UltraFold: Distributed Code-Based PCS from Packed Interleaved Merkle Trees

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

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

***Abstract*—Transparent, code-based polynomial commitment schemes (PCSs) are a foundation for large-scale succinct proofs: they avoid trusted setup, support post-quantum security, and achieve high performance. As proving workloads grow, the computational and memory demands of committing to and opening large polynomials motivate distribution. However, distributing these schemes requires coordinating data across encoding, folding, and Merkle authentication; poorly distributed data introduces repeated communication overhead that grows with the number of workers.**

In this paper, we present `UltraFold`, an efficient framework for distributing a class of code-based PCSs that we call *partition-compatible*. `UltraFold` combines a distributed encoder inspired by Bailey's FFT with an *interleaved* Merkle leaf layout. After a single all-to-all exchange of partially encoded values, each worker can complete encoding and generate subsequent oracle entries and local Merkle authentication paths without further worker-to-worker communication. `UltraFold` maintains a single global commitment to each oracle and preserves the underlying PCS's asymptotic proof size, and supports distribution to an arbitrary number of workers. We further incorporate *packed* Merkle trees to amortize hashing and authentication costs.

We instantiate `UltraFold` with BaseFold (CRYPTO'24), obtaining a distributed PCS for multilinear polynomials. Using 256 single-core workers, our implementation commits to and opens a 2^{30}-coefficient polynomial in under 5 seconds, with a proof size of 281 KB.

1. Introduction

Succinct non-interactive arguments of knowledge (SNARKs) [7], [8], [9], [11] enable efficient verification of computations. Many modern constructions reduce computation verification to polynomial relations and use a *polynomial commitment scheme* (PCS) to authenticate the polynomials involved. A PCS allows a prover to commit to a polynomial f using a short commitment and later prove an evaluation $f(\boldsymbol{z}) = v$. Code-based PCSs [4], [6], [10], [12], [26], [30] are attractive for these constructions because they avoid trusted setup, support post-quantum security, and offer efficient proof generation. These schemes are used in verifiable LLM inference systems, including DeepProve [2], which uses BaseFold [30], and FlashProve [21], as well as in ZK virtual machines such as Ceno [1].

As computation size grows, however, even efficient provers encounter the memory and throughput limits of a single server. Recent works therefore investigate *distributed proof generation*, using multiple machines to increase proving capacity and reduce latency [13], [15], [17], [22], [23], [24], [25]. For code-based PCS, the task is distributing the commitment to and opening of a single large polynomial. Such large polynomials arise from witness tables in general-purpose SNARKs and from workloads such as verifiable matrix multiplication and LLM inference [7], [12], [21].

For a polynomial of fixed size D, distributing the PCS to M workers should reduce the runtime to commit to and open this polynomial without increasing the proof size. Otherwise, scaling the prover also increases the cost of transmitting and verifying proofs, which is not desired. Communication must also be controlled, since repeated exchanges of large vectors, or routing these vectors through a coordinator, can offset the benefits of distribution. These costs become particularly important as the protocol scales to more workers.

We therefore seek a way to distribute code-based PCS with three desiderata: (1) **succinctness**, with proof size polylogarithmic in the polynomial size and independent of the worker count; (2) **scalability**, supporting a large number of workers to accommodate large workloads; and (3) **practical efficiency**, in prover time, proof size, and communication.

Limitations of prior works. Recent work offers preliminary methods for distributing code-based PCSs, but they do not meet the aforementioned desiderata. DeVirgo [25] distributes FRI and has succinct proofs after distribution, but has large communication during commit and open. HyperFond [28] and distributed Brakedown [16] distribute commitment construction, but their proof size is linear in the number of workers. FRIttata [27] makes the tradeoff between communication and proof size tunable through its Fold-and-Batch construction, but its proof size also depends on the number of workers. DePip-FRI [14] packs polynomial fragments into a shared Merkle commitment, while Veloz [29] distributes Ligerito by assigning matrix rows to workers. Both require exchanges of large-scale data during opening, and cannot scale their number of workers beyond 16 for practical polynomial sizes without harming the proof size, limiting their horizontal scalability. These prior works motivate a distribution framework that coordinates encoding, oracle transitions, and Merkle authentication to reduce

communication while keeping proof size independent of the worker count. We discuss related works in Section 7.

