HyperPianist: Pianist with Linear-Time Prover via Fully Distributed HyperPlonk

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 face challenges with high prover costs for large-scale applications. To accelerate proof generation, Pianist (Liu et al., S&P 2024) proposes to distribute the proof generation process across multiple machines, and achieves a significant reduction in overall prover time. However, Pianist inherits the quasi-linear computational complexity from its underlying SNARK proof system Plonk, limiting its scalability and efficiency with large circuits.

In this paper, we introduce HyperPianist, a fully distributed proof system with linear-time prover complexity and logarithmic communication cost among distributed machines. Starting from deVirgo (Xie et al., CCS 2022), we study their distributed multivariate SumCheck protocol and achieve logarithmic communication cost by using an additively homomorphic multivariate polynomial commitment scheme in the distributed setting. Given the distributed SumCheck protocol, we then adapt HyperPlonk (Chen et al., EuroptCrypt 2023), a proof system based on multivariate polynomials, to the distributed setting without extra overhead for witness re-distribution. In addition, we propose a more efficient construction of lookup arguments based on Lasso (Setty et al., Eurocrypt 2024), and adapt it to the distributed setting to enhance HyperPianist and obtain HyperPianist+.

1 Introduction

Zero-knowledge proofs (ZKPs) are cryptographic protocols where one party proves that a statement is true without revealing any additional information. 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 type of ZKPs where the proof is short and fast to verify ("succinct"). One of the most popular constructions of modern SNARKs is based on a polynomial interactive oracle proof (PIOP) in conjunction with a polynomial commitment scheme (PCS). Two of the most deployed SNARKs, Plonk [12] and Marlin [4], fall into this category. Plonk stands out with its compact proof size and fast verifier, as well as support for custom gates with a universal trusted setup; it has been adopted in various blockchain-related applications such as zkRollups and zkEVM (Zero-knowledge Ethereum Virtual Machine). An extension work Plonkup [24] enhances Plonk with the lookup arguments from Plookup [11] to allow 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 when applied to large-scale real-world applications, such as complex EVM traces on blockchains and large language models (LLMs) in machine learning. As reported in a recent work Pianist [18], Plonk requires about 200GB of memory for a circuit with 2^{25} gates.

Pianist [18] proposes to distribute the proof generation process of Plonk across multiple machines (sub-provers) to accelerate the proving phase. For a data-parallel circuit containing M identical sub-circuits each of size T, Pianist is able to achieve $O(T \log T + M \log M)$ sub-prover cost using M machines, while plain Plonk with one prover requires $O(MT \log MT)$ prover cost. Besides, Pianist achieves $O(1)$ communication cost among the sub-provers, and the proof size and verification cost are the same as Plonk (which are $O(1)$ too). Furthermore, Pianist can be generalized to a "fully" distributed proof system that applies to arbitrary circuits (in addition to data-parallel ones), with only minor modifications and the same asymptotic complexity.

Pianist has shown the great potential of distributed systems in accelerating proof generation. However, its underlying proof system, Plonk, does not scale well as the statement size gets larger. As a SNARK built upon univariate PIOP and PCS, Plonk inevitably requires FFT (Fast Fourier Transform) or NTT (Number Theoretic Transform) operations for polynomial interpolation, which incurs $O(N \log N)$ computational cost for the prover. Concretely, for a statement size larger than 2^{20}, the cost of FFTs or NTTs becomes the main bottleneck of prover efficiency [3]. Although Pianist distributes the proving process across multiple sub-provers using bivariate polynomials, each sub-prover still needs to perform polynomial interpolation, and thus faces the same challenge.

Recently, HyperPlonk [3] was proposed to improve the efficiency and scalability of Plonk: by adapting the Plonk univariate PIOP to the boolean hypercube using multivariate polynomials in conjunction with a suitable multilinear PCS, HyperPlonk can achieve linear-time prover complexity and handle custom gates more efficiently. This improvement arises from the linear-time multivariate PIOP and PCS, which scale better than the quasi-linear-time FFT or NTT operations required by univariate PIOPs used in Plonk. In practice, the improvement factor can be nearly $3\times$ when statement size gets to 2^{20}, and the gap will be larger as the statement size increases.

1.1 Our Contributions

HyperPianist. In this work, we propose HyperPianist (HyperPlonk vIA uNlimited dISTribution), a fully distributed proof system with linear prover time. HyperPianist is the distributed version of HyperPlonk, similar to how Pianist is the distributed version of Plonk. Our goal is to enhance the scalability and efficiency of Pianist by adapting the linear-time prover of HyperPlonk to the distributed setting. As a result, HyperPianist is able to achieve linear-time complexity for each sub-prover as well as logarithmic communication cost among the distributed sub-provers.

We observe that the constraints in HyperPlonk (i.e., gate identity and wiring identity) are eventually reduced to a multivariate SumCheck. An existing work, deVirgo [32], has already explored the SumCheck protocol in the distributed setting. However, naïvely applying their distributed SumCheck protocol to HyperPlonk would incur linear communication costs among the sub-provers and extra overhead for witness re-distribution. This mainly results from the following two issues.

First, the original distributed SumCheck protocol in deVirgo incurs linear communication cost among the distributed sub-provers. We note that the linear communication cost comes from the use of the FRI-based multivariate PCS: in the distributed setting, it requires exchanging extra witness data among the distributed machines for constructing Merkle proofs. Instead, we propose to make use of an additively horomorphic multivariate PCS, such as the IPA-based scheme Dory [17]. The additive homomorphic property enables aggregation of partial commitments to local witness held by the sub-provers, and thus enables logarithmic communication cost for each sub-prover. We show the comparison of our distributed PCS with the ones from deVirgo [32] and Pianist [18] in Table 1.

Second, in HyperPlonk, the wiring constraints (i.e., copy constraints) are reduced to a multiset check, which is then transformed into a grand product check and handled by the Quarks [27] PIOP system. The grand product PIOP from Quarks requires a helper polynomial, whose witness can not be locally obtained by the distributed sub-provers in the distributed setting. To compute witnesses of this polynomial, the sub-provers need extra linear communication cost to exchange intermediate data. To address this issue, we propose two solutions: (1) using a logarithmic derivatives-based PIOP for the multiset check, which naturally fits the distributed setting and has $O(\log n)$ proof size, and (2) using layered circuits to directly prove the multiset relation, with $O(\log^2 n)$ proof size but lower prover cost. Both solutions require no extra effort for witness re-distribution.

Table 1: Comparisons of our distributed PCS with the ones from deVirgo and Pianist. (N denotes the circuit size, M denotes the number of distributed machines and $T = N/M$ denotes the number of witnesses each distributed machine holds. "Communication" measures the communication cost among the distributed machines, and "Proof Size" measures the communication cost between the master sub-prover and the verifier. $\mathbb{H}, \mathbb{P}, \mathbb{F}$ and $\mathbb{G}, \mathbb{G}_T$ stand for hash, field, pairing, group operations respectively. $|\cdot|$ represents the size of corresponding elements. "Transparent" stands for transparent setup.)

PCS	Trans-parent	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Communication	Proof Size
deVirgo	✓	$O(T \log T)\,\mathbb{F} + O(T)\,\mathbb{H}$	$O(\log^2 N)\,\mathbb{H}$	$O(N)\,\lvert\mathbb{F}\rvert$	$O(\log^2 N)\,\lvert\mathbb{H}\rvert$
Pianist	✗	$O(T)\,\mathbb{G} + O(T)\mathbb{F}$	$O(1)\,\mathbb{P}$	$O(M)\,\lvert\mathbb{G}\rvert$	$O(1)\,\lvert\mathbb{G}\rvert$
Ours	✓	$O(T)\,\mathbb{G}$	$O(\log N)\,\mathbb{G}_T$	$O(M \log N)\,\lvert\mathbb{G}\rvert$	$O(\log N)\,\lvert\mathbb{G}_T\rvert$

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

Scheme	$\mathcal{P}_i$ Time	$\mathcal{V}$ Time	Communication	Proof Size
deVirgo	$O(T \log T)\,\mathbb{F}$ $+ O(T)\,\mathbb{H}$	$O(\log^2 N)\,\mathbb{H}$	$O(N)\,\lvert\mathbb{F}\rvert$	$O(\log^2 N)\,\lvert\mathbb{H}\rvert$
Pianist	$O(T \log T)\,\mathbb{G}$ $+ O(T \log T)\,\mathbb{F}$	$O(1)\,\mathbb{P}$	$O(M)\,\lvert\mathbb{G}\rvert$	$O(1)\,\lvert\mathbb{G}\rvert$
Ours	$O(T)\,\mathbb{G}$ $+ O(T)\,\mathbb{F}$	$O(\log N)\,\mathbb{G}_T$ $+ O(\log N)\,\mathbb{F}$	$O(M \log N)\,\lvert\mathbb{G}\rvert$ $+ O(M \log N)\,\lvert\mathbb{F}\rvert$	$O(\log N)\,\lvert\mathbb{G}_T\rvert$ $+ O(\log N)\,\lvert\mathbb{F}\rvert$

Combining the distributed SumCheck protocol and the distributed Hyper-Plonk PIOP system, we can obtain our fully distributed zero-knowledge proof system HyperPianist. We compare HyperPianist with deVrigo and Pianist in Table 2.

HyperPianist+. Our second contribution is HyperPianist+, an enhancement of HyperPianist with an optimized distributed proof system for lookup tables, similar to how HyperPlonk+ enhances HyperPlonk with lookup arguments. We propose an optimized construction of lookup arguments based on the state-of-the-art lookup argument Lasso [28], and extend it to the distributed setting.

Lookup arguments allow a party to prove that every element in a committed vector exists in a pre-determined table. Compared to conventional ZKP systems for arithmetic constraints, lookup arguments are especially suitable for non-linear functions like bitwise operations, range proofs, and even finite state machines. To build a distributed lookup argument compatible with HyperPianist, we studied Lasso [28], a multivariate proof system also built over the boolean hypercube. We found that one of its key components — the well-formation check based on Memory-in-the-Head techniques, is not optimal. We propose to use logarithmic derivative techniques from Logup [16] for the well-formation check and can immediately save 50% polynomial commitments.

We note a concurrent work [7] that 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 and get a more functional distributed ZKP system.

Implementation and Experiments. We have implemented our optimized lookup arguments using the Lasso framework[1] and conducted some preliminary experiments. Results show that our optimized lookup protocol is $1.41 \sim 1.89$ times as fast in proving 64-bit XOR computations compared to Lasso. Our fully distributed zero-knowledge proof system is still in progress, and we expect more efficiency gains in the future.

1.2 Related Works

Distributed ZKPs. Wu et al. introduced DIZK [30], the first distributed zero-knowledge proof protocol. However, DIZK incurs communication costs proportional to the circuit size due to the use of a distributed FFT algorithm. zkBridge [32] presented deVirgo, a distributed variant of Virgo [35], which achieves linear prover time but also suffers from linear communication costs due to the FRI protocol [2]. To mitigate the high communication overhead, Pianist [18] proposed a bivariate KZG commitment scheme to aggregate proofs from different machines, achieving optimal linear scalability in prover time and minimal communication among distributed machines. However, the prover complexity in Pianist remains quasi-linear to the circuit size. In contrast, our schemes provide strictly linear prover time with only logarithmic communication per machine.

