HyperFond: A Transparent and Post-Quantum Distributed SNARK with Polylogarithmic Communication

Yuanzhuo Yu[1], Mengling Liu[2], Yuncong Zhang[3], Shi-Feng Sun[1],

Tianyi Ma[1], Man Ho Au[2], and Dawu Gu[1]

Abstract

Recent years have witnessed the surge of academic researches and industrial implementations of *succinct non-interactive arguments of knowledge (SNARKs)*. However, proving time remains a bottleneck for applying SNARKs to large-scale circuits. To accelerate the proof generation process, a promising way is to distribute the workload to several machines running in parallel, the SNARKs with which feature are called *distributed SNARKs*. Nevertheless, most existing works either require a trusted setup, or rely on quantum-insecure assumptions, or suffer from linear communication costs.

In this paper, we introduce HyperFond, the *first* distributed SNARK that enjoys a transparent setup, post-quantum security and polylogarithmic communication cost, as well as field-agnosticity (no reliance on specific choices of fields). To this end, we first propose a distributed proof system based on HyperPlonk (by Chen et al. in EUROCRYPT 2023). To instantiate the system, we then put forward a *novel* approach to distribute the multilinear polynomial commitment scheme in BaseFold (by Zeilberger et al. in CRYPTO 2024), and present a trade-off between communication cost and proof size. In HyperFond, after committing to polynomial coefficients with quasilinear complexity, each sub-prover generates proofs with time linear in subcircuit size.

We implement HyperFond using up to 16 machines. Experimental results demonstrate that the proving time of HyperFond is 15.6 × faster than HyperPlonk instantiated with BaseFold. We also compare to deVirgo (by Xie et al. in CCS 2022), so far the only post-quantum distributed SNARK, and achieve a 1.45 × speedup.

[1]Shanghai Jiao Tong University
[2]The Hong Kong Polytechnic University
[3]Shandong University

Contents

1 Introduction

The Succinct Non-interactive ARgument of Knowledge (SNARK) allows a prover, who secretly possesses a witness $\mathbb{w}$ for a public statement $\mathbb{x}$ satisfying some **NP** relation $\mathcal{R}$, to non-interactively convince a verifier that $\mathbb{x}$ is true, during which the proof size and verification time are sublinear in the length of $\mathbb{w}$. Since the seminal work [1] by Bitansky et al., intensive studies have been carried out on SNARK constructions ([2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] and references therein), focusing on reducing reliance on trusted setups [4, 6, 9, 10, 13], achieving post-quantum security [2, 3, 6, 9, 13, 14], and optimizing performance [5, 7, 8, 11, 12].

Benifiting from their remarkable efficiency, SNARKs have met considerable real-world demands, as privacy-preserving cryptocurrencies (e.g., Zcash [15]) and scalability solutions for blockchains (e.g., zkRollups [16]), where $(\mathbb{x}; \mathbb{w})$ pairs are formulated as circuit satisfiability relations. However, SNARKs face a critical bottleneck in prover overhead, hindering large-scale adoption in applications such as proving machine learning inference. Moreover, the advent of quantum computing threatens traditional cryptographic primitives, including SNARKs. While post-quantum primitives have been extensively studied, with standards as the NIST Post-Quantum Cryptography project [17], little attention has been paid to the practical application of quantum resistant SNARKs.

In recent years, multiple works have focused on accelerating proof generation by distributing overall workload to several machines (sub-provers), with each proving a partial statement expressed as a subcircuit, but few demonstrate post-quantum security. Since the initiation of DIZK [18], the first distributed proof system with notable efficiency gains, several protocols have been proposed, including deVirgo [19], Pianist [20], Hekaton [21], HyperPianist [22], Cirrus [23] and Soloist [24]. Most protocols, excluding deVirgo, are *fully distributed* (supporting general arithmetic circuits) and achieve sublinear communication costs. In contrast, deVirgo only supports *data-parallel* circuits (displaying some parallel and repeated structures), and also faces linear communication costs among sub-provers, which, as discussed in Pianist and HyperPianist, is incurred by frequent witness exchange among sub-provers to compute Merkle roots. However, deVirgo is by far the *only* solution with post-quantum security and transparent setup, thanks to the FRI [25]-based Polynomial Commitment Scheme employed therein. Moreover, it enjoys, coming with FRI, a lighter implementation (for hash functions) than those of pairing-based counterparts, by operating on smaller fields for equivalent security (e.g., to provide at least 100 bits of security, the size of base field in deVirgo is about 2^{128}, while those of pairing-based SNARKs are around 2^{256}).

Based on these observations, a research question arises:

Can we design a transparent and post-quantum distributed SNARK that outperforms the state-of-the-art deVirgo?

In this paper, we make affirmative progress on this question. Following prior frameworks

Table 1: Comparisons of HyperFond to deVirgo. Here N denotes the whole circuit size, M the number of sub-provers, and $T := \frac{N}{M}$. "$\mathcal{P}_i$" stands for the running time of each sub-prover, "Comm." the total communication costs, "$|\pi|$" the proof size, and "$\mathcal{V}$" the verification time. "Fully?" denotes the support for general circuits, and "PQ?" means quantum resistance.

Protocols	$\mathcal{P}_i$ (PIOP)	$\mathcal{P}_i$ (PCS)	Comm.	$\|\pi\|$ and $\mathcal{V}$	Setup	Fully?	PQ?
deVirgo [19]	$O(T)$	$O(T \log T)$	$O(N)$	$O(\log^2(N))$	Transparent	×	✓
HyperFond **(ours)**	$O(T)$	$O(T \log T)$	$O(M \log^2(T))$	$O(M \log^2(T))$	Transparent	✓	✓

[8, 26, 10, 11, 27] we instantiate an information-theoretic object called Polynomial Interactive Oracle Proof (PIOP) [11, 28] with a cryptographic object known as Polynomial Commitment Scheme (PCS) [29], yielding a transparent, post-quantum secure distributed SNARK with ploylogrithmic communication costs among sub-provers, even for general arithmetic circuits.

1.1 Our Contributions

We present HyperFond, the first fully distributed SNARK featuring no trusted setup, quantum resistance, polylogarithmic communication costs, and field-agnosticity; we then compare to deVirgo [19], by far the only post-quantum secure distributed SNARK, in terms of asymptotic complexity (as shown in Table 1) and concrete efficiency. In detail, our main contributions are summarized as follows.

A distributed multivariate PIOP system. We propose a distributed PIOP system with polylogarithmic communication and linear proving time. Our construction is based on the framework of HyperPlonk [27], by refining its building blocks with more efficient and distribution-friendly constructions. In addition, we propose a new Lookup argument which generalizes the original approach proposed by Papini et al. [30] to the distributed setting for polynomials with *higher degrees*, and features supporting *multi-column* lookup without sending extra oracles.

A distributed multivariate PCS. We introduce, for the first time, a distributed version of BaseFold [31], which is a code-based PCS that generalizes FRI [25]. Retaining setup transparency and quantum resistance, the total communication costs are only polylogarithmic. Simultaneously, our distributed PCS can be built over agnostic fields with sufficient size, enjoying the same nice property as in BaseFold, which leads to better flexibility compared to SNARKs over FFT (Fast Fourier Transform)-friendly fields.

Implementations. To demonstrate practicality, we implement HyperFond and compare it to HyperPlonk instantiated with BaseFold, on vanilla circuits (with degree-two multiplications) and circuits with custom gates (of degree-five). Experimental results demonstrate that, with 16 machines, our proof generation is 15.6 × faster on a vanilla circuit, and 16.4 × on a custom one, both of sizes 2^{26}. We also compare HyperFond to deVirgo [19] under the same experimental environment therein, and achieve a 1.45 × speedup. Detailed

evaluations and comparisons can be found in Section 5.

It should be noted in Table 1, however, that HyperFond's polylogarithmic communication comes at the cost of an $O(M)$ overhead on proof size, and could be intolerable in applications requiring tiny proofs and fast verification. That said, we still reckon that HyperFond provides a solution in proving-oriented scenarios, say, proving the correctness of machine learning inference, where proofs are frequently requested by clients, while the bandwidth (typically 100Mbps $\sim$ 1Gbps [32, 33, 34] depending on plans) supports efficient downloading, so proofs of moderate sizes (around 200MB generated by 16 sub-provers in our case) would be acceptable.

1.2 Motivations and Challenges

To develop a distributed SNARK, the basic idea is to distribute a PIOP system and instantiate it with a distributed PCS, as in most previous works. To achieve desirable properties, for the PIOP part, we select HyperPlonk [27] as our cornerstone, because it achieves linear proving time by avoiding frequent FFTs, and supports proof generation for general circuits. As for PCS, we choose to distribute the code-based multilinear PCS from BaseFold [31], as it not only enjoys a transparent setup and post-quantum security, but also provides broader choices of fields benefiting from its field agnosticity. Before delving into our main ideas, let us first briefly review several prior works that HyperFond relies on, to better understand the technical challenges we are faced with during distribution.

The multivariate SumCheck protocol [35]. The multivariate SumCheck protocol is frequently invoked throughout HyperFond. Fix a finite field $\mathbb{F}$. The protocol aims to let a prover $\mathcal{P}$ convince a verifier $\mathcal{V}$ that, with high probability, the sum $\sum_{\mathbf{b}\in B_\mu} f(\mathbf{b})$ is equal to a purported value v, where $f \in \mathcal{F}_\mu^{(\leq d)} := \mathbb{F}^{(\leq d)}[X_1, \cdots, X_\mu]$ is a μ-variate polynomial and $B_\mu := \{0,1\}^\mu$ is the μ-dimension Boolean hypercube. In detail, $\mathcal{P}$ and $\mathcal{V}$ engage in a μ-round interaction, where in round k from μ downto 1, $\mathcal{P}$ computes a univariate polynomial $r_k(X) := \sum_{\mathbf{b}\in B_{k-1}} f(\mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$, and sends it to $\mathcal{V}$, who verifies that $r_k(0) + r_k(1) = r_{k+1}(\alpha_{k+1})$ (for $k = \mu$, this is $r_k(0) + r_k(1) = v$), and that $\deg(r_k) \leq d$. If the verification passes, $\mathcal{V}$ samples and sends $\alpha_k \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}$. In round $k = 1$, after verification $\mathcal{V}$ samples $\alpha_1 \leftarrow_\$ \mathbb{F}$, queries $\mathcal{P}$ for the value $f(\alpha_1, \cdots, \alpha_\mu)$ and checks whether it equals $r_1(\alpha_1)$. If all verifications pass, $\mathcal{V}$ outputs accept.

Featuring linear proving time, SumCheck has gained surging prominence in SNARK constructions. One of the most representative works exploiting this trait is the following.

HyperPlonk [27]. HyperPlonk is a multivariate proof system based on SumCheck. By transferring each building block in Plonk [26] from multiplicative groups to Boolean hypercubes, HyperPlonk achieves linear proving time for general arithmetic circuits and supports *custom* operation $G : \mathbb{F}^2 \to \mathbb{F}$ beyond additions and multiplications.

Roughly speaking, HyperPlonk expresses gate values and wires in an arithmetic circuit by a *gate identity* and a *wiring identity*, respectively. The proofs for both are reduced to SumCheck. The authors further enhance HyperPlonk to HyperPlonk$^+$, by incorporating

a Lookup protocol to prove that some gate values lie in a prescribed table. This protocol is also reduced to SumCheck. Nevertheless, the PCS in HyperPlonk, Orion$^+$ (adapted from Orion [13]), builds on a pairing-friendly field, which exhibits vulnerability against quantum threats and excludes broader choices of fields. In addition, the setup for Orion$^+$ is not transparent. Therefore, our construction can make the best use of HyperPlonk's PIOP system, but should seek other strategies for PCS.

The following work provides a strategy to distribute the SumCheck protocol, and achieves post-quantum security.

deVirgo [19]. Based on SumCheck, deVirgo is a distributed proof system for data-parallel circuits. It is by far the only distributed scheme that enjoys a transparent setup and post-quantum security. The feasibility of distribution results from the observation that $\sum_{\mathbf{b}\in B_\mu} f(\mathbf{b}) = \sum_{\mathbf{b}'\in B_m}\sum_{\mathbf{b}\in B_{\mu-m}} f(\mathbf{b}', \mathbf{b})$, for any integer $m \leq \mu$. Therefore, given identical and parallel subcircuits $\{\mathbf{C}_i\}_{i=0}^{2^m-1}$ that constitute a circuit $\mathbf{C}$, a SumCheck for $f \in \mathcal{F}_\mu^{(\leq d)}$, representing computations on some layer of $\mathbf{C}$, can be split into $M := 2^m$ parallel SumChecks, i.e., each $\mathcal{P}_i$ and $\mathcal{V}$ run SumCheck for $f^{(i)}(\mathbf{X}) := f(\mathbf{bin}_m(i), \mathbf{X}) \in \mathcal{F}_{\mu-m}^{(\leq d)}$ denoting computations on the same layer of $\mathbf{C}_i$, where $\mathbf{bin}_m(i) := (b_1, \cdots, b_m) \in B_m$ is i's μ-bit binary decomposition, for $\mu - m$ rounds with the same randomness $\alpha_\mu, \cdots, \alpha_{\mu-m+1}$. The master $\mathcal{P}_0$ then gathers data from other sub-provers, and runs with $\mathcal{V}$ the rest m rounds of SumCheck for $f'(\mathbf{X}) := f(\mathbf{X}, \alpha_{\mu-m+1}, \cdots, \alpha_\mu) \in \mathcal{F}_m^{(\leq d)}$.

Now that all sub-protocols in HyperPlonk$^{(+)}$ rely on SumCheck, an intuition for distribution is to directly utilize the distributed SumCheck above. However, during the reductions of wiring identity and Lookup to SumCheck, there is an intermediate *ProductCheck*, proving that $\prod_{\mathbf{b}\in B_\mu}(\beta + F(\mathbf{b})) = \prod_{\mathbf{b}\in B_\mu}(\beta + G(\mathbf{b}))$ for $F, G \in \mathcal{F}_\mu^{(\leq d)}$ and a randomly chosen β, for which HyperPlonk constructs a $(\mu+1)$-round interaction between $\mathcal{P}$ and $\mathcal{V}$, and introduces new witness polynomials with identical sizes. If ProductCheck were distributed in this trivial way, linear communication would arise for $\{\mathcal{P}_i\}_{i=0}^{M-1}$ to update their witness polynomials. Therefore, we need an alternative distribution strategy.

The PCS of deVirgo also poses a subtle constraint. To reduce proof size, sub-provers are required to share elements to compute multiple Merkle roots, incurring linear communication costs that might present challenges for deployment on large-scale circuits; it additionally expects that the finite field $\mathbb{F}$ has a proper multiplicative subgroup to conduct frequent FFTs, which is a limit compared to field-agnostic schemes.

Judging from the previous discussions, a transparent and quantum resistant PCS that applies to any sufficiently large field with no specific structure is desired.

BaseFold [31]. BaseFold introduces an *Interactive Oracle Proof of Proximity* (IOPP), validating that a given vector is close to a prescribed code space. It generalizes FRI [25] to any *foldable linear code*, from which an efficient multilinear PCS can be constructed. BaseFold enjoys a transparent setup, post-quantum security, field-agnostic property, and lighter implementations in engineering, compared to pairing-based PCSs. These nice properties

have fostered BaseFold's implementations (e.g., the Ceno [36] project[1]).

In BaseFold, to commit to a multilinear polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$, $\mathcal{P}$ first encodes its coefficients $\mathbf{f} \in \mathbb{F}^{2^\mu}$ to obtain π_μ, and runs an IOPP with $\mathcal{V}$ to convince the latter that π_μ is a valid codeword. To this end, $\mathcal{P}$ generates a list of string oracles $(\pi_\mu, \cdots, \pi_0)$ with sequentially halved lengths, using the randomness $\alpha_\mu, \cdots, \alpha_1$ chosen by $\mathcal{V}$. For the evaluation protocol, to prove that $f(\mathbf{z}) = y$ for $\mathbf{z} \in \mathbb{F}^\mu$ and $y \in \mathbb{F}$, $\mathcal{P}$ and $\mathcal{V}$ run the previous IOPP interleaved with a SumCheck using the same randomness, to ensure that (i) the committed polynomial is indeed f, and that (ii) the evaluation is correct.

The distribution for IOPP or SumCheck is straightforward, because those gradually halved lengths naturally raise fewer communication costs (up to polylogarithmic). The challenging part appears to be the encoding algorithm, which requires μ levels of recursion. A trivial distribution results in a doomed phase where the M sub-provers are stuck, holding intermediate codewords with lengths $O(\frac{2^\mu}{M})$, and forced to exchange them with other sub-provers. This naive approach causes linear communication. Therefore, a primary goal falls on distributing the encoding phase with minimal communication.

1.3 Technical Overview of Our Construction

In this section, we address all the aforementioned issues and show how to efficiently distribute HyperPlonk [27] and BaseFold [31], respectively, to obtain HyperFond with polylogarithmic communication.

The HyperFond PIOP system. At a high level, our PIOP system is akin to that of HyperPlonk, except that the ProductCheck is replaced with a more distribution-friendly one, called *Fractional SumCheck*, as illustrated in Figure 1.

To facilitate distribution, we first utilize the technique exploited in deVirgo [19] to distribute SumCheck, which all subsequent protocols heavily rely on. Given $f \in \mathcal{F}_\mu^{(\leq d)}$, each $\mathcal{P}_i$ determines $f^{(i)}(\mathbf{X}) := f(\mathbf{bin}_m(i), \mathbf{X}) \in \mathcal{F}_{\mu-m}^{(\leq d)}$. In round $\mu \geq k \geq m+1$, $\mathcal{P}_i$ calculates and sends $r_k^{(i)}(X) = \sum_{\mathbf{b} \in B_{k-m-1}} f^{(i)}(\mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$ to $\mathcal{P}_0$, who sums them to derive $r_k(X)$. Then in round $m \geq k \geq 1$, $\mathcal{P}_0$ alone runs with $\mathcal{V}$ to complete SumCheck. By this strategy, major computation during the first $\mu - m$ rounds is distributed to all sub-provers, instead of burdening the master alone. For detailed construction and explanation, please refer to Section 3.1.