Our work: `UltraFold`. We present `UltraFold`, a communication-efficient framework for distributing a class of code-based PCSs that we call *partition-compatible*. These schemes combine recursive encoding with opening computations that preserve a common partition of the encoded data. Concretely, each new oracle segment during polynomial opening can be generated from earlier segments with the same label and the public transcript. Once the encoded data are placed according to this partition, a worker can generate its subsequent oracle segments without fetching entries from other workers, making communication during opening extremely efficient. Some examples of code-based PCS that are partition-compatible include FRI, BaseFold, DeepFold [12], PolyFRIM [31], and the RS-code instantiation of Ligerito [20].

To efficiently distribute partition-compatible code-based PCSs, `UltraFold` adapts the local-transform and transposition ideas of Bailey's FFT [5] to recursive linear code encoders. Workers first perform partial encoding on their local inputs, exchange the intermediate values in one balanced all-to-all redistribution, and complete encoding locally. The redistribution places the output entries according to the partition used throughout the distributed opening phase.

`UltraFold` combines this encoding schedule with an *interleaved* Merkle leaf layout that groups each worker's leaves into a subtree. Workers send their subtree roots to the coordinator, which constructs a small top tree and obtains a single global commitment to the oracle. Each queried leaf is supplied by its owner together with a local authentication path, and the coordinator appends the path from that worker's subtree root to the global root. The resulting path has the same length as a path in the corresponding non-distributed PCS. Consequently, `UltraFold` has the same proof size as the underlying PCS.

To further improve runtime, we identify that the practical bottleneck in worker computation is building Merkle trees. `UltraFold` therefore incorporates *packing*, so that multiple polynomial coefficients are packed into a single Merkle leaf, amortizing the cost of Merkle tree construction. This also shortens Merkle paths, reducing proof sizes.

We instantiate `UltraFold` with BaseFold, FRI, and DeepFold. As summarized in Table 1, `UltraFold` instantiated with BaseFold simultaneously scales to a large number of workers (more than poly-logarithmic in D), achieves succinct proof size, and delivers practical prover efficiency.

Implementation and evaluation. We implement `UltraFold` instantiated with BaseFold in Rust using Arkworks 0.5 [3] and evaluate it with up to 256 single-core workers. With 256 workers, `UltraFold` commits to and opens a size-2^{30} polynomial in 4.66 seconds, including communication. The resulting proof is 281 KB and the verification time is 0.956 ms using eight threads. We also measure communication costs for both workers and the coordinator, which shows that although there is heavy communication during commit, `UltraFold`'s communication is very light during opening and is below 50 KB per worker with 16 workers and a size-2^{30} polynomial.

We compare `UltraFold` with the other distributed code-based PCS schemes, DeVirgo [25], DePip-FRI [14], and Veloz-RS [29] on the same hardware and with the same code rate and 16 workers. At polynomial size 2^{30}, `UltraFold` achieves $1.34\times$ and $4.87\times$ speedups in commitment and opening, respectively, over Veloz-RS, and produces proofs over 42% smaller than those of DePip-FRI and Veloz-RS. At this setting, its total opening communication is 0.50 MB, compared with 4,027 MB for DePip-FRI and 5,131 MB for Veloz-RS, while commitment communication is approximately the same across the three schemes.

Contributions

We present `UltraFold`, a communication-efficient framework for distributing partition-compatible code-based PCSs. We instantiate `UltraFold` with BaseFold, FRI, and DeepFold, and implement the BaseFold instantiation to demonstrate practical performance at scale.