Collaborative ZKPs. A series of recent works [22,5,6,19] have focused on distributing the proof generation process while maintaining the privacy of the witnesses. One popular approach relies on the notation of collaborative ZKPs introduced in [22]. This approach consists of two phases: First, each server sends and receives its part of the witness in a secret-sharing form. Then, all servers execute a certain secure multi-party computation (MPC) protocol for the proof generation circuit. We stress that these works are orthogonal to ours: their emphasis is on security and privacy, while we focus on scaling proof generation with sub-provers which trust each other.

1.3 Organization of the paper.

Section 2 presents the preliminaries. Section 3 introduces an optimized distributed SumCheck protocol as a main building block of HyperPianist, and then presents our construction of HyperPianist by adapting the HyperPlonk multivariate PIOP to the distributed setting. Section 5 presents the construction of

[1] Our implementation is in https://github.com/zhaowenlan1779/jolt/tree/logup.

our optimized lookup argument, and uses it to enhance HyperPianist to obtain HyperPianist+. Section 6 gives some preliminary experimental results.

2 Preliminaries

2.1 Notations

We use λ to denote the security parameter. For $n \in \mathbb{N}$, let $[n]$ be the set $\{1, 2, \ldots, n\}$; for $a, b \in \mathbb{N}$, let $[a, b)$ denote the set $\{a, a+1, \ldots, b-1\}$. 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**.

Vector, matrix and tensor indices will begin at 1. 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 $\boldsymbol{v_R}$.

We write $\leftarrow_\$ S$ for uniformly random samples of a set S, with the understanding that this encodes no additional structure.

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 $\mathbb{G}_1^n \times \mathbb{G}_2^n \to \mathbb{G}_T$.

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.

2.2 SNARKs

We adopt the definition of SNAKRs provided in [3].

Definition 1 (Interactive Argument of Knowledge). *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.*

It is well known that if the interactive argument of knowledge protocol is public-coin, then it can be made non-interactive by the Fiat-Shamir transformation [9]. 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 [3]). *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 $f \in \mathcal{F}_n^{(\le 1)}$. 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 (Polynomial Commitment Scheme (PCS)). *A polynomial commitment scheme Γ is a tuple $\Gamma = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ of PPT algorithms where:*

- $\mathsf{Gen}(1^\lambda, \mathcal{F}) \to \mathsf{pp}$ *generates public parameters* pp*;*
- $\mathsf{Commit}(\mathsf{pp}, f) \to \mathsf{com}_f$ *takes a secret polynomial $f(\boldsymbol{X})$ and outputs a public commitment* com_f*;*
- $\mathsf{Open}(\mathsf{pp}, \mathsf{com}_f, \boldsymbol{x}) \to (z, \pi_f)$ *evaluates the polynomial $y = f(\boldsymbol{x})$ at a point $\boldsymbol{x}$ and generates a proof π_f;*
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{com}_f, \boldsymbol{x}, z, \pi_f) \to b \in \{0,1\}$ *is an interactive protocol between the prover $\mathcal{P}$ and verifier $\mathcal{V}$, convincing the verifier that $f(\boldsymbol{x}) = z$.*

A commitment scheme Γ is ***binding*** *if for all PPT adversaries $\mathcal{A}$,*

$$\Pr\left[b_0 = b_1 \neq 0 \wedge x_0 \neq x_1 \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (C, x_0, x_1, r_0, r_1) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, C, x_0, r_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, C, x_1, r_1) \end{array} \right] \leq \mathsf{negl}(\lambda).$$

A commitment scheme Γ is ***hiding*** *if for all PPT adversaries $\mathcal{A}$,*

$$\left| \Pr\left[b = b' \;\middle|\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (x_0, x_1, st) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ b \leftarrow_\$ \{0,1\} \\ (C_b; r_b) \leftarrow \mathsf{Commit}(\mathsf{pp}; x_b) \\ b' \leftarrow \mathcal{A}(\mathsf{pp}, st, C_b) \end{array} \right] - 1/2 \right| \leq \mathsf{negl}(\lambda).$$

Our distributed polynomial commitment scheme is built upon Dory [17], 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 4 (Pedersen Commitment).

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (g \leftarrow_\$ \mathbb{G}_i^n, h \leftarrow_\$ \mathbb{G}_i)$
- $(\mathcal{C}, \mathcal{S}) \leftarrow \mathsf{Commit}(\mathsf{pp}; x) = \{r \leftarrow_\$ \mathbb{F}; (\langle x, g \rangle) + rh, r)\}$
- $\mathsf{Open}(\mathsf{pp}; \mathcal{C}, x, \mathcal{S})$: *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 5 (AFGHO Commitment [1]).

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

3 Distributed SumCheck Protocol with Sublinear Communication Cost

In this section, we present our key building block of HyperPianist, an optimized distributed SumCheck protocol with $O(\log n)$ communication cost per distributed machine for a SumCheck statement of size n. A previous work, deVirgo [32], has constructed a SumCheck PIOP in the distributed setting, but their construction is for data-parallel circuits and suffers from linear communication costs. Below we first review the distributed SumCheck PIOP from deVirgo [32], and then show how to reduce the communication cost down to $O(\log n)$ with an optimized distributed PCS.

3.1 Review: Distributed SumCheck PIOP From deVirgo

The SumCheck relation is defined as follows:

Definition 6 (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 $\sum_{\boldsymbol{x}\in\{0,1\}^n} f(\boldsymbol{x}) = v$.*

There is a well-known PIOP for SumCheck, as illustrated in Protocol 3.1.01, where the verifier runs in $O(n)$ time plus the time required to evaluate f at a single point $\boldsymbol{r} \in \mathbb{F}^n$. This is significantly faster than naïvely evaluating v on the hypercube.

Theorem 1. *The PIOP for $\mathcal{R}_{Sum}$ in Protocol 3.1.01 is perfectly complete and has knowledge error $dn/|\mathbb{F}|$.*

In deVirgo [32], the authors have explored the aggregation of multiple SumCheck instances for data-parallel circuits held by several distributed machines. Here we instead distribute a single SumCheck across multiple machines, assuming the initial witnesses have already been distributed. We present the protocol in Protocol 3.1.02.

Without loss of generality, suppose we have $M = 2^m$ distributed machines acting as sub-provers. Initially, we assume that the i-th sub-prover holds witnesses to indices $(x_1, \ldots, x_{n-m}, \boldsymbol{bin(i)})$ where $x_1, \ldots, x_{n-m} \in \{0, 1\}$. Following deVirgo [32], initially we can define

$$f^{(i)}(\boldsymbol{x}) := f(\boldsymbol{x}, \boldsymbol{bin(i)})$$

where $\boldsymbol{bin(i)}$ is the binary representation of the value i. Then we have

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

PROTOCOL 3.1.01 *SumCheck PIOP.*

$\mathcal{P}$ holds a multivariate polynomial $f : \mathbb{F}^n \to \mathbb{F}$, and wants to convince $\mathcal{V}$ that $v = \sum_{\boldsymbol{x} \in \{0,1\}^n} f(\boldsymbol{x})$.

– In the first round, $\mathcal{P}$ sends a univariate polynomial

$$f_1(X_1) := \sum_{\boldsymbol{x} \in \{0,1\}^{n-1}} f(X_1, \boldsymbol{x}).$$

$\mathcal{V}$ 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

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

$\mathcal{V}$ checks $f_{k-1}(r_{k-1}) = f_k(0) + f_k(1)$. If the check passed, $\mathcal{V}$ sends a random challenge $r_k \in \mathbb{F}$ to $\mathcal{P}$.

– In the n-th round, $\mathcal{P}$ sends a univariate polynomial

$$f_n(X_n) := f(r_1, \ldots, r_{n-1}, X_n).$$

$\mathcal{V}$ checks $f_{n-1}(r_{n-1}) = f_n(0) + f_n(1)$ and generates a random challenge $r_n \in \mathbb{F}$. $\mathcal{V}$ accepts if an only if $f_n(r_n) = f(r_1, \ldots, r_n)$ using one oracle call to $[[f]]$.

Note that, given witnesses to indices $(x_1, \ldots, x_{n-m}, \boldsymbol{bin}(i))$, each sub-prover $\mathcal{P}_i$ can locally obtain the polynomial $f^{(i)}(\boldsymbol{x})$. In the first $n-m$ rounds, each sub-prover can locally compute the univariate polynomial required by the SumCheck PIOP, and send it to a master sub-prover $\mathcal{P}_0$. $\mathcal{P}_0$ aggregates all these polynomials to obtain the polynomial to send to $\mathcal{V}$, and forwards the challenge from $\mathcal{V}$ to all sub-provers. By allocating each sub-prover a fixed binary suffix of length m, the computation of the univariate polynomial in the first $n-m$ rounds of the SumCheck PIOP can be distributed evenly among the sub-provers and then summed up via aggregation by a master sub-prover $\mathcal{P}_0$. More specifically, we let each sub-prover $\mathcal{P}_i$ locally compute the univariate polynomial $f_k^{(i)}(X_k)$ using its partial polynomial $f^{(i)}(\boldsymbol{x})$ with the fixed binary suffix $\boldsymbol{bin(i)}$ in each k-th round of the SumCheck PIOP where $1 \leq k \leq n-m$, and send it to a master sub-prover $\mathcal{P}_0$. $\mathcal{P}_0$ aggregates the univariate polynomials and obtains $f_k(X_k) = \sum_{\boldsymbol{bin(i)} \in \{0,1\}^m} f_k^{(i)}(X_k)$. After this, $\mathcal{P}_0$ acts as the prover in the normal SumCheck PIOP, i.e., interacting with $\mathcal{V}$ by sending $f_k(X_k)$ and receiving a random challenge r_k. Then, it transmits the random challenge to other sub-provers.

PROTOCOL 3.1.02 *Distributed SumCheck PIOP.*

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

- In the first round:
  - Each $\mathcal{P}_i$ computes its own univariate polynomial
  $$f_1^{(i)}(X_1) := \sum_{\boldsymbol{x}\in\{0,1\}^{n-m-1}} f(X_1, \boldsymbol{x}, \boldsymbol{bin(i)}),$$
  and sends it to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_1(X_1) = \sum_{i\in[0,M-1]} f_1^{(i)}(X_1)$, 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 sub-provers.
- In the k-th round where $2 \le k \le n - m$:
  - Each $\mathcal{P}_i$ computes its own univariate polynomial
  $$f_k^{(i)}(X_k) := \sum_{\boldsymbol{x}\in\{0,1\}^{n-m-k}} f(r_1, \ldots, r_{k-1}, X_k, \boldsymbol{x}, \boldsymbol{bin(i)})$$
  and sends it to $\mathcal{P}_0$.
  - $\mathcal{P}_0$ sums up all the univariate polynomials to get $f_k(X_k) = \sum_{i\in[0,M-1]} f_k^{(i)}(X_k)$, 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 sub-provers.
- After the $(n-m)$-th round, each $\mathcal{P}_i$ sends $f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$. $\mathcal{P}_0$ then computes $v' := \sum_{i\in[0,M-1]} f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$, and constructs the multivariate polynomial $g(\boldsymbol{x}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{x})$.
- In the final m rounds, $\mathcal{P}_0$ and $\mathcal{V}$ run the SumCheck PIOP (Protocol 3.1.01) on the statement
$$v' = \sum_{\boldsymbol{x}\in\{0,1\}^m} g(\boldsymbol{x}).$$

In the $(n-m)$-th round, after $\mathcal{V}$ returns the random challenge r_{n-m}, the multivariate polynomial to check has been reduced to $g(\boldsymbol{x}) := f(r_1, \cdots, r_{n-m}, \boldsymbol{x})$. At this point, all sub-provers send $f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$ to $\mathcal{P}_0$. $\mathcal{P}_0$ 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 $g(\boldsymbol{x})$ s.t. $g(\boldsymbol{bin(i)}) = f(r_1, \cdots, r_{n-m}, \boldsymbol{bin(i)})$. Now in the following rounds, the master sub-prover $\mathcal{P}_0$ acts as the single prover in a regular SumCheck PIOP to prove that $\sum_{\boldsymbol{x}\in\{0,1\}^m} g(\boldsymbol{x}) = v'$.

