$ as fast in proving when circuit size gets to 2^{20}, and the factor gets larger as the circuit size increases.

1.1 Our Contributions

HyperPianist. In this work, we propose HyperPianist[1], a distributed proof system featuring linear proving cost and sub-linear communication cost per sub-prover with a transparent setup. It satisfies the notion of "fully" distributed ZKP system introduced in Pianist [25], as it supports general circuits other than data-parallel ones (different from Pianist, the generalization of HyperPianist from data-parallel circuits to arbitrary ones incurs no extra cost). At the core of HyperPianist, we design a distributed multivariate PIOP system based on HyperPlonk, and a distributed multivariate PCS based on Dory.

Distributed Multivariate PIOP System. We observe that the constraint system of HyperPlonk can be reduced to multivariate SumCheck identities. We thus follow the distribution paradigm of deVirgo's multivariate SumCheck PIOP, and design a distributed multivariate PIOP system based on HyperPlonk without incurring extra communication cost or new constraints. To be more specific, in Pianist, the main challenge of distributing Plonk's PIOP system for general circuits arises from the wiring constraints (i.e., copy constraints). The constraints are handled by a grand product check PIOP, but unlike data-parallel circuits, each sub-prover is unable to construct a valid sub-proof in this case. Pianist introduces a helper polynomial and two additional constraints to solve this problem, which incurs extra communication and computational costs. Distributing HyperPlonk's PIOP system faces the same challenge: the copy constraints are reduced to a multiset check identity, and are further handled by the grand product check PIOP from Quarks [35]. This Quarks PIOP also involves a helper polynomial that is not distributively computable. Computing the polynomial requires an extra linear communication cost among the sub-provers.

To handle wiring constraints in the distributed setting without incurring extra overhead, we opt for the logarithmic derivative techniques [22] to reduce the multiset check identity to a rational sumcheck PIOP. The reduction process and the rational sumcheck PIOP are distributively computable, with only logarithm communication cost among sub-provers and proof size. We point out that using layered circuits to directly prove the multiset relation is also suitable for the distributed setting. This approach has logarithm communication cost as well, but $O(\log^2 N)$ proof size for circuit size N.

Distributed Multivariate PCS: deDory. We notice that the quasi-linear proving cost and linear communication cost of deVirgo's multivariate SumCheck protocol all come from its FRI-based multivariate PCS: it requires univariate polynomial interpolation and witness exchanging among sub-provers for constructing Merkle proofs. We thus design deDory, a distributed multivariate PCS based on an additively-homomorphic PCS Dory. Given a degree-N multivariate polynomial and M distributed sub-provers, deDory has $O(1)$ commitment size, $O(\log N + \log M)$ communication cost per sub-prover and $O(\log N)$ proof size as

[1] "HyperPianist" stands for "HyperPlonk vIA uNlimited dISTribution", analogously to "Pianist" stands for "Plonk vIA uNlimited dISTribution".

Table 1: Comparisons of deDory with other distributed PCS from deVirgo and Pianist. (Communication: the communication cost among the distributed machines. Proof Size: the communication cost between the master prover and the verifier. 2^n: circuit size, 2^m: the number of distributed machines, $2^t = 2^{n-m}$: the number of witnesses each machine holds. $\mathbb{H}, \mathbb{P}, \mathbb{F}$, $\mathbb{G}, \mathbb{G}_T$: hash, field, pairing, group operation. $|\cdot|$: the size of corresponding elements.)

PCS	Trans-parent	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Communication	Proof Size
deVirgo	✓	$O(t \cdot 2^t)\,\mathbb{F} + O(2^t)\,\mathbb{H}$	$O(n^2)\,\mathbb{H}$	$O(2^n)\,\lvert\mathbb{F}\rvert$	$O(n^2 + 2^m)\,\lvert\mathbb{H}\rvert$
Pianist	✗	$O(2^t)\,\mathbb{G} + O(2^t)\mathbb{F}$	$O(1)\,\mathbb{P}$	$O(2^m)\,\lvert\mathbb{G}\rvert$	$O(1)\,\lvert\mathbb{G}\rvert$
deDory	✓	$O(2^t)\,\mathbb{G}$	$O(n)\,\mathbb{G}_T$	$O(n \cdot 2^m)\,\lvert\mathbb{G}\rvert$	$O(n)\,\lvert\mathbb{G}_T\rvert$

Table 2: Comparisons of HyperPianist with other distributed ZKP systems. (Notations are the same as above.)

Scheme	Trans-parent	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Communication	Proof Size
deVirgo	✓	$O(t \cdot 2^t)\,\mathbb{F}$ $+O(2^t)\,\mathbb{H}$	$O(n^2)\,\mathbb{H}$	$O(2^n)\,\lvert\mathbb{F}\rvert$	$O(n^2 + 2^m)\,\lvert\mathbb{H}\rvert$
Pianist	✗	$O(t \cdot 2^t)\,\mathbb{F}$ $+O(2^t)\,\mathbb{G}$	$O(1)\,\mathbb{P}$	$O(2^m)\,\lvert\mathbb{G}\rvert$	$O(1)\,\lvert\mathbb{G}\rvert$
Ours	✓	$O(2^t)\,(\mathbb{F} + \mathbb{G})$	$O(n)\,(\mathbb{F} + \mathbb{G}_T)$	$O(n \cdot 2^m)\,(\lvert\mathbb{F}\rvert + \lvert\mathbb{G}\rvert)$	$O(n)\,(\lvert\mathbb{F}\rvert + \lvert\mathbb{G}_T\rvert)$

well as verifier time with a transparent setup. Intuitively, the additively homomorphic property of Dory enables efficient aggregation of partial results with constant communication cost per sub-prover. But due to the vector-matrix-vector argument nature of Dory's evaluation proof, naïvely distributing Dory would entail $O(\sqrt{N})$ communication cost per sub-prover. We thus re-organize the matrix representation of Dory in a distributively computable fashion to avoid communication costs for unnecessary aggregation, and bring down the communication cost from $O(\sqrt{N})$ to $O(\log N + \log M)$. We give the comparison of deDory with previous distributed PCS from deVirgo [41] and Pianist [25] in Table 1. By instantiating our distributed multivariate PIOP system with deDory, we can get HyperPianist. We compare HyperPianist with deVirgo and Pianist in Table 2.

HyperPianist+: Our second contribution is HyperPianist+, an enhancement of HyperPianist with an optimized distributed lookup argument based on Lasso [36]. A lookup argument allows a party to prove that every element in a committed vector exists in a pre-determined table. To build an optimal distributed lookup argument compatible with HyperPianist, we studied the state-of-the-art lookup argument Lasso [36] that is built over multivariate polynomials. We note that Lasso involves a well-formation check to guarantee the correctness of purported values given by some polynomials w.r.t. the table. In Lasso, this is done by using offline memory checking techniques from Spartan [34]. We identify that the well-formation check can be handled more efficiently using logarithmic derivative techniques from Logup [22]. By viewing the well-formation check as a set inclusion relation, we can reduce it to a rational SumCheck identity using logarithmic derivative techniques, and can immediately save 50% of commitment work and roughly 30% of the SumCheck prover cost.

We note a concurrent work [12] uses the same idea for optimizing Lasso. We emphasize that our construction of lookup arguments is developed independently of theirs, and we additionally adapt it to the distributed setting to get a more functional distributed ZKP system.

1.2 Organization of the paper.

Section 2 presents the preliminaries. Section 3 introduces the distributed multivariate PIOP system of HyperPianist. Section 4 present the distributed multivariate polynomial commitment scheme, deDory. Section 5 shows the construction of our optimized lookup argument for HyperPianist+. Section 6 gives some experimental results. We introduce some additional related works in Appendix I.

2 Preliminaries

2.1 Notations

We use λ to denote the statistical 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 also 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 $(\boldsymbol{v}_1 || \boldsymbol{v_2})$. We use $\otimes$ to denote the Kronecker product, mapping an $m \times n$ matrix A and $p \times q$ matrix B to an $mp \times nq$ matrix. 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.

2.2 SNARKs

Definition 1 (Interactive Argument of Knowledge [6]). *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:*

$$\Pr\left[\langle \mathcal{A}_2(\mathbb{i}, \mathbb{x}, \mathsf{st}), \mathcal{V}(\mathsf{vk}, \mathbb{x}) \rangle = 1 \wedge (\mathbb{i}, \mathbb{x}, \mathbb{w}) \notin \mathcal{R} \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathbb{i}, \mathbb{x}, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ (\mathsf{vk}, \mathsf{pk}) \leftarrow \mathcal{I}(\mathsf{pp}, \mathbb{i}) \\ \mathbb{w} \leftarrow \mathcal{E}^{\mathcal{A}_1, \mathcal{A}_2}(\mathsf{pp}, \mathbb{i}, \mathbb{x}) \end{array}\right] \le \delta(|\mathbb{i}| + |\mathbb{x}|).$$

  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 [14]. 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).

2.3 Polynomial Interactive Oracle Proof

Definition 2 (Public-coin Polynomial Interactive Oracle Proof [6]). *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.*

2.4 Multilinear Extension

We define the set $\mathcal{F}_n^{(\le d)}$ 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 $f \in \mathcal{F}_n^{(\leq 1)}$. It is well-known that 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 $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{X}) := \prod_{i=1}^n (\boldsymbol{x}_i\boldsymbol{X}_i + (1-\boldsymbol{x}_i)(1-\boldsymbol{X}_i))$.

2.5 Polynomial Commitment Scheme

Definition 3. *A commitment scheme [5] for some space of messages $\mathcal{X}$ is a tuple of three protocols* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$:

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{F})$ *generates public parameters* pp.
- $(com, \pi) \leftarrow \mathsf{Commit}(\mathsf{pp}; x)$ *takes as input some $x \in \mathcal{X}$; 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 $\mathcal{A}$,*

$$\Pr\left[b_0 = b_1 \neq 0 \wedge x_0 \neq x_1 \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (com, x_0, x_1, \pi_0, \pi_1) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, com, x_0, \pi_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, com, x_1, \pi_1) \end{array} \right] \leq \mathsf{negl}(\lambda).$$

Our distributed polynomial commitment scheme is built upon Dory [24], 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).

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

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 [1]).

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

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 follows.

Definition 7 (Polynomial Commitment Scheme [24]). *A tuple of protocols* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ *is a polynomial commitment scheme for n-variable multilinear polynomial if* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open})$ *is a commitment scheme for n-variable multilinear polynomials, 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. $f \in \mathcal{F}_n^{(\leq 1)}$, $f(\boldsymbol{x}) = v$, $\mathsf{Open}(\mathsf{pp}; com_f, f, \pi_f) = 1$ and $\mathsf{Open}_\mathbb{F}(\mathsf{pp}_\mathbb{F}; com_v, v, \pi_v) = 1$.*

3 Distributed Multivariate PIOP System

In this section, we present the distributed multivariate PIOP system of HyperPianist. As illustrated in Figure 1, we decompose all constraints into gate identities and wiring identities (i.e., copy constraints) as in HyperPlonk. But we use different building blocks to construct a PIOP system that is more compatible with the distributed setting. Since the bottom building block is the fundamental SumCheck protocol, below we start from the distributed SumCheck PIOP of deVirgo [41], and show how to construct a distributed multivariate PIOP system for HyperPianist with minimal communication cost in a bottom-up fashion.

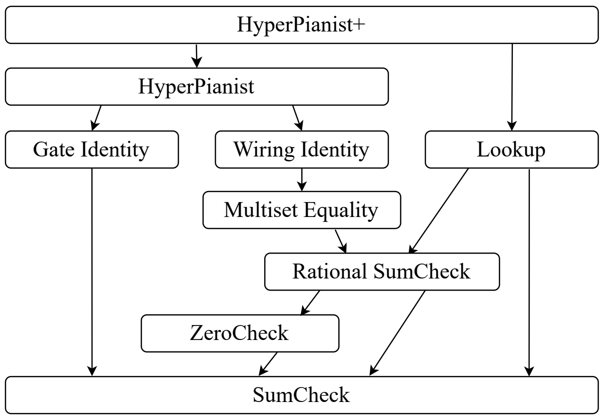

Fig. 1: Overview of the Multivariate PIOP System of HyperPianist(+).

3.1 Starting Point: Distributed SumCheck PIOP From deVirgo

A SumCheck relation is defined as follows.

Definition 8 (SumCheck Relation). *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 present the SumCheck PIOP from literature in Protocol 3.1.01. For an n-variate polynomial f, the SumCheck PIOP involves n rounds of interactions; in each round k, $\mathcal{P}$ computes a univariate polynomial $f_k(X_k)$ for $\mathcal{V}$ to check correctness of the claim in the last round $k-1$ using a random challenge r_{k-1}. After n rounds of reduction, $\mathcal{V}$ can check the final claim using an oracle call to the polynomial f. The prover cost of the SumCheck PIOP is $O(2^n)$ $\mathbb{F}$ operations, and the verifier cost is $O(n)$. The proof size is $O(n)$ $\mathbb{F}$ elements plus an oracle corresponding to the polynomial f.

Now we review deVirgo's distributed SumCheck PIOP. In deVirgo [41], the authors explored the aggregation of multiple SumCheck instances for data-parallel circuits held by several distributed machines. Here we generalize it to distributing a single SumCheck across multiple machines, given the assumption

PROTOCOL 3.1.01 *SumCheck PIOP*

$\mathcal{P}$ claims $((v, [[f]]; f) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$.

- In the first round:
  - $\mathcal{P}$ sends a univariate polynomial $f_1(X_1) = \sum_{\boldsymbol{x} \in \{0,1\}^{n-1}} f(X_1, \boldsymbol{x})$ to $\mathcal{V}$.
  - $\mathcal{V}$ checks $v = f_1(0) + f_1(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}$.
- In the k-th round where $2 \leq k \leq n-1$:
  - $\mathcal{P}$ sends a univariate polynomial $f_k(X_k) = \sum_{\boldsymbol{x} \in \{0,1\}^{n-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x})$ to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{k-1}(r_{k-1}) = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}$.
- In the n-th round:
  - $\mathcal{P}$ sends a univariate polynomial $f_n(X_n) = f(r_1, \ldots, r_{n-1}, X_n)$ to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{n-1}(r_{n-1}) = f_n(0) + f_n(1)$. If the check passes, $\mathcal{V}$ generates a random challenge $r_n \in \mathbb{F}$, and accepts iff $f_n(r_n) = f(r_1, \ldots, r_n)$ using one oracle call to $[[f]]$.

PROTOCOL 3.1.02 *Distributed SumCheck PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claim $((v, [[f]]); f) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$.

- In the first round:
  - Each $\mathcal{P}_i$ sends a local univariate polynomial $f_1^{(i)}(X_1) = \sum_{\boldsymbol{x} \in \{0,1\}^{n-m-1}} f(X_1, \boldsymbol{x}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_1(X_1) = \sum_{i \in [M]} f_1^{(i)}(X_1)$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks $v = f_1(0) + f_1(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ transmits r_1 to the other $\mathcal{P}_i$.
- In the k-th round where $2 \leq k \leq n-m$:
  - 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 $f_{k-1} = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ transmits r_k to the other $\mathcal{P}_i$.
- After the $(n-m)$-th round:
  - Each $\mathcal{P}_i$ sends $f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ computes $v' = \sum_{i \in [M]} f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$, and constructs the multilinear 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 $v' = \sum_{\boldsymbol{x} \in \{0,1\}^m} f'(\boldsymbol{x})$.

that the initial witnesses have already been distributed. We present the PIOP in Protocol 3.1.02, and elaborate on it below.

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 $\boldsymbol{bin}(i)$. Following deVirgo [41], we can define $f^{(i)}(\boldsymbol{x}) := f(\boldsymbol{x}, \boldsymbol{bin(i)})$, and have

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

The key observation for the distributed SumCheck PIOP is, in each round k where $1 \le k \le n-m$, the sub-prover $\mathcal{P}_i$ is able to locally compute a univariate polynomial $f_k^{(i)}(X_k) = \sum_{\boldsymbol{x}\in\{0,1\}^{n-m-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}, \boldsymbol{bin}(i))$. By construction, we have $\sum_{i\in[M]} f_k^{(i)}(X_k) = \sum_{\boldsymbol{x}\in\{0,1\}^{n-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x})$; the RHS is exactly the polynomial $f_k(X_k)$ defined 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 the univariate polynomial $f_k^{(i)}$, and then interact with $\mathcal{V}$ as in the regular SumCheck PIOP. Note that after receiving the random challenge r_k from $\mathcal{V}$ in each round, $\mathcal{P}_0$ needs to transmit it to other sub-provers.