2. Preliminaries

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

Notation. We use boldface $\boldsymbol{v}$ to denote a vector and $\boldsymbol{v}[i]$ to denote its i-th element. Let $\mathbb{F}$ denote a finite field of size $\Omega(2^\lambda)$, where λ is the security parameter, and let $B_\mu := \{0,1\}^\mu$ denote the μ-dimensional Boolean hypercube. We write $\mathbb{F}_\mu^{\leq d}[\boldsymbol{X}]$ for the set of μ-variate polynomials whose degree in each variable is at most d; a multivariate polynomial is *multilinear* if its degree in each variable is at most 1. For $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^\mu$, let $\tilde{eq}(\boldsymbol{a}, \boldsymbol{b}) := \prod_{i=1}^{\mu}(a_i b_i + (1-a_i)(1-b_i))$ denote the μ-dimension multilinear Lagrange polynomial. Note that when $\boldsymbol{a}, \boldsymbol{b} \in B_\mu$, $\tilde{eq}(\boldsymbol{a}, \boldsymbol{b})$ is 1 if and only if $\boldsymbol{a} = \boldsymbol{b}$, and is 0 otherwise. For $\boldsymbol{x} = (x_1, \ldots, x_\mu) \in \mathbb{F}^\mu$, define $[\boldsymbol{x}] := \sum_{i=0}^{\mu-1} 2^i \cdot x_{\mu-i}$, and let $\langle v \rangle_m$ denote the m-bit representation of an integer $v \in [0, 2^m - 1]$.

2.1. Definitions

Merkle tree. A Merkle tree [18] is a vector commitment that enables efficient openings of any entries of a length-n vector. It has three algorithms: (1) $\mathsf{Merkle.Commit}(\boldsymbol{x}) \to \mathsf{rt}$ outputs the Merkle root, (2) $\mathsf{Merkle.Open}(\mathsf{rt}, i, \boldsymbol{x}) \to (x_i, \mathsf{path})$ outputs the i-th entry of $\boldsymbol{x}$ with an opening path path, and (3) $\mathsf{Merkle.Verify}(\mathsf{rt}, i, x_i, \mathsf{path}) \to \{0,1\}$ outputs 1 if and only if rt and x_i agree with path.

Linear code. Let $\mathbb{F}$ be a finite field. A linear code $\mathsf{C} \subseteq \mathbb{F}^Q$ of blocklength Q and dimension D is a D-dimensional linear subspace of $\mathbb{F}^Q$. It has an injective linear encoder $\mathsf{Enc} : \mathbb{F}^D \to \mathbb{F}^Q$, represented by a matrix $E \in \mathbb{F}^{Q \times D}$ as $\mathsf{Enc}(\boldsymbol{t}) = E\boldsymbol{t}$. The code rate is $\rho := D/Q$, and the blow-up factor is $c := Q/D = 1/\rho$. Linearity implies that $\mathsf{Enc}(\alpha\boldsymbol{u} + \beta\boldsymbol{v}) = \alpha\mathsf{Enc}(\boldsymbol{u}) + \beta\mathsf{Enc}(\boldsymbol{v})$ for all $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^D$ and $\alpha, \beta \in \mathbb{F}$.

TABLE 1. COMPARISON WITH PRIOR TRANSPARENT DISTRIBUTED CODE-BASED PCSS. WE CONSIDER TOTAL POLYNOMIAL SIZE D AND M WORKERS, WITH $N := D/M$. COSTS INCLUDE ONE COMMITMENT AND ONE OPENING. COMMUNICATION IS AMORTIZED PER WORKER, COUNTS FIELD ELEMENTS AND HASH DIGESTS, INCLUDES PROOF COLLECTION, AND EXCLUDES DELIVERY FROM THE COORDINATOR TO THE VERIFIER. $\mathbb{F}$ COUNTS FIELD OPERATIONS AND $\mathbb{H}$ COUNTS HASH OPERATIONS.