3.2 Instantiation: Using Additively Homomorphic PCS

In the above distributed SumCheck PIOP, the verifier needs an oracle call to the polynomial f to get $f(r_1, \cdots, r_n)$. To instantiate this oracle call, we need a multivariate PCS suitable for the distributed setting.

In deVirgo [32], this multivariate PCS is obtained by adapting the FRI-based scheme to the distributed setting. Although deVrigo has optimized its distributed PCS algorithm by aggregating multiple commitments and proofs into one instance to reduce the proof size, it incurs linear communication costs for each sub-prover, as the FRI-based scheme requires exchanging data among the distributed sub-provers to construct a single Merkle proof.

To circumvent this issue, we consider adapting Dory [17], an IPA-based additively homomorphic PCS, to the distributed setting. The key insight here is that the additive homomorphism property enables each sub-prover to perform sub-computation locally and subsequently transmit only a constant number of elements to the master sub-prover for aggregation, which significantly reduces communication costs.

Review of Dory. In Section 2.5, we have introduced some building blocks used in Dory [17]. Below we first review the main technique in Dory, and then show how to adapt it to the distributed setting.

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 7 (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 $M = (M_{ij})$, where $M_{ij} = f(x_1, \ldots, x_n)$ for any $(x_1, \ldots, x_n) \in \{0,1\}^n$ and*

$$i = \sum_{t=1}^{k} 2^{k-t} \cdot x_t, \quad j = \sum_{t=k+1}^{n} 2^{n-t} \cdot x_t.$$

Dory Commitment. In order to generate polynomial commitments using the matrix representation, Dory [17] proposed to use a two-tiered homomorphic commitment [15] by combining the Pedersen and AFGHO commitments. Formally, for $M_{ij} \in \mathbb{F}^{n\times m}$, we have

– $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) = (\mathit{\Gamma}_1 \leftarrow_\$ \mathbb{G}_1^m, \mathit{\Gamma}_2 \leftarrow_\$ \mathbb{G}_2^n)$;

–
$$(\mathcal{C}, \mathcal{S}) \leftarrow \mathsf{Commit}(\mathsf{pp}; M_{ij}) = \begin{Bmatrix} V_i \leftarrow \mathsf{Commit}_{Pedersen}(\mathit{\Gamma}_1; M_{ij}); \\ C \leftarrow \mathsf{Commit}_{AFGHO}(\mathit{\Gamma}_2; \boldsymbol{V}); \end{Bmatrix};$$

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

Dory Evaluation Proof. Given the two-tiered commitment com_f to the polynomial f, the prover $\mathcal{P}$ would like to convince the verifier that f evaluates to y at some random point $\boldsymbol{r} \in \mathbb{F}^n$. The evaluation of f at some point $(r_1, \ldots, r_n) \in \mathbb{F}^n$ can be written as vector-matrix-vector products using its matrix representation. More specifically, we have:

$$f(r_1, \ldots, r_n) = (\otimes_{i<k}\boldsymbol{v_i})^T M(\otimes_{i\geq k}\boldsymbol{v_i}), \tag{1}$$

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

Definition 8. *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_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 $((\boldsymbol{L}, \boldsymbol{R}, com_M, com_y); (\boldsymbol{M}, y))$.*

Thus, 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

$$((\otimes_{i<k}\boldsymbol{v_i}, \otimes_{i\geq k}\boldsymbol{v_i}, com_M, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\text{VMV}}.$$

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 $\mathsf{Commit}_{Pedersen}(\Gamma_{1,fin}; y) = (y; com_y)$. $\mathcal{P}$ 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

$$com_v := \langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle = \mathsf{Commit}_{Pedersen}(\Gamma_1, \boldsymbol{v})$$

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_M is a commitment to $\boldsymbol{com_{row}} \in \mathbb{G}_1^n$. So to prove $((\boldsymbol{L}, \boldsymbol{R}, com_M, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\text{VMV}}$, it suffices to prove knowledge of $\boldsymbol{com_{row}} \in \mathbb{G}_1^n, \boldsymbol{v} \in \mathbb{F}^n$ such that $com_M = \langle \boldsymbol{com_{row}}, \Gamma_2\rangle$, $\langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle = \langle \boldsymbol{v}, \Gamma_1\rangle$ and $com_y = \langle \boldsymbol{v}, \boldsymbol{R}\rangle\Gamma_{1,fin}$, which are inner-product relations.

In this way, a vector-matrix-vector relation is reduced to inner-product relations. More formally, we define the inner-product relation as follows.

Definition 9. *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*

$$\begin{aligned} &D_1 = \langle \boldsymbol{v_1}, \Gamma_2\rangle, \quad D_2 = \langle \Gamma_1, \boldsymbol{v_2}\rangle, \\ &E_1 = \langle \boldsymbol{v_1}, \boldsymbol{s_2}\rangle, \quad E_2 = \langle \boldsymbol{s_1}, \boldsymbol{v_2}\rangle, \\ &C = \langle \boldsymbol{v_1}, \boldsymbol{v_2}\rangle. \end{aligned}$$

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 A, Protocol A.0.01.

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

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

For polynomial evaluation proof, $\boldsymbol{s_1}, \boldsymbol{s_2}$ have special multiplicative structure, and we have the identity:

$$\langle \otimes_{i=0}^{n-1}(l_i, r_i), \otimes_{i=0}^{n-1}(a_i, 1) \rangle = \prod_{i=0}^{N-1}(l_i a_i + r_i)$$

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

Combining the Protocol A.0.01 and the Protocol A.0.02, we obtain the protocol to prove the following 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 present this inner-product protocol in Protocol A.0.03.

Given the above sub-protocols, we describe the final protocol to prove the relation $((\boldsymbol{L}, \boldsymbol{R}, com_M, com_y); (\boldsymbol{M}, y)) \in \mathcal{R}_{\text{VMV}}$ in Protocol A.0.04.

Remark 1. Protocol A.0.04 only satisfies the weaker notation called *Random Evaluation Knowledge Soundness*, instead of *Knowledge Soundness*. To address this issue, Dory [17] suggested that if the prover can open the polynomial at a random challenge point using Protocol A.0.04, then we can achieve *Knowledge Soundness*. The final protocol for polynomial opening proof is described at Protocol A.0.05.

Adapting Dory to the Distributed Setting.

Distributed Dory Commitment. To adapt Dory into the distributed setting, we need some minor changes. Recall that in the distributed setting, the i-th sub-prover holds witnesses to indices $(x_1, \ldots, x_{n-m}, \boldsymbol{bin}(i))$ where $x_1, \ldots, x_{n-m} \in \{0, 1\}$ and $\boldsymbol{bin}(i)$ is the binary representation of the value i. In the vanilla matrix representation from Dory, each sub-prover ends up holding several different columns of the matrix, but will need to calculate the Pedersen commitments for the rows. Thus we use the transpose of the vanilla representation, i.e.:

$$f(x_1, \ldots, x_n) = (\otimes_{i \geq k} \boldsymbol{v_i})^T M^T (\otimes_{i<k} \boldsymbol{v_i}).$$

Then, each sub-prover can calculate the Pedersen commitments of the columns locally and send these commitments to the master sub-prover for aggregation, as in Protocol 3.2.01.

PROTOCOL 3.2.01 *Distributed-Dory-Commit*(f)

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Given a multivariate polynomial $f : \mathbb{F}^n \to \mathbb{F}$, suppose each sub-prover $\mathcal{P}_k$ holds a local partial polynomial $f^{(k)} : \mathbb{F}^{n-m} \to \mathbb{F}$ s.t. $f^{(k)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(k)})$.

– Each sub-prover $\mathcal{P}_k$ obtains the matrix representation of its local witnesses $\boldsymbol{M}_{ij}$ as in Equation 1, computes

$$\begin{aligned} \boldsymbol{com_{col}}^{(k)} &= \mathsf{Commit}_{Pedersen}(\Gamma_1^{(k)}; \boldsymbol{M}_{ij}), \\ com_M^{(k)} &= \mathsf{Commit}_{AFGHO}(\Gamma_2^{(k)}; \boldsymbol{com_{col}}^{(k)}), \end{aligned}$$

and sends $com_M^{(k)}$ to $\mathcal{P}_0$.

– The master sub-prover $\mathcal{P}_0$ computes commitment

$$com_M = \sum_{k \in [0, M-1]} com_M^{(k)}.$$

Distributed Dory Evaluation Proof. Since we use the transpose, the sub-provers in the distributed setting should instead prove that

$$((\otimes_{i \geq k} \boldsymbol{v_i}, \otimes_{i<k} \boldsymbol{v_i}, com_{\boldsymbol{M}^T}, com_y); (\boldsymbol{M^T}, y)) \in \mathcal{R}_{\text{VMV}}.$$

At a very high level, the above vector-matrix-vector relation can also be reduced to an inner-product relation, which is then handled by a reduction process and finally a fold-scalar proof. We note that the above protocols in Dory can be easily distributed across several sub-provers, since almost all calculations in this reduction are inner products, and each sub-prover can use their own partial sub-vector to compute their corresponding partial result.

We present the distributed protocols for polynomial evaluation proofs in Protocol 3.2.05. It needs two invocations of Protocol 3.2.04 to prove two distributed inner-product relations. Protocol 3.2.03 shows our distributed protocol for proving inner-product relations, which mainly involves a distributed reduction process described in Protocol 3.2.02. It is worth noting that, after the first $n - m$ rounds in the reduction process, the master sub-prover $\mathcal{P}_0$ needs to aggregate all partial data the other sub-provers hold, and then finishes the remaining reduction process with the verifier. The batch algorithm of Dory can also be distributed similarly.

PROTOCOL 3.2.02 *Distributed-Dory-Reduce*$_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Each $\mathcal{P}_i$ holds local partial 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 jointly pre-compute $\Delta_{1L} = \langle \Gamma_{1L}, \Gamma_2' \rangle$, $\Delta_{1R} = \langle \Gamma_{1R}, \Gamma_2' \rangle$, $\Delta_{2L} = \langle \Gamma_1', \Gamma_{2L} \rangle$, $\Delta_{2R} = \langle \Gamma_1', \Gamma_{2R} \rangle$, and $\chi = \langle \Gamma_1, \Gamma_2 \rangle$.