With distributed SumCheck in hand, almost all parental protocols in HyperPlonk can be trivially transformed to distributed settings, except the ProductCheck, i.e., checking that $\prod_{\mathbf{b} \in B_\mu} (\beta + F(\mathbf{b})) = \prod_{\mathbf{b} \in B_\mu} (\beta + G(\mathbf{b}))$. To avoid the surge of communication costs, we utilize the *logarithmic derivative* technique introduced in [37] and manage to avoid high communication costs. In more detail, the equation above can be transformed into fractional summations $\sum_{\mathbf{b} \in B_\mu} \frac{1}{\beta + F(\mathbf{b})} = \sum_{\mathbf{b} \in B_\mu} \frac{1}{\beta + G(\mathbf{b})}$. Now, focusing on the denominators, each $\mathcal{P}_i$ completes main computations locally. To further avoid sending extra polynomial

[1] https://github.com/scroll-tech/ceno

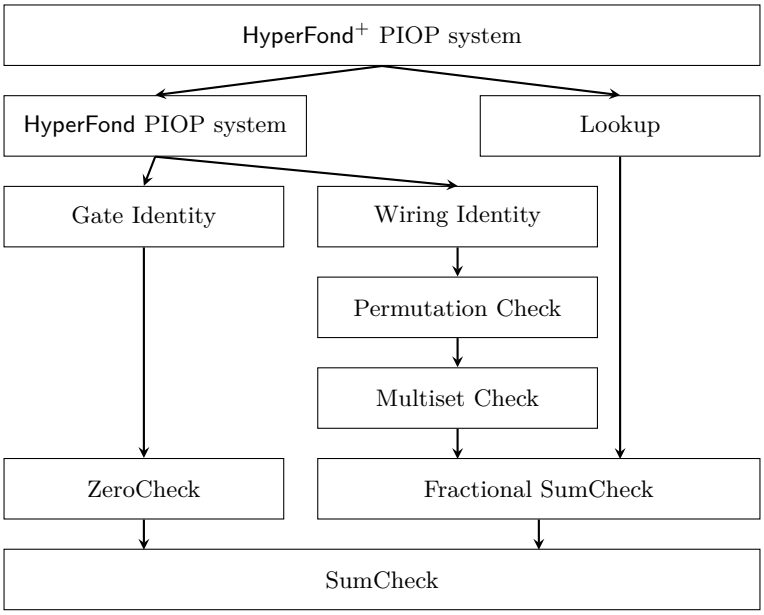

Figure 1: The PIOP system of HyperFond$^{(+)}$.

oracles, we resort to the layered circuit strategy introduced in [30] to support Fractional SumCheck for multilinear polynomials, and generalize it for multivariate ones. We then batch two summations into one as $\sum_{\mathbf{b}\in B_{\mu+1}} \frac{p(\mathbf{b})}{q(\mathbf{b})} = 0$, where $p, q \in \mathcal{F}_{\mu+1}^{(\leq d)}$, to accelerate proof generation. For details, please refer to Section 3.3.1.

The Lookup PIOP in HyperPlonk$^+$ is based on ProductCheck, bearing the similar distribution-unfriendliness as just mentioned. Therefore, we reuse the techniques in [37, 30] to derive a distributed Lookup PIOP. As a bonus, our protocol supports multi-column lookups, i.e., batching k lookups into one with only extra $\log k$ round complexity, instead of $(k-1)\mu$. For details, please refer to Section 3.5.

The HyperFond PCS. In BaseFold [31], to encode $\mathbf{f} \in \mathbb{F}^{2^\mu}$, the encoding algorithm $\mathbf{Enc}_\mu$ partitions $\mathbf{f}$ evenly into $\mathbf{f}_0$ and $\mathbf{f}_1$, invokes $\mathbf{Enc}_{\mu-1}$ for each, obtains $\mathbf{F}_1$ and $\mathbf{F}_2$, respectively, and outputs $(\mathbf{F}_1 + \mathbf{t} \circ \mathbf{F}_2 \| \mathbf{F}_1 + \mathbf{t}' \circ \mathbf{F}_2) \in \mathbb{F}^{\rho^{-1}\cdot 2^\mu}$, where ρ is code rate, $\mathbf{t}, \mathbf{t}' \in \mathbb{F}^{\rho^{-1}\cdot 2^{\mu-1}}$ are random and public vectors, and $\circ$ denotes Hadamard product. For $\mathbf{f} \in \mathbb{F}^{2^0}$, simply output $\mathbf{f} \cdot \mathbf{G}_0$ where $\mathbf{G}_0$ is the public generator matrix.

At first glance, it seems overly restrictive to optimize communication efficiency for such an intricate recursive algorithm. However, we observe that the algorithm essentially dives to the deepest level of recursion, partitions $\mathbf{f} \in \mathbb{F}^{2^\mu}$ into 2^μ equidistant pieces, and encodes each with $\mathbf{G}_0$. After that, as the recursion floats back, in each level the piece number halves and length doubles, until finally the desired codeword is obtained. Therefore, if we allocate $(\mathbf{f}^{(0)} \| \cdots \| \mathbf{f}^{(M-1)})$, to 2^m sub-provers, with $\mathcal{P}_i$ holding $\mathbf{f}^{(i)}$, and let them run

$\mathbf{Enc}_{\mu-m}$ separately, the encoding algorithm can indeed be made highly parallel with no communication until at the m-th level, when all sub-provers are stuck. Hence, we *terminate* the encoding phase at this moment to avoid communication, with $\mathcal{P}_i$ holding $\pi^{(i)}_{\mu-m} := \mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)}) \in \mathbb{F}^{\rho^{-1}\cdot 2^{\mu-m}}$ as a "partial" codeword.

Now, the IOPP phase seems infeasible to start, as it lacks the required codeword π_μ=$\mathbf{Enc}_\mu(\mathbf{f})$. We fix this by noting that the random field elements $\{\alpha_j\}_{j=1}^{\mu}$ are essentially used for linear combinations during IOPP. Specifically, we describe the case of four sub-provers. For $\mathbf{f}$=$(\mathbf{f}^{(0)}\|\cdots\|\mathbf{f}^{(3)})$∈$\mathbb{F}^{2^\mu}$, each $\mathcal{P}_i$ locally computes $\pi^{(i)}_{\mu-2}$. By the completeness of IOPP, $\pi_{\mu-2}$=$(\pi^{(0)}_{\mu-2}$+$\alpha_{\mu-1}$·$\pi^{(1)}_{\mu-2})$+α_μ·$(\pi^{(2)}_{\mu-2}$+$\alpha_{\mu-1}$·$\pi^{(3)}_{\mu-2})$. Yet instead of computing $\pi_{\mu-2}$ as above, which otherwise entails linear communication, each $\mathcal{P}_i$ starts computing $(\pi^{(i)}_{\mu-2},\cdots,\pi^{(i)}_0)$ alone with global randomness $\alpha_{\mu-2}$, $\cdots$, α_1, as if it were committing to a polynomial of size $2^{\mu-2}$. In the final round, $\mathcal{V}$ recovers $\pi_0 = (\pi^{(0)}_0 + \alpha_{\mu-1}\cdots\pi^{(1)}_0) + \alpha_\mu \cdot (\pi^{(2)}_0 + \alpha_{\mu-1}\cdots\pi^{(3)}_0)$, using the same linear relation as in $\pi_{\mu-2}$, which holds because scalar multiplication is commutative with linear operation. $\mathcal{V}$ then checks the correctness of π_0. Note that the number of oracles that $\{\mathcal{P}_i\}_{i=0}^3$ provide accordingly multiplies 4. This strategy generalizes to the case of M sub-provers by skipping the last m levels of recursion and the first m rounds of IOPP, with SumCheck intact to acquire $\{\alpha_j\}_{j=\mu-m+1}^{\mu}$. The oracles $\mathcal{V}$ receives multiply M, so there is an $O(M)$ overhead for proof size or verification time.

In summary, with distributed computation, the communication is *zero* for encoding, and $O(M(\mu-m)^2)$ during IOPP or SumCheck, dropping drastically from linear to polylogarithmic, without affecting intrinsic properties of BaseFold. The proof size and verification time increase to M times, essentially as a compensation for the shrunken communication costs. For details, please refer to Section 4.

1.4 Related Works

Distributed SNARKs. DIZK [18], provided by Wu et al., pioneers the construction of distributed SNARKs. In DIZK, a unique sub-prover called master allocates overall workload to others, and cooperates with them to generate a proof. The master also interacts with the verifier and synchronizes the latter's challenges to all workers. Based on the *Rank-One Constraint System (R1CS)*, DIZK is fully distributed, while its running time is quasilinear in subcircuit size, and communication costs are linear in total circuit size.

Following the footprint of DIZK, Xie et al. propose deVirgo [19] as reviewed in Section 1.2. Subsequently, Liu et al. introduce Pianist [20], a distributed proof system originating from Plonk [26]. Using bivariate polynomials, Pianist breaks each building block in Plonk into multiple ones locally processed by sub-provers, and ultimately yields constant proof size and communication costs. However, the proving time of each sub-prover is still quasilinear in the subcircuit size.

Concurrent to Pianist, Rosenberg et al. propose Hekaton [21] based on R1CS, with a "distribute-and-aggregate" framework that partitions the whole circuit into subcircuits,

proves each locally, and aggregates proofs into one. Each sub-prover checks memory consistency through a "global memory bank", enabling general aggregation where subcircuits share many wires. Nonetheless, each sub-prover still requires quasilinear time to generate a proof, and the setup is trusted.

Recently, Li et al. improve the Pianist proof system, and propose HyperPianist [22] that distributes HyperPlonk [27], and further instantiate it by adapting two PCSs, MKZG [38] and Dory [39], to distributed settings, respectively. Their strategy in the PIOP part is similar to ours in HyperFond, that is, utilize distributed SumCheck from deVirgo [19] and logarithmic derivative from [37], but in a slightly more expensive way that includes computing extra polynomial oracles (please refer to Remark 2 for further discussion). The proving complexity in HyperPianist is linear, and the communication costs are only logarithmic in the total circuit size. However, both PCSs are based on bilinear pairings, hence not quantum resistant.

Concurrent to HyperPianist, Cirrus [23], based on HyperPlonk [27] and MKZG [38], also achieves linear proving time and logarithmic communication costs. It further satisfies *accountability*, allowing the master (coordinator) to identify malicious workers. Another recent work is Soloist [24], which is based on R1CS and KZG [29] and yields constant proof size and communication as Pianist does. The drawbacks of both works, however, remain to be quantum vulnerability.

Collaborative SNARKs. Another line of works [40, 41, 42, 43, 44] focus on preserving witness privacy during distribution. The sub-provers collaborate by sharing witness in a secret form, and utilizing *multi-party computation* techniques to generate proof. Note that the goal of collaborative SNARKs is orthogonal to that of distributed SNARKs, where one is more concerned with privacy, while the other efficiency.

2 Preliminaries

Notations. We use λ to denote the security parameter. A function $f(\lambda)$ is $\mathsf{poly}(\lambda)$ if there exists some $c \in \mathbb{N}$ such that $f(\lambda) = O(\lambda^c)$. If for all $c \in \mathbb{N}, f(\lambda) = o(\lambda^{-c})$, then $f(\lambda)$ is in $\mathsf{negl}(\lambda)$ and is called *negligible*. The probability of $1 - \mathsf{negl}(\lambda)$ is *overwhelming*. Let $\mathbb{F}$ be a finite field so that $\log |\mathbb{F}| = \Omega(\lambda)$. Let $\leftarrow_{\$} S$ denote uniformly random sampling from a finite set S. For $a, b \in \mathbb{N}$ and $a < b$, $[a : b]$ denotes the discrete set $\{a, a+1, \cdots, b\}$. Boldface letters (e.g. $\mathbf{r}, \mathbf{G}$) denote vectors or matrices. For a polynomial $f \in \mathcal{F}_{\mu}^{(\leq d)}$, $[\![f]\!]$ denotes its oracle.

A *multiset* extends the concept of a set, where every element has a positive multiplicity. Two finite multisets are equal if they contain the same elements with identical multiplicities. A *relation* is a set of pairs $(\mathtt{x}; \mathtt{w})$. An *indexed relation* is a set of triples $(\mathtt{i}, \mathtt{x}; \mathtt{w})$, where the index $\mathtt{i}$ is fixed at the setup phase. Denote by $\mathcal{L}(\mathcal{R})$ the language corresponding to the indexed relation $\mathcal{R}$ such that $(\mathtt{i}, \mathtt{x}) \in \mathcal{L}(\mathcal{R})$ if and only if there exists $\mathtt{w}$ such that $(\mathtt{i}, \mathtt{x}; \mathtt{w}) \in \mathcal{R}$.

Multivariate polynomials. For every map $f : B_\mu \to \mathbb{F}$, there exists a unique multilinear polynomial $\tilde{f} \in \mathcal{F}_\mu^{(\leq 1)}$ such that $\tilde{f}(\mathbf{b}) = f(\mathbf{b})$ for all $\mathbf{b} \in B_\mu$. $\tilde{f}$ is called the *multilinear extension* (MLE) of f, and can be further expressed as $\tilde{f}(\mathbf{X}) = \sum_{\mathbf{b}\in B_\mu} f(\mathbf{b}) \cdot \tilde{eq}_{\mathbf{b}}(\mathbf{X})$, where $eq_{\mathbf{b}} : B_\mu \to \mathbb{F}$ maps $\mathbf{b}$ to 1, and others to 0, and $\tilde{eq}_{\mathbf{b}}(\mathbf{X}) := \prod_{i=1}^{\mu} (b_i X_i + (1 - b_i)(1 - X_i)) = \tilde{eq}_{\mathbf{X}}(\mathbf{b})$ is the MLE of $eq_{\mathbf{b}}$. The following result is a central property exploited in proof systems based on multivariate polynomials.

Lemma 1 (Schwartz-Zippel Lemma)**.** *Let $f \in \mathcal{F}_\mu^{(\leq d)}$ be a non-zero polynomial. Let S be any finite subset of $\mathbb{F}$ and $r_1, \cdots, r_\mu \leftarrow_\$ S$ be μ field elements chosen independently and uniformly from S. Then* $\Pr[f(r_1, \cdots, r_\mu) = 0] \leq \frac{d\mu}{|S|}$.

Logarithmic derivatives. Let $p(X) \in \mathbb{F}[X]$ be a univariate polynomial. The *logarithmic derivative* of p is the fractional function $\frac{p'}{p}$, where p' is the formal derivative of p. In particular, for $p(X)$ with the form $\prod_{i=1}^{n} (X + z_i)^{c_i}$, where $z_i \in \mathbb{F}$, and c_i denotes its multiplicity, it follows that $\frac{p'(X)}{p(X)} = \sum_{i=1}^{n} \frac{c_i}{X+z_i}$. This observation is useful in proving set relations, as captured by the following two lemmas.

Lemma 2 (Set Equivalence [37, Lemma 3])**.** *Let $\mathbb{F}$ be a field of characteristic greater than n. Let $\{a_i\}_{i=1}^n, \{b_i\}_{i=1}^n$ be multisets. Then $\prod_{i=1}^{n} (X + a_i) = \prod_{i=1}^{n} (X + b_i)$ if and only if $\sum_{i=1}^{n} \frac{1}{X+a_i} = \sum_{i=1}^{n} \frac{1}{X+b_i}$.*

Lemma 3 (Set Inclusion [37, Lemma 5])**.** *Let $\mathbb{F}$ be a field of characteristic greater than n. Let $\{a_i\}_{i=1}^n, \{b_i\}_{i=1}^n$ be multisets. Then $\{a_i\} \subseteq \{b_i\}$ (ignoring multiplicity), if and only if there exists a sequence of field elements $(c_i)_{i=1}^n$ such that $\sum_{i=1}^{n} \frac{1}{X+a_i} = \sum_{i=1}^{n} \frac{c_i}{X+b_i}$ in the fractional field $\mathbb{F}(X)$. Moreover, $\{a_i\} = \{b_i\}$ (ignoring multiplicity) if and only if $c_i \neq 0$ for all $i \in [1 : n]$.*

Linear codes. A linear error correcting code, with message length k and codeword length n, is an injective map from $\mathbb{F}^k$ to a linear subspace $\mathcal{C} \subseteq \mathbb{F}^n$. $\mathcal{C}$ is associated with a generator matrix $\mathbf{G} \in \mathbb{F}^{k\times n}$ such that the rows of $\mathbf{G}$ form a basis of $\mathcal{C}$, and that the encoding of a vector $\mathbf{v} \in \mathbb{F}^k$ is $\mathbf{v} \cdot \mathbf{G}$. The *minimum Hamming distance* of a code is the minimum number of different entries between any $\mathbf{c}_1 \neq \mathbf{c}_2 \in \mathcal{C}$. If $\mathcal{C}$ has a minimum Hamming distance $d \in [0 : n]$, we say that $\mathcal{C}$ is an $[n, k, d]$ *code*, and use $\Delta_{\mathcal{C}}$ to denote $\frac{d}{n}$, the *relative minimum distance*. Specifically, if $d = n - k + 1$, then $\mathcal{C}$ is *maximum distance separable* (MDS).

2.1 Proofs and Arguments

PoKs and ARKs. We adapt the definitions of Proofs and Arguments of Knowledge (PoKs, ARKs) from [27].

Definition 1 (PoKs and ARKs [27])**.** *An* argument of knowledge *(ARK) $\Pi = (\mathsf{Setup}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ for an indexed relation $\mathcal{R}$ with knowledge error $\delta : \mathbb{N} \to [0, 1]$ runs as follows, where given an index $\mathtt{i}$, common input $\mathtt{x}$ and prover witness*

$\mathbb{w}$, *the deterministic indexer outputs* $(\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathbb{i})$ *and the output of the verifier is denoted by the random variable* $\langle \mathcal{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathsf{vp}, \mathbb{x}) \rangle$.

Π *is* perfectly complete *if for all* $(\mathbb{i}, \mathbb{x}; \mathbb{w}) \in \mathcal{R}$,

$$\Pr\left[\langle \mathcal{P}(\mathsf{pp}, \mathbb{x}, \mathbb{w}), \mathcal{V}(\mathsf{vp}, \mathbb{x}) \rangle = 1 \,\middle|\, \begin{array}{c} \mathsf{gp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{vp}, \mathsf{pp}) \leftarrow \mathcal{I}(\mathsf{gp}, \mathbb{i}) \end{array}\right] = 1.$$

Π *is* δ-sound *(adaptive) if for every pair of probabilistic polynomial time (PPT) adversarial algorithm* $(\mathcal{A}_1, \mathcal{A}_2)$,

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

Π *is* computationally sound *if* Π *is* δ*-sound for negligible* δ*. If* Π *is* δ*-sound for negligible* δ *and unbounded* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*, then* Π *is* statistically sound.

Π *is* δ-knowledge sound *if 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 algorithm* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

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

for δ *negligible in* λ*. If the adversary is unbounded, then the ARK is called a* proof of knowledge *(PoK).*

Π *is* public-coin *if all messages from* $\mathcal{V}$ *can be computed as a deterministic function given a random public input.*

SNARKs. An ARK is *succinct* if the proof size satisfies $|\pi| = \mathsf{poly}(\lambda, \log|\mathbb{w}|)$ and the verification time is $\mathsf{poly}(\lambda, |\mathbb{x}|, \log|\mathbb{w}|)$. The interactiveness of a public-coin ARK can be removed via the Fiat-Shamir transform [45]. It is called a *SNARK* if further satisfying succinctness.

IOPs. HyperFond is constructed from an information-theoretic proof system called *interactive oracle proof* (IOP), where the verifier possesses oracle access to prover messages. We recall its formal definition below, followed by two special IOPs, PIOP and IOPP.