Scheme	Max workers for listed proof size	Proof Size	Worker Time	Worker Comm.
`UltraFold` (BaseFold)	$\boldsymbol{O(\sqrt{D/\log D})}$	$\boldsymbol{O(\lambda \log^2 D)}$	$\boldsymbol{O(N \log D)\mathbb{F}}$, $\boldsymbol{O(N/\log D)\mathbb{H}}$	$\boldsymbol{O(N + \lambda \log^2 D/M)}$
HyperFond [28]	$O(D)$	$O(\lambda M \log^2 N)$	$O(N \log N)\mathbb{F}$, $O(N)\mathbb{H}$	$O(\lambda \log^2 N)$
DeVirgo [25]	$O(D)$	$O(\lambda(M + \log^2 N))$	$O(N \log N)\mathbb{F}$, $O(N)\mathbb{H}$	$O(N + \lambda(M + \log^2 N)/M)$
FRIttata (Fold-and-Batch) [27]	$O(D)$	$O(\lambda\left[M(1 + \log(N/K)) \log N + \log^2 K\right])$	$O(N \log N)\mathbb{F}$, $O(N)\mathbb{H}$	$O(K + \lambda(1 + \log(N/K)) \log N)$
DePip-FRI [14]	$O(\log^2 N)^\dagger$	$O(\lambda \log^2 N))$	$O(N \log N)\mathbb{F}$, $O(N/M)\mathbb{H}$	$O(N + \lambda(M + \log^2 N)/M)$
Veloz-RS [29]	$O(\log D)^\P$	$O\left(\lambda \frac{\log^2 D}{\log \log D}\right)$	$O\left(N \log \frac{D}{\log D}\right)\mathbb{F}$, $O(N)\mathbb{H}$	$O\left(\frac{D}{\log D} + \frac{\lambda \log^2 D}{M \log \log D}\right)$

† DePip-FRI has proof size $O(\lambda(M + \log^2 N))$, which reduces to the listed bound when $M = O(\log^2 N)$. Larger worker counts increase the proof size. ¶ Veloz-RS uses its proof-size parameters. Larger worker counts increase the proof size.

Polynomial commitment scheme. On a high level, a PCS securely implements polynomial oracles and queries to such oracles. Here we define PCS as follows.

Definition 1 (Polynomial Commitment Scheme)**.** *A polynomial commit scheme for a class of functions $\mathcal{F}$ is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{VerPoly}, \mathsf{Eval})$*, where*

- $\mathsf{KeyGen}(1^\lambda, \mathcal{F}) \to \mathsf{pp}$*: generates the public parameter* pp*.*
- $\mathsf{Commit}(\mathsf{pp}, f) \to \mathsf{com}_f$*: generates a commitment* com_f *of a function* f*.*
- $\mathsf{VerPoly}(\mathsf{pp}, \mathsf{com}_f, f) \to \{0, 1\}$*: takes a polynomial and a commitment and outputs* 1 *if and only if* com_f *is the commitment of* f*.*
- $\mathsf{Eval}(\mathsf{pp}, \mathsf{com}_f, \boldsymbol{x}, y)$*: an interactive protocol between $\mathcal{P}$ and $\mathcal{V}$ where $\mathcal{P}$ convinces $\mathcal{V}$ that $f(\boldsymbol{x}) = y$ and com_f is a commitment of f.*

A PCS protocol should have the following properties:

- ***Completeness****: For all $f \in \mathcal{F}$ and for all $\boldsymbol{x}$*

$$\Pr\left[b = 1 \;\middle|\; \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{F}) \\ \mathsf{com}_f \leftarrow \mathsf{Commit}(f, \mathsf{crs}) \\ (y, \pi) \leftarrow \mathsf{Open}(f, \boldsymbol{x}, \mathsf{crs}) \\ b \leftarrow \mathsf{Verify}(\mathsf{com}_f, \boldsymbol{x}, y, \pi, \mathsf{crs}) \end{array}\right] = 1 - \mathsf{negl}(\lambda)$$

- ***Binding****: For all PPT adversary $\mathcal{A}$ the following probability is* $\mathsf{negl}(\lambda)$*.*

$$\Pr\left[b_1 = b_2 = 1 \wedge f_1 \neq f_2 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{F}) \\ (\mathsf{com}, f_1, f_2) \leftarrow \mathcal{A}(\mathsf{pp}) \\ b_1 \leftarrow \mathsf{VerPoly}(\mathsf{pp}, \mathsf{com}, f_1) \\ b_2 \leftarrow \mathsf{VerPoly}(\mathsf{pp}, \mathsf{com}, f_2) \end{array}\right]$$