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

$$D_{1L}^{(i)} = \langle \boldsymbol{v_{1L}}^{(i)}, \Gamma_2'^{(i)} \rangle, \quad D_{1R}^{(i)} = \langle \boldsymbol{v_{1R}}^{(i)}, \Gamma_2'^{(i)} \rangle, \quad E_{1\beta}^{(i)} = \langle \Gamma_1^{(i)}, \boldsymbol{s_2}^{(i)} \rangle,$$
$$D_{2L}^{(i)} = \langle \Gamma_1'^{(i)}, \boldsymbol{v_{2L}}^{(i)} \rangle, \quad D_{2R}^{(i)} = \langle \Gamma_1'^{(i)}, \boldsymbol{v_{2R}}^{(i)} \rangle, \quad E_{2\beta}^{(i)} = \langle \boldsymbol{s_1}^{(i)}, \Gamma_2^{(i)} \rangle,$$

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

$$D_{1L} = \sum D_{1L}^{(i)}, \quad D_{1R} = \sum D_{1R}^{(i)}, \quad E_{1\beta} = \sum E_{1\beta}^{(i)},$$
$$D_{2L} = \sum D_{2L}^{(i)}, \quad D_{2R} = \sum D_{2R}^{(i)}, \quad E_{2\beta} = \sum E_{1\beta}^{(i)},$$

  and sends it to the verifier $\mathcal{V}$.
- $\mathcal{V}$ samples $\beta \leftarrow_{\$} \mathbb{F}$ and sends it to the master sub-prover $\mathcal{P}_0$.
- The master sub-prover $\mathcal{P}_0$ transmits β to the other sub-provers $\mathcal{P}_i$.
- Each sub-prover $\mathcal{P}_i$ sets

$$\boldsymbol{v_1^{(i)}} \leftarrow \boldsymbol{v_1^{(i)}} + \beta \Gamma_1^{(i)}, \quad \boldsymbol{v_2^{(i)}} \leftarrow \boldsymbol{v_2^{(i)}} + \beta^{-1} \Gamma_2^{(i)}.$$

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

$$E_{1+}^{(i)} = \langle \boldsymbol{v_{1L}}^{(i)}, \boldsymbol{s_{2R}}^{(i)} \rangle, \quad E_{1-}^{(i)} = \langle \boldsymbol{v_{1R}}^{(i)}, \boldsymbol{s_{2L}}^{(i)} \rangle, \quad C_+^{(i)} = \langle \boldsymbol{v_{1L}}^{(i)}, \boldsymbol{v_{2R}}^{(i)} \rangle,$$
$$E_{2+}^{(i)} = \langle \boldsymbol{s_{1L}}^{(i)}, \boldsymbol{v_{2R}}^{(i)} \rangle, \quad E_{2-}^{(i)} = \langle \boldsymbol{s_{1R}}^{(i)}, \boldsymbol{v_{2L}}^{(i)} \rangle, \quad C_-^{(i)} = \langle \boldsymbol{v_{1R}}^{(i)}, \boldsymbol{v_{2L}}^{(i)} \rangle,$$

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

$$E_{1+} = \sum E_{1+}^{(i)}, \quad E_{1-} = \sum E_{1-}^{(i)}, \quad C_+ = \sum C_+^{(i)},$$
$$E_{2+} = \sum E_{2+}^{(i)}, \quad E_{2-} = \sum E_{2-}^{(i)}, \quad C_- = \sum C_-^{(i)},$$

  and sends them to the verifier $\mathcal{V}$.
- $\mathcal{V}$ samples $\alpha \leftarrow_{\$} \mathbb{F}$ and sends it to the master sub-prover $\mathcal{P}_0$.
- The master sub-prover $\mathcal{P}_0$ transmits α to the other sub-provers $\mathcal{P}_i$.
- Each sub-prover $\mathcal{P}_i$ sets

$$\boldsymbol{v_1'}^{(i)} \leftarrow \alpha \boldsymbol{v_{1L}}^{(i)} + \boldsymbol{v_{1R}}^{(i)}, \quad \boldsymbol{v_2'}^{(i)} \leftarrow \alpha^{-1} \boldsymbol{v_{1L}}^{(i)} + \boldsymbol{v_{1R}}^{(i)}.$$

- The verifier $\mathcal{V}$ computes

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

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

$$\boldsymbol{s_1}'^{(i)} \leftarrow \alpha \boldsymbol{s_{1L}}^{(i)} + \boldsymbol{s_{1R}}^{(i)}, \quad \boldsymbol{s_2}'^{(i)} \leftarrow \alpha^{-1} \boldsymbol{s_{2L}}^{(i)} + \boldsymbol{s_{2R}}^{(i)}.$$

- $\mathcal{V}$ 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 3.2.03 *Distribued-Dory-IPA*$_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Each $\mathcal{P}_i$ holds local partial 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 for all $j \in \{0, \dots, n-1\}$, the sub-provers jointly pre-compute $\Gamma_{1,j+1} = (\Gamma_{1,j})_L,\ \Gamma_{2,j+1} = (\Gamma_{2,j})_L$, for all $i \in \{0, \dots, n\}$ compute $\chi_i = \langle \Gamma_{1,i}, \Gamma_{2,i} \rangle$, and for all $i \in \{0, \dots, n-1\}$ compute

$$\Delta_{1L,i} = \langle (\Gamma_{1,i})_L, \Gamma_{2,i+1} \rangle, \quad \Delta_{2L,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_L \rangle,$$
$$\Delta_{1R,i} = \langle (\Gamma_{1,i})_R, \Gamma_{2,i+1} \rangle, \quad \Delta_{2R,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_R \rangle.$$

- For $j = 0, \dots, n-m-1$, $\{\mathcal{P}_i\}_{i \in [0,M-1]}, \mathcal{V}$ run

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

- Each sub-prover $\mathcal{P}_i$ 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 the master sub-prover $\mathcal{P}_0$.
- The master sub-prover $\mathcal{P}_0$ computes

  $$C = \sum_{i \in [0,M-1]} C^{(i)},$$
  $$D_1 = \sum_{i \in [0,M-1]} D_1^{(i)}, \quad D_2 = \sum_{i \in [0,M-1]} D_2^{(i)},$$
  $$E_1 = \sum_{i \in [0,M-1]} E_1^{(i)}, \quad E_2 = \sum E_2^{(i)}.$$

- The master sub-prover $\mathcal{P}_0$ sets

  $$\boldsymbol{v_1} = \left(v_1^{(i)}\right)_{i \in [0,M-1]}, \quad \boldsymbol{v_2} = \left(v_2^{(i)}\right)_{i \in [0,M-1]},$$
  $$\boldsymbol{s_1} = \left(s_1^{(i)}\right)_{i \in [0,M-1]}, \quad \boldsymbol{s_1} = \left(s_2^{(i)}\right)_{i \in [0,M-1]}.$$

- For $j = n-m, \dots, n-1$, the master sub-prover $\mathcal{P}_0$ and $\mathcal{V}$ run

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

- The master sub-prover $\mathcal{P}_0$ and $\mathcal{V}$ run

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

PROTOCOL 3.2.04 *Distributed-Dory-Eval-RE*$(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-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.

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

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

- Each sub-prover $\mathcal{P}_i$ computes
$$\boldsymbol{v}^{(i)} = {\boldsymbol{L}^{(i)}}^T \boldsymbol{M}^{(i)}, \quad y^{(i)} = \langle \boldsymbol{v}^{(i)}, \boldsymbol{R}^{(i)} \rangle,$$
and sends $y^{(i)}$ to the master sub-prover $\mathcal{P}_0$.
- The master sub-prover $\mathcal{P}_0$ computes
$$y = \sum_{i\in[0,M-1]} y^{(i)},$$
and sends it to the verifier $\mathcal{V}$.
- Each sub-prover $\mathcal{P}_i$ computes
$$\begin{aligned} C^{(i)} &= e(\langle \boldsymbol{v}^{(i)}, \boldsymbol{com_{col}}^{(i)} \rangle, \Gamma_{2,fin}), & D_2^{(i)} &= e(\langle \Gamma_1^{(i)}, \boldsymbol{v}^{(i)} \rangle, \Gamma_{2,fin}), \\ E_1^{(i)} &= \langle \boldsymbol{L}^{(i)}, \boldsymbol{com_{col}}^{(i)} \rangle, & E_2^{(i)} &= y^{(i)} \Gamma_{2,fin}, \end{aligned}$$
and sends them to the master sub-prover $\mathcal{P}_0$.
- The master sub-prover $\mathcal{P}_0$ computes
$$\begin{aligned} C &= \sum_{i\in[0,M-1]} C^{(i)}, & D_2 &= \sum_{i\in[0,M-1]} D_2^{(i)}, \\ E_1 &= \sum_{i\in[0,M-1]} E_1^{(i)}, & E_2 &= \sum_{i\in[0,M-1]} E_2^{(i)}, \end{aligned}$$
and sends it to the verifier $\mathcal{V}$.
- The verifier $\mathcal{V}$ checks that
$$\begin{aligned} &E_2 = y\Gamma_{2,fin}, \qquad com_y = y\Gamma_{1,fin}, \\ &e(E_1, \Gamma_{2,fin}) = D_2. \end{aligned}$$
- All sub-provers $\{\mathcal{P}_i\}_{i\in[0,M-1]}$ and $\mathcal{V}$ run
$$\text{Distributed-Dory-IPA}(\boldsymbol{L}, \boldsymbol{R}, C, com_M, D_2, E_1, E_2).$$

PROTOCOL 3.2.05 *Distributed-Dory-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 sub-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.

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

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 sub-prover $\mathcal{P}_0$.
- The master sub-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
$$com_{y'}^{(i)} = \boldsymbol{L'}^{(i)} \boldsymbol{M}^{(i)} \boldsymbol{R'}^{(i)} \Gamma_{1,fin},$$
and sends it to the master sub-prover $\mathcal{P}_0$.
- The master sub-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
$$\text{Distributed-Dory-Eval-RE}(com_M, com_y, \boldsymbol{L}, \boldsymbol{R}) \land$$
$$\text{Distributed-Dory-Eval-RE}(com_M, com_{y'}, \boldsymbol{L'}, \boldsymbol{R'}).$$

4 HyperPianist: Adapting HyperPlonk to the Distributed Setting

In this section, we show our construction of HyperPianist by adapting HyperPlonk to the distributed setting. At a very high level, all constraints in HyperPlonk(+) are reduced to SumCheck identities, as illustrated in Figure 1 from HyperPlonk [3]. Given the distributed SumCheck protocol described in Section 3 as a key building block, we now present how to adapt the multivariate PIOP system to the distributed setting in a bottom-up fashion.

4.1 Distributed ZeroCheck PIOP

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

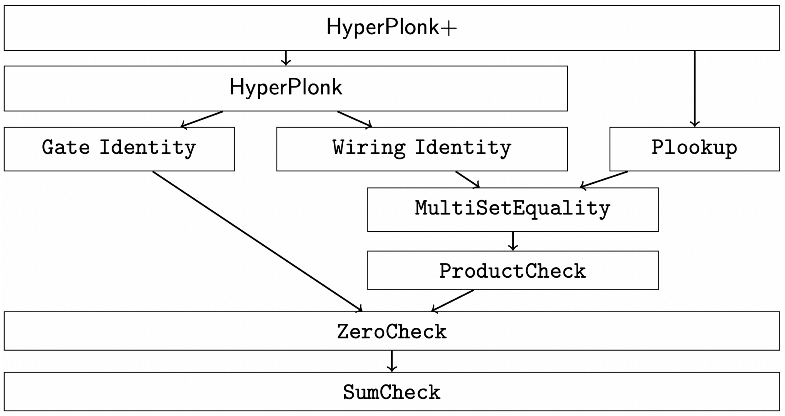

Fig. 1: Overview of the Multivariate PIOP System in HyperPlonk(+).