Definition 2 (Interactive Oracle Proofs (IOPs) [46, 47, 48, 31])**.** *A* k*-round public-coin* interactive oracle proof $\mathsf{IOP} = (\mathcal{P}, \mathcal{V})$ *for a relation* $\mathcal{R}$ *runs as follows. Initially* $\mathcal{P}$ *sends an oracle string* π_0*. In each round* $j \in [1:k]$*, the verifier* $\mathcal{V}$ *samples and sends a random challenge* α_j*, and the prover replies with an oracle string* π_j*. After* k *rounds of interaction the verifier queries some entries of the oracle strings* $(\pi_0, \cdots, \pi_k)$ *and outputs a bit* b.

IOP *is* complete *if for every* $(\mathbb{x}; \mathbb{w}) \in \mathcal{R}$,

$$\Pr\left[\mathcal{V}^{(\pi_0,\cdots,\pi_k)}(\mathbb{x}, \alpha_1, \cdots, \alpha_k) = 1 \;\middle|\; \begin{array}{c} \pi_0 \leftarrow \mathcal{P}(\mathbb{x}; \mathbb{w}) \\ \pi_1 \leftarrow \mathcal{P}(\mathbb{x}, \alpha_1; \mathbb{w}) \\ \cdots \\ \pi_k \leftarrow \mathcal{P}(\mathbb{x}, \alpha_1, \cdots, \alpha_k; \mathbb{w}) \end{array}\right] = 1,$$

taken over $\{\alpha_i \leftarrow_{\$} \mathbb{F}\}_{i=1}^{k}$ *(similarly hereinafter).*

IOP *is* sound *if for any* $\mathbb{x} \notin \mathcal{L}(\mathcal{R})$ *and any unbounded adversary* $\mathcal{A}$,

$$\Pr\left[\mathcal{V}^{(\pi_0,\cdots,\pi_k)}(\mathbb{x}, \alpha_1, \cdots, \alpha_k) = 1 \;\middle|\; \begin{array}{c} \pi_0 \leftarrow \mathcal{A}(\mathbb{x}) \\ \pi_1 \leftarrow \mathcal{A}(\mathbb{x}, \alpha_1) \\ \cdots \\ \pi_k \leftarrow \mathcal{A}(\mathbb{x}, \alpha_1, \cdots, \alpha_k) \end{array}\right] = \mathsf{negl}(\lambda).$$

Definition 3 (Polynomial Interactive Oracle Proofs (PIOPs) [27])**.** *A* polynomial interactive oracle proof *is an IOP in which the prover messages are polynomial oracles. For a* μ*-variate polynomial* f *over the field* $\mathbb{F}$*, its oracle* $[\![f]\!]$ *can be queried at an arbitrary point* $\mathbf{r} \in \mathbb{F}^{\mu}$ *to evaluate* $f(\mathbf{r})$*.*

Definition 4 (Interactive Oracle Proof of Proximity (IOPP) [49, 31])**.** *A public-coin IOP* $(\mathcal{P}, \mathcal{V})$ *for relation* $\mathcal{R}$ *is an* IOP of Proximity *if it satisfies IOP completeness and* $\nu(\cdot)$-IOPP soundness*: for every* $(\mathbb{x}; \mathbb{w})$ *where* $\mathbb{w}$ *is* δ*-far (in relative Hamming distance) from any* $\mathbb{w}'$ *so* $(\mathbb{x}; \mathbb{w}') \in \mathcal{R}$*, it holds that for any unbounded adversary* $\mathcal{A}$*,*

$$\Pr\left[\mathcal{V}^{(\pi_0,\cdots,\pi_k)}(\mathbb{x}, \alpha_1, \cdots, \alpha_k) = 1 \;\middle|\; \begin{array}{c} \pi_0 \leftarrow \mathcal{A}(\mathbb{x}; \mathbb{w}) \\ \pi_1 \leftarrow \mathcal{A}(\mathbb{x}, \alpha_1; \mathbb{w}) \\ \cdots \\ \pi_k \leftarrow \mathcal{A}(\mathbb{x}, \alpha_1, \cdots, \alpha_k; \mathbb{w}) \end{array}\right] \leq \nu(\delta).$$

2.2 Polynomial Commitment Schemes

Definition 5 (Polynomial Commitment Scheme [31])**.** *A* polynomial commitment scheme PCS *for multilinear polynomial is a tuple of polynomial time algorithms* (Setup, Commit, $\mathsf{VrfyOpen}$, Eval) *described as follows:*

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^{\lambda}, \mu)$*: takes security parameter* λ *and* $\mu \in \mathbb{N}$ *(number of variables), and outputs public parameter* pp*.*

$(C, r) \leftarrow \mathsf{Commit}(\mathsf{pp}, f)$*: takes a multilinear polynomial* f*, and outputs a commitment* C *with a hint* r*.*

$b \leftarrow \mathsf{VrfyOpen}(\mathsf{pp}, C, f, r)$*: takes a commitment* C*, a multilinear polynomial* f*, and a hint* r*, and outputs a bit* b*.*

$b \leftarrow \mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, y; f, r)$*: this is a protocol between the prover* $\mathcal{P}$ *and the verifier* $\mathcal{V}$ *with public inputs, a commitment* C*, an evaluation point* $\mathbf{z} \in \mathbb{F}^{\mu}$ *and a value* $y \in \mathbb{F}$*.* $\mathcal{P}$

additionally possesses a multilinear polynomial f and a hint r, and wants to convince $\mathcal{V}$ that C is a commitment of f and that $f(\mathbf{z}) = y$. $\mathcal{V}$ outputs a bit b *at the end of the protocol.*

PCS *is* complete *if for any $f \in \mathcal{F}_\mu^{(\leq 1)}$ and $\mathbf{z} \in \mathbb{F}^\mu$,*

$$\Pr\left[\mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, f(\mathbf{z}); f, r) = 1 \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mu) \\ (C, r) \leftarrow \mathsf{Commit}(\mathsf{pp}, f) \end{array}\right] = 1.$$

PCS *is* binding *if for any $\mu \in \mathbb{N}$ and PPT adversary $\mathcal{A}$,*

$$\Pr\left[b_0 = b_1 \wedge f \neq g \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \mu) \\ (C, f, g, r_0, r_1) \leftarrow \mathcal{A}(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{VrfyOpen}(\mathsf{pp}, C, f, r_0) \\ b_1 \leftarrow \mathsf{VrfyOpen}(\mathsf{pp}, C, g.r_1) \end{array}\right] = \mathsf{negl}(\lambda).$$

PCS *is* knowledge sound *if there exists an extractor $\mathcal{E}$ such that for all PPT adversary $\mathcal{A}$ and any instance $(C, \mathbf{z}, y)$ where $\Pr[\mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}, y)\langle \mathcal{A}, \mathcal{V}\rangle = 1] \geq \epsilon(\lambda)$ for some non-negligible function $\epsilon(\cdot)$, $\mathcal{E}^{\mathcal{A}}$ outputs witnesses f and r in expected polynomial time such that $f(\mathbf{z}) = y$ and $\mathsf{VrfyOpen}(\mathsf{pp}, C, f, r) = 1$ overwhelmingly.*

3 A Toolbox for Multivariate Polynomials in Distributed Manner

In this section, we follow the framework presented in Figure 1 to sequentially introduce several distributed PIOPs, and analyze their security and complexity. Our constraint system, which is similar to that of HyperPlonk, is presented in the end of this section.

3.1 Distributed SumCheck PIOP

As reviewed in Section 1.2, the SumCheck protocol [35] shows that the sum of the evaluations of a witness multivariate polynomial over a Boolean hypercube is equal to some claimed value, captured by the following relation.

Definition 6 (SumCheck Relation)**.** *The relation $\mathcal{R}_{\text{Sum}}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = ((s, [\![f]\!]); f)$, where $f \in \mathcal{F}_\mu^{(\leq d)}$, such that $\sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) = s$.*

To distribute SumCheck, in round $k > m$, we allocate the first m bits of $\mathbf{b} \in B_k$ to store $\mathbf{bin}_m(i)$, so $\mathcal{P}_i$ locally computes $r_k^{(i)}(X) := \sum_{\mathbf{b} \in B_{k-m-1}} f(\mathbf{bin}_m(i), \mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$. $\mathcal{P}_0$ then gathers and sums them up to derive $r_k(X)$ and sends to $\mathcal{V}$.

HyperPlonk [27] further exploits the inner structure of high degree multivariate polynomials, that is $f(\mathbf{X}) = h(g_1(\mathbf{X}), \cdots, g_c(\mathbf{X})) \in \mathcal{F}_\mu^{(\leq d)}$, where $g_1, \cdots, g_c \in \mathcal{F}_\mu^{(\leq 1)}$, $c = O(1)$, and h is a c-variate polynomial with total degree $\leq d$. In such a case, $\mathcal{P}$ processes $\{g_j\}_{j=1}^c$ separately, and evaluates f utilizing the *dynamic programming* technique

Protocol 1: Distributed SumCheck PIOP. Given a tuple $(\mathbb{x}, \mathbb{w}) = ((s, [\![f]\!]); f)$ for a multivariate polynomial $f(\mathbf{X}) = h(g_1(\mathbf{X}), \cdots, g_c(\mathbf{X})) \in \mathcal{F}_\mu^{(\leq d)}$, where $g_1, \cdots, g_c \in \mathcal{F}_\mu^{(\leq 1)}$, $c = O(1)$, and h is a c-variate polynomial with total degree $\leq d$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $\sum_{\mathbf{b}\in B_\mu} f(\mathbf{b}) = s$ as follows.

1. $\mathcal{P}_i$ computes $f^{(i)}(\mathbf{X}) = f(\mathbf{bin}_m(i), \mathbf{X})$ and derives $\{g_j^{(i)}(\mathbf{X}) = g_j(\mathbf{bin}_m(i), \mathbf{X})\}_{j=1}^c$. It also maintains $\{A_j^{(i)} : \mathbf{bin}_m(i) \times B_{\mu-m} \to \mathbb{F}\}_{j=1}^c$, denoting the evaluations of $\{g_j^{(i)}\}_{j=1}^c$ over $B_{\mu-m}$.
2. In round $k = \mu, \mu-1, \cdots, m+1$,
• $\mathcal{P}_i$ computes $r_{k,j}^{\mathbf{b},(i)}(X) = A_j^{(i)}(\mathbf{bin}_m(i), \mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$, $\forall\ \mathbf{b} \in B_{k-m-1}$ and $j \in [1:c]$, and obtains $r_k^{(i)}(X) := \sum_{\mathbf{b}\in B_{k-m-1}} h(r_{k,1}^{\mathbf{b},(i)}(X), \cdots, r_{k,c}^{\mathbf{b},(i)}(X))$. $\mathcal{P}_0$ sums them up to obtain $r_k(X)$ and forwards to $\mathcal{V}$.
• $\mathcal{V}$ checks that $r_k(0) + r_k(1) = r_{k+1}(\alpha_{k+1})$ (for $k = \mu$ this is $r_\mu(0) + r_\mu(1) = s$) and that $\deg(r_k) \leq d$. If it passes, $\mathcal{V}$ samples and sends $\alpha_k \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ then transmits α_k to other sub-provers.
• $\mathcal{P}_i$ updates $A_j^{(i)}(\mathbf{bin}_m(i), \mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu) \leftarrow A_j^{(i)}(\mathbf{bin}_m(i), \mathbf{b}, \alpha_k, \alpha_{k+1}, \cdots, \alpha_\mu)$ for each $\mathbf{b} \in B_{k-m-1}$ and $j \in [1:c]$.
3. $\{\mathcal{P}_i\}_{i=1}^{M-1}$ send $\{A_j^{(i)}(\mathbf{bin}_m(i), \alpha_k, \cdots, \alpha_\mu)\}_{j=1}^c$ to the master $\mathcal{P}_0$, who alone completes the following with $\mathcal{V}$.
4. In round $k = m, m-1, \cdots, 1$,
• $\forall \mathbf{b} \in B_{k-1}$ and $j \in [1:c]$, $\mathcal{P}_0$ computes $r_{k,j}^{\mathbf{b}}(X)$, obtains $r_k^{\mathbf{b}}(X)$, then $r_k(X)$, and forwards to $\mathcal{V}$.
• $\mathcal{V}$ checks that $r_k(0) + r_k(1) = r_{k+1}(\alpha_{k+1})$ and that $\deg(r_k) \leq d$. If the check passes, $\mathcal{V}$ samples and sends $\alpha_k \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ updates $A_j(\mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu) \leftarrow A_j(\mathbf{b}, \alpha_k, \alpha_{k+1}, \cdots, \alpha_\mu)$ for each $\mathbf{b} \in B_{k-1}$ and $j \in [1:c]$.
5. Finally, $\mathcal{V}$ queries $[\![f]\!]$ on $(\alpha_1, \cdots, \alpha_\mu)$ and accepts if $f(\alpha_1, \cdots, \alpha_\mu) = r_1(\alpha_1)$.

Figure 2: The distributed SumCheck PIOP.

in [50, 8]. Specifically, $\mathcal{P}$ maintains maps $\{A_j : B_\mu \to \mathbb{F}\}_{j=1}^c$, the evaluations of $\{g_j\}_{j=1}^c$ over B_μ. In each round k of SumCheck, $\mathcal{P}$ computes $r_{k,j}^{\mathbf{b}}(X) := A_j(\mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$ $:= (1-X) \cdot A_j(\mathbf{b}, 0, \alpha_{k+1}, \cdots, \alpha_\mu) + X \cdot A_j(\mathbf{b}, 1, \alpha_{k+1}, \cdots, \alpha_\mu)$ for each j, and derives $r_k^{\mathbf{b}}(X) := h(r_{k,1}^{\mathbf{b}}(X), \cdots, r_{k,c}^{\mathbf{b}}(X))$ using symbolic multiplication. The univariate polynomial r_k is then obtained by summing all $r_k^{\mathbf{b}}$'s up. On receiving $\mathcal{V}$'s challenge α_k, $\mathcal{P}$ updates $A_j(\mathbf{b}, X, \alpha_{k+1}, \cdots, \alpha_\mu)$ to be $A_j(\mathbf{b}, \alpha_k, \alpha_{k+1}, \cdots, \alpha_\mu)$.

The same distribution approach can be generalized for such case, i.e., let $\mathcal{P}_i$ process $g_j^{(i)}(\mathbf{X}) := g_j(\mathbf{bin}_m(i), \mathbf{X})$ for $1 \leq j \leq c$, and later utilize symbolic multiplication algorithm

Protocol 2: Distributed ZeroCheck PIOP. Given $(\mathbb{x}, \mathbb{w}) = ((\llbracket f \rrbracket); f)$, where $f \in \mathcal{F}_\mu^{(\leq d)}$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $f(\mathbf{b}) = 0$ for all $\mathbf{b} \in B_\mu$ by invoking Protocol 1 to prove that $((0, \llbracket \hat{f} \rrbracket); \hat{f}) \in \mathcal{R}_{\text{Sum}}$, where $\hat{f}(\mathbf{X}) := f(\mathbf{X}) \cdot \tilde{eq}_{\mathbf{r}}(\mathbf{X})$, for a $\mathbf{r} \leftarrow_{\$} \mathbb{F}^\mu$ sampled by $\mathcal{V}$.

Figure 3: The distributed ZeroCheck PIOP.

to obtain evaluations of f. We present the distributed SumCheck PIOP for high degree polynomials in Protocol 1.

3.2 Distributed ZeroCheck PIOP

Definition 7 (ZeroCheck Relation)**.** *The relation $\mathcal{R}_{\text{Zero}}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = ((\llbracket f \rrbracket); f)$ where $f \in \mathcal{F}_\mu^{(\leq d)}$ and $f(\mathbf{b}) = 0$ for all $\mathbf{b} \in B_\mu$.*

To prove the ZeroCheck relation for a polynomial f, by Lemma 1 it suffices to show that $f(\mathbf{r}) = 0$ for a random point $\mathbf{r} \in \mathbb{F}^\mu$, which can be further proved by multiplying a multilinear polynomial $\tilde{eq}_{\mathbf{r}}$ to f and running a SumCheck to show that the summation of its evaluations over B_μ is 0. We present the distributed ZeroCheck PIOP in Protocol 2.

3.3 Distributed Multiset Check PIOP

Definition 8 (Multiset Check Relation)**.** *For any $k \geq 1$, the relation $\mathcal{R}_{\text{MSet}}^k$ consists of tuples $(\mathbb{x}; \mathbb{w}) = ((\llbracket f_1 \rrbracket, \cdots, \llbracket f_k \rrbracket, \llbracket g_1 \rrbracket, \cdots, \llbracket g_k \rrbracket); (f_1, \cdots, f_k, g_1, \cdots, g_k))$, where $f_j, g_j \in \mathcal{F}_\mu^{(\leq d)}$, such that $\{[f_1(\mathbf{b}), \cdots, f_k(\mathbf{b})]\}_{\mathbf{b} \in B_\mu} = \{[g_1(\mathbf{b}), \cdots, g_k(\mathbf{b})]\}_{\mathbf{b} \in B_\mu}$.*

As in [27, Section 3.4], combine $\{f_j\}_{j=1}^k$ and $\{g_j\}_{j=1}^k$ with a universal hash family, that is $F := f_1 + \sum_{j=2}^k \gamma_j f_j$ and $G := g_1 + \sum_{j=2}^k \gamma_j g_j$, where $\gamma_2, \cdots, \gamma_k \leftarrow_{\$} \mathbb{F}$, and now it suffices to show that $\{F(\mathbf{b})\}_{\mathbf{b} \in B_\mu} = \{G(\mathbf{b})\}_{\mathbf{b} \in B_\mu}$. This is then reduced to proving the equality of two grand products $\prod_{\mathbf{b} \in B_\mu} (\beta + F(\mathbf{b})) = \prod_{\mathbf{b} \in B_\mu} (\beta + G(\mathbf{b}))$ for some $\beta \leftarrow_{\$} \mathbb{F}$ chosen by $\mathcal{V}$. By Lemma 2, Papini [37] shows that it suffices to prove $\sum_{\mathbf{b} \in B_\mu} \frac{1}{\beta + F(\mathbf{b})} = \sum_{\mathbf{b} \in B_\mu} \frac{1}{\beta + G(\mathbf{b})}$, the equality of fractional summations. The equation can be further proved using layered circuit technique introduced in [30]. The immediate advantage is that all polynomial oracles are sent at the beginning. We elaborate on this as follows.