Knowledge soundness*: For any PPT adversaries $\mathcal{A}_1, \mathcal{A}_2$, there exists a PPT extractor $\mathcal{E}$ with oracle access to $\mathcal{A}_1, \mathcal{A}_2$ such that the following probability is* $\mathsf{negl}(\lambda)$*:*

$$\Pr\left[\begin{array}{l} b = 1 \wedge \\ \big(\mathsf{VerPoly}(\mathsf{pp}, \mathsf{com}, f) = 0 \\ \quad \vee \;\; f(\boldsymbol{x}) \neq y\big) \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{F}) \\ (\mathsf{com}, \boldsymbol{x}, y, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{pp}) \\ f \leftarrow \mathcal{E}^{\mathcal{A}_1, \mathcal{A}_2}(\mathsf{pp}, \mathsf{com}, \boldsymbol{x}, y, \mathsf{st}) \\ b \leftarrow \langle \mathcal{A}_2(\mathsf{st}), \mathcal{V} \rangle(\mathsf{pp}, \mathsf{com}, \boldsymbol{x}, y) \end{array}\right].$$

2.2. Security Model

Participants in `UltraFold` consist of a coordinator $\mathcal{C}$, $M = 2^m$ workers $\mathcal{P}_1, \ldots, \mathcal{P}_M$, and a verifier $\mathcal{V}$. Note that $\mathcal{C}$ also counts as a party responsible for generating the proof alongside workers. We assume point-to-point communication channels between $\mathcal{C}$ and workers, between each pair of workers, and between $\mathcal{C}$ and $\mathcal{V}$. Note that $\mathcal{V}$ only communicates with $\mathcal{C}$. We assume the adversary may corrupt any subset $S \subseteq [M]$ of workers and/or the coordinator $\mathcal{C}$; corrupted parties can arbitrarily collude. We require the soundness and knowledge soundness properties of the PCS to hold against this adversary.

3. Partition-Compatible Code-Based PCSs

We identify a class of code-based polynomial commitment schemes, which we call *partition-compatible code-based PCSs*, whose computational structure supports communication-efficient distributed proving. These schemes combine recursive encoding during commitment with oracle transitions that preserve a common partition during opening. We describe these structural requirements following the two stages of the PCS.

Setting. Let $D = 2^d$, $M = 2^m$, and $N = D/M = 2^n$, where $0 \leq m \leq d$ and $n = d - m$. Here, M is the number of segments in a partition. We represent a multilinear polynomial f by its evaluation vector $\boldsymbol{t} \in \mathbb{F}^D$ over the Boolean hypercube B_d: $f(\boldsymbol{X}) = \sum_{\boldsymbol{a} \in B_d} \boldsymbol{t}[\boldsymbol{a}] \cdot \widetilde{\mathrm{eq}}(\boldsymbol{a}, \boldsymbol{X})$. For $\boldsymbol{a} \in B_n$ and $\boldsymbol{b} \in B_m$, we fix the input indexing by $\boldsymbol{t}[\boldsymbol{a} \| \boldsymbol{b}] = f(\boldsymbol{a}, \boldsymbol{b})$. All coordinate orders, input reshaping rules, and Merkle leaf groupings are public.

3.1. Commitment: Recursive Encoding and Merkle Authentication

Recursive encoding. We consider linear codes with encoders from a family $\mathsf{Enc}_h : \mathbb{F}^{2^h} \longrightarrow \mathbb{F}^{q_h}$ where $q_h := c2^h$, c is the code blow-up factor, and $\mathsf{Enc}_0 : \mathbb{F} \to \mathbb{F}^c$ is a linear map. At depth $h \geq 1$, split the input vector $\boldsymbol{v}$ into two equal-sized subvectors $\boldsymbol{v}_\ell$ and $\boldsymbol{v}_r$ according to the encoder's fixed coordinate order. Recursively compute $\boldsymbol{\ell} := \mathsf{Enc}_{h-1}(\boldsymbol{v}_\ell)$ and $\boldsymbol{r} := \mathsf{Enc}_{h-1}(\boldsymbol{v}_r)$.

For each $j \in [0, q_{h-1})$, combine the corresponding entries as

$$\begin{pmatrix} \mathsf{Enc}_h(\boldsymbol{v})[j] \\ \mathsf{Enc}_h(\boldsymbol{v})[j + q_{h-1}] \end{pmatrix} = \begin{pmatrix} a_{h,j} & b_{h,j} \\ c_{h,j} & d_{h,j} \end{pmatrix} \begin{pmatrix} \boldsymbol{\ell}[j] \\ \boldsymbol{r}[j] \end{pmatrix},$$

where each two-by-two matrix is public and nonsingular, with efficiently computable entries. Thus, an encoding layer combines pairs of corresponding entries from two smaller encodings. The coefficients and coordinate order are specified by the PCS instantiation.