PROTOCOL 4.1.01 *ZeroCheck PIOP.*

$\mathcal{P}$ holds a multivariate polynomial $f : \mathbb{F}^n \to \mathbb{F}$, and wants to convince $\mathcal{V}$ that $f(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$.

- $\mathcal{V}$ samples $\boldsymbol{r} \leftarrow_\$ \mathbb{F}^n$ and sends it to $\mathcal{P}$.
- $\mathcal{P}$ computes $\hat{f}(\boldsymbol{x}) := f(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r})$, where $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{y}) = \prod_{i=1}^{n}(x_i y_i + (1 - x_i)(1 - y_i))$.
- $\mathcal{P}, \mathcal{V}$ run the SumCheck PIOP (Protocol 3.1.01) to check the relation $((0, [[\hat{f}]]); \hat{f}) \in \mathcal{R}_{\text{Sum}}$.

Definition 10 (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 [3], all constraints are reduced to ZeroCheck relations, and then reduced to SumCheck relations using standard techniques. We show the reduction from ZeroCheck PIOP to SumCheck PIOP in Protocol 4.1.01.

In the distributed setting, we follow the same distribution mechanism as in the distributed SumCheck protocol: assuming there are $M = 2^m$ distributed sub-provers, each sub-prover $\mathcal{P}_i$ holds witnesses w.r.t. indices $\boldsymbol{bin(i)}$, and thus has a local partial multivariate polynomial $f^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ defined as $f^{(i)}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{bin(i)})$ where $f : \mathbb{F}^n \to \mathbb{F}$ is witness polynomial.

Given the verifier's random challenge vector $\boldsymbol{r}$, the i-th sub-prover can construct their 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 4.1.02.

PROTOCOL 4.1.02 *Distributed ZeroCheck PIOP.*

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

- $\mathcal{V}$ samples $\boldsymbol{r} \leftarrow_{\$} \mathbb{F}^n$ and sends it to the master sub-prover $\mathcal{P}_0$, who then transmits $\boldsymbol{r}$ to the other sub-provers.
- Each sub-prover $\mathcal{P}_i$ views $\boldsymbol{r}$ as $\boldsymbol{r} = (\boldsymbol{r}', \boldsymbol{r}'') \in \mathbb{F}^{n-m} \times \mathbb{F}^m$, and locally computes $\hat{f}^{(i)}(\boldsymbol{x}) := f^{(i)}(\boldsymbol{x}) \cdot \widetilde{eq}(\boldsymbol{x}, \boldsymbol{r}') \cdot \widetilde{eq}(\boldsymbol{bin(i)}, \boldsymbol{r}'')$, where $\widetilde{eq}(\boldsymbol{x}, \boldsymbol{y}) = \prod_{i=1}^n (x_i y_i + (1 - x_i)(1 - y_i))$.
- $\{\mathcal{P}_i\}_{\boldsymbol{bin(i)} \in \{0,1\}^m}, \mathcal{V}$ run the distributed SumCheck PIOP (Protocol 3.1.02) to check the relation $((0, [[\hat{f}]]); \hat{f}) \in \mathcal{R}_{\text{Sum}}$.

4.2 Distributed Multiset Check PIOP

We now focus on the Multiset Check PIOP, which demonstrates that two multisets are equal.

Definition 11 (Multiset Check Relation). *For any $k \geq 1$, the relation $\mathcal{R}_{MSet}$ is the set of all tuples*

$$(\mathbb{x}; \mathbb{w}) = (([[f_1]], \ldots, [[f_k]], [[g_1]], \ldots, [[g_k]]); (f_1, \ldots, f_k, g_1, \ldots, g_k))$$

where $f_i, g_i \in \mathcal{F}_n^{(\leq d)} (1 \leq i \leq k)$ and the following two multisets of tuples are equal:

$$\left\{ f_{\boldsymbol{x}} := (f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x})) \right\}_{\boldsymbol{x} \in \{0,1\}^n} = \left\{ g_{\boldsymbol{x}} := (g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x})) \right\}_{\boldsymbol{x} \in \{0,1\}^n}.$$

To prove the relation $\mathcal{R}_{\text{MSet}}$, HyperPlonk uses Reed-Solomon hash to reduce it into a grand product check. The product check relation is defined as follows.

Definition 12 (Product Check Relation). *The relation $\mathcal{R}_{Prod}$ is the set of all tuples*

$$(\mathbb{x}; \mathbb{w}) = ((s, [[f_1]], [[f_2]]); (f_1, f_2))$$

where $f_1, f_2 \in \mathcal{F}_n^{(\leq d)}, f_2(\boldsymbol{b}) \neq 0 \ \forall \boldsymbol{b} \in \{0,1\}^n$ and

$$\prod_{\boldsymbol{x} \in \{0,1\}^n} f'(\boldsymbol{x}) = s,$$

where f' is the rational polynomial $f' = f_1 / f_2$. In the case that $f_2 = 1$, we directly set $f = f_1$ and write $(\mathbb{x}; \mathbb{w}) = ((s, [[f]]); f)$.

PROTOCOL 4.2.01 *Multiset Check PIOP.*

$\mathcal{P}$ holds two multisets of tuples $\left\{f_{\boldsymbol{x}} = (f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\left\{g_{\boldsymbol{x}} := (g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ as defined in Definition 11, and wants to convince $\mathcal{V}$ that the two multisets are equal.

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

Protocol 4.2.04 shows the reduction from Multiset Check to Product Check. In the final step, HyperPlonk applies the Product Check PIOP from Quarks [27] to prove the two grand products are equal. In Quarks [27], the Product Check is transformed into a ZeroCheck relation and then proved by the aforementioned ZeroCheck PIOP.

Below we briefly review the Product Check PIOP from Quarks [27]. This PIOP relies on the following theorem.

Theorem 2. $P = \prod_{\boldsymbol{x}\in\{0,1\}^n} v(\boldsymbol{x})$ *if and only if there exists a multilinear polynomial in* $n+1$ *variables such that* $h(1, \ldots, 1, 0) = P$*, and* $\forall x \in \{0,1\}^n$*, the following hold:* $h(0, \boldsymbol{x}) = v(\boldsymbol{x}), h(1, \boldsymbol{x}) = h(\boldsymbol{x}, 0) \cdot h(\boldsymbol{x}, 1)$.

Theorem 2 shows that, to prove the original grand product relation, it is sufficient to prove the existence of such a multilinear polynomial h. Therefore, $\mathcal{P}$ commits a polynomial purported to be such a h, and proves that it is well-formed - namely, $h(1, \cdots, 1, 0) = P$, and the two constraints hold. To prove the two constraints, it is sufficient for $\mathcal{P}$ to prove that (1) $h(0, \boldsymbol{\gamma}) = v(\boldsymbol{\gamma})$ using the Schwart-Zipple lemma with a random challenge $\boldsymbol{\gamma} \in \mathbb{F}^n$, and (2) $h(1, \boldsymbol{x}) - h(\boldsymbol{x}, 0) \cdot h(\boldsymbol{x}, 1) = 0$ for all $\boldsymbol{x} \in \{0,1\}^n$ using a ZeroCheck PIOP.

Problems When Adapting Product Check PIOP to the Distributed Setting. Now we focus on the distributed setting. Note that after reducing to the Product Check identity, each sub-prover $\mathcal{P}_i$ holds its local partial polynomials $v^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ defined as $v^{(i)}(\boldsymbol{x}) := v(\boldsymbol{x}, (\boldsymbol{bin(i)}))$ with the fixed suffix $\boldsymbol{bin(i)}$. To apply the distributed ZeroCheck PIOP, each sub-prover needs to construct their partial polynomial $h^{(i)} : \mathbb{F}^{n-m+1} \to \mathbb{F}$ defined as $h^{(i)}(\boldsymbol{x}) := h(\boldsymbol{x}, \boldsymbol{bin(i)})$. In Quarks [27], h is constructed as follows:

- $h(1, \cdots, 1) = 0$, 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-provers are unable to construct $h^{(i)}$ from $v^{(i)}$ without interaction, as the immediate values required to construct $h^{(i)}$ are held by the

PROTOCOL 4.2.02 *Rational SumCheck PIOP.*

$\mathcal{P}$ holds two polynomials $p(\boldsymbol{x}), q(\boldsymbol{x}) : \mathbb{F}^n \to \mathbb{F}$ as defined in Definition 13, and wants to convince $\mathcal{V}$ that $\sum_{\boldsymbol{x}\in\{0,1\}^n} \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} = v$.

- $\mathcal{P}$ computes $f(\boldsymbol{x}) := \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}, \forall \boldsymbol{x} \in \{0,1\}^n$.
- $\mathcal{P}$ sends oracle $[[f]]$ to $\mathcal{V}$.
- $\mathcal{P}, \mathcal{V}$ run the ZeroCheck PIOP (Protocol 4.1.01) to check the relation $(([[f]]); f) \in \mathcal{R}_{\text{Zero}}$.
- $\mathcal{P}, \mathcal{V}$ run the SumCheck PIOP (Protocol 3.1.01) to check the relation $((v, [[f]]); f) \in \mathcal{R}_{\text{Sum}}$.

other sub-provers. To obtain the whole partial polynomial $h^{(i)}$, each sub-prover $\mathcal{P}_i$ needs a linear communication cost with the other sub-provers, which is undesirable in the distributed setting. To avoid the linear communication cost, we propose the following two solutions.

Solution 1: Logarithmic Derivatives. Our first solution is to use logarithmic derivatives techniques to construct the Multiset Check PIOP. It relies on the following theorem.

Theorem 3. $\prod_{i=1}^n (a_i + X) = \prod_{i=1}^n (b_i + X)$ *if and only if*

$$\sum_{i=1}^n \frac{1}{a_i + X} = \sum_{i=1}^n \frac{1}{b_i + X}.$$

In the final step of the Multiset PIOP (Protocol 4.2.04), to prove that $\prod_{\boldsymbol{x}\in\{0,1\}^n} \frac{f'(\boldsymbol{x})+\beta}{g'(\boldsymbol{x})+\beta} = 1$, i.e., $\prod_{\boldsymbol{x}\in\{0,1\}^n}(f'(\boldsymbol{x}) + \beta) = \prod_{\boldsymbol{x}\in\{0,1\}^n}(g'(\boldsymbol{x}) + \beta)$, the prover only needs to show

$$\sum_{\boldsymbol{x}\in\{0,1\}^n} \frac{1}{f'(\boldsymbol{x}) + \beta} = \sum_{\boldsymbol{x}\in\{0,1\}^n} \frac{1}{g'(\boldsymbol{x}) + \beta}.$$

To prove the above relation, we extend the SumCheck PIOP to fractions where both the numerator and the denominator are polynomials. 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 $p, q \in \mathcal{F}_n^{(\leq d)}$ and

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

PROTOCOL 4.2.03 *Distributed Rational SumCheck PIOP.*

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

- Each sub-prover $\mathcal{P}_i$ computes $f^{(i)}(\boldsymbol{x}) = \frac{p^{(i)}(\boldsymbol{x})}{q^{(i)}(\boldsymbol{x})}$, $\forall \boldsymbol{x} \in \{0,1\}^n$.
- The master sub-prover $\mathcal{P}_0$ sends an oracle $[[f]]$ to $\mathcal{V}$.
- $\{\mathcal{P}_i\}_{i \in [0, M-1]}, \mathcal{V}$ run the distributed ZeroCheck PIOP (Protocol 4.1.02) to check the relation $(([[f]]); f) \in \mathcal{R}_{\text{Zero}}$.
- $\{\mathcal{P}_i\}_{i \in [0, M-1]}, \mathcal{V}$ run the distributed SumCheck PIOP (Protocol 3.1.02) to check the relation $((v, [[f]]); f) \in \mathcal{R}_{\text{Sum}}$.