3.3.1 The Fractional SumCheck PIOP

Definition 9 (Fractional SumCheck Relation)**.** *The relation $\mathcal{R}_{\text{FSum}}$ is the set of all tuples $(\mathbb{x}; \mathbb{w}) = ((s, \llbracket p \rrbracket, \llbracket q \rrbracket); p, q)$ where $p(\mathbf{X}), q(\mathbf{X}) \in \mathcal{F}_\mu^{(\leq d)}$ so that $\sum_{\mathbf{b} \in B_\mu} \frac{p(\mathbf{b})}{q(\mathbf{b})} = s$.*

Protocol 3: Distributed Fractional SumCheck PIOP. Given $(\mathbb{x}; \mathbb{w}) = ((s, [\![p]\!], [\![q]\!]); (p, q))$ where $p, q \in \mathcal{F}_\mu^{(\leq d)}$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $\sum_{\mathbf{b}\in B_\mu} \frac{p(\mathbf{b})}{q(\mathbf{b})} = s$ as follows.
1. In round $k = 1$, $\mathcal{P}_0$ sends $\mathcal{V}$ claimed values $\tilde{p}_1(0)$, $\tilde{p}_1(1)$, $\tilde{q}_1(0)$, $\tilde{q}_1(1)$. $\mathcal{V}$ first checks that $\tilde{p}_1(0) \cdot \tilde{q}_1(1) + \tilde{p}_1(1) \cdot \tilde{q}_1(0) = s \cdot \tilde{q}_1(0) \cdot \tilde{q}_1(1)$. If the check passes, $\mathcal{V}$ samples a random $r_1 \leftarrow_\$ \mathbb{F}$ and sends it to $\mathcal{P}_0$.
2. In round $k = 2, \cdots, m$,
• $\mathcal{V}$ samples and sends $\lambda_{k-1} \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$. Then $\mathcal{P}_0$ computes the $(k-1)$-variate polynomial $h_{k-1}(\mathbf{X})$ as in equation 1.
• $\mathcal{P}_0$ and $\mathcal{V}$ run a classic SumCheck protocol to show that $((\tilde{p}_{k-1}(\mathbf{r}_{k-1}) + \lambda_{k-1} \cdot \tilde{q}_{k-1}(\mathbf{r}_{k-1}), [\![h_{k-1}]\!]); h_{k-1}) \in \mathcal{R}_{\text{Sum}}$, where in the final round $\mathcal{P}_0$ sends $\mathcal{V}$ $\tilde{p}_k(\boldsymbol{\alpha}_{k-1}, 0)$, $\tilde{p}_k(\boldsymbol{\alpha}_{k-1}, 1)$, $\tilde{q}_k(\boldsymbol{\alpha}_{k-1}, 0), \tilde{q}_k(\boldsymbol{\alpha}_{k-1}, 1)$.
• $\mathcal{V}$ samples and sends $r_k \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$. Let the randomness during the SumCheck be $\boldsymbol{\alpha}_{k-1}$ and set $\mathbf{r}_k := (\boldsymbol{\alpha}_{k-1}, r_k)$.
3. In round $k = m+1, \cdots, \mu$,
• $\mathcal{V}$ samples and sends $\lambda_{k-1} \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$, who transmits λ_{k-1} to $\{\mathcal{P}_i\}_{i=1}^{M-1}$. $\mathcal{P}_i$ computes $h_{k-1}^{(i)}(\mathbf{X}) := h_{k-1}(\mathbf{bin}_m(i), \mathbf{X})$.
• $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 1 to show $((\tilde{p}_{k-1}(\mathbf{r}_{k-1}) + \lambda_k \cdot \tilde{q}_{k-1}(\mathbf{r}_{k-1}), [\![h_{k-1}]\!]); h_{k-1}) \in \mathcal{R}_{\text{Sum}}$, where in the final round $\mathcal{P}_0$ sends $\mathcal{V}$ $\tilde{p}_k(\boldsymbol{\alpha}_{k-1}, 0)$, $\tilde{p}_k(\boldsymbol{\alpha}_{k-1}, 1)$, $\tilde{q}_k(\boldsymbol{\alpha}_{k-1}, 0)$, $\tilde{q}_k(\boldsymbol{\alpha}_{k-1}, 1)$.
• $\mathcal{V}$ samples and sends $r_k \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$. $\mathcal{P}_0$ then transmits r_k to other sub-provers. Let the randomness during the distributed SumCheck be $\boldsymbol{\alpha}_{k-1}$; set $\mathbf{r}_k := (\boldsymbol{\alpha}_{k-1}, r_k)$.
4. Finally, $\mathcal{V}$ samples and sends $\lambda_\mu \leftarrow_\$ \mathbb{F}$ to $\mathcal{P}_0$, who transmits λ_μ to $\{\mathcal{P}_i\}_{i=1}^{M-1}$. $\mathcal{P}_i$ computes $h_\mu^{(i)}$, where $h_\mu(\mathbf{X}) := \tilde{eq}_{\mathbf{X}}(\mathbf{r}_\mu) \cdot (p(\mathbf{X}) + \lambda_\mu q(\mathbf{X}))$. Then $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 1 to show $((\tilde{p}_\mu(\mathbf{r}_\mu) + \lambda_\mu \cdot \tilde{q}_\mu(\mathbf{r}_\mu), [\![h_\mu]\!]); h_\mu) \in \mathcal{R}_{\text{Sum}}$.

Figure 4: The distributed Fractional SumCheck PIOP.

To prove this relation, note that the summations are fully determined by the values of p and q over B_μ, so we can linearize them respectively and postpone consistency check at the final round. To this end, we reduce fractions to a common denominator in each round, and prove with a layered circuit. To be more specific, let layer 0 be the output layer, and layer μ the input layer. In layer μ, for all $\mathbf{b} \in B_\mu$ define $\tilde{p}_\mu(\mathbf{b}) = p(\mathbf{b})$ and $\tilde{q}_\mu(\mathbf{b}) = q(\mathbf{b})$. Then in layer $k \in [0 : \mu - 1]$, for all $\mathbf{b} \in B_k$ compute the numerator $\tilde{p}_k(\mathbf{b}) = \tilde{p}_{k+1}(\mathbf{b}, 0) \cdot \tilde{q}_{k+1}(\mathbf{b}, 1) + \tilde{p}_{k+1}(\mathbf{b}, 1) \cdot \tilde{q}_{k+1}(\mathbf{b}, 0)$, and the denominator $\tilde{q}_k(\mathbf{b}) = \tilde{q}_{k+1}(\mathbf{b}, 0) \cdot \tilde{q}_{k+1}(\mathbf{b}, 1)$.

Now we sketch how to prove the relation $\mathcal{R}_{\text{FSum}}$. Given a tuple $(\mathbb{x}, \mathbb{w}) = ((s, [\![p]\!], [\![q]\!]); p, q)$, $\mathcal{P}$ wants to convince $\mathcal{V}$ that $\sum_{\mathbf{b}\in B_\mu} \frac{p(\mathbf{b})}{q(\mathbf{b})} = s$. Having prepared the aforementioned

layered circuit, i.e., $\{\tilde{p}_k, \tilde{q}_k\}_{k=0}^{\mu}$, $\mathcal{P}$ first sends $\tilde{p}_1(0), \tilde{p}_1(1), \tilde{q}_1(0), \tilde{q}_1(1)$ to $\mathcal{V}$. $\mathcal{V}$ checks that $\tilde{p}_1(0)\cdot\tilde{q}_1(1)+\tilde{p}_1(1)\cdot\tilde{q}_1(0) = s\cdot\tilde{q}_1(0)\cdot\tilde{q}_1(1)$. If the check passes, $\mathcal{V}$ samples and sends $r_1 \leftarrow_{\$} \mathbb{F}$ to $\mathcal{P}$. Now, it suffices for $\mathcal{P}$ to convince $\mathcal{V}$ of the values $\tilde{p}_1(r_1)$ and $\tilde{q}_1(r_1)$. In layer $k \in [1:\mu-1]$, to confirm the values of $\tilde{p}_k(\mathbf{r}_k)$ and $\tilde{q}_k(\mathbf{r}_k)$ from the previous layer, $\mathcal{P}$ and $\mathcal{V}$ should run SumCheck protocols for the numerator, $\tilde{eq}_{\mathbf{X}}(\mathbf{r}_k)(\tilde{p}_{k+1}(\mathbf{X},0)\tilde{q}_{k+1}(\mathbf{X},1)+\tilde{p}_{k+1}(\mathbf{X},1)\tilde{q}_{k+1}(\mathbf{X},0))$, and the denominator, $\tilde{eq}_{\mathbf{X}}(\mathbf{r}_k)\tilde{q}_{k+1}(\mathbf{X},0)\tilde{q}_{k+1}(\mathbf{X},1)$, respectively, where the desired sums are exactly $\tilde{p}_k(\mathbf{r}_k)$ and $\tilde{q}_k(\mathbf{r}_k)$. These two SumChecks are then batched into one for

$$\begin{aligned} h_k(\mathbf{X}) := \tilde{eq}_{\mathbf{X}}(\mathbf{r}_k) \cdot [\lambda_k \cdot \tilde{q}_{k+1}(\mathbf{X},0) \cdot \tilde{q}_{k+1}(\mathbf{X},1) \\ + \tilde{p}_{k+1}(\mathbf{X},0)\tilde{q}_{k+1}(\mathbf{X},1) + \tilde{p}_{k+1}(\mathbf{X},1)\tilde{q}_{k+1}(\mathbf{X},0)], \end{aligned} \tag{1}$$

where $\lambda_k \leftarrow_{\$} \mathbb{F}$ is chosen by $\mathcal{V}$. The claimed value now becomes $\tilde{p}_k(\mathbf{r}_k)+\lambda_k\cdot\tilde{q}_k(\mathbf{r}_k)$. However, $\mathcal{V}$ queries $\mathcal{P}$ at a random point $\boldsymbol{\alpha}_k \in \mathbb{F}^k$ in the final round of the batched SumCheck, whose values depend on $\tilde{p}_{k+1}(\boldsymbol{\alpha}_k,0)$, $\tilde{p}_{k+1}(\boldsymbol{\alpha}_k,1)$, $\tilde{q}_{k+1}(\boldsymbol{\alpha}_k,0)$, and $\tilde{q}_{k+1}(\boldsymbol{\alpha}_k,1)$. To this end, $\mathcal{V}$ first acquires their values from $\mathcal{P}$ to finish the preceding SumCheck. Right after that, $\mathcal{V}$ samples $r_{k+1} \leftarrow_{\$} \mathbb{F}$ and sends it to $\mathcal{P}$. Let $\mathbf{r}_{k+1} := (\boldsymbol{\alpha}_k, r_{k+1})$. Now, it suffices for $\mathcal{P}$ to convince $\mathcal{V}$ of the values of $\tilde{p}_{k+1}(\mathbf{r}_{k+1})$ and $\tilde{q}_{k+1}(\mathbf{r}_{k+1})$.

In layer μ, the claims for $\tilde{p}_0$ and $\tilde{q}_0$ are reduced to $\tilde{p}_\mu(\mathbf{r}_\mu)$ and $\tilde{q}_\mu(\mathbf{r}_\mu)$, which can be further reduced to a batched SumCheck for $h_\mu(\mathbf{X}) := \tilde{eq}_{\mathbf{X}}(\mathbf{r}_\mu)\cdot(p(\mathbf{X})+\lambda_\mu q(\mathbf{X}))$, where λ_μ is the batching randomness sampled by $\mathcal{V}$.

The distributed PIOP for $\mathcal{R}_{\text{FSum}}$ is shown in Protocol 3.

3.3.2 Merging Fractional SumCheck into Multiset Check

Equipped with a distributed PIOP for $\mathcal{R}_{\text{FSum}}$, to prove $\sum_{\mathbf{b}\in B_\mu}\frac{1}{\beta+F(\mathbf{b})} = \sum_{\mathbf{b}\in B_\mu}\frac{1}{\beta+G(\mathbf{b})}$, we define

$$\begin{aligned} p(\mathbf{X}, y) &:= -\tilde{eq}_0(y)\cdot 1 + \tilde{eq}_1(y)\cdot 1 \text{ and} \\ q(\mathbf{X}, y) &:= \tilde{eq}_0(y)(\beta+G(\mathbf{X})) + \tilde{eq}_1(y)(\beta+F(\mathbf{X})) \end{aligned} \tag{2}$$

and batch them into one as $\sum_{\mathbf{b}\in B_{\mu+1}}\frac{p(\mathbf{b})}{q(\mathbf{b})} = 0$, so invoking the Fractional SumCheck once suffices. We formally present the distributed Multiset Check PIOP in Protocol 4.

Remark 1: The virtual oracle queries for $[\![p]\!]$ and $[\![q]\!]$ at a point $(\alpha_1, \cdots, \alpha_{\mu+1}) := (\boldsymbol{\alpha}, \alpha_{\mu+1})$ can be represented as

$$\begin{aligned} p(\boldsymbol{\alpha}) &:= -\tilde{eq}_0(\alpha_{\mu+1})\cdot 1 + \tilde{eq}_1(\alpha_{\mu+1})\cdot 1, \text{ and} \\ q(\boldsymbol{\alpha}) &:= \tilde{eq}_0(\alpha_{\mu+1})(\beta+G(\boldsymbol{\alpha})) + \tilde{eq}_1(\alpha_{\mu+1})\cdot(\beta+F(\boldsymbol{\alpha})). \end{aligned}$$

So it suffices for $\mathcal{V}$ to query $[\![F]\!]$ and $[\![G]\!]$ and do some minor computation. Thus, no extra oracles appear, as desired.

Remark 2: The distributed Multiset Check PIOP in HyperPianist [22] is similar to ours in that it also applies logarithmic derivative and aims to show $\sum_{\mathbf{b}\in B_\mu}\frac{p(\mathbf{b})}{q(\mathbf{b})} = s$ (named *Rational SumCheck* therein). However, the main goal of HyperPianist is to acheive strictly

Protocol 4: Distributed Multiset Check PIOP. Given $(\mathbb{x}; \mathbb{w}) = ((\llbracket f_1 \rrbracket, \cdots, \llbracket f_k \rrbracket, \llbracket g_1 \rrbracket, \cdots, \llbracket g_k \rrbracket); (f_1, \cdots, f_k, g_1, \cdots, g_k))$, where $f_j, g_j \in \mathcal{F}_\mu^{(\leq d)}$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $\{[f_1(\mathbf{b}), \cdots, f_k(\mathbf{b})]\}_{\mathbf{b}\in B_\mu} = \{[g_1(\mathbf{b}), \cdots, g_k(\mathbf{b})]\}_{\mathbf{b}\in B_\mu}$ as follows.

1. $\mathcal{V}$ samples $\beta, \gamma_2, \cdots, \gamma_k \leftarrow_\$ \mathbb{F}$ and sends them to $\mathcal{P}_0$. $\mathcal{P}_0$ then transmits them to other sub-provers.
2. $\mathcal{P}_i$ computes $F^{(i)} := f_1^{(i)} + \sum_{j=2}^k \gamma_j f_j^{(i)}$ and $G^{(i)} := g_1^{(i)} + \sum_{j=2}^k \gamma_j g_j^{(i)}$, and gets $p^{(i)}$ and $q^{(i)}$ as in equation 2.
3. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 3 to show $((0, \llbracket p \rrbracket, \llbracket q \rrbracket); (p, q)) \in \mathcal{R}_{\text{FSum}}$.

Figure 5: The distributed Multiset Check PIOP.

logarithmic communication costs, so the authors perform Rational SumCheck by interpolating an intermediate polynomial $q^{-1}(\mathbf{X}) \in \mathcal{F}_\mu^{(\leq 1)}$ such that $q^{-1}(\mathbf{b}) = \frac{1}{q(\mathbf{b})}$ over B_μ, and then invoke a ZeroCheck for $q(\mathbf{X}) \cdot q^{-1}(\mathbf{X}) - 1 = 0$ and a SumCheck for $\sum_{\mathbf{b}\in B_\mu} p(\mathbf{b}) \cdot q^{-1}(\mathbf{b}) = s$. Compared to our approach, the rational sumcheck in HyperPianist requires computing an extra oracle during the protocol, but provides less communication cost. We also note that Cirrus [23] yields a similar result as HyperPianist, i.e., trading the number of intermediate oracles for less communication. Therefore, our Fractional SumCheck can be viewed as orthogonal to those protocols in HyperPianist and Cirrus.

3.4 Distributed Permutation Check PIOP

The Permutation Check proves that for $f, g \in \mathcal{F}_\mu^{(\leq d)}$ there is a predefined one-to-one correspondence, a permutation σ, that connects the evaluations of f and g over B_μ.

Definition 10 (Permutation Relation)**.** *The indexed relation $\mathcal{R}_{Perm}$ is the set of all tuples $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\sigma, (\llbracket f \rrbracket, \llbracket g \rrbracket); f, g)$, where $f, g \in \mathcal{F}_\mu^{(d)}$, and $\sigma : B_\mu \to B_\mu$ is a permutation, such that $g(\mathbf{b}) = f(\sigma(\mathbf{b}))$ for all $\mathbf{b} \in B_\mu$.*

In HyperPlonk [27], given $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\sigma, (\llbracket f \rrbracket, \llbracket g \rrbracket); f, g)$, the indexer $\mathcal{I}_{\text{Perm}}$ generates two oracles $\llbracket s_{\text{id}} \rrbracket$ and $\llbracket s_\sigma \rrbracket$, where $s_{\text{id}} \in \mathcal{F}_\mu^{(\leq 1)}$ maps each $\mathbf{b} \in B_\mu$ to $\text{int}(\mathbf{b})$, and $s_\sigma \in \mathcal{F}_\mu^{(\leq 1)}$ maps each $\mathbf{b} \in B_\mu$ to $\text{int}(\sigma(\mathbf{b}))$. Now it suffices for $\mathcal{P}$ and $\mathcal{V}$ to run a Multiset Check PIOP to prove $(\mathbb{x}; \mathbb{w}) = ((\llbracket s_{\text{id}} \rrbracket, \llbracket f \rrbracket, \llbracket s_\sigma \rrbracket, \llbracket g \rrbracket); (s_{\text{id}}, f, s_\sigma, g)) \in \mathcal{R}_{\text{MSet}}^2$.

The distribution is trivial, as presented in Protocol 5.

3.5 Distributed Lookup PIOP

To support circuits with lookup gates, HyperPlonk$^+$ [27] presents a lookup PIOP that requires computing an extra oracle during the protocol. To avoid this, we reuse the layered

Protocol 5: Distributed Permutation Check PIOP. Given $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\sigma; (\llbracket f \rrbracket, \llbracket g \rrbracket); (f, g))$, where $f, g \in \mathcal{F}_\mu^{(\leq d)}$ and $\sigma : B_\mu \to B_\mu$ is a permutation, the indexer generates $(\llbracket s_{\text{id}} \rrbracket, \llbracket s_\sigma \rrbracket) \leftarrow \mathcal{I}_{\text{Perm}}(\sigma)$. Then $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $g(\mathbf{b}) = f(\sigma(\mathbf{b}))$ for all $\mathbf{b} \in B_\mu$ by invoking Protocol 4 to prove that $((\llbracket s_{\text{id}} \rrbracket, \llbracket f \rrbracket, \llbracket s_\sigma \rrbracket, \llbracket g \rrbracket); (s_{\text{id}}, f, s_\sigma, g)) \in \mathcal{R}^2_{\text{MSet}}$.

Figure 6: The distributed Permutation Check PIOP.