For a single encoded vector, the initial oracle is $\boldsymbol{\pi}_0 = \mathsf{Enc}_d(\boldsymbol{t})$. We write $\boldsymbol{\pi}_0 = E\boldsymbol{t}$, where $E \in \mathbb{F}^{Q_0 \times D}$ and $Q_0 = cD$ for the complete encoded data.

Merkle commitment. The prover constructs a Merkle tree over $\boldsymbol{\pi}_0$ using the prescribed leaf grouping and ordering, and publishes its root $\mathsf{rt}_0 := \mathsf{MerkleRoot}(\boldsymbol{\pi}_0)$. The root rt_0 is part of commitment to f.

3.2. Opening: Partition-Preserving Oracle Transitions

To open the commitment at $\boldsymbol{z} \in \mathbb{F}^d$ with claimed value v, the prover generates additional authenticated oracles whose consistency with the initial oracle supports verification of the evaluation claim. We characterize the structure of these oracle computations.

Oracle transitions. During opening, the prover generates additional oracles $\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_R$, where $\boldsymbol{\pi}_i \in \mathbb{F}^{Q_i}$. We require each complete oracle transition before the specified terminal stage to admit an affine representation

$$\boldsymbol{\pi}_i = \boldsymbol{a}_i(\tau_i) + \sum_{h=0}^{i-1} T_{i,h}(\tau_i)\boldsymbol{\pi}_h, \tag{1}$$

where τ_i contains the public inputs, challenges, and short transcript messages available before $\boldsymbol{\pi}_i$ is generated. The matrices $T_{i,h}(\tau_i) \in \mathbb{F}^{Q_i \times Q_h}$ and the offset $\boldsymbol{a}_i(\tau_i) \in \mathbb{F}^{Q_i}$ are publicly determined.

Partition compatibility. Fix a partition parameter $M = 2^m$. For the initial oracle and every subsequent oracle before the specified terminal stage, we require $M \mid Q_i$ and use the cyclic partition $S_i^{(\boldsymbol{b})} := \{b + Mt : 0 \le t < Q_i/M\}$ and $\boldsymbol{\pi}_i^{(\boldsymbol{b})} := \boldsymbol{\pi}_i\big|_{S_i^{(\boldsymbol{b})}}$, where b is the integer represented by $\boldsymbol{b} \in B_m$. Therefore, position j belongs to segment $j \bmod M$, and each segment contains exactly Q_i/M elements. The partition is fixed and is independent of the polynomial and verifier challenges.

We require each oracle transition to preserve these segment labels. Specifically, the affine representation in Eq. (1) must satisfy $T_{i,h}(\tau_i)[u, v] \neq 0$ only if $u \bmod M = v \bmod M$ for every $h < i$ and every possible transcript τ_i.

Define $T_{i,h}^{(\boldsymbol{b})}(\tau_i) := T_{i,h}(\tau_i)[S_i^{(\boldsymbol{b})}, S_h^{(\boldsymbol{b})}]$ and let $\boldsymbol{a}_i^{(\boldsymbol{b})}(\tau_i)[t] := \boldsymbol{a}_i(\tau_i)[b + Mt]$ for $t \in [0, Q_i/M)$. The transition then decomposes as

$$\boldsymbol{\pi}_i^{(\boldsymbol{b})} = \boldsymbol{a}_i^{(\boldsymbol{b})}(\tau_i) + \sum_{h=0}^{i-1} T_{i,h}^{(\boldsymbol{b})}(\tau_i)\boldsymbol{\pi}_h^{(\boldsymbol{b})}. \tag{2}$$

The segmentwise maps and offsets must admit efficient computation. Consequently, given the public transcript, every segment labeled $\boldsymbol{b}$ can be derived from $\boldsymbol{\pi}_0^{(\boldsymbol{b})}$ without accessing oracle segments with other labels.

Terminal computation. Each instantiation specifies a terminal stage at which the remaining active polynomial and oracle state contains at most $H = H(D, M)$ field elements. Partition preservation is required until this stage, after which the residual computation may operate on this state as a whole. The bound H excludes earlier committed oracle data retained solely for authentication queries.