Note that the distributed reduction happens only in the first $n-m$ rounds. After the $(n-m)$-th round, the multivariate polynomial has been reduced to $f'(\boldsymbol{x}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{x})$ where $\boldsymbol{x} \in \{0,1\}^m$. At this point, all sub-provers need to send $f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$, who then computes $v' = \sum_{\boldsymbol{bin(i)}\in\{0,1\}^m} f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$, and constructs the multivariate polynomial $f'(\boldsymbol{x})$ s.t. $f'(\boldsymbol{bin(i)}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$ for all $\boldsymbol{bin(i)} \in \{0,1\}^m$. Now in the following rounds, the master prover $\mathcal{P}_0$ acts as a single prover in the regular SumCheck PIOP to prove that $\sum_{\boldsymbol{x}\in\{0,1\}^m} f'(\boldsymbol{x}) = v'$. In the last round, $\mathcal{V}$ also invokes an oracle call to evaluate $f(r_1, \cdots, r_n)$ and checks the final identity.

Remark 1. HyperPlonk proposed a SumCheck PIOP for high-degree polynomials, achieving a prover time of $O(d \log^2 d \cdot 2^n)$ for $f \in \mathcal{F}_n^{(\le d)}$, where can be evaluated in $O(d)$ operations. This algorithm can be adapted similarly to the distributed setting and we give the detailed description in Protocol A.0.01.

3.2 Distributed ZeroCheck PIOP

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

PROTOCOL 3.2.01 *ZeroCheck PIOP (in HyperPlonk)*

$\mathcal{P}$ claims $(([[f]]); f) \in \mathcal{R}_{\text{Zero}}$ to $\mathcal{V}$.

- $\mathcal{V}$ samples $\boldsymbol{r} \leftarrow_{\$} \mathbb{F}^n$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ computes $\hat{f}(\boldsymbol{x}) = f(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r})$.
- $\mathcal{P}, \mathcal{V}$ run the SumCheck PIOP (Protocol 3.1.01) to check $((0, [[\hat{f}]]); \hat{f}) \in \mathcal{R}_{\text{Sum}}$.

PROTOCOL 3.2.02 *Distributed ZeroCheck PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claim $(([[f]]); f) \in \mathcal{R}_{\text{Zero}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$.

- $\mathcal{V}$ 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 3.1.02) to check $((0, [[\hat{f}]]); \hat{f}) \in \mathcal{R}_{\text{Sum}}$.

Definition 9 (ZeroCheck Relation). *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 HyperPlonk [6], a ZeroCheck relation is reduced to a SumCheck relation using a random challenge $\boldsymbol{r}$. We show the reduction in Protocol 3.2.01.

In the distributed setting, we follow the same distribution paradigm as in the distributed SumCheck PIOP, assuming that each sub-prover $\mathcal{P}_i$ holds a sub-polynomial $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$. Given the verifier's random challenge vector $\boldsymbol{r}$, the i-th sub-prover $\mathcal{P}_i$ can construct its local witness for $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{r})$ by naturally splitting the random challenge vector into $\boldsymbol{r} = (\boldsymbol{r'}, \boldsymbol{r''}) \in \mathbb{F}^{n-m} \times \mathbb{F}^m$ and then calculating $\{\widetilde{eq}(\boldsymbol{x}, \boldsymbol{r'}) \cdot \widetilde{eq}(\boldsymbol{bin(i)}, \boldsymbol{r''}) | \boldsymbol{x} \in \{0,1\}^{n-m}\}$. From here, the adaptation of ZeroCheck PIOP to the distributed setting follows naturally. We give the distributed ZeroCheck PIOP in Protocol 3.2.02.

Theorem 1. *The PIOP for $\mathcal{R}_{Zero}$ in Protocol 3.2.01 is perfectly complete and has knowledge error $O(dn/|\mathbb{F}|)$.*

3.3 Distributed Multiset Check PIOP

We now focus on the 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.

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

$$(\mathbb{x}; \mathbb{w}) = (([[f_0]], \ldots, [[f_{k-1}]], [[g_0]], \ldots, [[g_{k-1}]]); (f_0, \ldots, f_k, g_1, \ldots, g_{k-1}))$$

where $f_j, g_j \in \mathcal{F}_n^{(\le d)} (j \in [k])$ and the following two multisets of tuples are equal:

$$\left\{ f_{\boldsymbol{x}} := (f_0(\boldsymbol{x}), \ldots, f_{k-1}(\boldsymbol{x})) \right\}_{\boldsymbol{x} \in \{0,1\}^n} = \left\{ g_{\boldsymbol{x}} := (g_0(\boldsymbol{x}), \ldots, g_{k-1}(\boldsymbol{x})) \right\}_{\boldsymbol{x} \in \{0,1\}^n}.$$

As shown in Figure 3.3.01, HyperPlonk utilizes Reed-Solomon Hash and multiset hash to reduce a $\mathcal{R}^k_{\mathrm{MSet}}$ relation to a $\mathcal{R}^1_{\mathrm{MSet}}$ relation and further to the equality of two grand products.

PROTOCOL 3.3.01 *Multiset Check PIOP (in HyperPlonk)*

$\mathcal{P}$ claims $(([[f_0]], \ldots, [[f_{k-1}]], [[g_0]], \ldots, [[g_{k-1}]]) \in \mathcal{R}^k_{\mathrm{MSet}}$ to $\mathcal{V}$.

- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_{\$} \mathbb{F}$ and sends them to $\mathcal{P}$.
- $\mathcal{P}$ computes $f'(\boldsymbol{x}) = \sum_{j=0}^{k-1} \gamma^j f_j(\boldsymbol{x})$ and $g'(\boldsymbol{x}) = \sum_{j=0}^{k-1} \gamma^j g_j(\boldsymbol{x})$.
- $\mathcal{P}, \mathcal{V}$ run a Product Check PIOP to check $((1, [[f' + \beta]], [[g' + \beta]]); (f' + \beta, g' + \beta)) \in \mathcal{R}_{\mathrm{Prod}}$.

Theorem 2. *The PIOP for $\mathcal{R}_{MSet}$ is perfectly complete and has knowledge error $O((2^n + dn)/|\mathbb{F}|)$.*

We formally define the grand product check relation as follows.

Definition 11 (Product Check Relation). *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 $f_0, f_1 \in \mathcal{F}_n^{(\le d)}, f_1(\boldsymbol{b}) \neq 0,\ \forall \boldsymbol{b} \in \{0,1\}^n$ and $\prod_{\boldsymbol{x} \in \{0,1\}^n} f'(\boldsymbol{x}) = s$, where f' is defined as $f' = f_0/f_1$.*

HyperPlonk applies the Product Check PIOP from Quarks [35] to prove equality of two products. Below we briefly review the Product Check PIOP from Quarks [35]. It relies on the following theorem.

Theorem 3 (Product Check in Quarks [35]). *$P = \prod_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x})$ if and only if there exists $h \in \mathcal{F}_{n+1}^{(\le 1)}$ such that $h(1, \ldots, 1, 0) = P$, and $\forall x \in \{0,1\}^n$, the following hold: $h(0, \boldsymbol{x}) = f(\boldsymbol{x}), h(1, \boldsymbol{x}) = h(\boldsymbol{x}, 0) \cdot h(\boldsymbol{x}, 1)$.*

Theorem 3 shows that to prove a product check relation, it suffices to prove the existence of such a multilinear polynomial h. Therefore, $\mathcal{P}$ can provide an oracle call to a polynomial 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 $\boldsymbol{x} \in \{0,1\}^n$. Condition (1) is straightforward to verify, while conditions (2) and (3) can be verified using ZeroCheck PIOP.

Problems When Distributing Quarks 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 $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, (\boldsymbol{bin(i)}))$. To apply the distributed ZeroCheck PIOP, $\mathcal{P}_i$ needs to construct its sub-polynomial $h^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ defined as $h^{(i)}(\boldsymbol{x}) = h(\boldsymbol{x}, \boldsymbol{bin(i)})$. In Quarks [35], h is constructed as follows:

PROTOCOL 3.3.02 *Rational SumCheck PIOP*

$\mathcal{P}$ claims $((v,[[p]],[[q]]);(p,q)) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$.

- $\mathcal{P}$ computes $f(\boldsymbol{x}) = \frac{1}{q(\boldsymbol{x})}, \forall \boldsymbol{x} \in \{0,1\}^n$ and sends an oracle $[[f]]$ to $\mathcal{V}$.
- $\mathcal{P}, \mathcal{V}$ run the ZeroCheck PIOP (Protocol 3.2.01) to check $(([[f \cdot q - 1]]); f \cdot q - 1) \in \mathcal{R}_{\text{Zero}}$.
- $\mathcal{P}, \mathcal{V}$ run the SumCheck PIOP (Protocol 3.1.01) to check $((v,[[p \cdot f]]); p \cdot f) \in \mathcal{R}_{\text{Sum}}$.

- $h(1,\cdots,1) = 0$, and
- 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, the sub-prover $\mathcal{P}_i$ is unable to construct $h^{(i)}$ from $f^{(i)}$ without introducing extra communication cost. Due to the definition of h, $\mathcal{P}_i$ needs other necessary values that are held by the other sub-provers to construct $h^{(i)}$, which would incur a linear communication cost in total. We thus seek for other approaches to avoid the linear communication cost.

Our Solution: Logarithmic Derivatives. Our insight here is to use logarithmic derivatives techniques [22] to construct a Multiset Check PIOP. By using logarithmic derivatives, we can avoid introducing helper polynomials that are incompatible with the distributed setting (such as h in the Quarks Product Check PIOP). We explain this below.

Definition 12. *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 4. *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 $\mathbb{F}[X]$ if and only if $\sum_{i=1}^n \frac{1}{a_i+X} = \sum_{i=1}^n \frac{1}{b_i+X}$ in the fractional field $\mathbb{F}(X)$.*

Recall that the final step of the Multiset Check PIOP in HyperPlonk is to prove $\prod_{\boldsymbol{x}\in\{0,1\}^n} \frac{f'(\boldsymbol{x})+\beta}{g'(\boldsymbol{x})+\beta} = 1$. Instead of utilizing a grand product check, here we extend the SumCheck PIOP to fractions (where both the numerator and the denominator are polynomials) to prove the relation. We define the Rational SumCheck relation 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 $\sum_{\boldsymbol{x}\in\{0,1\}^n} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} = v$.*

PROTOCOL 3.3.03 *Distributed Rational SumCheck PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claim $((v, [[p]], [[q]]); (p, q)) \in \mathcal{R}_{\text{Sum}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $p^{(i)}(\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{bin(i)})$, $q^{(i)}(\boldsymbol{x}) = q(\boldsymbol{x}, \boldsymbol{bin(i)})$.

- 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 3.2.02) 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 3.1.02) to check $((v, [[p \cdot f]]); p \cdot f) \in \mathcal{R}_{\text{Sum}}$.

PROTOCOL 3.3.04 *Distributed Multiset Check PIOP*

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

- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_\$ \mathbb{F}$ and sends them to $\mathcal{P}_0$. $\mathcal{P}_0$ sends them to the other $\mathcal{P}_i$.
- Each $\mathcal{P}_i$ computes $f'^{(i)}(\boldsymbol{x}) = \sum_{i=1}^{k} \gamma^{i-1} f^{(i)}_i(\boldsymbol{x})$, $g'^{(i)}(\boldsymbol{x}) = \sum_{i=1}^{k} \gamma^{i-1} g^{(i)}_i(\boldsymbol{x})$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed Rational SumCheck PIOP (Protocol 3.3.03) to check $((1, [[f' + \beta]], [[g' + \beta]]); (f' + \beta, g' + \beta)) \in \mathcal{R}_{\text{RSum}}$.

This is in the form of SumCheck, but the SumCheck PIOP does not apply directly to fractions. To work it around, we can find the multilinear interpolation of the fraction, i.e., $f(\boldsymbol{x}) = \frac{1}{q(\boldsymbol{x})}$, and reduce it to a normal SumCheck PIOP as well as a ZeroCheck PIOP to guarantee the well formation of the helper polynomial f. We present the Rational SumCheck PIOP in Protocol 3.3.02.

Theorem 5. *The PIOP for $\mathcal{R}_{RSum}$ in Protocol 3.3.02 is perfectly complete and has knowledge error $O(dn/|\mathbb{F}|)$.*

This Rational SumCheck PIOP is well suited to the distributed setting: given the distributed polynomial $q(\boldsymbol{x})$, the sub-polynomials of the introduced helper polynomial f can be locally computed by the sub-provers without any additional communication cost. We give the distributed Rational SumCheck PIOP in Protocol 3.3.03. And based on it, we present our distributed Multiset Check PIOP in Protocol 3.3.04.

Remark 2. Other than the logarithmic derivative-based approach, directly using layered circuits to prove the grand product check is also compatible with the distributed setting, and adds no extra overhead in communication or the number of constraints. It incurs a slightly larger proof size of $O(n^2)$, but with lower prover cost. We elaborate on this approach in Appendix B.

PROTOCOL 3.4.01 *Permutation Check PIOP (in HyperPlonk)*

$\mathcal{P}$ claims $(\sigma; ([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{Prod}}$ to $\mathcal{V}$.

– $\mathcal{P}, \mathcal{V}$ run a Multiset Check PIOP (Protocol 3.3.04 or Protocol B.0.02) to check $((1, [[(s_{id}]], [[f]], [[s_\sigma]], [[g]])); (s_{id}, f, s_\sigma, g) \in \mathcal{R}^2_{\text{MSet}}$.

PROTOCOL 3.4.02 *Distributed Permutation Check PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claims $(\sigma; ([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{Prod}}$ to $\mathcal{V}$. $\mathcal{P}_i$ holds $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $g^{(i)}(\boldsymbol{x}) = g(\boldsymbol{x}, \boldsymbol{bin(i)})$.

– Each sub-prover $\mathcal{P}_i$ locally computes $s^{(i)}_{id} := \{s_{id}(\boldsymbol{x}, bin(i)) \mid \boldsymbol{x} \in \{0,1\}^{n-m}\}$ and $s^{(i)}_\sigma := \{s_\sigma(\boldsymbol{x}, bin(i)) \mid \boldsymbol{x} \in \{0,1\}^{n-m}\}$.
– $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run a distributed Multiset Check PIOP (Protocol 3.3.04) to check $((1, [[(s_{id}]], [[f]], [[s_\sigma]], [[g]])); (s_{id}, f, s_\sigma, g) \in \mathcal{R}^2_{\text{MSet}}$.

3.4 Distributed Permutation Check PIOP

A permutation relation shows that for two multivariate polynomials $f, g \in \mathcal{F}^{(\leq d)}_n$, the evaluations of g on boolean hypercube is a predefined permutation σ of f's evaluations on the boolean hypercube.

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

Given a predefined permutation σ, HyperPlonk introduces two helper polynomials s_{id}, s_σ s.t. the polynomial $s_{id} \in \mathcal{F}^{(\leq 1)}_n$ maps each binary string $\boldsymbol{x} \in \{0,1\}^n$ to the corresponding integer value $[\boldsymbol{x}] = \sum_{i=1}^n x_i \cdot 2^{i-1} \in \mathbb{F}$, and analogously, $s_\sigma \in \mathcal{F}^{(\leq 1)}_n$ 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 if $f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x})$, then the multisets $\{([\boldsymbol{x}], f(\boldsymbol{x}))\}_{\boldsymbol{x} \in \{0,1\}^n}$ must be identical to $\{([\sigma(\boldsymbol{x})], g(\boldsymbol{x}))\}_{\boldsymbol{x} \in \{0,1\}^n}$. We formulate this PIOP in Protocol 3.4.01.

The above Permutation Check PIOP can be distributed naturally, as each $\mathcal{P}_i$ is able to locally compute its sub-polynomial $s^{(i)}_{id} = s_{id}(\boldsymbol{x}, \boldsymbol{bin(i)})$ and $s^{(i)}_\sigma = s_\sigma(\boldsymbol{x}, \boldsymbol{bin(i)})$ for $\boldsymbol{x} \in \{0,1\}^{n-m}$. We present it in Protocol 3.4.02.

Theorem 6. *The PIOP for $\mathcal{R}_{Perm}$ is perfectly complete and has knowledge error $O((2^n + dn)/|\mathbb{F}|)$.*

The constraint system of HyperPianist directly follows HyperPlonk, and given the building blocks described above, the overall distributed PIOP system for the constraints of HyperPianist is trivial. We present them in Appendix F.

Remark 3. The above distributed PIOP construction is not zero-knowledge, but it can be achieved using the standard techniques from HyperPlonk [6, Appendix A]. Therefore, we omit the details here for simplicity.