This is in the form of SumCheck, but the SumCheck PIOP does not apply directly to fractions. A simple workaround is to find the multilinear interpolation of the fraction and reduce it to a normal SumCheck PIOP plus a ZeroCheck PIOP for the well formation check of the helper polynomial. We present the Rational SumCheck PIOP in Protocol 4.2.02.

This Rational SumCheck PIOP is well suited to the distributed setting, as each sub-prover can calculate the new witnesses locally. We give the distributed Rational SumCheck PIOP in Protocol 4.2.03.

Now we are ready to present our distributed Multiset Check PIOP based on the distributed Rational SumCheck PIOP. The PIOP is shown in Protocol 4.2.03.

Solution 2: Layered Circuit. Our second solution is to directly use a layered circuit to prove grand products, which is also suitable for the distributed setting. We elaborate on this below.

The layered circuit to prove a grand product $s = \prod_{\boldsymbol{z} \in \{0,1\}^n} f(\boldsymbol{z})$ has depth n. 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

$$V_j(\boldsymbol{z}) = \sum_{\boldsymbol{x} \in \{0,1\}^j} \widetilde{eq}(\boldsymbol{x}, \boldsymbol{z}) V_{j+1}(0, \boldsymbol{x}) V_{j+1}(1, \boldsymbol{x}). \tag{2}$$

To prove the grand product relation, $\mathcal{P}$ and $\mathcal{V}$ need $O(n)$ invocations of the SumCheck protocol. The proof starts from the Layer 0, and finally reduced to some random point evaluation of input polynomial f.

PROTOCOL 4.2.04 *Distributed Multiset Check PIOP.*

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Given two multisets of tuples $\left\{f_{\boldsymbol{x}} := (f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\left\{g_{\boldsymbol{x}} := (g_1(\boldsymbol{x}), \ldots, g_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ as defined in Definition 11, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $\left\{f^{(i)}_{\boldsymbol{x}} := (f^{(i)}_1(\boldsymbol{x}), \ldots, f^{(i)}_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$ and $\left\{g^{(i)}_{\boldsymbol{x}} := (g^{(i)}_1(\boldsymbol{x}), \ldots, g^{(i)}_k(\boldsymbol{x}))\right\}_{\boldsymbol{x}\in\{0,1\}^n}$. 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 sub-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 Rational SumCheck PIOP (Protocol 4.2.03) to check the relation $((1, [[f' + \beta]], [[g' + \beta]]); (f' + \beta, g' + \beta)) \in \mathcal{R}_{\mathrm{RSum}}$.

For the 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 $1 \le j \le n-1$, the random opening v_j can be verified by checking

$$v_j = \sum_{\boldsymbol{x}\in\{0,1\}^j} \widetilde{eq}(x, \boldsymbol{r^j} || \gamma^j) V_{j+1}(0, \boldsymbol{x}) V_{j+1}(1, \boldsymbol{x}), \tag{3}$$

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 into one in this round. At the 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 to the $V_n(\boldsymbol{z})$.

Now we consider witness polynomial generation in the distributed setting. At the input layer n, each sub-prover $\mathcal{P}_i$ holds a partial 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 partial witness polynomial $V_j^{(i)} : \mathbb{F}^{j-m}$ 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 sub-prover $\mathcal{P}_0$. Then $\mathcal{P}_0$ reconstructs the polynomial $V_m : \mathbb{F}^m \rightarrow \mathbb{F}$ 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 describe 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 sub-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.

4.3 Distributed Permutation Check PIOP

In HyperPlonk, the wiring constraints are reduced to a permutation relation and then transformed into a multiset check relation. The permutation relation shows that for two multivariate polynomials $f, g \in \mathcal{F}_n^{(\leq d)}$, 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 $\{0,1\}^n \rightarrow \{0,1\}^n$, $f, g \in \mathcal{F}_n^{(\leq d)}$, such that for all $\boldsymbol{x} \in \{0,1\}^n$,

$$f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x}).$$

We first review the construction in HyperPlonk, which originates from Plonk. Given a tuple $(\sigma; ([[f]], [[g]]); (f, g))$ where σ is the predefined permutation, the indexer generates two oracles $[[s_{id}]], [[s_\sigma]]$ such that the polynomial $s_{id} \in \mathcal{F}_n^{(\leq 1)}$ 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, $s_\sigma \in \mathcal{F}_n^{(\leq 1)}$ maps $\boldsymbol{x} \in \{0,1\}^n$ to $[\sigma(\boldsymbol{x})]$. Given this, HyperPlonk reduces the Permutation check 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 4.3.01.

PROTOCOL 4.3.01 *Permutation Check PIOP.*

$\mathcal{P}$ holds two polynomials $f(\boldsymbol{x}), g(\boldsymbol{x}) : \mathbb{F}^n \to \mathbb{F}$ as defined in Definition 13, and wants to convince $\mathcal{V}$ that for all $\boldsymbol{x} \in \{0,1\}^n, f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x})$ where σ is a permutation $\{0,1\}^n \to \{0,1\}^n$.

– $\mathcal{P}, \mathcal{V}$ run a Multiset Check PIOP (Protocol 4.2.04) to check the relation $((1, [[(s_{id}]], [[f]], [[s_\sigma]], [[g]])); (s_{id}, f, s_\sigma, g) \in \mathcal{R}_{\text{MSet}}$.

The sub-provers are able to locally compute their partial polynomials $s_{id}^{(i)} := \{s_{id}(\boldsymbol{x}, bin(i)) \mid \boldsymbol{x} \in \{0,1\}^{n-m}\}$ and $s_\sigma^{(i)} := \{s_\sigma(\boldsymbol{x}, bin(i)) \mid \boldsymbol{x} \in \{0,1\}^{n-m}\}$. With the distributed Multiset Check PIOP, the above Permutation Check PIOP adapts to the distributed setting naturally. We give the distributed Permutation Check PIOP in Protocol 4.3.02.

PROTOCOL 4.3.02 *Distributed Permutation Check PIOP.*

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Given two polynomials $f(\boldsymbol{x}), g(\boldsymbol{x}) : \mathbb{F}^n \to \mathbb{F}$ as defined in Definition 13, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $f^{(i)}(\boldsymbol{x}), g^{(i)}(\boldsymbol{x})$. All sub-provers want to convince $\mathcal{V}$ that for all $\boldsymbol{x} \in \{0,1\}^n, f(\sigma(\boldsymbol{x})) = g(\boldsymbol{x})$ where σ is a permutation $\{0,1\}^n \to \{0,1\}^n$.

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

4.4 Putting Everything Together: HyperPianist

Our fully distributed SNARK is constructed upon the building blocks introduced above. The constraint system is from HyperPlonk [3], but we additionally assume that all initial witnesses are distributed among the sub-provers.

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 15 (Constraint System of HyperPianist [3]). *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* $\mathcal{R}_{HyperPianist}$ *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*

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

PROTOCOL 4.4.01 *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 sub-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 PIOP for the gate constraint, which is a distributed ZeroCheck PIOP (Protocol 4.1.02) 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. $\mathcal{P}$ and $\mathcal{V}$ run a 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 $p(\boldsymbol{r}) = w(\langle 0\rangle_{\mu+\nu_w-\nu}, \boldsymbol{r})$.

– *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 4.4.01. We can instantiate it with the distributed Dory PCS. Note that in the above protocols we omit the zero-knowledge property for simplicity, and this can be achieved using standard techniques from previous works [12,18,3].

5 HyperPianist+: HyperPianist with Optimized Lookup Arguments

In this section, we construct an optimized lookup argument and adapt it to the distributed setting. We then enhance HyperPianist with this lookup argument to obtain HyperPianist+.

5.1 Distributed Lookup Arguments.

The lookup relation is essentially a set inclusion relation on committed vectors, defined as follows.

Definition 16 (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 $\boldsymbol{T} \in \mathbb{F}^N$, such that for all $i \in \{1, \cdots, \ell\}$

$$\boldsymbol{a}_i = \boldsymbol{T}[\boldsymbol{b}_i].$$

There have been a series of works [33,25,34,10,8,16,28] aimed at improving the efficiency of the lookup table argument. In the univariate setting, with some expensive setup, prover costs can be made only quasi-linear to the number of queries. In the multivariate setting, which is our main focus, HyperPlonk+ has shown how to transform the univariate PIOP from Plookup [11] into a multivariate one. However, as noted in Lasso [28], this transformation introduces additional overhead. As far as we know, Lasso is the most cost-effective construction for lookup arguments, and thus we use it as our starting point.

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 subtables, such that for some $r = (r_1, \cdots, r_c)$ and some (simple) algebraic function g,

$$T[r] = g(T_1[r_1], \cdots, T_k[r_1], T_{k+1}[r_2], \cdots, T_{2k}[r_2], \cdots, T_{\alpha-k+1}[r_c], \cdots, T_\alpha[r_c]),$$

where T_i are the subtables, 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 $\boldsymbol{t} \in \mathbb{F}^n$, 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 nonzero entry of value 1, and $\boldsymbol{t} = \boldsymbol{M} \cdot \boldsymbol{t}$. Then, they run the SumCheck protocol to check that

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

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. Let $v := \tilde{a}(\boldsymbol{r})$. Then $\mathcal{V}$ is able to obtain v via an oracle call of $\tilde{a}$. Now the main task is to efficiently run the SumCheck protocol on the following claim

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

Observe that one key feature of the matrix $\boldsymbol{M}$ is that it is extremely sparse, i.e., only one entry in each row of $\boldsymbol{M}$ can be non-zero, and the non-zero entry should have value 1. Making use of this feature, Lasso designs an efficient method to prove Equation 4.

Given that $\boldsymbol{M}$ is sparse, Lasso transforms Equation 4 into the following equation

$$\sum_{\boldsymbol{i} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{i}, \boldsymbol{r}) \cdot T[nz(\boldsymbol{i})] = v,$$

where for each $\boldsymbol{i}$-th row, $nz(\boldsymbol{i})$ denotes the column corresponding to the non-zero entry in this row, and $T[nz(\boldsymbol{i})]$ denotes the $nz(\boldsymbol{i})$-th entry of the table t. Since we assume that the table T is decomposable, we can write the LHS of the above equation as

$$\sum_{\boldsymbol{i}\in\{0,1\}^{\log\ell}} \widetilde{eq}(\boldsymbol{i},\boldsymbol{r}) \cdot g(T_1[nz_1(\boldsymbol{i})], \cdots, T_k[nz_1(\boldsymbol{i})], T_{k+1}[nz_2(\boldsymbol{i})], \cdots, \\ T_{2k}[nz_2(\boldsymbol{i})], \cdots, T_{\alpha-k+1}[nz_c(\boldsymbol{i})], \cdots, T_\alpha[nz_c(\boldsymbol{i})]).$$

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

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

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 Memory-in-the-Head technique from Spartan [26]. We briefly review the technique below.