Protocol 6: Distributed Lookup PIOP. Given a tuple $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\mathbf{t}; (\llbracket f_1 \rrbracket, \cdots, \llbracket f_k \rrbracket, \llbracket t \rrbracket, \llbracket c \rrbracket); (f_1, \cdots, f_k, t, c))$, where $f_1, \cdots, f_k, t, c \in \mathcal{F}_\mu^{(\leq 1)}$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ convince $\mathcal{V}$ that $\bigcup_{i=1}^k \{f_i(\mathbf{b}) : \mathbf{b} \in B_\mu\} \subseteq \{t(\mathbf{b}) : \mathbf{b} \in B_\mu\}$ by invoking Protocol 3 to show that $((0, \llbracket p \rrbracket, \llbracket q \rrbracket); (p, q)) \in \mathcal{R}_{\text{FSum}}$, where $p(\mathbf{X}, \mathbf{Y})$ and $q(\mathbf{X}, \mathbf{Y})$ are defined in equation 3.

Figure 7: The distributed Lookup PIOP.

circuit technique in [30]. Furthermore, we extend it to support Lookup for multiple columns at once. The lookup relation for multiple columns is defined as follows.

Definition 11 (Lookup Relation)**.** *The lookup relation* $\mathcal{R}^k_{\text{Lookup}}$ *is the set of tuples* $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = (\mathbf{t}; (\llbracket f_1 \rrbracket, \cdots, \llbracket f_k \rrbracket, \llbracket t \rrbracket, \llbracket c \rrbracket); (f_1, \cdots, f_k, t, c))$*, where* $f_1, \cdots, f_k, t, c \in \mathcal{F}_\mu^{(\leq 1)}$*. Here* t *is the representation of the table* $\mathbf{t} \in \mathbb{F}^{2^\mu}$*, such that* $\bigcup_{i=1}^k \{f_i(\mathbf{b}) : \mathbf{b} \in B_\mu\} \subseteq \{t(\mathbf{b}) : \mathbf{b} \in B_\mu\}$*, and* c *counts the multiplicities (explained below).*

By Lemma 3, the inclusion relation can be transformed to $\sum_{j=1}^k \sum_{\mathbf{b} \in B_\mu} \frac{1}{X + f_j(\mathbf{b})} = \sum_{\mathbf{b} \in B_\mu} \frac{c(\mathbf{b})}{X + t(\mathbf{b})}$, where $c(\mathbf{b}) := \sum_{j=1}^k |\{\mathbf{b} \in B_\mu : f_j(\mathbf{b}) = t(\mathbf{b})\}|$. This expression is akin to the equation 2 from Multiset Check PIOP, so we similarly represent the numerator and denominator by

$$\begin{aligned} p(\mathbf{X}, \mathbf{Y}) &:= -\tilde{eq}_{\mathbf{0}}(\mathbf{Y}) \cdot c(\mathbf{X}) + \sum_{\mathbf{b} \in B_e \setminus \{\mathbf{0}\}} \tilde{eq}_{\mathbf{b}}(\mathbf{Y}) \cdot 1, \\ q(\mathbf{X}, \mathbf{Y}) &:= \tilde{eq}_{\mathbf{0}}(\mathbf{Y}) \cdot (\beta + t(\mathbf{X})) + \sum_{\mathbf{b} \in B_e \setminus \{\mathbf{0}\}} \tilde{eq}_{\mathbf{b}}(\mathbf{Y}) \cdot (\beta + f_{\text{int}(\mathbf{b})}(\mathbf{X})), \end{aligned} \tag{3}$$

two polynomials $\in \mathcal{F}_{\mu+e}^{(\leq 1)}$ (w.l.o.g. assume $k = 2^e - 1$). We present the distributed Lookup PIOP in Protocol 6.

3.6 Security and Complexity Analysis

3.6.1 Security analysis

Since our PIOP system aims at accelerating proof generation, we assume sub-provers trust each other. Therefore, we analyze security as in the single-prover setting. We have the following theorem.

Theorem 1. *The PIOPs for* $\mathcal{R}_{\text{Sum}}$, $\mathcal{R}_{\text{Zero}}$, $\mathcal{R}_{\text{FSum}}$, $\mathcal{R}^k_{\text{MSet}}$, $\mathcal{R}_{\text{Perm}}$, *and* $\mathcal{R}^k_{\text{Lookup}}$ *are perfectly complete, with knowledge soundness error* $\frac{d\mu}{|\mathbb{F}|}$, $O(\frac{d\mu}{|\mathbb{F}|})$, $O(\frac{\mu^2+d\mu}{|\mathbb{F}|})$, $O(\frac{\mu^2+d\mu+2^\mu}{|\mathbb{F}|})$, $O(\frac{\mu^2+d\mu+2^\mu}{|\mathbb{F}|})$, *and* $O(\frac{(\mu+\log k)^2+d(\mu+\log k)}{|\mathbb{F}|})$, *respectively.*

To prove this theorem, we first quote several lemmas from HyperPlonk [27, Lemma 2.3, Theorem 3.1, Theorem 3.2], whose proofs can be found accordingly.

Lemma 4 (Sound PIOPs are knowledge sound)**.** *Consider a* δ*-sound PIOP for an oracle relation* $\mathcal{R}$*, where* $\forall(\mathtt{i}, \mathtt{x}; \mathtt{w}) \in \mathcal{R}$, $\mathtt{w}$ *consists only of polynomials and* $(\mathtt{i}, \mathtt{x})$ *contain oracles to these polynomials. The PIOP has* δ *knowledge-soundness, and the extractor runs in time* $O(|\mathtt{w}|)$.

Lemma 5. *The PIOP for* $\mathcal{R}_{\text{Sum}}$ *is perfectly complete and has knowledge error* $\delta^{d,\mu}_{\text{Sum}} = \frac{d\mu}{|\mathbb{F}|}$.

Lemma 6. *The PIOP for* $\mathcal{R}_{\text{Zero}}$ *is perfectly complete and has knowledge error* $\delta^{d,\mu}_{\text{Zero}} = \frac{d\mu}{|\mathbb{F}|} + \delta^{d+1,\mu}_{\text{Sum}} = O(\frac{d\mu}{|\mathbb{F}|})$.

Note that our distributed PIOP system differs from that of HyperPlonk only in that we merge the Product Check and the MultisetCheck into one, and that the MultisetCheck and Lookup PIOPs are recompiled based on the Fractional SumCheck PIOP respectively, with other PIOPs intact. Thus, based on the lemmas above, it suffices for us to analyze the perfect completeness and soundness errors starting from the distributed Fractional SumCheck PIOP.

Proof. We prove perfect completeness and soundness of each PIOP subsequently.
(1) **Distributed Fractional SumCheck PIOP.**
Perfect completeness: this has been explained detailedly in the form of layered circuit in Section 3.3.1
Soundness: for any $((s, [\![p]\!], [\![q]\!]); (p, q)) \notin \mathcal{R}_{\text{FSum}}$, the probability that $\mathcal{V}$ accepts in the k-th invocation of SumCheck PIOP is $\delta^{1,k}_{\text{Sum}}$, and that of the final invocation is $\delta^{d,\mu}_{\text{Sum}}$. Considering the $\frac{1}{|\mathbb{F}|}$ error resulting from the batching in each layer, the total soundness error is $\delta^{d,\mu}_{\text{FSum}} = O(\frac{\mu^2+d\mu}{|\mathbb{F}|})$.
(2) **Distributed MultiSet Check PIOP.**
Perfect completeness: for any $(([\![f_1]\!], \cdots, [\![f_k]\!], [\![g_1]\!], \cdots, [\![g_k]\!]); (f_1, \cdots, g_k)) \in \mathcal{R}^k_{\text{MSet}}$, the product equivalence $\prod_{\mathbf{b}\in B_\mu}(\beta + F(\mathbf{b})) = \prod_{\mathbf{b}\in B_\mu}(\beta + G(\mathbf{b}))$ holds, and by Lemma 2 it

follows that $\sum_{\mathbf{b}\in B_\mu} \frac{1}{\beta+F(\mathbf{b})} = \sum_{\mathbf{b}\in B_\mu} \frac{1}{\beta+G(\mathbf{b})}$. By defining $p(\mathbf{X}, y) := -\tilde{eq}_0(y)\cdot 1 + \tilde{eq}_1(y)\cdot 1$, and $q(\mathbf{X}, y) := \tilde{eq}_0(y)(\beta+G(\mathbf{X})) + \tilde{eq}_1(y)\cdot(\beta+F(\mathbf{X}))$, it then holds that $\sum_{\mathbf{b}\in B_{\mu+1}} \frac{p(\mathbf{b})}{q(\mathbf{b})} = 0$. Thus the perfect completeness follows, conditioned on that of the distributed Fractional SumCheck PIOP.

Soundness: for any $((\llbracket f_1\rrbracket, \cdots, \llbracket f_k\rrbracket, \llbracket g_1\rrbracket, \cdots, \llbracket g_k\rrbracket); (f_1, \cdots, g_k)) \notin \mathcal{R}^k_{\text{MSet}}$, F and G are determined by $\phi_{\mathbf{r}} : (x_1, \cdots, x_k) \to x_1 + r_2 x_2 + \cdots + r_k x_k$ [27, Theorem 3.5]. Therefore $((\llbracket F\rrbracket, \llbracket G\rrbracket); (F, G)) \notin \mathcal{R}^1_{\text{MSet}}$ with probability at least $1 - \frac{2^\mu}{|\mathbb{F}|}$. Conditioned on this, $\prod_{\mathbf{b}\in B_\mu}(X + F(\mathbf{b})) \neq \prod_{\mathbf{b}\in B_\mu}(X + G(\mathbf{b}))$. By Lemma 1, given some randomly chosen β, the probability that $\prod_{\mathbf{b}\in B_\mu}(\beta + F(\mathbf{b})) \neq \prod_{\mathbf{b}\in B_\mu}(\beta + G(\mathbf{b}))$ is at least $1 - \frac{2^\mu}{|\mathbb{F}|}$. Further conditioned on this, $\sum_{\mathbf{b}\in B_\mu} \frac{1}{\beta+F(\mathbf{b})} \neq \sum_{\mathbf{b}\in B_\mu} \frac{1}{\beta+G(\mathbf{b})}$. Therefore the probability that $\mathcal{V}$ accepts is at most $\delta^{d,\mu+1}_{\text{FSum}}$. Finally by a union bound, the soundness error is $\delta^{d,\mu}_{\text{MSet}} = O(\frac{\mu^2+d\mu+2^\mu}{|\mathbb{F}|})$.

(3) **Distributed Permutation PIOP.**

Perfect completeness and soundness: this PIOP is a special case of the distributed Multiset Check PIOP with $k = 2$, so perfect completeness immediately follows, and the soundness error is $\delta^{d,\mu}_{\text{Perm}} = \delta^{d,\mu}_{\text{MSet},2} = O(\frac{\mu^2+d\mu+2^\mu}{|\mathbb{F}|})$.

(4) **Distributed Lookup PIOP.**

Perfect completeness and soundness: this PIOP is essentially a distributed Fractional SumCheck PIOP with $\mu + \log k$ variables, so perfect completeness follows, and the soundness error is $\delta^{d,\mu}_{\text{Lookup},k} = \delta^{d,\mu+\log k}_{\text{FSum}} = O(\frac{(\mu+\log k)^2+d(\mu+\log k)}{|\mathbb{F}|})$. □

Remark 3: For ease of description, our PIOPs are presented without considering the zero-knowledge property. But this can be standardly achieved by augmenting SumCheck-based PIOPs with a compiler. For detailed constructions, please refer to HyperPlonk [27, Appendix A].

3.6.2 Complexity analysis

The complexity of each PIOP is summarized in Table 2. Here, $\mathcal{P}_0$ completes some additional $O(2^m(d\mu + d\log^2 d))$ computations in each PIOP, which are minor compared to the total complexity per sub-prover, hence omitted. Communication costs and round complexity rise as the distributed Fractional SumCheck kicks in, originating from the layered circuit with μ depth. But we stress that this is acceptable since our distributed polynomial commitment scheme to be presented in Section 4 requires the same amount of communication. Moreover, we benefit from the layered circuit technique by avoiding sending extra oracles during the protocols.

Table 2: The complexity analysis of PIOPs. Here $m = \log M$. d denotes degree, μ the number of variables, k in dMSetCheck the length of each tuple in multisets, and $k = 2^e - 1$ in dLookup the number of columns.

PIOPs	$\mathcal{P}_i$	Communication	$\mathcal{V}$	Num. of queries	Num. of rounds
dSumCheck	$O(2^{\mu-m} d \log^2 d)$	$O(2^m d(\mu-m))$	$O(d\mu)$	1	μ
dZeroCheck	$O(2^{\mu-m} d \log^2 d)$	$O(2^m d(\mu-m))$	$O(d\mu)$	1	$\mu+1$
dFSumCheck	$O(2^{\mu-m} d \log^2 d)$	$O(2^m d(\mu-m)^2)$	$O(\mu^2+d\mu)$	2	$(\mu^2-\mu)/2$
dMSetCheck	$O(2^{\mu-m} d \log^2 d)$	$O(2^m d(\mu-m)^2)$	$O(\mu^2+d\mu)$	$2k$	$(\mu^2+\mu)/2+1$
dPermCheck	$O(2^{\mu-m} d \log^2 d)$	$O(2^m d(\mu-m)^2)$	$O(\mu^2+d\mu)$	4	$(\mu^2+\mu)/2+1$
dLookup	$O(2^{\mu+e-m} d \log^2 d)$	$O(2^m d(\mu+e-m)^2)$	$O((\mu+e)^2+d(\mu+e))$	2^e+1	$((\mu+e)^2-(\mu+e))/2$

3.7 HyperFond$^{(+)}$ Constraint System

3.7.1 Basic Construction

HyperPlonk [27] considers a fan-in two arithmetic circuit over a finite field $\mathbb{F}$, supporting additions, multiplications, and custom operations $G : \mathbb{F}^2 \to \mathbb{F}$. Let n denote the number of input gates, and s that of operation gates. Without loss of generality, let $2^\mu = n + s + 1$ be the circuit size. Determine an $H \in \mathcal{F}_{\mu+2}^{(\leq 1)}$, so $H(0,0,\mathbf{bin}_\mu(i)), H(0,1,\mathbf{bin}_\mu(i))$, and $H(1,0,\mathbf{bin}_\mu(i))$ denote the left input, right input and output of gate i, respectively. We thus obtain

$$\begin{aligned} f(\mathbf{X}) = {} & q_{\text{add}}(\mathbf{X}) \cdot (H(0,0,\mathbf{X}) + H(0,1,\mathbf{X})) + q_{\text{mult}}(\mathbf{X}) \cdot (H(0,0,\mathbf{X}) \cdot H(0,1,\mathbf{X})) \\ & + q_{\text{custom}}(\mathbf{X}) \cdot G(H(0,0,\mathbf{X}), H(0,1,\mathbf{X})) + q_{\text{in}}(\mathbf{X}) - q_{\text{out}}(\mathbf{X}) \cdot H(1,0,\mathbf{X}) \in \mathcal{F}_\mu^{(\leq d)}, \end{aligned}$$

where q_{add}, (resp.$q_{\text{mult}}, q_{\text{custom}}, q_{\text{in}}$ and q_{out}) is the *selector* multilinear polynomial, mapping $\mathbf{bin}_\mu(i)$ to 1 if gate i is an addition (resp. multiplication, custom, input, output) gate, else 0. Now, the correct execution of the circuit is reduced to $f(\mathbf{X}) = 0$ over B_μ. In addition, each wire in the circuit induces an equality between two elements from the evaluations of H over $B_{\mu+2}$, captured by a permutation $\sigma : B_{\mu+2} \to B_{\mu+2}$ such that $H(\mathbf{X}) = H(\sigma(\mathbf{X}))$ over $B_{\mu+2}$.

Our constraint system for HyperFond is similar to that of HyperPlonk [27], from which we obtain the following relation and construction.

Definition 12 (HyperFond Indexed Relation)**.** *Fix public parameters* $\mathsf{gp} := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$*, where* $\mathbb{F}$ *is the field,* $\ell = 2^\nu$ *is the public input length,* $n = 2^\mu$ *is the number of constraints,* $\ell_w = 2^{\nu_w}, \ell_q = 2^{\nu_q}$ *are the numbers of witnesses and selectors per constraint, and* $f : \mathbb{F}^{\ell_q + \ell_w} \to \mathbb{F}$ *is a map. The indexed relation* $\mathcal{R}_{\mathsf{HyperFond}}$ *is the set of all tuples* $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((q,\sigma); (p, [\![w]\!]); w)$*, where* $\sigma : B_{\mu+\nu_w} \to B_{\mu+\nu_w}$ *is a permutation,* $q \in \mathcal{F}_{\mu+\nu_q}^{(\leq 1)}$*,* $p \in \mathcal{F}_\nu^{(\leq 1)}$*,* $w \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$*, so that (1)* $(\sigma; ([\![w]\!], [\![w]\!]); w) \in \mathcal{R}_{Perm}$*; (2)* $([\![\hat{f}]\!]; \hat{f}) \in \mathcal{R}_{\text{Zero}}$*, where*

$$\begin{aligned} \hat{f}(\mathbf{X}) := f(& q(\mathbf{bin}_{\nu_q}(0), \mathbf{X}), \cdots, q(\mathbf{bin}_{\nu_q}(\ell_q - 1), \mathbf{X}), \\ & w(\mathbf{bin}_{\nu_w}(0), \mathbf{X}), \cdots, w(\mathbf{bin}_{\nu_w}(\ell_w - 1), \mathbf{X})); \end{aligned}$$

Protocol 7: Distributed PIOP for $\mathcal{R}_{\mathsf{HyperFond}}$. Let public parameters $\mathsf{gp} = (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$. Given a tuple $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((q, \sigma); (p, [\![w]\!]); w)$, the indexer $\mathcal{I}(q, \sigma)$ calls the Permutation Check PIOP indexer $\mathcal{I}_{\mathrm{Perm}}$ (from Protocol 5), derives $([\![s_{\mathrm{id}}]\!], [\![s_\sigma]\!]) \leftarrow \mathcal{I}_{\mathrm{Perm}}(\sigma)$, and outputs oracles $([\![q]\!], [\![s_{\mathrm{id}}]\!], [\![s_\sigma]\!])$ where $q \in \mathcal{F}_{\mu+\nu_q}^{(\leq 1)}$, and $s_{\mathrm{id}}, s_\sigma \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$. $\{\mathcal{P}_i(\mathsf{gp}, \mathbb{i}, p, w^{(i)})\}_{i=0}^{M-1}$ and $\mathcal{V}(\mathsf{gp}, p, [\![q]\!], [\![s_{\mathrm{id}}]\!], [\![s_\sigma]\!])$ then run as follows.
1. $\mathcal{P}_0$ sends $\mathcal{V}$ the witness oracle $[\![w]\!]$.
2. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 2 to show that $([\![\hat{f}]\!]; \hat{f}) \in \mathcal{R}_{\mathrm{Zero}}$;
3. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 5 to show that $(\sigma; ([\![w]\!], [\![w]\!]); (w, w)) \in \mathcal{R}_{\mathrm{Perm}}$;
4. $\mathcal{V}$ samples and sends $\mathbf{r} \leftarrow_{\$} \mathbb{F}^\nu$, and queries $[\![w]\!]$ on input $(0^{\mu+\nu_w-\nu}, \mathbf{r})$ to check its consistency with $p(\mathbf{r})$.