3.5 Complexity Analysis

Since all of our distributed PIOPs are reduced to the distributed SumCheck protocol, we primarily focus on the complexity of the SumCheck PIOP in the following analysis. Using Protocol A.0.01, the workload for each sub-prover is $O(d\log^2 d \cdot 2^{n-m})$ for polynomials $f \in \mathcal{F}_n^{(\leq d)}$. The additional workload for the master prover during the first $n-m$ rounds involves summing 2^m degree-d univariate polynomials, which can be done in $O(d \cdot 2^m)$ per round. In the final m rounds, the master prover performs the remaining SumCheck, incurring a cost of $O(d\log^2 d \cdot 2^m)$. Therefore, the total additional workload for the master prover is $O((d(n-m) + d\log^2 d) \cdot 2^m)$.

In each round, each sub-prover must send its univariate polynomial, consisting of $d+1$ $\mathbb{F}$ elements, to the master prover, while the master prover needs to transmit the random challenge from the verifier to each sub-prover. Consequently, the total communication overhead for each sub-prover is $O(d(n-m))$.

For each invocation of the distributed Rational SumCheck, each sub-prover needs to additionally generate an oracle for the helper polynomial, which can be done with $O(2^{n-m})$ operations locally.

4 Distribuetd Multivariate PCS: deDory

In this section, we present deDory, a distributed multivariate PCS for instantiating the multivariate PIOP system. It is a key component of HyperPianist, and also may be of independent interest.

4.1 Review: Dory PCS

We first review the original Dory PCS in the following. The Dory PCS focuses on multilinear polynomials, with the observation that on the boolean hypercube, any polynomial (univariate or multivariate) can be transformed into an equivalent multilinear polynomial. Given a multilinear polynomial $f \in \mathcal{F}_n^{(\leq 1)}$, its matrix representation is defined as follows.

Definition 15 (Matrix Representation of Multilinear Polynomial). *For a multilinear polynomial $f : \mathbb{F}^n \to \mathbb{F}$, without loss of generality, we assume n is even and let $k := n/2$. Then the polynomial f can be represented as 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 $i = \sum_{t=1}^{k} 2^{k-t} \cdot x_t, j = \sum_{t=k+1}^{n} 2^{n-t} \cdot x_t$.*

Dory Commitment. In order to generate polynomial commitments using the matrix representation, Dory [24] proposed to use a two-tiered homomorphic commitment [20] by combining the Pedersen and AFGHO commitments. Formally, for $\boldsymbol{M}_{ij} \in \mathbb{F}^{n\times m}$, we have

– $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (\mathit{\Gamma}_1 \leftarrow_\$ \mathbb{G}_1^m, \mathit{\Gamma}_2 \leftarrow_\$ \mathbb{G}_2^n)$;
–
$$(com, \pi) \leftarrow \mathsf{Commit}(\mathsf{pp}; \boldsymbol{M}_{ij}) = \begin{Bmatrix} V_i \leftarrow \mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_1; \boldsymbol{M}_{ij}); \\ C \leftarrow \mathsf{Commit}_{AFGHO}(\mathit{\Gamma}_2; \boldsymbol{V}); \end{Bmatrix};$$

– $\mathsf{Open}(\mathsf{pp}; com, x, \pi)$: Check whether $\sum_i \mathit{\Gamma}_{2i}(\sum_j \boldsymbol{M}_{ij}\mathit{\Gamma}_{1j}) = C$.

Dory Evaluation Proof. The evaluation proof of Dory is built on the following observation: given the matrix representation of f, the evaluation of f at some point $(r_1, \dots, r_n) \in \mathbb{F}^n$ can be written as a vector-matrix-vector product. More specifically, we have

$$f(r_1, \dots, r_n) = (\otimes_{k\le n/2}\boldsymbol{v_k})^T \boldsymbol{M}(\otimes_{k>n/2}\boldsymbol{v_k}), \tag{1}$$

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

Definition 16. *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 vector-matrix-vector relation

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

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 $\mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_{1,fin}; y) = (y; com_y)$. $\mathcal{P}$ 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 $com_v := \langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle = \mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_1, \boldsymbol{v})$ is a commitment to $\boldsymbol{v}$, where $\boldsymbol{com_{row}}$ is a vector of Pedersen commitments to the rows of matrix $\boldsymbol{M}$. Recall that $com_{\boldsymbol{M}}$ is a commitment to $\boldsymbol{com_{row}} \in \mathbb{G}_1^n$. 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. $com_{\boldsymbol{M}} = \langle \boldsymbol{com_{row}}, \mathit{\Gamma}_2\rangle$, $\langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle = \langle \boldsymbol{v}, \mathit{\Gamma}_1\rangle$ and $com_y = \langle \boldsymbol{v}, \boldsymbol{R}\rangle \mathit{\Gamma}_{1,fin}$. This is further proved by two sigma protocols and an Inner-Product Argument (IPA) showing $C = \langle \boldsymbol{com_{row}}, \boldsymbol{v}\mathit{\Gamma}_{2,fin}\rangle$. We formally define the inner-product relation below.

Definition 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 $D_1 = \langle \boldsymbol{v_1}, \mathit{\Gamma}_2\rangle$, $D_2 = \langle \mathit{\Gamma}_1, \boldsymbol{v_2}\rangle$, $E_1 = \langle \boldsymbol{v_1}, \boldsymbol{s_2}\rangle$, $E_2 = \langle \boldsymbol{s_1}, \boldsymbol{v_2}\rangle$, $C = \langle \boldsymbol{v_1}, \boldsymbol{v_2}\rangle$.*

In Dory, we have $\boldsymbol{v_1} = \boldsymbol{com_{row}}, \boldsymbol{v_2} = \boldsymbol{v}\Gamma_{2,fin}$, and two public vectors $\boldsymbol{s_1} = \boldsymbol{R}, \boldsymbol{s_2} = \boldsymbol{L}$. To prove the inner-product relation, Dory utilizes an observation that for any vector $\boldsymbol{u_L}, \boldsymbol{u_R}, \boldsymbol{v_L}, \boldsymbol{v_R}$, and any non-zero scalar a,

$$\langle \boldsymbol{u_L} || \boldsymbol{u_R}, \boldsymbol{v_L} || \boldsymbol{v_R} \rangle = \langle a\boldsymbol{u_L} + \boldsymbol{u_R}, a^{-1}\boldsymbol{v_L} + \boldsymbol{v_R} \rangle - a\langle \boldsymbol{u_L}, \boldsymbol{v_R} \rangle - a^{-1}\langle \boldsymbol{u_R}, \boldsymbol{v_L} \rangle.$$

Thus a claim about the inner-product $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ of length n can be reduced to a claim about the inner-product of vectors of length $n/2$. We give a detailed description of the reduction process in Appendix C, Protocol C.0.01.

After $\log n$ iterations, the length of the inner-product is eventually reduced to 1. $\mathcal{V}$ must also compute the final $\boldsymbol{s_1}, \boldsymbol{s_2}$ to verify the *Fold-Scalars*, as described in Protocol C.0.02. In particular, there are the following scalars:

$$\langle \boldsymbol{s_1}, \otimes_{k=0}^{n-1}(\alpha_k, 1) \rangle, \ \langle \boldsymbol{s_2}, \otimes_{k=0}^{n-1}(\alpha_k^{-1}, 1) \rangle.$$

For polynomial evaluation proof, $\boldsymbol{s_1}, \boldsymbol{s_2}$ have special multiplicative structure, where each vector is a tensor product of n vectors of length 2. Thus we have

$$\langle \otimes_{k=0}^{n-1}(l_k, r_k), \otimes_{k=0}^{n-1}(a_k, 1) \rangle = \prod_{i=0}^{N-1} (l_k a_k + r_k),$$

which allows computation of the product in $O(\log n)$ operations on $\mathbb{F}$.

Combining the IPA reduction Protocol C.0.01 and Fold-Scalar verification Protocol C.0.02, we can obtain the full IPA protocol to prove the inner-product relation $((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2})) \in \mathcal{R}_{\text{Inner}}$. We show the IPA protocol in Protocol C.0.03. Based on this, we give the protocol for proving the vector-matrix-vector relation in Protocol C.0.04.

Remark 4. Protocol C.0.04 only satisfies the weaker notation called *Random Evaluation Knowledge Soundness*, instead of *Knowledge Soundness*. To address this issue, Dory [24] suggested that if the prover can open the polynomial at a random challenge point using Protocol C.0.04, then we can achieve *Knowledge Soundness*. We describe the final polynomial evaluation proof in Protocol C.0.05.

4.2 deDory: Adapting Dory to the Distributed Setting

A Naïve Attempt. At first thought, one may think Dory is naturally suitable for distribution: the polynomial is represented as a matrix, the local witnesses each sub-prover holds can be viewed as sub-columns of the matrix (or sub-rows if we transpose the matrix), and all the intermediate values involved in the PCS can be constructed by the sub-provers computing over their sub-columns (or sub-rows) then aggregating their partial results together. This straightforward approach works, but would inevitably incur a communication of $O(2^{n/2})$ between each sub-prover and the master prover. We elaborate on this issue below.

Recall that for an n-variate polynomial f, each sub-prover holds 2^{n-m} witnesses where $M = 2^m$ is the number of sub-provers. Given the matrix representation $\boldsymbol{M}$ of the polynomial f, it is clear that these witnesses construct $2^{n/2-m}$ columns (each of size $2^{n/2}$) of the matrix $\boldsymbol{M}$. To commit to $\boldsymbol{M}$ in a distributed fashion, one may transpose $\boldsymbol{M}$ so that each sub-prover holds $2^{n/2-m}$ rows of the matrix and it can apply Pedersen commitment procedure to these rows and then aggregate the commitments together using AFGHO commitment.

Now consider the evaluation procedure. In the original Dory evaluation protocol, the prover needs to compute and prove an inner-product $C = \langle \boldsymbol{v}, \boldsymbol{com_{row}} \rangle$, which implies that the prover should be able to calculate both $\boldsymbol{v}, \boldsymbol{com_{row}}$ at the same time to obtain C. In the distributed setting, it's easy for the sub-provers to calculate $\boldsymbol{com_{row}}$ as each of them holds full vectors of some sub-rows of $\boldsymbol{M}$. However, the columns they hold are incomplete; given that $\boldsymbol{v} = \boldsymbol{L}^T \boldsymbol{M}$, to compute $\boldsymbol{v}$, each sub-prover needs to send its partial results (which have size linearly in the number of columns) to the master prover for aggregation, which incurs a communication cost of $O(2^{n/2})$. Besides, after obtaining $\boldsymbol{v}$, the master prover needs to additionally re-allocate the corresponding sub-elements of $\boldsymbol{v}$ to each sub-prover as witnesses to the following IPA reduction procedure.

Achieving Logarithm Communication Cost. Below we show how to optimize the above process and construct a distributed Dory PCS with logarithm communication cost. The key insight here is to re-organize the matrix to enable the computation compatible with the distributed setting. Below we first give the high-level idea of our construction.

Since each sub-prover $\mathcal{P}_i$ holds the witnesses defining the sub-polynomial $f^{(i)}$, we could represent it using the matrix representation as well, denoted as $\boldsymbol{M}^{(i)}$ 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 re-organize their sub-matrices into a large matrix carefully. To achieve this, we opt for the block-diagonal form and define a new matrix $\hat{\boldsymbol{M}}$ as

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

To be more specific, all sub-matrices $\boldsymbol{M}^{(i)}$ are placed diagonally into the new matrix and all the other entries are set to zero. In this way, the sub-provers can calculate their local partial results $\boldsymbol{com^{(i)}_{row}}, \boldsymbol{v}^{(i)}$ and $C^{(i)}$, and then further use them as local witnesses directly in the following IPA reduction procedure. This entirely avoids the two-round interaction between the sub-provers and the master prover in the naïve approach, and thus saves $O(2^{n/2})$ communication cost. Below we give a detailed explanation of our construction, and formally describe the procedures of deDory in Protocol 4.2.02.

Distributed Dory Commitment. Since each sub-matrix $M^{(i)}$ has size $2^{(n-m)/2} \times 2^{(n-m)/2}$, the new matrix $\hat{M}$ 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 $\boldsymbol{M}^{(i)}$ using the corresponding sub-vectors $\Gamma_1^{(i)}, \Gamma_2^{(i)}$, and then aggregating the results together by the master prover (as described in Protocol 4.2.01).

PROTOCOL 4.2.01 $\mathsf{deDory\text{-}Commit}(f)$

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. $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $\forall i \in [2^m]$.

- $\mathcal{P}_i$ obtains the matrix representation of its sub-polynomial $\boldsymbol{M}^{(i)}$, computes
$$\boldsymbol{com_{row}}^{(i)} = \mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_1^{(i)}; \boldsymbol{M}^{(i)}),$$
$$com_M^{(i)} = \mathsf{Commit}_{AFGHO}(\mathit{\Gamma}_2^{(i)}; \boldsymbol{com_{row}}^{(i)}),$$
and sends $com_M^{(i)}$ to $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes commitment $com_M = \sum_{i \in [0, M-1]} com_M^{(i)}$.

PROTOCOL 4.2.02 *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. $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$, $\forall i \in [2^m]$.

- $\mathsf{deDory.Gen}(1^\lambda)$: Sample $\mathsf{pp} = (\mathit{\Gamma}_1 \in \mathbb{G}_1^N, \mathit{\Gamma}_2 \in \mathbb{G}_2^N)$.
- $\mathsf{deDory.Commit}(pp; f)$: $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ run $\mathsf{deDory\text{-}Commit}(f)$, and $\mathcal{P}_0$ outputs com_M as the commitment to f.
- $\mathsf{deDory.Open}(pp; f, \boldsymbol{r})$
  - Each $\mathcal{P}_i$ locally computes $\boldsymbol{L}^{(i)}$ and $\boldsymbol{R}^{(i)}$ corresponding to $\boldsymbol{r}$.
  - $\mathcal{P}_0$ computes the commitment to the purported value $y = f(\boldsymbol{r})$ as com_y, and sends it to the verifier $\mathcal{V}$.
  - $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run $\mathsf{deDory\text{-}Eval}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$.

Distributed Dory Evaluation Proof. As we re-arranged the structure of the matrix, we need to ensure that under this new representation, we can construct a valid vector-matrix-vector argument that guarantees the correctness of $f(\boldsymbol{r})$ w.r.t. the commitment to $\hat{\boldsymbol{M}}$. To this end, we need to also re-arrange the public vectors $\hat{\boldsymbol{L}}$ and $\hat{\boldsymbol{R}}$ accordingly s.t.
$$\hat{\boldsymbol{L}}^T \hat{\boldsymbol{M}} \hat{\boldsymbol{R}} = f(\boldsymbol{r}). \tag{2}$$
To see how to arrange the two vectors, first recall that each sub-prover $\mathcal{P}_i$ holds 2^{n-m} witnesses $f(\boldsymbol{x}, \boldsymbol{bin}(i))$ where $\boldsymbol{x} \in \{0,1\}^{n-m}$. Given the fact that the last m variables are fixed for each sub-prover, we divide the evaluation vector $\boldsymbol{r}$ into three parts, namely: $\boldsymbol{r}^{(0)} = \boldsymbol{r}[0 : (n-m)/2]$, $\boldsymbol{r}^{(1)} = \boldsymbol{r}[(n-m)/2 : n-m]$, $\boldsymbol{r}^{(2)} = \boldsymbol{r}[n-m : n]$.

Recall that $\boldsymbol{M}^{(i)}$ is represented as in Definition 15 with a fixed suffix $\boldsymbol{bin}(i)$. Then for each $\mathcal{P}_i$, we can initially set its sub-vectors $\hat{\boldsymbol{L}} = \otimes_{r_k \in \boldsymbol{r}^{(0)}}(1 - r_k, r_k)$ and $\hat{\boldsymbol{R}} = \otimes_{r_k \in \boldsymbol{r}^{(1)}}(1 - r_k, r_k)$. To guarantee Equation (2), we need additionally multiply $\hat{\boldsymbol{L}}$ (or $\hat{\boldsymbol{R}}$ equivalently) by the i-th element of the vector $\otimes_{r_k \in \boldsymbol{r}^{(2)}}(1 - r_k, r_k)$. We then can get the full vectors of $\hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$ by collecting $\hat{\boldsymbol{L}}^{(i)}, \hat{\boldsymbol{R}}^{(i)}$ together. We give the formal construction of $\hat{\boldsymbol{L}}, \hat{\boldsymbol{R}}$ in the following theorem.