Memory-in-The-Head in a Nutshell. 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 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.

Memory-in-The-Head as Applied 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

PROTOCOL 5.1.01 *Well-Formation Check PIOP.*

$\mathcal{P}$ has provided oracles to $\widetilde{nz}(\boldsymbol{x})$ and $E(\boldsymbol{x})$, where $\widetilde{nz}$ is the MLE of nz, and wants to convince $\mathcal{V}$ that $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})]$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

- $\mathcal{P}$ computes $m(X)$ (as defined in the text), and sends the oracle of $m(X)$ to $\mathcal{V}$.
- $\mathcal{V}$ samples $\beta, \gamma \leftarrow_{\$} \mathbb{F}$, and sends them to $\mathcal{P}$.
- $\mathcal{P}$ sends the rational sum claim to $\mathcal{V}$

$$v = \sum_{\boldsymbol{x} \in \{0,1\}^{\log l}} \frac{1}{\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x})}.$$

- $\mathcal{P}, \mathcal{V}$ run the Rational SumCheck PIOP (Protocol 4.2.02) to check the relations $((v, [[m(\boldsymbol{x})]], [[\beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x})]]); (m(\boldsymbol{x}), \beta + s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$, and $((v, [[1]], [[\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}))]]); (1, \beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$.

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

Claim 1. 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 $\mathsf{write\_cts} = \mathsf{read\_ts} + 1$.

Prover Time. For each memory checking, the prover needs to commit two polynomials $\mathsf{read\_ts}$ and $\mathsf{final\_cts}$, both of which require time $O(m)$. To demonstrate multisets are equal, the prover engages in four grand product checks (or their logarithmic derivative counterparts), with two of size $O(m)$ and two of size $O(N)$.

Our Optimization: Using Logup Instead. We make the observation that

Claim 2. 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 [16] (or its layered circuit compilation version [23]). Logup [16] depends heavily on the logarithmic derivative technique, which is a generalized version of Theorem 3.

Theorem 4. *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 $z \in \mathbb{F}$.*

PROTOCOL 5.1.02 *Lookup PIOP.*

T is a decomposable lookup table of size N, and $\boldsymbol{a}$ is the vector of lookups of size ℓ. $\mathcal{P}$ knows polynomials $\widetilde{nz}_1, \cdots, \widetilde{nz}_c : \mathbb{F}^{\log \ell} \to \mathbb{F}$ which are MLE of $nz_1, \cdots, nz_c$. $\mathcal{P}$ wants to convince $\mathcal{V}$ that each entry of $\boldsymbol{a}$ is in the table T.

- $\mathcal{P}$ provides an oracle of the MLE polynomial $\tilde{a} : \mathbb{F}^{\log \ell} \to \mathbb{F}$ of $\boldsymbol{a}$, and oracles of $nz_1, \cdots, nz_c : \mathbb{F}^{\log \ell} \to \mathbb{F}$.
- $\mathcal{V}$ 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 the relation that $v = \sum_{\boldsymbol{k} \in \{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{r}, \boldsymbol{k}) g(E_1(\boldsymbol{k}), \cdots, E_\alpha(\boldsymbol{k}))$.
- $\mathcal{P}, \mathcal{V}$ run a Well-Formation Check PIOP (Protocol 5.1.01) to check that $E_j(\boldsymbol{k}) = T_j(nz(k))$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

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

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

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. $\square$

This leads to the following algebraic criterion for set inclusion.

Theorem 5. *Suppose that $(a_i)_{i=1}^{\ell}$, $(b_j)_{j=1}^{N}$ are arbitrary 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 2)

$$\{(nz(\boldsymbol{k}), E(\boldsymbol{k})) | \boldsymbol{k} \in \{0,1\}^{\log \ell}\} \subset \{(\boldsymbol{x}, T(\boldsymbol{x})) | \boldsymbol{x} \in \{0,1\}^{\log N}\}.$$

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 5. We

PROTOCOL 5.1.03 *Distributed Well-Formation Check PIOP.*

Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Given polynomials $\widetilde{nz}(\boldsymbol{x}), E(\boldsymbol{x}) : \mathbb{F}^{\log \ell} \to \mathbb{F}$ and $T : \mathbb{F}^{\log N} \to \mathbb{F}$ as defined in text, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $\widetilde{nz}^{(i)}(\boldsymbol{x}) = \widetilde{nz}(\boldsymbol{x}, \boldsymbol{bin(i)}), E^{(i)}(\boldsymbol{x}) = E(\boldsymbol{x}, \boldsymbol{bin(i)})$. All sub-provers want to convince $\mathcal{V}$ that $E(\boldsymbol{k}) = T[nz(\boldsymbol{k})]$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

- Each sub-prover $\mathcal{P}_i$ locally computes $m^{(i)}$ as described in the text.
- Sub-provers calculate the rational sumcheck claim v.
- $\{\mathcal{P}_i\}_{i \in [0, M-1]}, \mathcal{V}$ run the distributed Rational SumCheck PIOP (Protocol 4.2.03) 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 $((v, [[1]], [[\beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x})]]); (1, \beta + \widetilde{nz}(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}))) \in \mathcal{R}_{\text{RSum}}$.

can get the following relation

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

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 4.3, i.e., it maps each binary string $\boldsymbol{x} \in \{0,1\}^n$ to the corresponding integer value $[\boldsymbol{x}] = \sum_{i=1}^{n} x_i \cdot 2^{i-1} \in \mathbb{F}$.

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 Memory-in-the-Head approach). Now, to prove the well-formation of E, the prover needs to compute the polynomial $m(\boldsymbol{x})$ with $O(m)$ non-zero entries, and engage in a rational SumCheck to prove that $v = \sum_{\boldsymbol{x} \in \{0,1\}^{\log N}} \frac{m(\boldsymbol{x})}{s_{id}(\boldsymbol{x}) + \gamma \cdot T(\boldsymbol{x}) + \beta}$ as well as $v = \sum_{\boldsymbol{x} \in \{0,1\}^{\log \ell}} \frac{1}{nz(\boldsymbol{x}) + \gamma \cdot E(\boldsymbol{x}) + \beta}$. The details are in Protocol 5.1.01.

Prover Time. For each logup, the prover needs to commit one polynomial m, requiring $O(m)$ time. To demonstrate the set inclusion, the prover engages in two logarithmic derivative checks, of size $O(m)$ and size $O(N)$ respectively. This directly leads to a 50% save in the constant compared with using memory in the heads.

We show the complete Lookup PIOP in Protocol 5.1.02. It follows Lasso [28], except that we use the logup based techniques for the Well-Formation Check PIOP rather than the Memory-in-The-Head approach.

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 partial polynomials of $nz^{(i)}(\boldsymbol{x})$. Given this, the sub-provers are able to locally construct $E^{(i)}(\boldsymbol{x}) = T[nz^{(i)}(\boldsymbol{x})]$. The main task of

PROTOCOL 5.1.04 *Distributed Lookup PIOP.*

T is a decomposable lookup table of size N, and $\boldsymbol{a}$ is the vector of lookups of size ℓ. Suppose there are M distributed sub-provers $\mathcal{P}_0, \cdots, \mathcal{P}_{M-1}$ and $\mathcal{P}_0$ is the master sub-prover. Given polynomials $\widetilde{nz}_1(\boldsymbol{x}), \cdots, \widetilde{nz}_c(\boldsymbol{x})$ as defined in Equation 5, suppose each sub-prover $\mathcal{P}_i$ holds local partial polynomials $\widetilde{nz}_1^{(i)}(\boldsymbol{x}), \cdots, \widetilde{nz}_c^{(i)}(\boldsymbol{x})$. All sub-provers want to convince $\mathcal{V}$ that each entry of $\boldsymbol{a}$ is in the table T.

- The master sub-prover $\mathcal{P}_0$ provide an oracle of the MLE polynomial $\tilde{a} : \mathbb{F}^{\log \ell} \to \mathbb{F}$ of $\boldsymbol{a}$, and oracles of $\widetilde{nz}_1, \cdots, \widetilde{nz}_c : \mathbb{F}^{\log \ell} \to \mathbb{F}$.
- $\mathcal{V}$ picks a random $\boldsymbol{r} \in \mathbb{F}^{\log \ell}$, and sends it to the master sub-prover $\mathcal{P}_0$, who then transmits it to the other sub-provers.
- $\mathcal{V}$ makes an oracle call to $\tilde{a}$ and obtains $\tilde{a}(\boldsymbol{r})$.
- $\{\mathcal{P}\}_{i\in[0,M-1]}, \mathcal{V}$ run a distributed SumCheck PIOP (Protocol 3.1.02) to check the relation that $v = \sum_{\boldsymbol{k}\in\{0,1\}^{\log \ell}} \widetilde{eq}(\boldsymbol{r}, \boldsymbol{k}) g(E_1(\boldsymbol{k}), \cdots, E_\alpha(\boldsymbol{k}))$.
- $\{\mathcal{P}\}_{i\in[0,M-1]}, \mathcal{V}$ run a distributed Well-Formation Check PIOP (Protocol 5.1.03) to check that $E_j(\boldsymbol{k}) = T_j(nz(k))$ for all $\boldsymbol{k} \in \{0,1\}^{\log \ell}$.

the sub-provers is to compute $m(\boldsymbol{x})$. We let $m^{(i)} : \mathbb{F}^{n-m} \to \mathbb{F}$ be a multivariate polynomial that maps $\boldsymbol{x} \in \{0,1\}^{n-m}$ 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 definition, we have

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

This demonstrates that the computation of $m(\boldsymbol{x})$ can be locally distributed among the sub-provers. As shown in Section 4.3, the polynomial s_{id} is also suitable for distribution. We thus give the distributed Well-Formation Check PIOP in Protocol 5.1.03.

Given the above distributed Well-Formation Check PIOP, we now present our distributed Lookup PIOP in Protocol 5.1.04.

5.2 Putting Everything Together: HyperPianist+

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

Definition 17 (Constraint System of HyperPianist+ [3]). *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* $f_{lk} : \mathbb{F}^{\ell_{lk}+\ell_w} \to \mathbb{F}$ *is an algebraic map. The indexed relation* $\mathcal{R}_{HyperPianist+}$

PROTOCOL 5.2.01 *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 sub-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 4.4.01) for $(\mathbb{i}; \mathbb{x}; \mathbb{w}) \in \mathcal{R}_{\mathrm{HyperPianist}}$.
3. $\{\mathcal{P}_i\}_{i \in [0, M-1]}$ and $\mathcal{V}$ run a distributed Lookup PIOP (Protocol 5.1.04) for $(\mathsf{table}; [[g]]) \in \mathcal{L}(\mathcal{R}_{\mathrm{Lookup}})$ where g is as defined in Definition 16.