Figure 8: The distributed PIOP for $\mathcal{R}_{\mathsf{HyperFond}}$.

and (3) $p(\mathbf{X}) = w(0^{\mu+\nu_w-\nu}, \mathbf{X})$.

Based on the original PIOP in [27, Section 4.1], we derive the distributed version by letting each $\mathcal{P}_i$ compute $w^{(i)}$, as presented in Protocol 7. Its security and complexity analysis can be reduced to those of PIOPs for $\mathcal{R}_{\mathrm{Zero}}$ and $\mathcal{R}_{\mathrm{Perm}}$, respectively, so omitted for simplicity.

3.7.2 Adding Lookup Gates

To enable the non-algebraic constraints that bind some function over witness values to a predescribed table $\mathbf{t}$, HyperPlonk$^+$ [27] uses an algebraic function f_{lk} as

$$\begin{aligned} g(\mathbf{X}) :=& f_{\mathrm{lk}}(q_{\mathrm{lk}}(\mathbf{bin}_{\nu_{\mathrm{lk}}}(0), \mathbf{X}), \cdots, q_{\mathrm{lk}}(\mathbf{bin}_{\nu_{\mathrm{lk}}}(\ell_{\mathrm{lk}}-1), \mathbf{X}), \\ & w(\mathbf{bin}_{\nu_w}(0), \mathbf{X}), \cdots, w(\mathbf{bin}_{\nu_w}(\ell_w-1), \mathbf{X})), \end{aligned}$$

to capture these constraints, where $q_{\mathrm{lk}} \in \mathcal{F}_{\mu+\nu_{\mathrm{lk}}}^{(\leq 1)}$ denotes the selector polynomial, $\ell_{\mathrm{lk}} = 2^{\nu_{\mathrm{lk}}}$ its number of lookup selectors, and $\ell_w = 2^{\nu_w}$ the number of witness wires. Thus i-th constraint is satisfied if $g(\mathbf{bin}_\mu(i)) \in \mathbf{t}$.

In $\mathsf{HyperFond}^+$, to support k-column lookup within only one table, we generalize g to $g_1, \cdots, g_k$ as described above with k algebraic maps $f_{\mathrm{lk},1}, \cdots, f_{\mathrm{lk},k} : \mathbb{F}^{\ell_{\mathrm{lk}}+\ell_w} \to \mathbb{F}$. Then we define k multilinear polynomials $h_1, \cdots, h_k \in \mathcal{F}_\mu^{(\leq 1)}$ conditioned on $h_j(\mathbf{X}) = \sum_{\mathbf{b} \in B_\mu} g_j(\mathbf{b}) \tilde{eq}_{\mathbf{b}}(\mathbf{X})$ for all $j \in [1:k]$, and run a distributed Lookup PIOP (Protocol 6) to check that the evaluations of h_j over B_μ lie in $\mathbf{t}$.

Definition 13 ($\mathsf{HyperFond}^+$ Indexed Relation)**.** *Let* $\mathsf{gp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ *be public parameters in Definition 12, and* $\mathsf{gp}_2 := (\ell_{lk}, f_{lk,1}, \cdots, f_{lk,k})$ *where* $\ell_{lk} = 2^{\nu_{lk}}$ *is the number*

Protocol 8: Distributed PIOP for $\mathcal{R}_{\mathsf{HyperFond}^+}$. Let public parameters $\mathsf{gp} = (\mathsf{gp}_1, \mathsf{gp}_2)$ where $\mathsf{gp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ and $\mathsf{gp}_2 := (\ell_{\mathrm{lk}}, f_{\mathrm{lk},1}, \cdots, f_{\mathrm{lk},k})$. Given a tuple $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((\mathbb{i}_1, \mathbb{i}_2); (p, [\![w]\!]); w)$ where $\mathbb{i}_2 := (\mathbf{t} \in \mathbb{F}^{2^\mu}, q_{\mathrm{lk}} \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_{\mathrm{lk}}})$, the indexer $\mathcal{I}(\mathbb{i}_1, \mathbb{i}_2)$ calls the HyperFond PIOP indexer $\mathcal{I}_{\mathsf{HyperFond}}$ (from Protocol 7) to derive $\mathsf{vp}_{\mathsf{HyperFond}} \leftarrow \mathcal{I}_{\mathsf{HyperFond}}(\mathbb{i}_1)$, calls the Lookup PIOP indexer $\mathcal{I}_{\mathrm{Lookup}}$ for $\mathsf{vp}_{\mathrm{t}} \leftarrow \mathcal{I}_{\mathrm{Lookup}}(\mathbf{t})$, and outputs $\mathsf{vp} := ([\![q_{\mathrm{lk}}]\!], \mathsf{vp}_{\mathsf{HyperFond}}, \mathsf{vp}_{\mathrm{t}})$. $\{\mathcal{P}_i(\mathsf{gp}, \mathbb{i}, p, w^{(i)})\}_{i=0}^{M-1}$ and $\mathcal{V}(\mathsf{gp}, p, \mathsf{vp}, [\![w]\!])$ then run as follows.
1. $\mathcal{P}_0$ sends $\mathcal{V}$ the witness oracle $[\![w]\!]$, and oracles $[\![h_1]\!], \cdots, [\![h_k]\!], [\![c]\!]$ for values and their multiplicities to lookup.
2. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 7 to show that $(\mathbb{i}_1; \mathbb{x}; \mathbb{w}) \in \mathcal{R}_{\mathsf{HyperFond}}$;
3. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ and $\mathcal{V}$ run Protocol 6 to show that $(\mathbf{t}; ([\![h_1]\!], \cdots, [\![h_k]\!], [\![t]\!], [\![c]\!]); (h_1, \cdots, h_k, t, c)) \in \mathcal{R}^k_{\mathrm{Lookup}}$.

Figure 9: The distributed PIOP for $\mathcal{R}_{\mathsf{HyperFond}^+}$.

of selectors and $f_{lk,j} : \mathbb{F}^{\ell_{lk}+\ell_w} \to \mathbb{F}$ are k algebraic maps. Indexed relation $\mathcal{R}_{\mathsf{HyperFond}^+}$ is the set of all tuples $(\mathbb{i}; \mathbb{x}; \mathbb{w}) = ((\mathbb{i}_1, \mathbb{i}_2); (p, [\![w]\!]); w)$, where $\mathbb{i}_2 := (\mathbf{t} \in \mathbb{F}^{2^\mu}, q_{lk} \in \mathcal{F}^{(\leq 1)}_{\mu+\nu_{lk}})$, such that (1) $(\mathbb{i}_1; \mathbb{x}; \mathbb{w}) \in \mathcal{R}_{\mathsf{HyperFond}}$; and (2) $(\mathbf{t}; ([\![h_1]\!], \cdots, [\![h_k]\!], [\![t]\!], [\![c]\!]); (h_1, \cdots, h_k, t, c)) \in \mathcal{R}^k_{\mathrm{Lookup}}$ as in Section 3.5

We present the distributed PIOP for $\mathcal{R}_{\mathsf{HyperFond}^+}$ in Protocol 8. Its security and complexity analysis can be reduced to those of PIOPs for $\mathcal{R}_{\mathsf{HyperFond}}, \mathcal{R}_{\mathrm{Zero}}$ and $\mathcal{R}^k_{\mathrm{Lookup}}$, respectively, so omitted for simplicity.

4 Distributed Multivariate PCS

To compile our distributed PIOP system into a distributed SNARK, we need to distribute a multilinear PCS, which, by standard batch opening [27, Subsection 3.7], can be transformed into a PCS for general multivariate polynomials. In this section, we present such a distributed scheme based on BaseFold [31], and analyze its security and complexity.

4.1 Distributed BaseFold

Since the PCS in BaseFold consists of an encoding algorithm, an IOPP between the prover $\mathcal{P}$ and verifier $\mathcal{V}$, and the final evluation protocol that interleaves the IOPP and another SumCheck protocol, we distribute them step by step.

4.1.1 Distributed Encoding Algorithm

We start with the definition of foldable linear codes. Let $\mathcal{C}_0$ be an $[n_0, k_0, d_0]$-linear MDS code, with rate $\rho := \frac{k_0}{n_0}$, and $\mathbf{G}_0 \in \mathbb{F}^{k_0 \times n_0}$ its generator matrix. A foldable linear code $\mathcal{C}_\mu$ is defined inductively as follows. For $1 \leq j \leq \mu$, let $k_j = 2^j k_0$ and $n_j = \rho^{-1} k_j$. Let $\mathbf{T}_{j-1}$, $\mathbf{T}'_{j-1} \leftarrow_{\$} (\mathbb{F}^\times)^{n_{j-1}}$ be diagonal matrices. The generator matrix $\mathbf{G}_j$ for $\mathcal{C}_j$ is defined as

$$\mathbf{G}_j := \begin{bmatrix} \mathbf{G}_{j-1} & \mathbf{G}_{j-1} \\ \mathbf{G}_{j-1} \cdot \mathbf{T}_{j-1} & \mathbf{G}_{j-1} \cdot \mathbf{T}'_{j-1} \end{bmatrix}.$$

Given $\mathbf{G}_0$ and $\{\mathbf{T}_j, \mathbf{T}'_j\}_{j=0}^{\mu-1}$, the original encoding algorithm for $\mathcal{C}_\mu$ is then recursively defined in Algorithm 1.

Algorithm 1 $\mathbf{Enc}_\mu$: BaseFold Encoding Algorithm [31]

Input: $\mathbf{f} \in \mathbb{F}^{k_\mu}$
Output: $\mathbf{f} \cdot \mathbf{G}_\mu \in \mathbb{F}^{n_\mu}$
if $\mu = 0$ (i.e. $\mathbf{f} \in \mathbb{F}^{k_0}$) **then**
  return $\mathbf{f} \cdot \mathbf{G}_0$
else
  Parse $\mathbf{f} = (\mathbf{f}_l, \mathbf{f}_r)$
  Set $\mathbf{l} = \mathbf{Enc}_{\mu-1}(\mathbf{m_l})$, $\mathbf{r} = \mathbf{Enc}_{\mu-1}(\mathbf{m_r})$, $\mathbf{t} = \mathrm{diag}(\mathbf{T}_{\mu-1})$ and $\mathbf{t}' = \mathrm{diag}(\mathbf{T}'_{\mu-1})$
  return $(\mathbf{l} + \mathbf{t} \circ \mathbf{r}, \mathbf{l} + \mathbf{t}' \circ \mathbf{r})$
end if

To distribute computation and reduce communication, we terminate Algorithm 1 at the m-th level of recursion, as in Algorithm 2, where each $\mathcal{P}_i$ computes $\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$, a "partial" codeword, and communication is not required.

Algorithm 2 Distributed Encoding Algorithm

Input: $\mathbf{f} = (\mathbf{f}^{(0)} \| \cdots \| \mathbf{f}^{(M-1)}) \in \mathbb{F}^{k_\mu}$
Output: $(\mathbf{f}^{(i)} \cdot \mathbf{G}_{\mu-m})_{i=0}^{M-1} \in (\mathbb{F}^{n_{\mu-m}})^M$
for $i = 0$ to $M - 1$ **do**
  Compute $\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$ Using Algorithm 1
end for

4.1.2 Distributed IOPP

In the original IOPP, $\mathcal{P}$ wants to convince $\mathcal{V}$ that an oracle $\pi_\mu = \mathbf{Enc}_\mu(\mathbf{f}) \in \mathbb{F}^{n_\mu}$ is close to a codeword in $\mathcal{C}_\mu$. To this end, the protocol is split into two phases, IOPP.Commit and IOPP.Query. In the commit phase, $\mathcal{P}$ inductively generates a list of oracles $(\pi_{\mu-1}, \cdots, \pi_0)$

Protocol 9: Distributed Commit Phase $\mathsf{IOPP.dCommit}$**.** On input oracle $\Pi_{\mu-m} \in (\mathbb{F}^{n_{\mu-m}})^M$, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ generate oracles $(\Pi_{\mu-m-1}, \cdots, \Pi_0) \in (\mathbb{F}^{n_{\mu-m-1}})^M \times \cdots \times (\mathbb{F}^{n_0})^M$ as follows. For j from $\mu - m - 1$ downto 0:

1. $\mathcal{V}$ samples and sends $\alpha_{j+1} \leftarrow_\$ \mathbb{F}$ to the master prover $\mathcal{P}_0$. $\mathcal{P}_0$ then transmits it to other sub-provers.
2. $\mathcal{P}_i$ interpolates $Q_{j,p}^{(i)}(X) := \mathsf{interpolate}((\mathrm{diag}(\mathbf{T}_j)[p], \pi_{j+1}^{(i)}[p]), (\mathrm{diag}(\mathbf{T}'_j)[p], \pi_{j+1}^{(i)}[p + n_j]))$ for each $p \in [1 : n_j]$, and sets $\pi_j^{(i)}[p] = Q_j^{(i)}(\alpha_{j+1})$. $\mathcal{P}_i$ then sends oracle $\pi_j^{(i)} \in \mathbb{F}^{n_j}$ to $\mathcal{P}_0$, who outputs oracle $\Pi_j = (\pi_j^{(i)})_{i=0}^{M-1} \in (\mathbb{F}^{n_j})^M$.

Protocol 10: Distributed Query Phase $\mathsf{IOPP.dQuery}$**.** Given a list of oracles $(\Pi_{\mu-m}, \cdots, \Pi_0) \in (\mathbb{F}^{n_{\mu-m}})^M \times \cdots \times (\mathbb{F}^{n_0})^M$, $\mathcal{V}$ samples and sends each $\mathcal{P}_i$ an index $p^{(i)} \leftarrow_\$ [1 : n_{\mu-m-1}]$. Then for j from $\mu - m - 1$ downto 0, $\mathcal{V}$:

1. acquires $\pi_{j+1}^{(i)}[p^{(i)}]$, $\pi_{j+1}^{(i)}[p^{(i)} + n_j]$ and $\pi_j^{(i)}[p^{(i)}]$ from each sub-prover $\mathcal{P}_i$ via $\mathcal{P}_0$;
2. computes $Q_j^{(i)}(X) := \mathsf{interpolate}((\mathrm{diag}(\mathbf{T}_j)[p^{(i)}], \pi_{j+1}^{(i)}[p^{(i)}]), (\mathrm{diag}(\mathbf{T}'_j)[p^{(i)}], \pi_{j+1}^{(i)}[p^{(i)} + n_j]))$;
3. checks that $Q_j^{(i)}(\alpha_{j+1}) = \pi_j^{(i)}[p^{(i)}]$. If $j > 0$ and $p^{(i)} > n_{j-1}$, updates $p^{(i)} \leftarrow p^{(i)} - n_{j-1}$.

Finally, if Π_0 consists of valid codewords $(\pi_0^{(0)}, \cdots, \pi_0^{(M-1)})$ w.r.t. $\mathbf{G}_0$, $\mathcal{V}$ outputs accept, else reject.

Figure 10: The distributed IOPP.

with $\mathcal{V}$'s challenges $\alpha_\mu, \cdots, \alpha_1$. Specifically, for an oracle π_j, $\mathcal{P}$ evenly splits it into $\pi_{j,1}$ and $\pi_{j,2}$, and folds them into π_{j-1} as follows. For each index p, $\mathcal{P}$ interpolates a linear polynomial $Q_{j,p}(X)$ with points $\mathrm{diag}(\mathbf{T}_j)[p]$, $\mathrm{diag}(\mathbf{T}'_j)[p]$ and values $\pi_{j,1}[p]$, $\pi_{j,2}[p]$. $\mathcal{P}$ then sets $\pi_{j-1}[p] = Q_{j,p}(\alpha_j)$. In the query phase, $\mathcal{V}$ checks (i) the consistency among these oracles, by randomly selecting a p, acquiring $\pi_{j,1}[p], \pi_{j,2}[p]$ and $\pi_{j-1}[p]$ from $\mathcal{P}$, interpolating a linear polynomial $Q_{j,p}(X)$, and verifying that $Q_{j,p}(\alpha_j) = \pi_{j-1}[p]$, and (ii) π_0 is a valid codeword.

In our distributed setting, encoding stops at the m-th level of recursion in Algorithm 2, and the encoding of $\mathbf{f}$ becomes $(\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)}))_{i=0}^{M-1}$. In round j of commit and query, $\mathcal{P}_i$ interpolates $Q_{j,p}^{(i)}(X)$ for each p, and computes $\pi_j^{(i)}$, with global randomness α_{j+1}. The oracle Π_j is a vector $(\pi_j^{(i)})_{i=0}^{M-1}$. The distributed commit and query phases are presented in Protocols 9 and 10, where $\mathsf{interpolate}((x_1, y_1), (x_2, y_2))$ denotes the unique linear polynomial $Q(X)$, such that $Q(x_1) = y_1$ and $Q(x_2) = y_2$. As in BaseFold, we require $\mathbf{T}_j[p] \neq \mathbf{T}'_j[p]$ for each $j \in [0 : \mu - 1]$ and p, so interpolation is feasible.

4.1.3 Distributed PCS

In BaseFold, for a multilinear polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$, the oracle $\pi_\mu = \mathbf{Enc}_\mu(\mathbf{f})$ is the encoding of $\mathbf{f}$, the coefficients of f, and then replaced with a Merkle root with leaves being π_μ, forwarded to $\mathcal{V}$. To prove the correctness of $y = f(\mathbf{z})$, $\mathcal{P}$ and $\mathcal{V}$ run a SumCheck protocol for $\hat{f}(\mathbf{X}) := f(\mathbf{X}) \cdot \tilde{eq}_{\mathbf{z}}(\mathbf{X})$, combined with the preceding IOPP to generate $\pi_{\mu-1}, \cdots, \pi_0$, also in the form of Merkle roots. $\mathcal{V}$ checks the correctness of π_0, and that $\pi_\mu, \cdots, \pi_0$ are consistent.

The distributed multilinear PCS follows by replacing each π_j with $\Pi_j = (\pi_j^{(i)})_{i=0}^{M-1}$, and letting $\mathcal{V}$ perform consistency check for each i. In the final round, $\mathcal{V}$ recovers π_0, and checks its correctness. Formally, we have the following.

Field choices and Setup. By field-agnosticity, $\mathbb{F}$ is only required to satisfy $\frac{\mu}{|\mathbb{F}|} = \mathsf{negl}(\lambda)$. On input 1^λ and μ, $\{\mathcal{P}_i\}_{i=0}^{M-1}$ acquire parameters $\mathbf{G}_0$ and $\{\mathbf{T}_j, \mathbf{T}'_{\mu-1}\}_{j=0}^{\mu-1}$ for $\mathcal{C}_\mu$.