PROTOCOL 4.2.03 $\mathsf{deDory\text{-}IPA}_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

$\mathcal{P}_i$ 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 $j \in \{0, \ldots, n-1\}$, the sub-provers pre-compute $\Gamma_{1,j+1} = (\Gamma_{1,j})_L$, $\Gamma_{2,j+1} = (\Gamma_{2,j})_L$, for all $i \in \{0, \ldots, n\}$ compute $\chi_i = \langle \Gamma_{1,i}, \Gamma_{2,i} \rangle$, and for all $i \in \{0, \ldots, n-1\}$ compute $\Delta_{1L,i} = \langle (\Gamma_{1,i})_L, \Gamma_{2,i+1} \rangle$, $\Delta_{2L,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_L \rangle$, $\Delta_{1R,i} = \langle (\Gamma_{1,i})_R, \Gamma_{2,i+1} \rangle$, $\Delta_{2R,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_R \rangle$.

- For $j = 0, \ldots, n-m-1$, $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run
  $(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2) \leftarrow \mathsf{deDory\text{-}Reduce}_{2^{n-j}}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.
- 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 $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes $C = \sum_{i\in[0,M-1]} C^{(i)}$, $D_1 = \sum_{i\in[0,M-1]} D_1^{(i)}$,
  $D_2 = \sum_{i\in[0,M-1]} D_2^{(i)}$, $E_1 = \sum_{i\in[0,M-1]} E_1^{(i)}$, and $E_2 = \sum_{i\in[0,M-1]} E_2^{(i)}$.
- $\mathcal{P}_0$ sets $\boldsymbol{v_1} = \left(v_1^{(i)}\right)_{i\in[0,M-1]}$, $\boldsymbol{v_2} = \left(v_2^{(i)}\right)_{i\in[0,M-1]}$, $\boldsymbol{s_1} = \left(s_1^{(i)}\right)_{i\in[0,M-1]}$,
  $\boldsymbol{s_2} = \left(s_2^{(i)}\right)_{i\in[0,M-1]}$.
- For $j = n-m, \ldots, n-1$, $\mathcal{P}_0$ and $\mathcal{V}$ run
  $(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2) \leftarrow \mathsf{Dory\text{-}Reduce}_{2^{n-j}}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.
- $\mathcal{P}_0$ and $\mathcal{V}$ run $\mathsf{Dory\text{-}Fold\text{-}Scalar}(s_1, s_2, C, D_1, D_2, E_1, E_2)$.

Theorem 7. *Let* $\hat{\boldsymbol{L}} = \otimes_{r_k \in (\boldsymbol{r}^{(2)}||\boldsymbol{r}^{(0)})}(1-r_k, r_k)$*, and* $\hat{\boldsymbol{R}} = \otimes_{k\in[2^m]}(1,1) \otimes_{r_k\in\boldsymbol{r}^{(1)}} (1-r_k, r_k)$*, then* $\hat{\boldsymbol{L}}^T \hat{\boldsymbol{M}} \hat{\boldsymbol{R}} = f(\boldsymbol{r})$*.*

Proof. Let $\boldsymbol{L} = \otimes_{r_k\in\boldsymbol{r}^{(0)}}(1-r_k, r_k)$ and $\boldsymbol{R} = \otimes_{r_k\in\boldsymbol{r}^{(1)}}(1-r_k, r_k)$. Then we have $\hat{\boldsymbol{R}} = [\boldsymbol{R}||\cdots||\boldsymbol{R}]$. For $\hat{\boldsymbol{L}}$, since $\otimes_{r_k\in\boldsymbol{r}^{(2)}}(1-r_k, r_k) = \{\widetilde{eq}(\boldsymbol{r}^{(2)}, \boldsymbol{bin}(i))\}_{i\in[2^m]}$, then $\hat{\boldsymbol{L}} = [\widetilde{eq}(\boldsymbol{r}^{(2)}, \boldsymbol{bin}(0))\boldsymbol{L}||\cdots||\widetilde{eq}(\boldsymbol{r}^{(2)}, \boldsymbol{bin}(2^m-1))\boldsymbol{L}]$.

Then, $\hat{\boldsymbol{L}}^T \hat{\boldsymbol{M}} \hat{\boldsymbol{R}} = \sum_{i=0}^{2^m-1} \widetilde{eq}(\boldsymbol{r}'', \boldsymbol{bin}(i))\boldsymbol{L}^T\boldsymbol{M}^{(i)}\boldsymbol{R}$. Since $\boldsymbol{M}^{(i)}$ is the matrix representation of sub-polynomial $f^{(i)}$, and $f^{(i)}(\boldsymbol{r^{(0)}}||\,\boldsymbol{r^{(1)}}) = \boldsymbol{L}\boldsymbol{M}^{(i)}\boldsymbol{R}$, we have $\hat{\boldsymbol{L}}^T \hat{\boldsymbol{M}} \hat{\boldsymbol{R}} = \sum_{i=0}^{2^m-1} \widetilde{eq}(\boldsymbol{r}^{(2)}, \boldsymbol{bin}(i)) f^{(i)}(\boldsymbol{r^{(0)}}||\boldsymbol{r^{(1)}}) = f(\boldsymbol{r})$. □

Let $\hat{\boldsymbol{L}}^{(i)}, \hat{\boldsymbol{R}}^{(i)}$ 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, the master prover $\mathcal{P}_0$ aggregates all the reduced results from the sub-provers, and performs the last m rounds of IPA reduction and the final Fold-Scalars verification with $\mathcal{V}$ as in the regular Dory. We present the full distributed IPA protocol in Protocol 4.2.03 (the distributed IPA reduction protocol and the regular Fold-Scalars verification can be found in Protocol D.0.01 and Protocol C.0.02 respectively).

Given the distributed IPA protocol, we now present the distributed vector-matrix-vector protocol in Protocol 4.2.04. This approach reduces the communication overhead from $O(2^{n/2})$ to $O(n)$, with only a modest increase in proof size, from $O(n)$ to $O(n+m)$, due to the matrix size expanding from $2^{n/2}$ to $2^{(n+m)/2}$.

PROTOCOL 4.2.04 deDory-Eval-Re($com_M, com_y, \boldsymbol{L}, \boldsymbol{R}$)

$\mathcal{P}_i$ holds sub-matrix $\boldsymbol{M}^{(i)}, \boldsymbol{com_{row}}^{(i)}$ w.r.t. $\boldsymbol{M}, \boldsymbol{com_{row}}$ and common input $\boldsymbol{L}^{(i)}, \boldsymbol{R}^{(i)}$ w.r.t. $\boldsymbol{L}, \boldsymbol{R}$ s.t.

$$\boldsymbol{com_{row}^{(i)}} = \mathsf{Commit}_{Pedersen}(\Gamma_1; M^{(i)}),\ com_M = \mathsf{Commit}_{AFGHO}(\Gamma_2; \boldsymbol{com_{row}}),$$
$$((\boldsymbol{L}, \boldsymbol{R}, com_{\boldsymbol{M}}, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\mathrm{VMV}}.$$

- Each $\mathcal{P}_i$ computes $\boldsymbol{v}^{(i)} = \boldsymbol{L}^{(i)^T}\boldsymbol{M}^{(i)}$, $y^{(i)} = \langle \boldsymbol{v}^{(i)}, \boldsymbol{R}^{(i)} \rangle$, and sends $y^{(i)}$ to $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes $y = \sum_{i \in [0,M-1]} y^{(i)}$, and sends it to $\mathcal{V}$.
- Each $\mathcal{P}_i$ computes $C^{(i)} = e(\langle \boldsymbol{v}^{(i)}, \boldsymbol{com_{row}}^{(i)} \rangle, \Gamma_{2,fin})$, $E_2^{(i)} = y^{(i)}\Gamma_{2,fin}$, $E_1^{(i)} = \langle \boldsymbol{L}^{(i)}, \boldsymbol{com_{row}}^{(i)} \rangle$, $D_2^{(i)} = e(\langle \Gamma_1^{(i)}, \boldsymbol{v}^{(i)} \rangle, \Gamma_{2,fin})$, and sends them to $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes $C = \sum_{i \in [0,M-1]} C^{(i)}$, $D_2 = \sum_{i \in [0,M-1]} D_2^{(i)}$, $E_1 = \sum_{i \in [0,M-1]} E_1^{(i)}$, $E_2 = \sum_{i \in [0,M-1]} E_2^{(i)}$, 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($\boldsymbol{L}, \boldsymbol{R}, C, com_M, D_2, E_1, E_2$).

We present the final distributed protocol for polynomial evaluation proofs in Protocol D.0.02. The batch algorithm of Dory can also be distributed similarly.

Theorem 8. *Given polynomial $f(\boldsymbol{X}) \in \mathcal{F}_n^{(\leq 1)}$ and $M = 2^m$ sub-prover, Protocol 4.2.02 is PCS satisfying completeness and knowledge soundness. The total proving computation consists of $O(N = 2^n)$ group operations, while $O(\frac{N}{M})$ group operations for each sub-prover and $O(\frac{N}{M} + M)$ group operations for the master prover. The total communication between $\mathcal{P}_i$ and $\mathcal{P}_0$ is $O(n)$. The commitment size is 1 $\mathbb{G}_T$ element, and the proof size is $O(n+m)$ $\mathbb{G}_T$ elements. The verification cost is $O(n+m)$ group operations plus $O(1)$ pairing.*

5 HyperPianist+: with Optimized Lookup Arguments

In this section, we present an optimized lookup argument in the distributed setting and obtain HyperPianist+.

The lookup relation can be interpreted as a set inclusion relation on committed vectors, as defined in Definition 18. HyperPlonk+ demonstrates how to extend the univariate PIOP from Plookup [16] into a multivariate one. However, this transformation entails sorting the union table of public table and prover's witnesses, making the prover not strict linear. Our optimization builds on Lasso [36], the most efficient known construction of multivariate lookup arguments so far.

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

5.1 Review: Lookup Arguments from Lasso

Lasso is specialized for structured tables. It makes the observation that the lookup tables for many non-linear operations (like bitwise AND) can be broken down into smaller sub-tables, such that for some $r = (r_1, \cdots, r_c)$ and some (simple) algebraic function g,

$$T[r] = g(T_1[r_1], \cdots, T_k[r_1], \cdots, T_{\alpha-k+1}[r_c], \cdots, T_\alpha[r_c]), \tag{3}$$

where T_i are the sub-tables of the table T, and $\alpha = kc$.

In Lasso, if the prover wants to convince the verifier that a committed vector $\boldsymbol{a} \in \mathbb{F}^\ell$ is contained in another committed vector (i.e., table) $\boldsymbol{t} \in \mathbb{F}^n$, it turns to prove the existence of some sparse matrix $\boldsymbol{M} \in \mathbb{F}^{\ell \times n}$ such that in each row, there is only one non-zero entry of value 1, and $\boldsymbol{M} \cdot \boldsymbol{t} = \boldsymbol{a}$. Then, they run the SumCheck protocol on the claim

$$\sum_{\boldsymbol{y} \in \{0,1\}^{\log n}} \widetilde{M}(\boldsymbol{r}, \boldsymbol{y}) \cdot \tilde{t}(\boldsymbol{y}) = \tilde{a}(\boldsymbol{r}), \tag{4}$$

where $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$ is a random vector chose by the verifier, and $\widetilde{M}(x, y), \tilde{t}(y), \tilde{a}(x)$ is the MLE of $\boldsymbol{M}, \boldsymbol{t}, \boldsymbol{a}$ respectively.

Note that $\mathcal{V}$ can obtain $\tilde{a}(\boldsymbol{r})$ via an oracle call of $\tilde{a}$. However, naïvely committing to $\widetilde{M}$ for $\mathcal{V}$ to making oracle calls is expensive for $\mathcal{P}$. To efficiently prove the SumCheck claim (4), 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 (4) into $\sum_{\boldsymbol{i} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{i}, \boldsymbol{r}) \cdot T[nz(\boldsymbol{i})] = \tilde{a}(\boldsymbol{r})$, where for each $\boldsymbol{i}$-th row of the matrix $\boldsymbol{M}$, $nz(\boldsymbol{i})$ denotes the column index corresponding to the non-zero entry in this row, and $T[nz(\boldsymbol{i})]$ denotes the corresponding $nz(\boldsymbol{i})$-th entry of the table $\boldsymbol{t}$. Since we assume the table is decomposable, we can write the LHS of the equation as

$$\sum_{\boldsymbol{i} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{i}, \boldsymbol{r}) \cdot g(T_1[nz_1(\boldsymbol{i})], \cdots, T_k[nz_1(\boldsymbol{i})], \cdots, T_{\alpha-k+1}[nz_c(\boldsymbol{i})], \cdots, T_\alpha[nz_c(\boldsymbol{i})]).$$

Let $E_j(i)$ be the MLE of $T_j[nz_{\lceil j/k \rceil}(i)]$. Then we have

$$\sum_{\boldsymbol{i} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{i}, \boldsymbol{r}) \cdot g(E_1(i), E_2(i), \cdots, E_\alpha(i)) = \tilde{a}(\boldsymbol{r}). \tag{5}$$

Now $\mathcal{P}$ and $\mathcal{V}$ can engage in a new SumCheck instance to check Equation (5). To this end, the prover now needs to provide oracles to new polynomials E_j, and additionally, it needs to show that they are well-formed, i.e., indeed equal to the MLE of $T_j[nz_{\lceil j/k \rceil}(i)]$. In Lasso, this well-formation check is done with the offline memory checking technique from Spartan [34]. Due to page limit, we review the technique in Appendix G.

Using the offline memory techniques, for each lookup argument, the prover needs to commit $2c$ polynomials of size ℓ and c polynomials of size $N^{1/c}$. The prover also needs to run a SumCheck PIOP for Equation 5, which can be done in $2 \cdot c \cdot \ell$ using the algorithm in Remark 1. Finally, the prover needs to run c instances of memory checking, which leads to c sized-$2(N^{1/c} + \ell)$ commitment and $2c$ invocation of sized-$(N^{1/c} + \ell)$ SumCheck PIOP of degree 2 with the grand product check PIOP from Quarks [35], or $4c$ invocation of sized-$(N^{1/c} + \ell)$ SumCheck PIOP of degree 2 using layered circuits [37].

PROTOCOL 5.2.01 *Well-Formation Check PIOP*

$\mathcal{P}$ claims $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})], \forall \boldsymbol{k} \in \{0,1\}^{\log \ell}$. $\mathcal{V}$ have oracles to $E(\boldsymbol{x})$ and $\widetilde{nz}(\boldsymbol{x})$.

- $\mathcal{P}$ sends an oracle of $m(X)$ (defined in Equation (6)) to $\mathcal{V}$.
- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_\$ \mathbb{F}$, and sends them to $\mathcal{P}$.
- $\mathcal{P}$ sends the rational sum claim $v = \sum_{\boldsymbol{x} \in \{0,1\}^{\log l}} \frac{1}{\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x})}$ to $\mathcal{V}$.
- $\mathcal{P}, \mathcal{V}$ run the Rational SumCheck PIOP (Protocol 3.3.02) to check $((v, [[m(\boldsymbol{x})]], [[\beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x})]]); (m(\boldsymbol{x}), \beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$, and $((v, [[1]], [[\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x})]]); (1, \beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$.

PROTOCOL 5.2.02 *Lookup PIOP*

$\mathcal{P}$ claims $(\boldsymbol{b}; ([[a]], [[T]]); (a, T)) \in \mathcal{R}_{\text{Lookup}}$ to $\mathcal{V}$, where T is defined as in Equ. (3).

- $\mathcal{P}$ sends oracles of $E_1, \cdots, E_\alpha$ and $\widetilde{nz_1}, \cdots, \widetilde{nz_\alpha}$ to $\mathcal{V}$.
- $\mathcal{V}$ picks a random $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$, and sends it to $\mathcal{P}$.
- $\mathcal{V}$ makes an oracle call to $\tilde{a}$ and obtains $\tilde{a}(\boldsymbol{r})$.
- $\mathcal{P}, \mathcal{V}$ run a SumCheck PIOP (Protocol 3.1.01) to check $v = \sum_{\boldsymbol{k} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{r}, \boldsymbol{k}) g(E_1(\boldsymbol{k}), \cdots, E_\alpha(\boldsymbol{k}))$.
- $\mathcal{P}, \mathcal{V}$ run a Well-Formation Check PIOP (Protocol 5.2.01) to check $E_j(\boldsymbol{k}) = T_j(nz(k)), \forall \boldsymbol{k} \in \{0,1\}^{\log \ell}, j \in [\alpha]$.

5.2 Our Optimization: Using Logup Instead

We optimize the well-formation check of Lasso by making the observation that

Claim 1. If $\{(nz(\boldsymbol{k}), E(\boldsymbol{k})) | \boldsymbol{k} \in \{0,1\}^{\log \ell}\} \subset \{(\boldsymbol{x}, T[\boldsymbol{x}]) | \boldsymbol{x} \in \{0,1\}^{\log N}\}$, then $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})]$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

It is worth noting that here the two sides are sets, not multisets. This statement can be proved more efficiently with techniques from Logup [22] (or its layered circuit compilation version [30]). Logup [22] depends heavily on the logarithmic derivative technique, which is a generalized version of Theorem 4.