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*

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

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

We present the distributed PIOP for HyperPlonk+ in Protocol 5.2.01. It is the combination of the distributed PIOPs for HyperPlonk and the Lookup relation.

6 Evaluation

In this section, we show some preliminary results of our implementation and experiments.

6.1 Implementation

While a full implementation of our distributed proof system is still in the works, to better gauge the effectiveness of our techniques, we have implemented our optimized lookup arguments as patches to Jolt (a SNARK framework for zkVMs based on Lasso). [2] Below we give some more details on the implementation of our optimized lookup arguments.

[2] Our implementation is in https://github.com/zhaowenlan1779/jolt/tree/logup.

Sparse-Dense Layered Circuits. Our rational sum argument on f can be considered "sparse" - even though it is defined over the hypercube of size $\log N$, at most m entries have non-zero numerators.

The implementation of Lasso in the Jolt framework relied on layered-circuit-based grand product checks. In Lasso, the grand products are dense; but in Lasso, for each CPU tick, the next instruction needs to be fetched and decoded from memory. The results of all possible lookups for all instruction types are evaluated, and only one of them is used, depending on the actual type of the instruction. That is, the "primary sumcheck" becomes

$$\sum_{\boldsymbol{x}} eq(\boldsymbol{r}, \boldsymbol{x}) \left[\mathrm{flags}_0(\boldsymbol{x}) g_0(E_0(\boldsymbol{x})) + \cdots \mathrm{flags}_K(\boldsymbol{x}) g_K(E_K(\boldsymbol{x}))\right]$$

where $\mathrm{flags}_k(\boldsymbol{x})$ (on the hypercube) determines whether the instruction in index $\boldsymbol{x}$ is of type k, and K is the number of instruction types. This meant that the grand product tree became sparse.

Lasso developed a sparse-dense layered circuit proof for this purpose. In Lasso, each layer on the grand product tree may be dense or sparse, as determined by a "densification" threshold. A sparse layer is stored as a list of pairs of indices and values, and indices not in the list have value 1 (and thus do not change the product). To prove the SumCheck for each layer, it is necessary to sequentially bind the layer to challenges, and to compute a univariate polynomial on the next variable and sum up the rest (i.e. $\sum_{\boldsymbol{x}} f(r_1, \cdots, r_{k-1}, X_k, \boldsymbol{x})$). Both can be adapted to suit the sparse representation, although the sparsity will decrease with each bind.

Unfortunately this approach does not directly apply to our rational sum argument. Since each node now holds a rational (p, q), and the two polynomials are committed separately, we cannot just dispose of the nodes where $p \equiv 0$, because the q commitments must be opened to their proper values.

In our current implementation, for q we effectively just prove a regular dense grand product, while p is a sparse rational sum (the proofs for p and q are still related in the sense that challenges are shared). Since $q = \beta + (s_{\mathrm{id}}(x) + \gamma \cdot T(x))$ does not depend on the dimension index, there are only k different q polynomials, not α, and the computation on q is not typically the dominant part. Preliminary experiments show that the sparse-dense layered circuit can be up to 40% faster, though the concrete benefits still need to be evaluated with more practical tests.

6.2 Experiments

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 $2^{16} = 65536$). All experiments are run 6 times (consecutively), with the first run discarded, on a laptop equipped with AMD Ryzen 7 5800H @ 3.20 GHz. Proof size reported here is the compressed size. Prover time does not include time needed to setup generators; verifier time does

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**

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. 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
3. 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)
4. 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)
5. 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)

6. 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)
7. 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
8. 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
9. 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)
10. 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
11. 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
12. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
13. 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)
14. 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
15. 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)
16. 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
17. 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)
18. 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
19. 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)
20. 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

21. 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)
22. 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)
23. 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
24. Pearson, L., Fitzgerald, J., Masip, H., Bellés-Muñoz, M., Muñoz-Tapia, J.L.: Plonkup: Reconciling plonk with plookup. Cryptology ePrint Archive (2022)
25. 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
26. 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)
27. 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
28. 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
29. 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)
30. 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
31. 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)
32. 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
33. 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
34. 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
35. 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 Dory Evaluation Proof

In this section, we present the formal protocols of the evaluation proof in Dory [17].

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

Suppose the prover $\mathcal{P}$ holds the witness $\boldsymbol{v_1}, \boldsymbol{v_2}$ 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 $\chi = \langle \varGamma_1, \varGamma_2 \rangle$.

- The prover $\mathcal{P}$ computes

$$D_{1L} = \langle \boldsymbol{v_{1L}}, \varGamma_2' \rangle, \quad D_{1R} = \langle \boldsymbol{v_{1R}}, \varGamma_2' \rangle, \quad E_{1\beta} = \langle \varGamma_1, \boldsymbol{s_2} \rangle,$$
$$D_{2L} = \langle \varGamma_1', \boldsymbol{v_{2L}} \rangle, \quad D_{2R} = \langle \varGamma_1', \boldsymbol{v_{2R}} \rangle, \quad E_{2\beta} = \langle \boldsymbol{s_1}, \varGamma_2 \rangle,$$

  and sends them to verifier $\mathcal{V}$.
- The verifier $\mathcal{V}$ samples $\beta \leftarrow_\$ \mathbb{F}$ and sends it to the prover $\mathcal{P}$.
- The prover $\mathcal{P}$ sets

$$\boldsymbol{v_1} \leftarrow \boldsymbol{v_1} + \beta \varGamma_1, \quad \boldsymbol{v_2} \leftarrow \boldsymbol{v_2} + \beta^{-1} \varGamma_2.$$

- The prover $\mathcal{P}$ computes

$$E_{1+} = \langle \boldsymbol{v_{1L}}, \boldsymbol{s_{2R}} \rangle, \quad E_{1-} = \langle \boldsymbol{v_{1R}}, \boldsymbol{s_{2L}} \rangle, \quad C_+ = \langle \boldsymbol{v_{1L}}, \boldsymbol{v_{2R}} \rangle,$$
$$E_{2+} = \langle \boldsymbol{s_{1L}}, \boldsymbol{v_{2R}} \rangle, \quad E_{2-} = \langle \boldsymbol{s_{1R}}, \boldsymbol{v_{2L}} \rangle, \quad C_- = \langle \boldsymbol{v_{1R}}, \boldsymbol{v_{2L}} \rangle,$$

  and sends them to the verifier $\mathcal{V}$.
- The verifier $\mathcal{V}$ samples $\alpha \leftarrow_\$ \mathbb{F}$ and sends it to the prover $\mathcal{P}$.
- The prover $\mathcal{P}$ sets

$$\boldsymbol{v_1'} \leftarrow \alpha \boldsymbol{v_{1L}} + \boldsymbol{v_{1R}}, \quad \boldsymbol{v_2'} \leftarrow \alpha^{-1} \boldsymbol{v_{1L}} + \boldsymbol{v_{1R}}.$$

- The verifier $\mathcal{V}$ computes

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

- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ both set

$$\boldsymbol{s_1}' \leftarrow \alpha \boldsymbol{s_{1L}} + \boldsymbol{s_{1R}}, \quad \boldsymbol{s_2}' \leftarrow \alpha^{-1} \boldsymbol{s_{2L}} + \boldsymbol{s_{2R}}.$$

- The verifier $\mathcal{V}$ 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 A.0.02 *Dory-Fold-Scalar*$(s_1, s_2, C, D_1, D_2, E_1, E_2)$.

Suppose the prover $\mathcal{P}$ holds the witness 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 $\chi = \langle \Gamma_1, \Gamma_2 \rangle$.

– The verifier $\mathcal{V}$ samples $\gamma \leftarrow_{\$} \mathbb{F}$ and sends it to the prover $\mathcal{P}$.
– The prover $\mathcal{P}$ defines

$$v_1' = v_1 + \gamma s_1 H_1, \quad v_2' = v_2 + \gamma^{-1} s_2 H_2,$$

and sends them to the verifier $\mathcal{V}$.
– The verifier computes

$$\begin{aligned} C' &= C + s_1 s_2 H_T + \gamma \cdot e(H_1, E_2) + \gamma^{-1} \cdot e(E_1, H_2), \\ D_1' &= D_1 + e(H_1, s_1 \gamma \Gamma_2), \\ D_2' &= D_2 + e(s_2 \gamma^{-1} \Gamma_1, H_2) \end{aligned}$$

– The verifier $\mathcal{V}$ 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 A.0.03 *Dory-IPA*$_{2^n}(\boldsymbol{s_1}, \boldsymbol{s_2}, C, D_1, D_2, E_1, E_2)$.

Suppose the prover $\mathcal{P}$ holds witness $\boldsymbol{v_1}, \boldsymbol{v_2}$ 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

$$\begin{aligned} \Delta_{1L,i} &= \langle (\Gamma_{1,i})_L, \Gamma_{2,i+1} \rangle, \quad \Delta_{2L,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_L \rangle, \\ \Delta_{1R,i} &= \langle (\Gamma_{1,i})_R, \Gamma_{2,i+1} \rangle, \quad \Delta_{2R,i} = \langle \Gamma_{1,i+1}, (\Gamma_{2,i})_R \rangle. \end{aligned}$$

– For $j = 0, \ldots, n-1$, the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ run

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

– The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ run

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

PROTOCOL A.0.04 *Dory-Eval-RE*$(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

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

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

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

- The prover $\mathcal{P}$ computes

$$\boldsymbol{v} = \boldsymbol{L}^T\boldsymbol{M}, \quad y = \langle \boldsymbol{v}, \boldsymbol{R}\rangle,$$

  and sends y to the verifier.
- The prover $\mathcal{P}$ computes

$$\begin{aligned} C &= e(\langle \boldsymbol{v}, \boldsymbol{com_{row}}\rangle, \Gamma_{2,fin}), & D_2 &= e(\langle \Gamma_1, \boldsymbol{v}\rangle, \Gamma_{2,fin}), \\ E_1 &= \langle \boldsymbol{L}, \boldsymbol{com_{row}}\rangle, & E_2 &= y\Gamma_{2,fin}, \end{aligned}$$

  and sends them to the verifier $\mathcal{V}$.
- The verifier $\mathcal{V}$ checks that

$$\begin{aligned} E_2 &= y\Gamma_{2,fin}, & com_y &= y\Gamma_{1,fin}, \\ e(E_1, \Gamma_{2,fin}) &= D_2. \end{aligned}$$

- The prover $\mathcal{P}$ and $\mathcal{V}$ run

$$\text{Dory-IPA}(\boldsymbol{L}, \boldsymbol{R}, C, com_M, D_2, E_1, E_2).$$

PROTOCOL A.0.05 *Dory-Eval*$(com_M, com_y, \boldsymbol{L}, \boldsymbol{R})$

Suppose the prover $\mathcal{P}$ 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 $(r_1 \cdots, r_n) \in \mathbb{F}^n$.

- The verifier samples $u \leftarrow_{\$} \mathbb{F}$ and sends it to the prover $\mathcal{P}$.
- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ both set

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

- The prover $\mathcal{P}$ computes

$$com_{y'} = \boldsymbol{L'} \boldsymbol{M} \boldsymbol{R'} \Gamma_{1,fin},$$

  and sends it to the verifier $\mathcal{V}$.
- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ run

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