Distributed commitment phase. Given $f \in \mathcal{F}_\mu^{(\leq 1)}$ with $\mathbf{f} \in \mathbb{F}^{2^\mu}$, $\mathcal{P}_i$ computes $\pi_{\mu-m}^{(i)} = \mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$. Denote the encoding of f by $\Pi_{\mu-m} := (\pi_{\mu-m}^{(i)})_{i=0}^{M-1}$, which in the IOP is the commitment to f that $\mathcal{V}$ queries. The commitments in the random oracle model consist of M Merkle roots, $(\mathsf{MC}(\pi_{\mu-m}^{(i)}))_{i=0}^{M-1}$.

Distributed opening phase. $\mathcal{P}_0$ opens a commitment to f by gathering and sending $\Pi_{\mu-m}$ with f to $\mathcal{V}$. $\mathcal{V}$ checks that (i) the M Merkle roots of the words are valid, and (ii) the relative distance between each pair of $\pi_{\mu-m}^{(i)}$ and $\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$ is less than $\Delta_{\mathcal{C}_{\mu-m}}/2$.

Distributed evaluation protocol. The protocol interleaves distributed SumCheck (Protocol 1) with distributed IOPP (highlighted in blue). We present it in Protocol 11.

Remark 4: The matrices $\{\mathbf{T}_j, \mathbf{T}'_{\mu-1}\}_{j=\mu-m}^{\mu-1}$ are not explicitly used here. We retain them merely for security analysis.

4.2 Security and Complexity Analysis

4.2.1 Security Analysis

Theorem 2. *The distributed algorithm is correct; the distributed IOPP satisfies completeness and soundness; and the distributed evaluation protocol satisfies completeness, bindingness, and knowledge soundness.*

The core observation is that m levels of recursion and m rounds of IOPP are skipped. Thus, we view the distributed encoding algorithm (resp. IOPP and PCS) of magnitude 2^μ as 2^m independent single-prover processes, each of magnitude $2^{\mu-m}$.

We first present some background knowledge that will be used during the security proof.

Definition 14 (Coset Relative Minimum Distance [31])**.** *Let n be an even integer and let $\mathcal{C}$ be a $[n, k, d]$error-correcting code. Let $\mathbf{v} \in \mathbb{F}^n$ be a vector and let $c \in \mathcal{C}$ be a codeword.*

Protocol 11: Distributed Evaluation Protocol $\mathsf{PC.dEval}$.

Public input: a point $\mathbf{z} \in \mathbb{F}^\mu$, a claimed value $y \in \mathbb{F}$, and Merkle roots $\Pi_{\mu-m} = (\mathsf{MC}(\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})))_{i=0}^{M-1} \in \mathbb{F}^M$.

Prover witness: $\{f^{(i)}\}_{i=0}^{M-1}$ of multilinear polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$, and $\{\mathbf{f}^{(i)}\}_{i=0}^{M-1}$ of its coefficients $\mathbf{f} \in \mathbb{F}^{2^\mu}$.

Parameters: The generator matrix $\mathbf{G}_0$, diagonal matrices $\{\mathbf{T}_j, \mathbf{T}'_j\}_{j=0}^{\mu-1}$, and the parallel repetition parameter $\ell \in \mathbb{N}$.

1. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ compute $r_\mu(X) := \sum_{\mathbf{b}\in B_{\mu-1}} f(\mathbf{b}, X) \cdot \tilde{eq}_{\mathbf{z}}(\mathbf{b}, X)$; $\mathcal{P}_0$ sends it to $\mathcal{V}$, who checks that $r_\mu(0) + r_\mu(1) = y$.
2. For j from $\mu - 1$ downto m, $\mathcal{V}$ samples and sends $\alpha_{j+1} \leftarrow \mathbb{F}$ to $\{\mathcal{P}_i\}_{i=0}^{M-1}$ via $\mathcal{P}_0$. $\{\mathcal{P}_i\}_{i=0}^{M-1}$ compute $r_j(X) := \sum_{\mathbf{b}\in B_{j-1}} f(\mathbf{b}, X, \alpha_{j+1}, \cdots, \alpha_\mu) \cdot \tilde{eq}_{\mathbf{z}}(\mathbf{b}, X, \alpha_{j+1}, \cdots, \alpha_\mu)$; $\mathcal{P}_0$ sends it to $\mathcal{V}$, who checks that $r_j(0) + r_j(1) = r_{j+1}(\alpha_{j+1})$.
3. For j from $m - 1$ downto 0, $\mathcal{P}_0$ runs the rest of the SumCheck protocol with $\mathcal{V}$.
4. Parallelly using the randomness $\alpha_{\mu-m}, \cdots, \alpha_1$ from the SumCheck protocol, for the same j from $\mu - m - 1$ downto 0, each $\mathcal{P}_i$ interpolates $Q_j^{(i)}(X) := \mathsf{interpolate}((\mathrm{diag}(\mathbf{T}_j)[p], \pi_{j+1}^{(i)}[p]), (\mathrm{diag}(\mathbf{T}'_j)[p], \pi_{j+1}^{(i)}[p + n_j]))$ for $p \in [1 : n_j]$, and sets $\pi_j^{(i)}[p] = Q_j^{(i)}(\alpha_{j+1})$. $\mathcal{P}_0$ then aggregates and outputs $(\mathsf{MC}(\pi_j^{(i)}))_{i=0}^{M-1}$.
5. $\mathcal{V}$ checks that $\mathsf{IOPP.dQuery}^{(\Pi_{\mu-m}, \cdots, \Pi_0)}$ outputs accept for all ℓ independent calls; $\mathcal{V}$ also verifies the equation $\mathbf{Enc}_0(\frac{r_1(\alpha_1)}{\tilde{eq}_{\mathbf{z}}(\alpha_1, \cdots, \alpha_\mu)}) = \pi_0^{(0)}$, where $\pi_0^{(0)}$ is recovered using $\Pi_0 = (\pi_0^{(i)})_{i=0}^{M-1}$ as follows. For j from 1 to m:
- for each $i \in \{0, 2^j, \cdots, M - 2^j\}$, $\mathcal{V}$ updates $\pi_0^{(i)} \leftarrow \pi_0^{(i)} + \alpha_{j+\mu-m} \cdot \pi_0^{(i+2^{j-1})}$.

Figure 11: The distributed evaluation protocol.

The coset relative distance $\Delta^*(\mathbf{v}, c)$ *between* $\mathbf{v}$ *and* c *is*

$$\Delta^*(\mathbf{v}, c) := \frac{2 \cdot \sharp\{j \in [1 : n/2] : \mathbf{v}[j] \neq c[j] \vee \mathbf{v}[j + n/2] \neq c[j + n/2]\}}{n}.$$

The coset relative minimum distance *of* $\mathbf{v}$ *to the code* $\mathcal{C}$*, denoted by* $\Delta^*(\mathbf{v}, \mathcal{C})$*, is* $\Delta^*(\mathbf{v}, \mathcal{C}) = \min_{c\in\mathcal{C}} \Delta^*(\mathbf{v}, c)$.

The Johnson bound estimates a radius around every code as its minimum distance, within which it is list-decodable.

Definition 15 (Johnson Bound [31])**.** *For every* $\gamma \in (0, 1]$*, define* $J_\gamma : [0, 1] \to [0, 1]$ *as* $J_\gamma(\lambda) := 1 - \sqrt{1 - \lambda(1 - \gamma)}$.

To complete our proof for the distributed versions of encoding algorithm, IOPP and

PCS, we respectively quote several useful lemmas from BaseFold [31], whose proofs can be found accordingly.

(1) **Correctness of the distributed encoding algorithm (Algorithm 2)**

Lemma 7 (Correctness of BaseFold Encoding Algorithm [31, Lemma 1])**.** *Let* $\rho^{-1}, \mu \in \mathbb{N}$ *and let* $\mathbf{G}_0$ *be the generator matrix of* $\mathcal{C}_0$*, with associated diagonal matrices* $\{\mathbf{T}_j, \mathbf{T}'_j\}_{j=0}^{\mu-1}$*. Then* $\forall \mathbf{f} \in \mathbb{F}^{2^\mu}$*,* $\mathbf{Enc}_\mu(\mathbf{f}) = \mathbf{f} \cdot \mathbf{G}_\mu$*.*

As mentioned before, each sub-prover pauses at the moment when it possesses the codeword of $\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$, instead of continuing collaborating with other sub-provers to ultimately derive the $\mathbf{Enc}_\mu(\mathbf{f})$. The correctness of the distributed version is thus straight forward.

Proof. **Correctness:** By Lemma 7, $\mathbf{Enc}_\mu(\mathbf{f}) = \mathbf{f} \cdot \mathbf{G}_\mu$, so $\mathbf{Enc}_{\mu-m}(\mathbf{m^{(i)}}) = \mathbf{m^{(i)}} \cdot \mathbf{G}_{\mu-m}$, $\forall i \in [0 : M-1]$. □

(2) **Completeness and soundness of the distributed IOPP (Protocols 9 and 10)**

Lemma 8 (Completeness of BaseFold IOPP [31, Lemma 5])**.** *If* π_μ *is a valid codeword in* $\mathcal{C}_\mu$*, then the verifier always outputs* accept *in the* IOPP.Query *phase given oracles* $(\pi_\mu, \cdots, \pi_0)$ *output by the honest prover in the* IOPP.Commit *phase.*

Lemma 9 (IOPP Soundness for Foldable Linear Codes [31, Theorem 3])**.** *Let* $(\mathcal{C}_\mu)$ *be a* (ρ^{-1}, k_0, μ)*-foldable linear code with generator matrices* $(\mathbf{G}_0, \cdots, \mathbf{G}_\mu)$ *and corresponding codes* $\mathcal{C}_0, \cdots, \mathcal{C}_{\mu-1}$*. Also assume that the relative minimum distance* $\Delta_{\mathcal{C}_i} \geq \Delta_{\mathcal{C}_{i+1}}$ *for all* $i \in [0 : \mu - 1]$*. Let* $\gamma > 0$ *and set* $\delta := \min(\Delta^*(\pi_\mu, \mathcal{C}_\mu), J_\gamma(J_\gamma(\Delta_{\mathcal{C}_\mu})))$ *where* $\Delta^*(\pi_\mu, \mathcal{C}_\mu)$ *is the relative coset minimum distance between* $\mathbf{v}$ *and* $\mathcal{C}_\mu$ *(Definition 14) and* J_γ *is the Johnson bound (Definition 15). Then with probability at least* $1 - \frac{2\mu}{\gamma^3|\mathbb{F}|}$ *(over the challenges* $\alpha_1, \cdots, \alpha_\mu$ *in* IOPP.Commit*), for any (adaptively chosen) prover oracle tuple* $(\pi_{\mu-1}, \cdots, \pi_0)$*, the verifier outputs* accept *in all of the* ℓ *repetitions of* IOPP.Query *with probability at most* $(1 - \delta + \gamma\mu)^\ell$*.*

Essentially, IOPP.Commit and IOPP.Query are decomposed into M sub-phases, when sub-provers can compute locally with public parameters. Thus, we can view the distributed IOPP of magnitude 2^μ as M independent BaseFold IOPPs. The following proof is based on Lemmas 8 and 9.

Proof. **Completeness:** by Lemma 8, if $\pi^{(i)}_{\mu-m}$ is a valid codeword in $\mathcal{C}_{\mu-m}$ for each $i \in [0 : M-1]$, then given oracle tuple $(\pi^{(i)}_{\mu-m}, \cdots, \pi^{(i)}_0)$ computed by an honest $\mathcal{P}_i$, $\mathcal{V}$ always outputs accept. Since we assume that the sub-provers $\{\mathcal{P}_i\}_{i=0}^{M-1}$ are internally honest to each other, $\mathcal{V}$ outputs accept given the aggregated oracles $(\Pi_{\mu-m}, \cdots, \Pi_0)$.
Soundness: by Lemma 9, in the single-prover IOPP, with probability at least $1 - \frac{2\mu}{\gamma^3|\mathbb{F}|}$ (over the challenges $\alpha_0, \cdots, \alpha_{\mu-1}$ in IOPP.Commit), for any (adaptively chosen) prover

oracle tuple $(\pi_{\mu-1}, \cdots, \pi_0)$, $\mathcal{V}$ outputs accept in all of the ℓ repetitions of IOPP.Query with probability at most $(1-\delta+\gamma\mu)^\ell$.

In the distributed setting, the number of rounds of IOPP decreases to $\mu - m$, and that of verification multiplies M, so the rejection probability is at least $\delta' - \gamma(\mu - m)$, where $\delta' := \min(\min_i(\Delta^*(\pi^{(i)}_{\mu-m}, \mathcal{C}_{\mu-m})), J_\gamma(J_\gamma(\Delta_{\mathcal{C}_{\mu-m}})))$. Therefore, in one repetition of IOPP.dQuery, the rejecting probability is $1-(1-\delta'+\gamma(\mu-m))^M$. By a union bound, with probability at least $1-\frac{2M(\mu-m)}{\gamma^3|\mathbb{F}|}$, for any prover oracle tuple $(\Pi_{\mu-m-1}, \cdots, \Pi_0)$, $\mathcal{V}$ outputs accept in ℓ repetitions of IOPP.dQuery with probability at most $(1-\delta'+\gamma(\mu-m))^{M\ell}$. □

(3) **Completeness, bindingness, and knowledge soundness of the distributed PCS (Protocol 11)**

Lemma 10 (Completeness and Bindingness of the BaseFold PCS [31, Section 5.1])**.** *The BaseFold multilinear polynomial commitment scheme is complete and binding.*

Lemma 11 (Knowledge Soundness of the BaseFold PCS [31, Section 5.1])**.** *Let $\gamma, \delta \in (0,1)$ satisfy $\delta < J_\gamma(J_\gamma(\Delta_{\mathcal{C}_\mu}))$, and $\Delta_{\mathcal{C}_i} \geq \Delta_{\mathcal{C}_{i+1}}$. Fix finite field $\mathbb{F}$ and set $\ell \in \mathbb{N}$ such that $\frac{2\mu}{\gamma^3|\mathbb{F}|} + (1-\delta+\mu\gamma)^\ell \leq \mathsf{negl}(\lambda)$. Assume that $3\delta - \mu\gamma < \Delta_{\mathcal{C}_\mu}$. For any PCS evaluation instance $(\pi_f, \mathbf{z}, y)$ and any malicious prover $\mathcal{P}^*$ that succeeds in the BaseFold PCS evaluation protocol with non-negligible probability, there is a polynomial-time extractor $\mathcal{E}^{\mathcal{P}}$ such that with overwhelming probability outputs a polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$ where $f(\mathbf{z}) = y$ and $\Delta^*(\mathbf{Enc}_\mu(\mathbf{f}), \pi_f) \leq \delta$.*

Similar to the observation in the previous proofs, we present the following proof based on Lemmas 10 and 11.

Proof. **Completeness:** the evaluation protocol consists of two parts —SumCheck and IOPP. The completeness of the former one is idential to that in PIOP, so we only consider the IOPP part. Recall that in the single-prover case,

$$\begin{aligned}\pi_{\mu,\text{single}} =&(\mathbf{Enc}_{\mu-1}(\mathbf{f}^{(0)}) + \text{diag}(\mathbf{T}_{\mu-1}) \circ \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(1)})\| \\ &\mathbf{Enc}_{\mu-1}(\mathbf{f}^{(0)}) + \text{diag}(\mathbf{T}'_{\mu-1}) \circ \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(1)})).\end{aligned}$$

Then $\pi_{\mu-1,\text{single}}[p]$ for each index p is computed by interpolating a linear polynomial $Q_{\mu-1,p}(X)$, with

$$\begin{aligned}&\text{points: } x_1 = \text{diag}(\mathbf{T}_{\mu-1})[p], x_2 = \text{diag}(\mathbf{T}'_{\mu-1})[p], \\ &\text{values: } y_1 = \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(0)})[p], y_2 = \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(1)})[p],\end{aligned}$$

and evaluating $Q_{\mu-1,p}(X)$ at α_μ.

We prove the correctness by induction on m. Consider the case where $m = 1$. By Lemma 10, honest $\mathcal{P}_0$ and $\mathcal{P}_1$ always derive the correct codewords $\pi^{(0)}_{\mu-1} = \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(0)})$

and $\pi_{\mu-1}^{(1)} = \mathbf{Enc}_{\mu-1}(\mathbf{f}^{(1)})$, respectively. Consequently, we have $\pi_{\mu-1,\text{single}} = \pi_{\mu-1}^{(0)} + \alpha_\mu \cdot \pi_{\mu-1}^{(1)}$. This linear relation w.r.t. α_μ preserves till the final round, where $\pi_{0,\text{single}} = \pi_0^{(0)} + \alpha_\mu \cdot \pi_0^{(1)}$, because scalar multiplication is commutative with linear operation. Thus, once the verifier obtains $\pi_0^{(0)}$ and $\pi_0^{(1)}$, and updates $\pi_0^{(0)} \leftarrow \pi_0^{(0)} + \alpha_\mu \cdot \pi_0^{(1)}$, it holds that $\mathbf{Enc}_0(\frac{r_1(\alpha_1)}{\tilde{eq}_{\mathbf{z}}(\alpha_1,\cdots,\alpha_\mu)}) = \pi_0^{(0)}$.

Assume that the case $M = 2^m$ holds, that is, the equation $\mathbf{Enc}_0(\frac{r_1(\alpha_1)}{\tilde{eq}_{\mathbf{z}}(\alpha_1,\cdots,\alpha_\mu)}) = \pi_0^{(0)}$ holds when $\pi_0^{(0)}$ is computed as $\pi_0^{(i)} \leftarrow \pi_0^{(i)} + \alpha_{j+\mu-m} \cdot \pi_0^{(i+2^{j-1})}$ for j from 1 to m for each $i \in \{0, 2^j, \cdots, M-2^j\}$, using randomness $\{\alpha_{j+\mu-m}\}_{j=1}^m$. Then in the case $M' = 2^{m+1}$, simply replace $1 \le j \le m$ with $1 \le j \le m+1$, with other procedures unchanged. The only difference is that an additional randomness $\alpha_{\mu-m}$ joins the update, to merge $\pi_0^{(1)}, \cdots, \pi_0^{(M'-1)}$ into $\pi_0^{(0)}, \cdots, \pi_0^{(M'-2)}$, respectively, and the rest m rounds of updates follows that during the case $M = 2^m$. Therefore, the equation $\mathbf{Enc}_0(\frac{r_1(\alpha_1)}{\tilde{eq}_{\mathbf{z}}(\alpha_1,\cdots,\alpha_\mu)}) = \pi_0^{(0)}$ still holds.