Theorem 9. *Let $\mathbb{F}$ be a field of characteristic $p > \max(\ell, N)$. Suppose that $(a_i)_{i=1}^{\ell}$, $(b_j)_{j=1}^{N}$ 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 $\sum_{i=1}^{\ell} \frac{1}{a_i + X} = \sum_{j=1}^{N} \frac{m_j}{b_j + X}$.*

Recall that our task is to prove the following set inclusion relation (see Claim 1) $\{(nz(\boldsymbol{k}), E(\boldsymbol{k})) | \boldsymbol{k} \in \{0,1\}^{\log \ell}\} \subset \{(\boldsymbol{x}, T(\boldsymbol{x})) | \boldsymbol{x} \in \{0,1\}^{\log N}\}$. We first use Reed-Solomon to fingerprint each tuple, i.e., using a random challenge to combine each tuple into one element, and then apply Theorem 9. We can get the following relation

$$\sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \frac{1}{nz(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}) + \beta} = \sum_{\boldsymbol{x} \in \{0,1\}^{\log N}} \frac{m(\boldsymbol{x})}{s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}) + \beta}, \quad (6)$$

where γ, β are random challenges picked by $\mathcal{V}$, and the polynomial $s_{id} \in \mathcal{F}_n^{(\leq 1)}$ is defined the same as in Section 3.4 (i.e., mapping each binary string $\boldsymbol{x} \in \{0,1\}^n$ to the corresponding integer value $[\boldsymbol{x}] = \sum_{i=1}^{n} x_i \cdot 2^{i-1} \in \mathbb{F}$).

It is worth noting that $m(\boldsymbol{k})$ for $\boldsymbol{k} \in \{0,1\}^\ell$ denotes the multiplicity of the entry $(nz(\boldsymbol{k}), E(\boldsymbol{k}))$ in the lookup vector (in fact, it is equal to $\mathsf{final\_cts}(\boldsymbol{k})$ in the offline memory checking approach). Now, to prove the well-formation of E, the prover needs to compute the polynomial $m(\boldsymbol{x})$ and invokes two rational SumCheck to prove Equation (6). The details are in Protocol 5.2.01.

We show the complete Lookup PIOP in Protocol 5.2.02. It follows Lasso [36], except that we use the Logup-based techniques for the Well-Formation Check PIOP rather than the offline memory checking approach.

Prover Time. We give a rough estimation on the prover time using our approach. For each lookup argument, the prover needs to commit c polynomials of size ℓ and size $N^{1/c}$. With the PIOP compilation, the prover needs to additionally commit c polynomials of size ℓ and size $N^{1/c}$, and invoke c degree-2 SumCheck protocol of size ℓ and size $N^{1/c}$, and c multilinear SumCheck protocol. With layered circuits, the additional prover's work is equivalent to $4c$ invocation of degree 2 SumCheck protocol of size $(\ell + N^{1/c})$. Compared with the Lasso PIOP version, our optimization roughly saves 50% of commitment work and 30% of SumCheck work. Compared with the Lasso layered circuit version, we can save approximately 30% of commitment work.

5.3 Adapting Lookup PIOP to the Distributed Setting

Now we show how to distribute the above Well-Formation Check PIOP. As before, we assume that each sub-prover holds a sub-polynomials $nz^{(i)}(\boldsymbol{x})$. Given this, the sub-provers are able to locally construct $E^{(i)}(\boldsymbol{x}) = T[nz^{(i)}(\boldsymbol{x})]$. The main task of the sub-provers is to compute $m(\boldsymbol{x})$. We let $m'^{(i)} : \mathbb{F}^n \to \mathbb{F}$ be a multivariate polynomial that maps $\boldsymbol{x} \in \{0,1\}^n$ to the multiplicity of the entry $nz(\boldsymbol{x}, \boldsymbol{bin(i)}), E(\boldsymbol{x}, \boldsymbol{bin(i)})$ in the lookup vector. Note that the polynomial $m'^{(i)}$ can be locally computed by the sub-prover $\mathcal{P}_i$ given $E^{(i)}(\boldsymbol{x}), nz^{(i)}(\boldsymbol{x})$. Then by the definition of m, we have

$$\sum_{i \in [0, M-1]} m'^{(i)}(\boldsymbol{x}) = m(\boldsymbol{x}).$$

Thus we can let the master prover adds up all the $m'^{(i)}(\boldsymbol{x})$ from the sub-provers, and then re-allocate the corresponding sub-polynomial $m^{(i)}(\boldsymbol{x}) = m(\boldsymbol{x}, (\boldsymbol{bin(x)}))$ where $\boldsymbol{x} \in \{0,1\}^{n-m}$. Note that there are at most m non-zero multiplicity values, thus the communication between each sub-prover and the master prover is at most m.

Due to page limit, we present the distributed Well-Formation Check PIOP in Protocol H.0.01, and the full distributed Lookup PIOP in Protocol H.0.02. We then can get the final PIOP system for $\mathsf{HyperPianist}+$ by combining the distributed PIOPs for HyperPlonk and the Lookup relation. We describe the constraint system and the distributed PIOP system $\mathsf{HyperPianist}+$ in Appendix H.

Table 3: Experiment results of proving 64-bit XOR statements (using $C = 4$ decomposed subtables each of size 2^{16}).

Statement Size	Scheme	Prover Time (ms)	Proof Size (KB)	Verifier Time (ms)
2	Lasso	70.42	54	3.979
	Ours	**37.44 (1.89×)**	54	**3.746**
32	Lasso	73.28	58	7.863
	Ours	**42.88 (1.71×)**	58	**7.128**
256	Lasso	74.35	66	17.73
	Ours	**52.58 (1.41×)**	**65**	**16.55**

6 Evaluation

In this section, we conduct some evaluations on HyperPianist (+), and compare it with the state-of-the-art distributed ZKP system Pianist.

6.1 Implementation

We fully implemented HyperPianist and HyperPianist+ [2] using the Rust arkworks[3] ecosystem. We build HyperPianist by adapting the multivariate PIOP system of the open-source HyperPlonk and fully realizing the distributed multivariate PCS deDory. Our optimized lookup argument is implemented as a patch to Lasso from the Jolt project [2] (a SNARK framework for zkVMs based on Lasso). More implementation details can be found in Appendix **??**.

6.2 Experiments

Setup. We conduct three types of experiments: (1) we first fix the number of distributed machines as 32, and use random circuits of size varying from 2^{21} to 2^{25} to evaluate both HyperPianist and Painist, (2) we fix the circuit size to 2^25, and use different number of distributed machines from 4 to 32 to measure the scalability of HyperPianist in the size of the distributed system, (3) we test the performance of the lookup arguments of HyperPianist+ compared with the original Lasso.

All the distributed machines are Alibaba ecs.r8i.8xlarge cloud servers with 32 vCPUs and 256 GB memory. These machines are located in two regions.

We performed some preliminary experiments generating proofs for m random lookups of XOR statements. The subtables are the same as those used in Lasso, and parameters are set to match Lasso's default for RV32 (number of dimensions $C = 4$, memory size/subtable size $2^{16} = 65536$). Proof size reported here

[2] Our implementation is in https://anonymous.4open.science/r/HyperPianist.
[3] https://github.com/arkworks-rs

is the compressed size. Prover time does not include time needed to setup generators; verifier time does not include decompression, but includes time needed to deserialize an uncompressed proof in memory.

We show the experimental results in Table 3. Compared with Lasso, our construction of lookup arguments is $1.41 \sim 1.89\times$ as fast in proving time, and slightly faster in verifier time with the same (or smaller) proof size. The preliminary results have already shown the efficiency of our scheme. As our implementation and optimization are still ongoing, we expect more efficiency gains in the future.

References

1. Abe, M., Fuchsbauer, G., Groth, J., Haralambiev, K., Ohkubo, M.: Structure-preserving signatures and commitments to group elements. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. pp. 209–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Arun, A., Setty, S., Thaler, J.: Jolt: Snarks for virtual machines via lookups. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2024)
3. Bagad, S., Domb, Y., Thaler, J.: The sum-check protocol over fields of small characteristic. Cryptology ePrint Archive (2024)
4. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 107, pp. 14:1–14:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ICALP.2018.14, https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14
5. Bünz, B., Fisch, B., Szepieniec, A.: 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. pp. 677–706. Springer (2020)
6. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology – EUROCRYPT 2023. pp. 499–530. Springer Nature Switzerland, Cham (2023)
7. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: 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. pp. 738–768. Springer (2020)
8. Chiesa, A., Lehmkuhl, R., Mishra, P., Zhang, Y.: Eos: Efficient private delegation of zksnark provers. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6453–6469 (2023)
9. Dao, Q., Thaler, J.: Constraint-packing and the sum-check protocol over binary tower fields. Cryptology ePrint Archive (2024)
10. Dao, Q., Thaler, J.: More optimizations to sum-check proving. Cryptology ePrint Archive (2024)

11. Dayama, P., Patra, A., Paul, P., Singh, N., Vinayagamurthy, D.: How to prove any np statement jointly? efficient distributed-prover zero-knowledge protocols. Proceedings on Privacy Enhancing Technologies (2022)
12. Dore, D.: TaSSLE: Lasso for the commitment-phobic. Cryptology ePrint Archive, Paper 2024/1075 (2024), https://eprint.iacr.org/2024/1075, https://eprint.iacr.org/2024/1075
13. Eagen, L., Fiore, D., Gabizon, A.: cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Paper 2022/1763 (2022), https://eprint.iacr.org/2022/1763, https://eprint.iacr.org/2022/1763
14. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. pp. 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
15. Gabizon, A., Khovratovich, D.: flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. Cryptology ePrint Archive, Paper 2022/1447 (2022), https://eprint.iacr.org/2022/1447, https://eprint.iacr.org/2022/1447
16. Gabizon, A., Williamson, Z.J.: plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315 (2020), https://eprint.iacr.org/2020/315, https://eprint.iacr.org/2020/315
17. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
18. Garg, S., Goel, A., Jain, A., Policharla, G.V., Sekar, S.: zksaas: Zero-knowledge snarks as a service. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4427–4444 (2023)
19. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing. p. 113–122. STOC '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1374376.1374396, https://doi.org/10.1145/1374376.1374396
20. Groth, J.: Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 431–448. Springer (2011)
21. Gruen, A.: Some improvements for the piop for zerocheck. Cryptology ePrint Archive (2024)
22. Haböck, U.: Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530 (2022), https://eprint.iacr.org/2022/1530, https://eprint.iacr.org/2022/1530
23. Kate, A., Zaverucha, G.M., Goldberg, I.: 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. pp. 177–194. Springer (2010)
24. Lee, J.: Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) Theory of Cryptography. pp. 1–34. Springer International Publishing, Cham (2021)
25. Liu, T., Xie, T., Zhang, J., Song, D., Zhang, Y.: Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 35–35. IEEE Computer Society, Los Alamitos, CA, USA (may 2024). https://doi.org/10.1109/SP54263.2024.00035, https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00035

26. Liu, X., Zhou, Z., Wang, Y., He, J., Zhang, B., Yang, X., Zhang, J.: Scalable collaborative zk-snark and its application to efficient proof outsourcing. Cryptology ePrint Archive (2024)
27. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. J. ACM **39**(4), 859–868 (oct 1992). https://doi.org/10.1145/146585.146605, https://doi.org/10.1145/146585.146605
28. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87. pp. 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
29. Ozdemir, A., Boneh, D.: Experimenting with collaborative zk-snarks: Zero-knowledge proofs for distributed secrets. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4291–4308 (2022)
30. Papini, S., Haböck, U.: Improving logarithmic derivative lookups using gkr. Cryptology ePrint Archive, Paper 2023/1284 (2023), https://eprint.iacr.org/2023/1284, https://eprint.iacr.org/2023/1284
31. Pearson, L., Fitzgerald, J., Masip, H., Bellés-Muñoz, M., Muñoz-Tapia, J.L.: Plonkup: Reconciling plonk with plookup. Cryptology ePrint Archive (2022)
32. Posen, J., Kattis, A.A.: Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Paper 2022/957 (2022), https://eprint.iacr.org/2022/957, https://eprint.iacr.org/2022/957
33. Rothblum, R.D.: A note on efficient computation of the multilinear extension. Cryptology ePrint Archive (2024)
34. Setty, S.: Spartan: Efficient and general-purpose zksnarks without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020. pp. 704–737. Springer International Publishing, Cham (2020)
35. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zksnarks. Cryptology ePrint Archive, Paper 2020/1275 (2020), https://eprint.iacr.org/2020/1275, https://eprint.iacr.org/2020/1275
36. Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216 (2023), https://eprint.iacr.org/2023/1216, https://eprint.iacr.org/2023/1216
37. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013. pp. 71–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
38. Thaler, J., et al.: Proofs, arguments, and zero-knowledge. Foundations and Trends® in Privacy and Security **4**(2–4), 117–660 (2022)
39. Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: DIZK: A distributed zero knowledge proof system. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 675–692. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/wu
40. Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., Song, D.: Libra: Succinct zero-knowledge proofs with optimal prover computation. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019. pp. 733–764. Springer International Publishing, Cham (2019)
41. Xie, T., Zhang, J., Cheng, Z., Zhang, F., Zhang, Y., Jia, Y., Boneh, D., Song, D.: zkbridge: Trustless cross-chain bridges made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 3003–3017. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560652, https://doi.org/10.1145/3548606.3560652

42. Xie, T., Zhang, Y., Song, D.: Orion: Zero knowledge proof with linear prover time. In: Annual International Cryptology Conference. pp. 299–328. Springer (2022)
43. Zapico, A., Buterin, V., Khovratovich, D., Maller, M., Nitulescu, A., Simkin, M.: Caulk: Lookup arguments in sublinear time. Cryptology ePrint Archive, Paper 2022/621 (2022), https://eprint.iacr.org/2022/621, https://eprint.iacr.org/2022/621
44. Zapico, A., Gabizon, A., Khovratovich, D., Maller, M., Ràfols, C.: Baloo: Nearly optimal lookup arguments. Cryptology ePrint Archive, Paper 2022/1565 (2022), https://eprint.iacr.org/2022/1565, https://eprint.iacr.org/2022/1565
45. Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 859–876 (2020). https://doi.org/10.1109/SP40000.2020.00052

A Distributed SumCheck PIOP for high degree polynomials

In HyperPlonk [6], the authors proposed an algorithm for high-degree polynomials with special structures. Consider a multivariate polynomial

$$f(\boldsymbol{X}) = h(g_0(\boldsymbol{X}), \dots, g_{c-1}(\boldsymbol{X})) \tag{7}$$

such that $h \in \mathcal{F}_c^{(\leq d)}$ can be evaluated through an arithmetic circuits with $O(d)$ gates and $g_i, \forall i \in [1, c]$ are all multi-linear polynomials. The core idea is to compute the univariate polynomial $r_i(X)$ symbolically.

Algorithm 1 Evaluating $f_k(X)$ for each round. [6]

```
Input: The evaluation tables for current $g_0, \dots, g_{c-1}$, current table length $\ell$.
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 $f_{\boldsymbol{x}} := h(g_{0,\boldsymbol{x}}(X), \dots, g_{c-1,\boldsymbol{x}}(X)), \forall \boldsymbol{x}\{0,1\}^\ell$ 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 $f_{\boldsymbol{x}}(X) = \prod_{j=1}^{d-1} g_{j,\boldsymbol{x}}(X)$ [6]

```
Input: $g_0, \dots, g_{d-1}$ 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 fast polynomial multiplication
5:     end for
6: end for
7: return $f_{\boldsymbol{x}}(X) = t_{\log d,1}(X)$
```

PROTOCOL A.0.01 *Distributed SumCheck PIOP for high degree polynomials.*

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

- In the first round:
  - Each $\mathcal{P}_i$ computes its univariate polynomial $f_1^{(i)}(X)$ using Algorithm 1 with tables for $g_j^{(i)}$, and table length $n-m-1$, and sends it to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_1(X) = \sum_{i\in[M]} f_1^{(i)}(X)$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks $v = f_1(0) + f_1(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_1 \in \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ transmits r_1 to the other $\mathcal{P}_i$.
  - Each $\mathcal{P}_i$ updates the table corresponding to each $g_j^{(i)}$.
- In the k-th round where $2 \le k \le n-m$:
  - Each $\mathcal{P}_i$ computes its univariate polynomial $f_k^{(i)}(X)$ 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 $f_{k-1}(r_{k-1}) = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ 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.
- In the k-th round where $n-m+1 \le k \le n$:
  - $\mathcal{P}_0$ computes the univariate polynomial $f_k(X)$ using Algorithm 1 with tables for g_j, and table length $n-m-k$, and sends it to $\mathcal{V}$.
  - $\mathcal{V}$ checks $f_{k-1}(r_{k-1}) = f_k(0) + f_k(1)$. If the check passes, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ 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 $g_0, \ldots, g_{c-1}$.

Complexity. Let 2^m be the number of sub-provers, $f \in \mathcal{F}_n^{(\le d)}$ be the polynomial defined as Equation 7. The complexity of Protocol A.0.01 is as follows:

- 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.

B Distributed Grand Product PIOP Using Layered Circuits

Suppose we have a grand product $s = \prod_{\boldsymbol{z}\in\{0,1\}^n} f(\boldsymbol{z})$. We first introduce how to prove the relation using a layered circuit regularly. The circuit has depth n, where layer 0 is the output layer and layer n is the input layer. The input polynomial in layer n is specified by

$$V_n(\boldsymbol{z}) = f(\boldsymbol{z}).$$

Then in each j-th layer where $n-1 \geq j \geq 0$, each gate takes inputs from two gates in the $(j+1)$-th layer, and the witness polynomial V_j for the j-th layer is specified by

$$V_j(\boldsymbol{z}) = \sum_{\boldsymbol{x}\in\{0,1\}^j} \widetilde{eq}(\boldsymbol{x},\boldsymbol{z})V_{j+1}(0,\boldsymbol{x})V_{j+1}(1,\boldsymbol{x}). \tag{8}$$

To prove the grand product, $\mathcal{P}$ and $\mathcal{V}$ need n invocations of the SumCheck protocol. The proof starts from layer 0, and finally reduced to some random point evaluation of input polynomial f. For layer 0, $\mathcal{P}$ sends two value v_1^0 and v_1^0 purported to be equal to $V_1(0)$ and $V_1(1)$ respectively. $\mathcal{V}$ checks that $s = v_1^0 \cdot v_1^1$, and using some random challenge γ^1 to reduce the proof of v_1^0 and v_1^0 into a single point v_1, which can be done by proving the following equation with SumCheck protocol.

$$v_1 = \sum_{x\in\{0,1\}} \widetilde{eq}(x,\gamma^1)V_2(0,x)V_2(1,x).$$

In each following layer j where $1 \leq j \leq n-1$, the random opening v_j can be verified by checking

$$v_j = \sum_{\boldsymbol{x}\in\{0,1\}^j} \widetilde{eq}(x,\boldsymbol{r^j}||\gamma^j)V_{j+1}(0,\boldsymbol{x})V_{j+1}(1,\boldsymbol{x}), \tag{9}$$

where $\boldsymbol{r^j} \in \mathbb{F}^{j-1}$ is the random challenge vector chosen by $\mathcal{V}$ during the previous SumCheck protocol, and γ^j is the random challenge used to combine two proofs together in this round. At layer $n-1$, the proof is finally reduced to a random opening of $V_n(\boldsymbol{z})$, which can be directly verified by $\mathcal{V}$ using one oracle call.

Now we consider the distributed setting. We first focus on the generation of the witness polynomial for sub-provers. At the input layer n, each sub-prover $\mathcal{P}_i$ holds a sub-polynomial of the input polynomial $V_n^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ defined as $V_n^{(i)}(\boldsymbol{z}) := f(\boldsymbol{z},\boldsymbol{bin(i)})$. Then in each layer j where $n-1 \geq j \geq m-1$, the sub-prover $\mathcal{P}_i$ can locally compute a sub-polynomial $V_j^{(i)} : \mathbb{F}^{j-m} \to \mathbb{F}$ by

$$V_j^{(i)}(\boldsymbol{z}) := \sum_{\boldsymbol{x}\in\{0,1\}^{j-m}} \widetilde{eq}^{(i)}(\boldsymbol{x},\boldsymbol{z})V_{j+1}^{(i)}(0,\boldsymbol{x})V_{j+1}^{(i)}(1,\boldsymbol{x}).$$

At layer m, each sub-prover $\mathcal{P}_i$ sends $V_m^{(i)} = V_m(\boldsymbol{bin(i)})$ to the master prover $\mathcal{P}_0$. Then $\mathcal{P}_0$ reconstructs the polynomial $V_m : \mathbb{F}^m \to \mathbb{F}$ using the received $V_m^{(i)}$ from the sub-provers. The witness polynomials corresponding to the remaining layers can be constructed by $\mathcal{P}_0$ using $V_m(\boldsymbol{z})$ locally.

After all the witness polynomials are generated properly in a distributed manner, we now consider the distributed proving procedure. The protocol goes from the output layer 0 to the input layer n as in the normal setting in the first m layers: the master prover $\mathcal{P}_0$ and the verifier $\mathcal{V}$ invoke the normal SumCheck protocol to prove the claim in each layer. Then after m layers, the protocol differs — the computation can be distributed among the sub-provers. Thus, the later $n-m$ invocations of SumCheck protocol are executed in a distributed fashion using the distributed SumCheck protocol. We present the full protocol of this construction in Protocol B.0.01. To prove equality of two grand products, we need two invocations of Protocol B.0.01. We present the Multiset Check protocol using layered circuits in Protocol B.0.02.

PROTOCOL B.0.01 *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 (Protocol 3.1.01) 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 3.1.02) to check that
  $$v_j = \sum_{x\in\{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \gamma^j) V_{j+1}(0, \boldsymbol{x}) v_{j+1}(1, \boldsymbol{x}).$$

PROTOCOL B.0.02 *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 B.0.01) to check the relation $((1, [[f' + \beta]], [[g' + \beta]]); (f' + \beta, g' + \beta)) \in \mathcal{R}_{\text{Prod}}$.

Theorem 10. *The PIOP for $\mathcal{R}_{Prod}$ in Protocol B.0.01 is perfectly complete and has knowledge error $O(n/|\mathbb{F}|)$.*

Complexity. Let 2^m be the number of sub-provers, $f, g \in \mathcal{F}_n^{(\leq d)}$ be the polynomial for Rational SumCheck protocol. The complexity of Protocol B.0.02 is as follows:

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

C Dory Evaluation Proof

In this section, we present the formal protocols of the evaluation proof in Dory [24].

D Distributed Dory PCS

In this section, we give formal descriptions of the protocols in our distributed Dory PCS.

PROTOCOL C.0.01 $\mathsf{Dory\text{-}Reduce}_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$

$\mathcal{P}$ 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 $\varDelta_{1L} = \langle \varGamma_{1L}, \varGamma_2' \rangle$, $\varDelta_{1R} = \langle \varGamma_{1R}, \varGamma_2' \rangle$, $\varDelta_{2L} = \langle \varGamma_1', \varGamma_{2L} \rangle$, $\varDelta_{2R} = \langle \varGamma_1', \varGamma_{2R} \rangle$, and $\chi = \langle \varGamma_1, \varGamma_2 \rangle$.

- $\mathcal{P}$ computes $D_{1L} = \langle \boldsymbol{v_{1L}}, \varGamma_2' \rangle$, $D_{1R} = \langle \boldsymbol{v_{1R}}, \varGamma_2' \rangle$, $D_{2L} = \langle \varGamma_1', \boldsymbol{v_{2L}} \rangle$, $D_{2R} = \langle \varGamma_1', \boldsymbol{v_{2R}} \rangle$, $E_{1\beta} = \langle \varGamma_1, \boldsymbol{s_2} \rangle$, $E_{2\beta} = \langle \boldsymbol{s_1}, \varGamma_2 \rangle$, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ samples $\beta \leftarrow_{\$} \mathbb{F}$ and sends it to the prover $\mathcal{P}$.
- $\mathcal{P}$ sets $\boldsymbol{v_1} \leftarrow \boldsymbol{v_1} + \beta \varGamma_1$, and $\boldsymbol{v_2} \leftarrow \boldsymbol{v_2} + \beta^{-1} \varGamma_2$.
- $\mathcal{P}$ computes $E_{1+} = \langle \boldsymbol{v_{1L}}, \boldsymbol{s_{2R}} \rangle$, $E_{1-} = \langle \boldsymbol{v_{1R}}, \boldsymbol{s_{2L}} \rangle$, $E_{2+} = \langle \boldsymbol{s_{1L}}, \boldsymbol{v_{2R}} \rangle$, $E_{2-} = \langle \boldsymbol{s_{1R}}, \boldsymbol{v_{2L}} \rangle$, $C_+ = \langle \boldsymbol{v_{1L}}, \boldsymbol{v_{2R}} \rangle$, $C_- = \langle \boldsymbol{v_{1R}}, \boldsymbol{v_{2L}} \rangle$, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ samples $\alpha \leftarrow_{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ 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

$$\begin{aligned} &C' = C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_-, \\ &D_1' = \alpha D_{1L} + D_{1R} + \alpha\beta \varDelta_{1L} + \beta \varDelta_{1R}, \qquad &&D_2' = \alpha^{-1} D_{2L} + D_{2R} + \alpha^{-1}\beta^{-1} \varDelta_{2L} + \beta^{-1} \varDelta_{2R}, \\ &E_1' = E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}, &&E_2' = E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}. \end{aligned}$$

- $\mathcal{P}$ 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 C.0.02 Dory-Fold-Scalar$(s_1, s_2, C, D_1, D_2, E_1, E_2)$.

$\mathcal{P}$ holds v_1, v_2 s.t. $((s_1, s_2, C, D_1, D_2, E_1, E_2); (v_1, v_2)) \in \mathcal{R}_{\text{Inner}}$.
$\mathcal{P}$ pre-computes $\chi = \langle \Gamma_1, \Gamma_2 \rangle$.

- $\mathcal{V}$ samples $\gamma \leftarrow_\$ \mathbb{F}$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ defines $v_1' = v_1 + \gamma s_1 H_1, v_2' = v_2 + \gamma^{-1} s_2 H_2$, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ computes $C' = C + s_1 s_2 H_T + \gamma \cdot e(H_1, E_2) + \gamma^{-1} \cdot e(E_1, H_2)$,
  $D_1' = D_1 + e(H_1, s_1 \gamma \Gamma_2)$, $D_2' = D_2 + e(s_2 \gamma^{-1} \Gamma_1, H_2)$.
- $\mathcal{V}$ samples $d \leftarrow_\$ \mathbb{F}$ and accepts if
  $e(v_1' + d\Gamma_1, v_2' + d^{-1}\Gamma_2) = \chi + C + dcD_2 + d^{-1}cD_1$.

PROTOCOL C.0.03 Dory-IPA$_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

$\mathcal{P}$ 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 $\mathcal{P}$ pre-computes $\Gamma_{1,j+1} = (\Gamma_{1,j})_L$, $\Gamma_{2,j+1} = (\Gamma_{2,j})_L$, for all $i \in \{0, \ldots, n\}$ computes $\chi_i = \langle \Gamma_{1,i}, \Gamma_{2,i} \rangle$, and for all $i \in \{0, \ldots, n-1\}$ computes $\Delta_{1L,i} = \langle (\Gamma_{1,i})_L, \Gamma_{2,i+1} \rangle$, $\Delta_{2L,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_L \rangle$, $\Delta_{1R,i} = \langle (\Gamma_{1,i})_R, \Gamma_{2,i+1} \rangle$, and $\Delta_{2R,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_R \rangle$.

- For $j = 0, \ldots, n-1$, $\mathcal{P}$ and $\mathcal{V}$ run
$$(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2) \leftarrow \text{Dory-Reduce}_{2^{n-j}}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2).$$
- $\mathcal{P}$ and $\mathcal{V}$ run Dory-Fold-Scalar$(s_1, s_2, C, D_1, D_2, E_1, E_2)$.

E Distributed Rational SumCheck with Layered Circuits

In this section, we elaborate the distirbuted Rational SumCheck protocol with layered circuits. The original protocol was proposed in [30], and the key insight of it is to represent the fractions with projective coordinates.

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 $n-1 \leq j \leq 0$, each gate takes inputs from two gates in the $(j+1)$-th layer, and the witness polynomial $(p_j(\boldsymbol{x}), q_j(\boldsymbol{x}))$ for j-th

PROTOCOL C.0.04 $\mathsf{Dory\text{-}Eval\text{-}RE}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

$\mathcal{P}$ holds witness $\boldsymbol{M}, \boldsymbol{com_{col}}$ and common input $\boldsymbol{L}, \boldsymbol{R}$ s.t.

$$\boldsymbol{com_{col}} = \mathsf{Commit}_{Pedersen}(\Gamma_1; M_{ij}),\ com_M = \mathsf{Commit}_{AFGHO}(\Gamma_2; \boldsymbol{com_{col}}),$$
$$((\otimes_{i\geq k}\boldsymbol{v_i}, \otimes_{i<k}\boldsymbol{v_i}, com_{\boldsymbol{M^T}}, com_y); (\boldsymbol{M^T}, y)) \in \mathcal{R}_{\mathrm{VMV}}$$

where $\boldsymbol{v_i} = (1 - r_i, r_i)$ for $(r_1 \cdots, r_n) \in \mathbb{F}^n$.

- $\mathcal{P}$ computes $\boldsymbol{v} = \boldsymbol{L}^T\boldsymbol{M}$ and $y = \langle \boldsymbol{v}, \boldsymbol{R}\rangle$ and sends y to $\mathcal{V}$.
- $\mathcal{P}$ computes $C = e(\langle \boldsymbol{v}, \boldsymbol{com_{row}}\rangle, \Gamma_{2,fin})$, $D_2 = e(\langle \Gamma_1, \boldsymbol{v}\rangle, \Gamma_{2,fin})$, $E_1 = \langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle$, $E_2 = y\Gamma_{2,fin}$, and sends them to $\mathcal{V}$.
- $\mathcal{V}$ checks that $E_2 = y\Gamma_{2,fin}$, $com_y = y\Gamma_{1,fin}$ and $e(E_1, \Gamma_{2,fin}) = D_2$.
- $\mathcal{P}$ and $\mathcal{V}$ run $\mathsf{Dory\text{-}IPA}(\boldsymbol{L}, \boldsymbol{R}, C, com_M, D_2, E_1, E_2)$.

PROTOCOL C.0.05 $\mathsf{Dory\text{-}Eval}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

$\mathcal{P}$ holds witness $\boldsymbol{M}, \boldsymbol{com_{col}}$ and common input $\boldsymbol{L}, \boldsymbol{R}$ s.t.

$$\boldsymbol{com_{col}} = \mathsf{Commit}_{Pedersen}(\Gamma_1; M_{ij}),\ com_M = \mathsf{Commit}_{AFGHO}(\Gamma_2; \boldsymbol{com_{col}}),$$
$$((\otimes_{i\geq k}\boldsymbol{v_i}, \otimes_{i<k}\boldsymbol{v_i}, com_{\boldsymbol{M^T}}, com_y); (\boldsymbol{M^T}, y)) \in \mathcal{R}_{\mathrm{VMV}}$$

where $\boldsymbol{v_i} = (1 - r_i, r_i)$ for $(r_1 \cdots, r_n) \in \mathbb{F}^n$.

- $\mathcal{V}$ samples $u \leftarrow_{\$} \mathbb{F}$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ and $\mathcal{V}$ both set $\boldsymbol{L'} = (1, u, \ldots, u^{n-1})$, and $\boldsymbol{R'} = (1, u^n, \ldots, u^{n(n-1)})$.