Bindingness: by Lemma 10, $\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)})$ binds to $\pi_{\mu-m}^{(i)}$ for each i. Therefore the tuple $(\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)}))_{i=0}^{M-1}$ binds to $\Pi_{\mu-m} = (\pi_{\mu-m}^{(i)})_{i=0}^{M-1}$.

Knowledge soundness: by Lemma 11, there exists an extractor $\mathcal{E}_{\mathsf{PC.Eval}}$ that, given any $\mathsf{PC.Eval}$ instance $(\pi, \mathbf{z}, y)$, can extract a valid witness polynomial. On the other hand, by the IOP-to-NARK transformation [46, 31], given any PCS evaluation prover that convinces the verifier with non-negligible probability, there is an efficient extractor $\mathcal{E}_{\mathsf{IOP}}$ that outputs the IOP oracle string, which opens the Merkle commitment sent by the prover.

Now, let $\gamma, \delta' \in (0,1)$ satisfy $\delta' < J_\gamma(J_\gamma(\Delta_{\mathcal{C}_{\mu-m}}))$, and $\Delta_{\mathcal{C}_i} \ge \Delta_{\mathcal{C}_{i+1}}$. Set $\ell \in \mathbb{N}$ such that $\frac{2M(\mu-m)}{\gamma^3|\mathbb{F}|} + (1-\delta'+\gamma(\mu-m))^{M\ell} \le \mathsf{negl}(\lambda)$. Assume that $3\delta' - \gamma(\mu-m) < \Delta_{\mathcal{C}_{\mu-m}}$. We construct the extractor $\mathcal{E}_{\text{dis}}$ in the distributed setting. Given an evaluation instance $(\Pi_{\mu-m}, \mathbf{z}, y)$ that passes the verifier's verification, $\mathcal{E}_{\text{dis}}$ invokes $\mathcal{E}_{\mathsf{IOP}}(\pi_{\mu-m}^{(i)})$ for each $i \in [0 : M-1]$, and derives $(\mathbf{Enc}_{\mu-m}(\mathbf{f}^{(i)}))_{i=0}^{M-1}$. Using $\{\mathbf{T}_j, \mathbf{T}'_j\}_{j=\mu-m}^{\mu-1}$, $\mathcal{E}_{\text{dis}}$ complements the computation of $\pi_\mu = \mathbf{Enc}_\mu(\mathbf{f})$. Then $\mathcal{E}_{\text{dis}}$ computes $\pi = \mathsf{MC}(\pi_\mu)$, and invokes $\mathcal{E}_{\mathsf{PC.Eval}}(\pi, \mathbf{z}, y)$ to derive the witness polynomial f with overwhelming probability. □

4.2.2 Complexity Analysis

Prover cost. For each $\mathcal{P}_i$, encoding costs $O(2^{\mu-m}(\mu-m))$ field operations, and running $\mathsf{IOPP.dCommit}$ and distributed SumCheck costs $O(2^{\mu-m})$ field operations and hashes.
Verifier cost. Verification is dominated by ℓ parallel calls to the protocol $\mathsf{IOPP.dQuery}$, with total cost $O(2^m \ell(\mu-m))$ field operations and $O(2^m \ell(\mu-m)^2)$ hash verifications.
Proof size. The proof size is dominated by $O(2^m \ell(\mu-m))$ Merkle paths for ℓ groups of $\{\pi_j^{(i)}\}_{j\in[0:\mu-m]}^{i\in[0:M-1]}$, each with length $O(\mu-m)$. Hence, the proof size is $O(2^m \ell(\mu-m)^2)$.
Communication costs. The communication costs mainly consist of Merkle paths, so $O(2^m \ell(\mu-m)^2)$ in total.

5 Implementation and Evaluation

Implementation. To implement HyperFond[2], the codes for its distributed PIOP system are modified and revised from the publicly available HyperPlonk[3], while that for the distributed PCS is based on a BaseFold implementation[4]. We then substitute and merge their arithmetic backends into one where the base field $\mathbb{F}$ is a degree-two extension of the *Goldilocks field* $\mathbb{F}_p$, with $p = 2^{64} - 2^{32} + 1$.

Choice of field $\mathbb{F}$ and parallel repetition parameter ℓ. Since our underlying PCS is field-agnostic, it suffices to ensure that $\frac{\mu}{|\mathbb{F}|} = \mathsf{negl}(\lambda)$, without considering the inner structure of $\mathbb{F}$. However, a smaller $|\mathbb{F}|$ results in a larger ℓ to ensure lower soundness error (please refer to Section 4.2.1 for details). To enhance efficiency without compromising security, we set $\mathbb{F} = \mathbb{F}_{p^2}$ as aforementioned; we set $\ell = 565$, the same as in BaseFold when conducting experiments over a 128-bit field for polynomials with around 25 variables, and achieve at least 100 bits of security.

Setup. To evaluate HyperFond, we conduct experiments on vanilla and custom gates, respectively, using up to 16 Ubuntu 24.04 c7n.16xlarge.4 cloud servers, each with 64 vCPUs and 256 GB RAM, in a local area network.

5.1 Linear Scalability

We evaluate HyperFond in various aspects, including proving time, communication cost, proof size, etc., to demonstrate its linear scalability. In each following figure or table, a single machine denotes HyperPlonk [27] instantiated with BaseFold [31].

As in Figure 12, the proving time on both types of circuits reduces as the number of machines increases, as expected in a distributed system. In particular, the proving time of a single machine is 319s and 509s, respectively, on two types of circuits both of sizes 2^{26}; with 16 machines we achieve a 15.6 $\times$ speedup on vanilla circuits, and 16.4 $\times$ on circuits with custom gates.

Other indicators for HyperFond are presented in Table 3. For a fixed circuit size, as the number of machines M doubles, the total communication increases, while that per machine decreases, because the number of rounds -1 as M doubles; the proof size and verification time increase because they depend on M. In particular, for 16 machines generating a proof for 2^{26} circuit, the communication costs are 222MB, while 14.8MB per machine; the proof size is 237MB, and verification time 2.81s. From another perspective, for fixed M, all indicators increase roughly in a polylogarithmic magnitude, as the circuit scales up, which fits our theoretical analysis. In conclusion, the overall efficiency of HyperFond is considerable for large-scale general arithmetic circuits.

[2] https://github.com/n96409816/HyperFond
[3] https://github.com/EspressoSystems/hyperplonk
[4] https://github.com/hadasz/plonkish_basefold

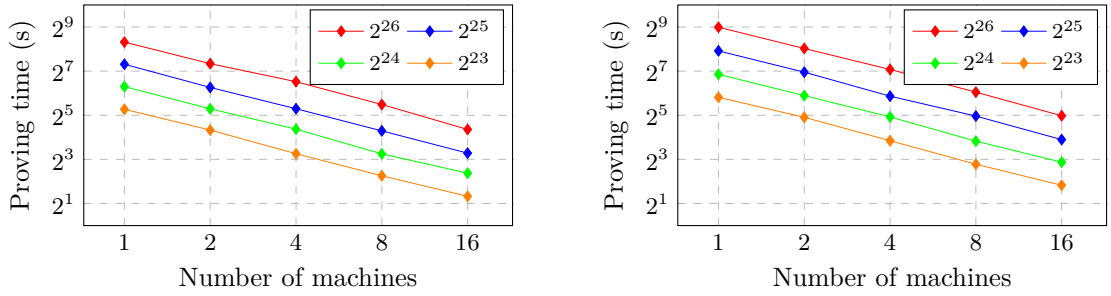

Figure 12: The proving time of HyperFond on vanilla circuits (left) and those with custom gates (right) with various sizes.

Table 3: Other indicators for linear scalability of HyperFond on vanilla circuits.

Circuit size	2^{23}					2^{24}				
Number of machines	1	2	4	8	16	1	2	4	8	16
$\mathcal{P}_i$ / Total Comm. (MB)	-	14.8/14.8	13.2/39.5	11.7/81.8	10.4/156	-	16.7/16.7	14.8/44.4	13.2/92.2	11.7/175
Proof size (MB)	16.7	29.6	52.7	93.5	166	18.8	33.4	59.2	105	187
Verification time (s)	0.190	0.268	0.392	0.594	0.896	0.262	0.375	0.582	0.913	1.33
Circuit size	2^{25}					2^{26}				
Number of machines	1	2	4	8	16	1	2	4	8	16
$\mathcal{P}_i$ / Total Comm. (MB)	-	18.9/18.9	16.7/50.1	14.8/104	13.2/198	-	21.3/21.3	18.9/56.6	16.7/117	14.8/222
Proof size (MB)	21.2	37.7	66.8	118	211	24.0	42.6	75.4	134	237
Verification time (s)	0.361	0.577	0.873	1.29	1.91	0.562	0.869	1.28	1.87	2.81

5.2 Comparing to deVirgo

We conduct a comprehensive complexity comparison of HyperFond to other distributed SNARK generation protocols, as in Table 4. Since deVirgo [19] is by far the only one featuring quantum resistance, we further compare HyperFond to it experimentally. Although deVirgo does not provide source code, it is revised from the non-distributed analogue, Virgo [51], whose underlying field is $\mathbb{F}_{q^2}$, where $q{=}2^{61}{-}1$ is a Mersenne prime. Its repetition parameter is 33, also providing 100+ bits of security. Therefore, our previous setting of the base field $\mathbb{F}$ yields a fair comparison to deVirgo in terms of security, with 17 $\times$ more parallel repetitions. Besides, Virgo is implemented[5] in C++, while the base codes of HyperFond are based on Rust. Since the two languages exhibit comparable engineering performance (typically $< 5\%$), we ignore the tiny difference.

To compare HyperFond to deVirgo under its experimental conditions, hardware is altered to 32 Ubuntu 24.04 ac8.24xlarge.2 cloud servers, each with 96 vCPUs and 192 GB RAM, and we carry out experiments on EdDSA signature verification circuits. By [19, Section 6.1], the size of one circuit is approximately 2^{21}, so we set $\mu = 21 + m$ and use $M = 2^m$ machines in HyperFond to generate proof for M parallel subcircuits. A remaining issue is that deVirgo presents verification costs in gas consumption [19, Table 2] instead of

[5] https://github.com/sunblaze-ucb/Virgo

Table 4: Comparisons of HyperFond to other distributed SNARK generation protocols. Notations are the same as those in Table 1, except that Setup is classified into three categories as circuit-specific (each circuit requires a unique trusted setup), universal (a single trusted setup generates some parameters that can be reused for all circuits) and transparent (requires no trusted setup).

Protocols	$\mathcal{P}_i$ (PIOP)	$\mathcal{P}_i$ (PCS)	Comm.	$\|\pi\|$ and $\mathcal{V}$	Setup	Fully?	PQ?
DIZK [18]	$O(T\log^2(T))$	$O(T\log^2(T))$	$O(N)$	$O(1)$	Circuit-specific	✓	×
Pianist [20]	$O(T\log T)$	$O(T)$	$O(M)$	$O(1)$	Universal	✓	×
Hekaton [21]	$O(T\log T)$	$O(T)$	$O(M)$	$O(\log M)$	Circuit-specific	✓	×
HyperPianist$^{\mathrm{K}}$ [22]	$O(T)$	$O(T)$	$O(M\log T)$	$O(\log N)$	Universal	✓	×
HyperPianist$^{\mathrm{D}}$ [22]	$O(T)$	$O(T)$	$O(M\log T)$	$O(\log N)$	Transparent	✓	×
Cirrus [23]	$O(T)$	$O(T)$	$O(M\log T)$	$O(\log N)$	Universal	✓	×
Soloist [24]	$O(T\log T)$	$O(T\log T)$	$O(M)$	$O(1)$	Universal	✓	×
deVirgo [19]	$O(T)$	$O(T\log T)$	$O(N)$	$O(\log^2(N))$	Transparent	×	✓
HyperFond **(ours)**	$O(T)$	$O(T\log T)$	$O(M\log^2(T))$	$O(M\log^2(T))$	Transparent	✓	✓

Table 5: Comparisons of HyperFond to deVirgo. As in deVirgo, to generate a proof for M signature verifications, we use M machines.

		Proving time (s)		$\mathcal{P}_i$ / Total Comm. (MB)		Proof size (MB)	
Num. of sigs	Circuit size	deVirgo	HyperFond **(ours)**	deVirgo	HyperFond **(ours)**	deVirgo	HyperFond **(ours)**
8	2^{24}	12.52	8.44	942/7516	13.2/92.2	1.86	105
32	2^{26}	12.80	8.82	1034/33013	13.2/408	1.86	421

time. To keep fairness, our comparison focuses on proving time, communication cost, and proof size, which still align with the theme of distribution.

As in Table 5, for each M, HyperFond achieves a significant improvement in proving time and communication costs, at the cost of larger proof size. In particular, for 32 parallel circuits, HyperFond takes 8.82s to generate a proof using 408MB communication costs, and the derived proof size is 421MB. Compared to deVirgo, we achieve a 1.45 × speedup, with 98.8% reduction in communication costs, while 226 × proof size. In conclusion, the proving efficiency of HyperFond, including time and communication, is remarkable in data-parallel circuits (in particular, EdDSA signature verification parallel circuits), even when we set $\ell = 565$; while from the perspective of verification, the rather large proof size reveals its limited capability for large number of parallel subcircuits.

Remark 5: The large proof size essentially results from (i) the trade-off between field-agnosticity and repetition parameter ℓ, and (ii) the way we distribute BaseFold with poly-logarithmic communication. For the former part, if we apply the BaseFold PCS to Reed-Solomon codes [52] (i.e., trading field-agnosticity for a lower soundness error), ℓ can be adjusted to be around 250 to maintain 100+ bits of security, *shrinking more than 50%* of the proof size; for the latter part, this is actually an interesting trade-off between communication cost and proof size, within the context of post-quantum security.

6 Conclusions and Future Directions

We introduced HyperFond, the *first* distributed SNARK enjoying a transparent setup, post-quantum security, and polylogarithmic communication costs, as well as field-agnostic property. Experimental results demonstrate its linear scalability and high efficiency.

For future directions, a possible optimization is to reduce the prover complexity to be strictly linear. Besides, we successfully bound the communication costs to be polylogarithmic, but the proof size and verification time in HyperFond PCS grow linearly with M, the number of sub-provers. It remains open whether we can remove the $O(M)$ overhead while keeping sublinear communication costs.

References

[1] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 326–349, New York, NY, USA, 2012. Association for Computing Machinery.

[2] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K. Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *Advances in Cryptology – ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part III*, page 336–365, Berlin, Heidelberg, 2017. Springer-Verlag.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2087–2104, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943, 2018.

[5] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, ed-

itors, *Advances in Cryptology – CRYPTO 2019*, pages 701–732, Cham, 2019. Springer International Publishing.

[7] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2111–2128, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 733–764, Cham, 2019. Springer International Publishing.

[9] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. In *Advances in Cryptology – EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I*, page 103–128, Berlin, Heidelberg, 2019. Springer-Verlag.

[10] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 704–737, Cham, 2020. Springer International Publishing.

[11] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 738–768, Cham, 2020. Springer International Publishing.

[12] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrú. Gemini: Elastic snarks for diverse environments. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 427–457, Cham, 2022. Springer International Publishing.

[13] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Advances in Cryptology – CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*, page 299–328, Berlin, Heidelberg, 2022. Springer-Verlag.

[14] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023,*

Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part II, page 193–226, Berlin, Heidelberg, 2023. Springer-Verlag.

[15] Hopwood Sean Bowe, T. George Hornby, and Nathan Wilcox. Zcash protocol specication. 2016.

[16] Ethereum Foundation. "zk-rollups: Scaling solutions for ethereum. 2024.

[17] NIST. https://csrc.nist.gov/projects/post-quantum-cryptography.

[18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. Dizk: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, Baltimore, MD, August 2018. USENIX Association.

[19] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3003–3017, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1777–1793, 2024.

[21] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. Hekaton: Horizontally-scalable zksnarks via proof aggregation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 929–940, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Chongrong Li, Pengfei Zhu, Yun Li, Cheng Hong, Wenjie Qu, and Jiaheng Zhang. HyperPianist: Pianist with Linear-Time Prover and Logarithmic Communication Cost . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3383–3401. IEEE Computer Society, 2025.

[23] Wenhao Wang, Fangyan Shi, Dani Vilardell, and Fan Zhang. Cirrus: Performant and accountable distributed SNARK. Cryptology ePrint Archive, Paper 2024/1873, 2024.

[24] Weihan Li, Zongyang Zhang, Yun Li, Pengfei Zhu, Cheng Hong, and Jianwei Liu. Soloist: Distributed SNARKs for rank-one constraint system. Cryptology ePrint Archive, Paper 2025/557, 2025.

[25] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *Electron. Colloquium Comput. Complex.*, 2017.

[26] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[27] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 499–530, Cham, 2023. Springer Nature Switzerland.

[28] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I*, page 677–706, Berlin, Heidelberg, 2020. Springer-Verlag.

[29] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[30] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284, 2023.

[31] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: Efficient field-agnostic polynomial commitment schemes from foldable codes. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 138–169, Cham, 2024. Springer Nature Switzerland.

[32] Verizon. https://www.verizon.com.

[33] AT&T. https://www.att.com.

[34] Deutsche Telekom. https://www.telekom.de.

[35] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, oct 1992.

[36] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. *J. Cryptol.*, 38(2), January 2025.

[37] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.

[38] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography Conference*, pages 222–242. Springer, 2013.

[39] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 1–34, Cham, 2021. Springer International Publishing.

[40] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4291–4308, Boston, MA, August 2022. USENIX Association.

[41] Pankaj Dayama, Arpita Patra, Protik Paul, Nitin Singh, and Dhinakaran Vinayagamurthy. How to prove any NP statement jointly? efficient distributed-prover zero-knowledge protocols. In *Proceedings on Privacy Enhancing Technologies*, 2022.

[42] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zkSNARK provers. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6453–6469, Anaheim, CA, August 2023. USENIX Association.

[43] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: Zero-Knowledge SNARKs as a service. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4427–4444, Anaheim, CA, August 2023. USENIX Association.

[44] Xuanming Liu, Zhelei Zhou, Yinghao Wang, Yanxin Pang, Jinye He, Bingsheng Zhang, Xiaohu Yang, and Jiaheng Zhang. Scalable collaborative zk-SNARK and its application to fully distributed proof delegation. Cryptology ePrint Archive, Paper 2024/940, 2024.

[45] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[46] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings, Part II, of the 14th International Conference on Theory of Cryptography - Volume 9986*, page 31–60, Berlin, Heidelberg, 2016. Springer-Verlag.

[47] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 49–62, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Gal Arnon, Alessandro Chiesa, and Eylon Yogev. Iops with inverse polynomial soundness error. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 752–761, 2023.

[49] Guy N. Rothblum, Salil P. Vadhan, and Avi Wigderson. Interactive proofs of proximity: delegating computation in sublinear time. In *Symposium on the Theory of Computing*, 2013.

[50] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 71–89, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[51] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876, 2020.

[52] Ulrich Haböck. Basefold in the list decoding regime. Cryptology ePrint Archive, Paper 2024/1571, 2024.