- $\mathcal{P}$ computes $com_{y'} = \boldsymbol{L'}\boldsymbol{M}\boldsymbol{R'}\Gamma_{1,fin}$ and sends it to $\mathcal{V}$.
- $\mathcal{P}$ and $\mathcal{V}$ run $\mathsf{Dory\text{-}Eval\text{-}RE}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$ and $\mathsf{Dory\text{-}Eval\text{-}RE}(com_M, com_{y'}, \boldsymbol{L'}, \boldsymbol{R'})$.

layer is specified by

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

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 finally reduced to some random point evaluation of input polynomial p and q.

PROTOCOL D.0.01 $\mathsf{deDory\text{-}Reduce}_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

$\mathcal{P}_i$ holds witness $\boldsymbol{v}_1^{(i)}, \boldsymbol{v}_2^{(i)}$ w.r.t. $\boldsymbol{v}_1, \boldsymbol{v}_2$ s.t.

$$((\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2); (\boldsymbol{v_1}, \boldsymbol{v_2})) \in \mathcal{R}_{\text{Inner}}.$$

The sub-provers pre-compute $\varDelta_{1L} = \langle \varGamma_{1L}, \varGamma_2' \rangle$, $\varDelta_{1R} = \langle \varGamma_{1R}, \varGamma_2' \rangle$, $\varDelta_{2L} = \langle \varGamma_1', \varGamma_{2L} \rangle$, $\varDelta_{2R} = \langle \varGamma_1', \varGamma_{2R} \rangle$, and $\chi = \langle \varGamma_1, \varGamma_2 \rangle$.

- Each $\mathcal{P}_i$ computes $D_{1L}^{(i)} = \langle \boldsymbol{v_{1L}}^{(i)}, \varGamma_2'^{(i)} \rangle$, $D_{1R}^{(i)} = \langle \boldsymbol{v_{1R}}^{(i)}, \varGamma_2'^{(i)} \rangle$, $D_{2L}^{(i)} = \langle \varGamma_1'^{(i)}, \boldsymbol{v_{2L}}^{(i)} \rangle$, $D_{2R}^{(i)} = \langle \varGamma_1'^{(i)}, \boldsymbol{v_{2R}}^{(i)} \rangle$, $E_{1\beta}^{(i)} = \langle \varGamma_1^{(i)}, \boldsymbol{s_2}^{(i)} \rangle$, $E_{2\beta}^{(i)} = \langle \boldsymbol{s_1}^{(i)}, \varGamma_2^{(i)} \rangle$, and sends them to $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes $D_{1L} = \sum D_{1L}^{(i)}$, $D_{1R} = \sum D_{1R}^{(i)}$, $D_{2L} = \sum D_{2L}^{(i)}$, $D_{2R} = \sum D_{2R}^{(i)}$, $E_{1\beta} = \sum E_{1\beta}^{(i)}$, $E_{2\beta} = \sum E_{1\beta}^{(i)}$ 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 $\boldsymbol{v_1}^{(i)} \leftarrow \boldsymbol{v_1}^{(i)} + \beta \varGamma_1^{(i)}$ and $\boldsymbol{v_2}^{(i)} \leftarrow \boldsymbol{v_2}^{(i)} + \beta^{-1} \varGamma_2^{(i)}$.
- 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 $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes $E_{1+} = \sum E_{1+}^{(i)}$, $E_{1-} = \sum E_{1-}^{(i)}$, $E_{2+} = \sum E_{2+}^{(i)}$, $E_{2-} = \sum E_{2-}^{(i)}$, $C_+ = \sum C_+^{(i)}$, $C_- = \sum C_-^{(i)}$, 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
  $C' = C + \chi + \beta D_2 + \beta^{-1} D_1 + \alpha C_+ + \alpha^{-1} C_-$,
  $D_1' = \alpha D_{1L} + D_{1R} + \alpha\beta \varDelta_{1L} + \beta \varDelta_{1R}, D_2' = \alpha^{-1} D_{2L} + D_{2R} + \alpha^{-1}\beta^{-1} \varDelta_{2L} + \beta^{-1} \varDelta_{2R}$,
  $E_1' = E_1 + \beta E_{1\beta} + \alpha E_{1+} + \alpha^{-1} E_{1-}, E_2' = E_2 + \beta^{-1} E_{2\beta} + \alpha E_{2+} + \alpha^{-1} E_{2-}$.
- 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 $((\boldsymbol{s_1}', \boldsymbol{s_2}', C', D_1', D_2', E_1', E_2'); (\boldsymbol{v_1'}, \boldsymbol{v_2'})) \in \mathcal{R}_{\text{Inner}}$.

PROTOCOL D.0.02 deDory-Eval$(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master prover. Each sub-prover $\mathcal{P}_i$ holds local partial witness $\boldsymbol{M}^{(i)}, \boldsymbol{com_{col}}^{(i)}$ w.r.t. $\boldsymbol{M}, \boldsymbol{com_{col}}$ and common input $\boldsymbol{L}^{(i)}, \boldsymbol{R}^{(i)}$ w.r.t. $\boldsymbol{L}, \boldsymbol{R}$ s.t.

$$\boldsymbol{com_{col}} = \mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_1; M_{ij}),\ com_M = \mathsf{Commit}_{AFGHO}(\mathit{\Gamma}_2; \boldsymbol{com_{col}}),$$
$$((\otimes_{i \geq k} \boldsymbol{v_i}, \otimes_{i<k} \boldsymbol{v_i}, com_{\boldsymbol{M}^T}, com_y); (\boldsymbol{M^T}, y)) \in \mathcal{R}_{\text{VMV}}$$

where $\boldsymbol{v_i} = (1 - r_i, r_i)$ for $(r_1 \cdots, r_n) \in \mathbb{F}^n$.

- The verifier samples $u \leftarrow_{\$} \mathbb{F}$ and sends it to the master prover $\mathcal{P}_0$.
- The master prover $\mathcal{P}_0$ transmits u to the other sub-provers $\mathcal{P}_i$.
- Each sub-prover $\mathcal{P}_i$ sets the corresponding local vector $\boldsymbol{L'}^{(i)}, \boldsymbol{R'}^{(i)}$ w.r.t.

$$\boldsymbol{L'} = (1, u, u^2, \ldots, u^{n-1}), \quad \boldsymbol{R'} = (1, u^n, u^{2n}, \ldots, u^{n(n-1)}).$$

- Each sub-prover $\mathcal{P}_i$ computes

$$com_{y'}^{(i)} = \boldsymbol{L'}^{(i)} \boldsymbol{M}^{(i)} \boldsymbol{R'}^{(i)} \mathit{\Gamma}_{1,fin},$$

  and sends it to the master prover $\mathcal{P}_0$.
- The master prover $\mathcal{P}_0$ computes

$$com_{y'} = \sum_{i \in [0, M-1]} com_{y'}^{(i)},$$

  and sends it to the verifier $\mathcal{V}$.
- All sub-provers $\{\mathcal{P}_i\}_{i \in [0,M-1]}$ and the verifier $\mathcal{V}$ run

$$\text{Distributed-Dory-Eval-RE}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R}) \wedge$$
$$\text{Distributed-Dory-Eval-RE}(com_M, com_{y'}, \boldsymbol{L'}, \boldsymbol{R'}).$$

F Overall Distributed PIOP System of HyperPianist

In this section, we describe the overall PIOP system of HyperPianist.

F.1 Constraint System

The original Plonk considers fan-in-two circuits, where each gate takes at most two inputs. The left input, the right input, and the output of each gate are encoded by three univariate polynomials. The verifier can check the computation of each gate by a polynomial equation, which we refer to as the *gate constraint*. Additionally, the verifier also checks that the input and output of the gates are connected correctly as defined by the structure of the circuit, which we refer to as the *wiring constraint* (also called *copy constraint*).

For gate j of the circuit C, let a_j, b_j and o_j be its left input, right input, and output, respectively. We define the multivariate polynomial $\tilde{a}(\boldsymbol{X})$ to be the multilinear extension of the vector $\{a_j\}$, and similarly define polynomials $\tilde{b}(\boldsymbol{X})$ and $\tilde{o}(\boldsymbol{X})$. If gate j is an addition gate, then $a_j + b_j = o_j$, and thus $\tilde{a}(\langle j \rangle) + \tilde{b}(\langle j \rangle) = \tilde{o}(\langle j \rangle)$; if gate j is a multiplication gate, then $a_j \cdot b_j = o_j$, and thus $\tilde{a}(\langle j \rangle) \cdot \tilde{b}(\langle j \rangle) = \tilde{o}(\langle j \rangle)$. Then we can express the gate constraints as follows $g(\boldsymbol{X}) = q_a(\boldsymbol{X})\tilde{a}(\boldsymbol{X}) + q_b(\boldsymbol{X})\tilde{b}(\boldsymbol{x}) + q_o(\boldsymbol{X})\tilde{o}(\boldsymbol{X}) + q_{ab}(\boldsymbol{X})(\tilde{a}(\boldsymbol{X}) \cdot \tilde{b}(\boldsymbol{X})) + q_c(\boldsymbol{X})$, where

– if gate $j = \boldsymbol{X}$ is an addition gate, $q_a(\boldsymbol{X}) = q_b(\boldsymbol{X}) = 1$, $q_o(\boldsymbol{X}) = -1$, $q_{ab}(\boldsymbol{X}) = q_c(\boldsymbol{X}) = 0$,
– if gate $j = \boldsymbol{X}$ is a multiplication gate, $q_{ab}(\boldsymbol{X}) = 1$, $q_o(\boldsymbol{X}) = -1$, $q_a(\boldsymbol{X}) = q_b(\boldsymbol{X}) = q_c(\boldsymbol{X}) = 0$,
– if gate $j = \boldsymbol{X}$ is a public input, $q_c(\boldsymbol{X}) = in_j$, $q_o(\boldsymbol{X}) = -1$, $q_a(\boldsymbol{X}) = q_b(\boldsymbol{X}) = q_{ab}(\boldsymbol{X}) = 0$, where in_j is the public input value of gate j.

In this way, the correct evaluation of the circuit is equivalent to $g(\boldsymbol{X}) = 0$ for all $\boldsymbol{X} \in \{0,1\}^n$.

To check the wiring constraints, where a set of values are required to be equal, Plonk uses a cycle connecting all indices to be checked, σ. Then if the following sets are equal $\{(f_j, j)\} = \{(f_j, \sigma(j))\}$, all f_j must be equal.

Definition 19 (Constraint System of HyperPianist [6]). *Fix public parameters* $\mathsf{pp} := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ *where* $\mathbb{F}$ *is the field,* $\ell = 2^\nu$ *is the public input length,* $n = 2^\nu$ *is the number of constraints,* $\ell_w = 2^{\nu_m}, \ell_q = 2^{\nu_q}$ *are the number of witnesses and selector per constraint, and* $f : \mathbb{F}^{\ell_q + \ell_w} \to \mathbb{F}$ *is an algebraic map with degree* d. *The indexed relation* $\mathcal{R}_{\mathsf{HyperPianist}}$ *is the set of all tuples*

$$(\mathbb{i};\ \mathbb{x};\ \mathbb{w}) = ((q,\ \sigma);\ (p,\ [[w]]);\ w),$$

where $\sigma : B_{\mu+\nu_m} \to B_{\mu+\nu_m}$ *is a permutation,* $q \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_q}$, $p \in \mathcal{F}^{(\leq 1)}_{\mu+\nu}$, $w \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_w}$, *such that*

– *the wiring constraint is satisfied, that is,* $(\sigma;\ ([[w]],\ [[w]]);\ w) \in \mathcal{R}_{Perm}$*;*
– *the gate constraint is satisfied, that is,* $([[\tilde{f}]], \tilde{f}) \in \mathcal{R}_{Zero}$*, where the virtual polynomial* $\tilde{f} \in \mathcal{F}^{\leq d}_{\mu}$ *is defined as*

$$\begin{aligned}\tilde{f}(\boldsymbol{X}) := & f(q(\langle 0 \rangle_{\nu_q}, \boldsymbol{X}), \ldots, q(\langle \ell_q - 1 \rangle_{\nu_q}, \boldsymbol{X}), \\ & w(\langle 0 \rangle_{\nu_w}, \boldsymbol{X}), \ldots, w(\langle \ell_w - 1 \rangle_{\nu_w}, \boldsymbol{X}));\end{aligned}$$

– *the public input is consistent with the witness, that is, the public input polynomial* $p \in \mathcal{F}^{(\leq 1)}_{\nu}$ *is identical to* $w(0^{\mu+\nu_w-\nu}, \boldsymbol{X}) \in \mathcal{F}^{(\leq 1)}_{\nu}$.

We present the PIOP protocol for HyperPianist in Protocol F.1.01. We can instantiate it with the distributed multivariate PCS deDory.

G Memory-in-The-Head Used in Lasso

In the offline memory checking protocol, a checker performs a series of operations on an untrusted memory, and then checks that all operations are done

PROTOCOL F.1.01 *Distributed PIOP for* HyperPianist.

Indexer. $\mathcal{I}(q,\sigma)$ calls the permutation PIOP indexer $([[s_{id}]], [[s_\sigma]]) \leftarrow \mathcal{I}(\sigma)$. The oracle output is $([[q]], [[s_{id}]], [[s_\sigma]])$, where $q \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_q}$, s_{id}, $s_\sigma \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_m}$.
The Protocol. $\{\mathcal{P}_i(\mathsf{pp}, \mathbb{i}, p, w^{(i)})\}_{i\in[0,M-1]}$ and $\mathcal{V}(\mathsf{pp}, p, [[q]], [[s_{id}]], [[s_\sigma]])$ run the following protocol.

1. The master prover $\mathcal{P}_0$ sends $\mathcal{V}$ the witness oracle $[[w]]$ where $w \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_m}$.
2. $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run a distributed PIOP for the gate constraint, which is a distributed ZeroCheck PIOP (Protocol 3.2.02) for the relation $([[\tilde{f}]], \tilde{f}) \in \mathcal{R}_{\text{Zero}}$ where $\tilde{f} \in \mathcal{F}^{(\leq d)}_\mu$ is as defined previously.
3. $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run a distributed PIOP for the wiring constraint, which is a distributed Permutation Check PIOP for $(\sigma; ([[w]], [[w]]); (w, w)) \in \mathcal{R}_{\text{Perm}}$.
4. $\mathcal{V}$ checks the consistency between witness and public input. It samples $\boldsymbol{r} \leftarrow \mathbb{F}^n u$, queries $[[w]]$ on input $(\langle 0\rangle_{\mu+\nu_w-\nu}, \boldsymbol{r})$ and checks whether $p(\boldsymbol{r}) = w(\langle 0\rangle_{\mu+\nu_w-\nu}, \boldsymbol{r})$.

honestly. The untrusted memory maintains one multiset: S, representing the data stored. The checker locally maintains two multisets: WS and RS, representing the data written to and read from the untrusted memory, respectively. Each multiset element is three-tuple of (1) the value's address, (2) the value, and (3) the value's timestamp. During setup, The checker writes to the untrusted memory $(i, v_i, 0), \forall i \in [N]$. These tuples are added to S and WS. On each read call to address i, the untrusted memory provides an output (v_i, t_i), purported to be the value stored in address i and its corresponding timestamp. The checker adds (i, v_i, t_i) to RS and $(i, v_i, t_i + 1)$ to WS. The untrusted memory updates the timestamp $t_i = t_i + 1$ in S; that is, it removes (i, v_i, t_i) from S and adds $(i, v_i, t_i + 1)$. After all queries are done, the untrusted memory returns the local set S. Finally, the checker checks that the $\mathsf{RS} \cup \mathsf{S}$ and WS are equal as multisets.

In Lasso, for each polynomial E, the prover wants to show that $\forall \boldsymbol{k} \in \{0,1\}^{\log \ell}$, $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})]$. To do so, the prover plays the memory checking protocol in the head. It commits two more polynomials $\mathsf{read\_ts}$ and $\mathsf{final\_cts}$. For $k \in [m]$, $\mathsf{read\_ts}(k)$ represents the timestamps returned by the untrusted memory for k-th read. For $j \in [N]$, $\mathsf{final\_cts}(j)$ represents the final timestamp for the value stored at location j. Let $\mathsf{write\_cts} = \mathsf{read\_ts}+1$ denote the new timestamp the untrusted memory writes for each read call. Then, the prover only needs to show that $\mathsf{RS} \cup \mathsf{S} = \mathsf{WS}$ as multisets, where

- $\mathsf{WS} = \{(s_{id}(i), T(i), 0) \mid i \in \{0,1\}^{\log N}\} \cup \{(nz(\boldsymbol{k}), E(\boldsymbol{k}), \mathsf{write\_cts}(k)) \mid \boldsymbol{k} \in \{0,1\}^{\log \ell}\}$;
- $\mathsf{RS} = \{(nz(\boldsymbol{k}), E(\boldsymbol{k}), \mathsf{read\_ts}(k)) \mid \boldsymbol{k} \in \{0,1\}^{\log \ell}\}$;
- $\mathsf{S} = \{(s_{id}(i), T(i), \mathsf{final\_cts}(i)) \mid i \in \{0,1\}^{\log N}\}$.

PROTOCOL H.0.01 *Distributed Well-Formation Check PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claim $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})], \forall \boldsymbol{k} \in \{0,1\}^{\log \ell}$. $\mathcal{V}$ have oracles to $E(\boldsymbol{x})$ and $\widetilde{nz}(\boldsymbol{x})$.

- $\mathcal{P}_0$ sends an oracle of $m(X)$ (defined in Equation (6)) to $\mathcal{V}$.
- Each $\mathcal{P}_i$ computes $v^{(i)} = \sum_{\boldsymbol{x} \in \{0,1\}^\ell} \frac{m^{(i)}(\boldsymbol{x})}{\beta + s_{id}^{(i)}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}, \boldsymbol{bin(i)})}$, and sends it to $\mathcal{P}_0$.
- $\mathcal{P}_0$ computes the rational sumcheck claim $v = \sum_{i=0}^{M} v^{(i)}$, and sends it to $\mathcal{V}$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run the distributed Rational SumCheck PIOP (Protocol 3.3.03) to check the relation $((v, [[m(\boldsymbol{x})]], [[\beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x})]]); (m(\boldsymbol{x}), \beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$, and $((v, [[1]], [[\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x})]]); (1, \beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$.

PROTOCOL H.0.02 *Distributed Lookup PIOP*

$\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ claim $(\boldsymbol{b}; ([[a]], [[T]]); (a, T)) \in \mathcal{R}_{\text{Lookup}}$ to $\mathcal{V}$, where T is defined as in Equ. (3).

- $\mathcal{P}_0$ sends oracles of $E_1, \cdots, E_\alpha$ and $\widetilde{nz_1}, \cdots, \widetilde{nz_\alpha}$ to $\mathcal{V}$.
- $\mathcal{V}$ picks a random $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$, and sends it to $\mathcal{P}_0$. $\mathcal{P}_0$ sends it to the other $\mathcal{P}_i$.
- $\mathcal{V}$ makes an oracle call to $\tilde{a}$ and obtains $\tilde{a}(\boldsymbol{r})$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run a distributed SumCheck PIOP (Protocol 3.1.02) to check the relation that $v = \sum_{\boldsymbol{k} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{r}, \boldsymbol{k}) g(E_1(\boldsymbol{k}), \cdots, E_\alpha(\boldsymbol{k}))$.
- $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{V}$ run a distributed Well-Formation Check PIOP (Protocol H.0.01) to check that $E_j(\boldsymbol{k}) = T_j(nz(k))$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

Claim 2. If $E(\boldsymbol{k}) \neq T[nz(\boldsymbol{k})]$ for some k, there do not exist $\mathsf{read\_ts}$, $\mathsf{final\_cts}$ such that $\mathsf{RS} \cup \mathsf{S} = \mathsf{WS}$ holds given $\mathsf{write\_cts} = \mathsf{read\_ts} + 1$.

H Distributed PIOP System of HyperPianist+

To integrate the lookup argument, we only need to add constraints enforcing that some function over the witness values belongs to a pre-determined table.

Definition 20 (Constraint System of HyperPianist+ [6]). *Let* $\mathsf{pp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ *be the public parameters for Plonk. Let* $\mathsf{pp}_2 := (\ell_{lk}, f_{lk})$ *be the additional public parameters where* $\ell_{lk} = 2^{\nu_{lk}}$ *is the number of lookup selectors and* $f_{lk} : \mathbb{F}^{\ell_{lk} + \ell_w} \rightarrow \mathbb{F}$ *is an algebraic map. The indexed relation* $\mathcal{R}_{\mathsf{HyperPianist+}}$ *is the set of all triples*

$$(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((\mathbb{i}_1, \mathbb{i}_2); (p, [[w]]); w)$$

where $\mathbb{i}_2 := (\mathsf{table} \in \mathbb{F}^{n-1},\ q_{lk} \in \mathcal{F}^{\leq 1}_{\mu + \nu_{lk}})$ *such that*

- $(\mathbb{i}_1; \mathbb{x}; \mathbb{w}) \in \mathcal{R}_{\mathsf{HyperPianist}}$

PROTOCOL H.0.03 *Distributed PIOP for HyperPianist+*

Indexer. $\mathcal{I}(\mathbb{i}_1, \mathbb{i}_2 = (\mathsf{table}, q_{\mathrm{lk}}))$ calls the distributed HyperPlonk PIOP indexer $\mathsf{vp}_{\mathrm{plonk}} \leftarrow \mathcal{I}_{\mathrm{plonk}}(\mathbb{i}_1)$, and calls the distributed Lookup PIOP indexer $\mathsf{vp}_{\mathrm{t}} \leftarrow \mathcal{I}_{\mathrm{lkup}}(\mathbb{i}_2)$. The oracle output is $\mathsf{vp} = ([[q_{lk}]], \mathsf{vp}_{\mathrm{t}}, \mathsf{vp}_{\mathrm{plonk}})$.

The Protocol. $\{\mathcal{P}_i(\mathsf{pp}, \mathbb{i}, p, w^{(i)})\}_{i \in [0, M-1]}$ and $\mathcal{V}(\mathsf{pp}, p, \mathsf{vp})$ run the following protocol.

1. The master prover $\mathcal{P}_0$ sends $\mathcal{V}$ the witness oracle $[[w]]$ where $w \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_m}$.
2. $\{\mathcal{P}_i\}_{i \in [0, M-1]}$ and $\mathcal{V}$ run a distributed HyperPianist PIOP (Protocol F.1.01) for $(\mathbb{i}; \mathbb{x}; \mathbb{w}) \in \mathcal{R}_{\mathsf{HyperPianist}}$.
3. $\{\mathcal{P}_i\}_{i \in [0, M-1]}$ and $\mathcal{V}$ run a distributed Lookup PIOP (Protocol H.0.02) for $(\mathsf{table}; [[g]]) \in \mathcal{L}(\mathcal{R}_{\mathrm{Lookup}})$ where g is as defined in Definition 18.

– *there exists* $\mathsf{addr} : B_\mu \to [1, 2^\mu)$ *such that* $(\mathsf{table}; [[g]]; (g, \mathsf{addr})) \in \mathcal{R}_{Lookup}$, *where* $g \in \mathcal{F}^{(deg(f_{lk})}_\mu$ *is defined as*

$$\begin{aligned}\tilde{g}(\boldsymbol{X}) :=& f_{lk}(q_{lk}(\langle 0\rangle_{\nu_{lk}}, \boldsymbol{X}), \ldots, q_{lk}(\langle \ell_{\nu_{lk}} - 1\rangle_{\nu_{lk}}, \boldsymbol{X}),\\ & w(\langle 0\rangle_{\nu_w}, \boldsymbol{X}), \ldots, w(\langle \ell_w - 1\rangle_{\nu_w}, \boldsymbol{X})).\end{aligned}$$

I Additional Related Works

I.1 Collaborative ZKPs.

A series of recent works [29,8,11,26] 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 [29]. 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.

I.2 Lookup Argument

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 [43,32,44,15,13,22,36] 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 the one-time expensive setup, prover complexity can be made only quasi-linear to the number of queries. In the multivariate setting, Lasso [36] provides a generalized approach for proving lookup arguments on structured tables, significantly enhancing efficiency. Building on Lasso, our scheme introduces additional optimizations and further improves the prover's performance.

I.3 Optimization of SumCheck PIOP

The SumCheck PIOP underpins many fast prover SNARKs[19,40,6,42]. In many of its applications, the polynomial can be evaluated with a few operations on several multilinear polynomials. The naïve algorithm has quadratic complexity in the number of operations. HyperPlonk [6] introduced a quasi-linear algorithm that performs well for high-degree polynomial, including custom gates in Plonkish arithmetization. However, its advantage diminishes in low-degree settings, due to its heavy reliance on FFTs. Recent works [3,9,10,21] focus on optimizing the constant factor of the vanilla protocol, with quadratic scaling factor unchanged. These optimizations speed up provers for low-degree polynomials, which is beneficial for many applications (e.g., [36,2,19,37]). Notably, all these optimizations can be adapted to distributed settings with no overhead, further benefiting our distributed SNARKS.

J Proof of Previous Theorem

J.1 Proof of Theorem 1

Proof. **Completeness.** For every $([[f]]; f) \in \mathcal{R}_{\text{Zero}}$, $\hat{f}$ is also zero everywhere on the boolean hypercube, thus the summation of $\hat{f}$ over boolean hypercube is zero, and completeness follows from SumCheck protocol's completeness.
Knowledge soundness. It is suffice to argue the soundness error of the protocol. We note that $([[f]]; f) \in \mathcal{R}_{\text{Zero}}$ if and only if the following holds

$$g(\boldsymbol{Y}) = \sum_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{Y})$$

is identically zero. This is because for $x, y \in \{0,1\}^n$, $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{y})$ equals to 1 if $\boldsymbol{x} = \boldsymbol{y}$ and equals to 0 otherwise. So $g(\boldsymbol{y}) = f(\boldsymbol{y})$ for all $\boldsymbol{y} \in \{0,1\}^n$. Therefore, for any $([[f]]; f) \notin \mathcal{R}_{\text{Zero}}$, the corresponding g is a non-zero polynomials and thus

$$g(\boldsymbol{r}) = \sum_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}) = 0$$

with probability $n/|\mathbb{F}|$ over the choice of $\boldsymbol{r}$. Thus the probability that the verifier accepts is at most $n/|\mathbb{F}|$ plus the probability that the SumCheck PIOP verifier accepts when $((0, [[\hat{f}]]); \hat{f}) \notin \mathcal{R}_{\text{Sum}}$, which is $O(n/|\mathbb{F}|)$.

J.2 Proof of Theorem 4

We follow the proof from [22]. In order to prover Theorem 4, we first give some useful lemma.

Lemma 1. *Let $\mathbb{F}$ be a field of characteristic p, and $p(X)/q(X)$ a rational function over $\mathbb{F}$ with both degree being less than p. If the derivative $(\frac{p(X)}{q(X)})' = 0$, then $\frac{p(X)}{q(X)} = c$ for some constant $c \in \mathbb{F}$.*

Proof. If $q(X)$ is a constant, then the assertion of the Lemma follows from the corresponding statement for polynomials. Hence we only need to consider the non-constant $q(X)$. Use polynomial division to obtain the representation

$$\frac{p(X)}{q(X)} = m(X) + \frac{r(X)}{q(X)},$$

where $m(x), r(x) \in \mathbb{F}[X]$. By linearity of the derivative, we have

$$0 = \left(\frac{p(X)}{q(X)}\right)' = m'(X) + \left(\frac{r(X)}{q(X)}\right)',$$

and therefore

$$r'(X) \cdot q(X) - r(X) \cdot q'(X) = -m'(X) \cdot q(X)^2.$$

Comparing the degrees of left and right hand side, we conclude that $m'(X) = 0$. Since the degree of $m(X)$ is less than p, we have $m(X) = c$ for some constant $c \in \mathbb{F}$. Furthurmore, if we had $r(X) \neq 0$, tehn the leading term of the left hand side would be

$$(k - n) \cdot c_n \cdot d_k \cdot X^{n+k-1},$$

with $c_n \cdot X^n, n > 0$ being the leading term of $q(X)$, and $d_k \cdot X^k, 0 \leq k < n$, being the leading term of $r(X)$. As $0 < n - k < p$, and both $c_n \neq 0$ and $d_k \neq 0$, the leading term of the left hand side would not vanish. Therefore it must hold that $r(X) = 0$ and the proof is complete.

Proof. If $p_a(X) = \prod_{i=1}^{n}(X + a_i)$ and $p_b(X) = \prod(X + b_i)$ coincide, so do their logarithmic derivative. To show the other direction, assume that

$$\frac{p_a'(X)}{p_a(X)} = \frac{p_b'(X)}{p_b(X)}.$$

Then

$$\left(\frac{p_a(X)}{p_b(X)}\right)' = \frac{p_a'(X) \cdot p_b(X) - p_a(X) \cdot p_b'(X)}{p_b^2(X)} = 0.$$

Hence by Lemma 1, we have $p_a(X)/p_b(X) = c$ for some constant $c \in \mathbb{F}$. As both $p_a(X)$ and $p_b(X)$ have leading coefficient 1, we conclude that $c = 1$.

J.3 Proof of Theorem 5

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. It is sufficient to argue 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

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

For 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 $O(dn/|\mathbb{F}|)$. To 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 $O(dn/|\mathbb{F}|)$.

J.4 Proof of Theorem 6

Proof. **Completeness.** For any $(\sigma; ([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{Perm}}$, it holds that the multiset $(s_{id}x, f(\boldsymbol{x})_{x\in\{0,1\}^n}$ is identical to the multiset $(s_\sigma x, g(\boldsymbol{x})_{x\in\{0,1\}^n}$. Thus

$$(([[s_{id}]], [[f]], [[s_\sigma]], [[g]]); (s_{id}, f, s_\sigma, g)) \in \mathcal{R}_{\text{MSet}}$$

and completeness follows from the completeness of the PIOP for $\mathcal{R}_{\text{MSet}}$.
Knowledge soundness. It is sufficient to argue the soundness error of the protocol. Since the permutation relation holds if and only if the multiset check relation holds, the PIOP has the same soundness error as Multiset Check PIOP.

J.5 Proof of Theorem 8

Proof. The completeness and soundness follow the original Dory protocol and 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 $\boldsymbol{com_{row}}^{(i)}$ and $com_M^{(i)}$, which costs $O(\frac{N}{M})$ $\mathbb{G}_1$ operations and $O(\sqrt{\frac{N}{M}})$ parings, and the master prover products them up using $O(M)$ $\mathbb{G}_T$ 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 $O(\sqrt{\frac{N}{M}})$ $\mathbb{F}$, $\mathbb{G}_1$ operations and pairings and the master prover reconstruct the elements using $O(M)$ $\mathbb{G}_1$ and $\mathbb{G}_T$ operations. The operations cost in IPA protocol is $O(N/M)$ $\mathbb{G}_1$ and $\mathbb{G}_T$ operations for each sub-prover, and the additional operations for master prover is $O(M)$ $\mathbb{G}_1$ 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 sending 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 all $O(n + m)$.

J.6 Proof of Theorem 9

We follow the proof from [22]. The proof of Theorem 9 depends on the following lemma:

Lemma 2. *Let $\mathbb{F}$ be an arbitrary field and $m_1, m_2 : \mathbb{F} \to \mathbb{F}$ be any functions. Then $\sum_{z\in\mathbb{F}} \frac{m_1(z)}{X-z} = \sum_{z\in\mathbb{F}} \frac{m_2(z)}{X-z}$ in the rational function field $\mathbb{F}(X)$ if and only if $m_1(z) = m_2(z)$ for every $z \in \mathbb{F}$.*

Proof. Suppose that $\sum_{z\in\mathbb{F}} \frac{m_1(z)}{X-z} = \sum_{z\in\mathbb{F}} \frac{m_2(z)}{X-z}$. Then we have $\sum_{z\in\mathbb{F}} \frac{m_1(z)-m_2(z)}{X-z} = 0$, and therefore

$$\begin{aligned} p(X) &= \prod_{w\in\mathbb{F}}(X-w)\sum_{z\in\mathbb{F}}\frac{m_1(z)-m_2(z)}{X-z} \\ &= \sum_{z\in\mathbb{F}}(m_1(z)-m_2(z))\prod_{w\in\mathbb{F}\setminus\{z\}}(X-w) \\ &= 0. \end{aligned}$$

In particular, $p(z) = (m_1(z)-m_2(z)) \cdot \prod_{w\in\mathbb{F}\setminus\{z\}}(X-w) = 0$ for every $z \in \mathbb{F}$. Since $\prod_{w\in\mathbb{F}\setminus\{z\}}(z-w) \neq 0$, we must have $m_1(z) = m_2(z)$ for every $z \in \mathbb{F}$. The other direction is obvious.

With this lemma, we proceed to proof of Theorem 9

Proof. Let us denote by $m_a(z)$ be the multiplicity of a filed element z in the sequence $(a_i)_{i=1}^{\ell}$. Likewise, we do for $\{b_j\}_{j=1}^{N}$. Suppose that $\{a_i\} \subset \{b_j\}$ as sets. Set (m_i) as the normalized multiplicities $m_i = \frac{m_a(b_i)}{m_b(b_i)}$. This choice of (m_i) obviously satisfies $\sum_{i=1}^{\ell} \frac{1}{X+a_i} = \sum_{i=1}^{N} \frac{m_i}{X+b_i}$.

Conversely, suppose that $\sum_{i=1}^{\ell} \frac{1}{X+a_i} = \sum_{i=1}^{N} \frac{m_i}{X+b_i}$ holds. Collecting fraction with the same denominator we obtain fractional representations for both sides of the equation

$$\sum_{i=1}^{N} \frac{1}{X+a_i} = \sum_{z\in\mathbb{F}} \frac{m_a(z)}{X+z},$$

$$\sum_{i=1}^{N} \frac{m_i}{X+b_i} = \sum_{z\in\mathbb{F}} \frac{\mu(z)}{X+z}.$$

Note that since $p > \max(\ell, N)$, we know that for each $z \in \{a_i\}$ we have $m_a(z) \neq 0$. By the uniqueness of fractional representations (Lemma 2), $m_a(z) = \mu(z)$ for every $z \in \{a_i\}$, and therefore each $z \in \{a_i\}$ must occur also in $\{b_j\}$.