Oracle authentication and verification. The prover authenticates each newly generated oracle by a Merkle root $\mathsf{rt}_i := \mathsf{MerkleRoot}(\boldsymbol{\pi}_i)$. At the query stages prescribed by the PCS, the verifier requests oracle entries together with their Merkle authentication paths. It checks the paths against the recorded roots and verifies the corresponding algebraic consistency relations. The prover also supplies the terminal polynomial or oracle data required by the protocol.

Definition 2 (Partition-compatible code-based PCS)**.** *Fix a partition parameter $M = 2^m$ and a terminal state size bound $H(D, M)$. A code-based PCS is* partition-compatible *for M segments if its commitment uses the recursive encoding and Merkle authentication described above, and its honest opening computation admits affine oracle transitions that preserve the cyclic partitions, with efficient segmentwise implementations as in Eq.* (2)*, until the specified terminal stage. The prescribed Merkle leaf grouping must keep all coordinates of each leaf within one segment.*

3.3. Examples of Partition-Compatible Code-Based PCSs

We instantiate the recursive encoding and partition-preserving transitions for several constructions. Let $K_{h,j}$ denote the public two-by-two combining matrix at encoding depth h and position j.

FRI. We use the multilinear representation of the Reed-Solomon encoding underlying FRI [6]. For $h \ge 1$, define $\boldsymbol{\Gamma}_h(\zeta) := (\zeta, \zeta^2, \ldots, \zeta^{2^{h-1}})$ and let $\boldsymbol{\Gamma}_0$ denote the empty tuple. Let $\mathcal{H}_h = \{\zeta_{h,j}\}_{j=0}^{q_h-1}$ be a multiplicative coset of size $q_h = c2^h$, with the domains and their orderings chosen so that $\zeta_{h,j+q_h/2} = -\zeta_{h,j}$ and $\zeta_{h-1,j} = \zeta_{h,j}^2$ for $0 \le j < q_h/2$. For a Boolean evaluation vector $\boldsymbol{v} \in \mathbb{F}^{2^h}$ representing the multilinear polynomial f, define $\mathsf{Enc}_h(\boldsymbol{v})[j] := f(\boldsymbol{\Gamma}_h(\zeta_{h,j}))$.

To obtain the recursive encoding rule, $f(0, \cdot)$ and $f(1, \cdot)$, respectively. the combining matrix is

$$K_{h,j}^{\mathsf{RS}} = \begin{pmatrix} 1 - \zeta_{h,j} & \zeta_{h,j} \\ 1 + \zeta_{h,j} & -\zeta_{h,j} \end{pmatrix},$$

which is nonsingular since $\zeta_{h,j} \neq 0$.

To derive oracle transitions, let $f_0 = f$ and define $f_i(\boldsymbol{X}) := f(\alpha_0, \ldots, \alpha_{i-1}, \boldsymbol{X})$. f_i has $d - i$ remaining variables and $f_{i+1}(\boldsymbol{X}) = f_i(\alpha_i, \boldsymbol{X})$. The corresponding oracle has length $Q_i = c2^{d-i}$ and entries $\boldsymbol{\pi}_i[j] = f_i(\boldsymbol{\Gamma}_{d-i}(x_{i,j}))$ with $x_{i,j} := \zeta_{d-i,j}$. Interpolating in this coordinate at α_i gives $\boldsymbol{\pi}_{i+1}[j] = f_i\big(\alpha_i, \boldsymbol{\Gamma}_{d-i-1}(x_{i,j}^2)\big) = \frac{x_{i,j}+\alpha_i}{2x_{i,j}}\boldsymbol{\pi}_i[j] +$

$\frac{x_{i,j}-\alpha_i}{2x_{i,j}}\boldsymbol{\pi}_i[j+Q_i/2]$. Therefore, the transition is $\boldsymbol{\pi}_{i+1} = T_{i+1,i}(\tau_{i+1})\boldsymbol{\pi}_i$ where τ_{i+1} includes α_i, and

$$T_{i+1,i}(\tau_{i+1})[j,k] = \begin{cases} \dfrac{x_{i,j}+\alpha_i}{2x_{i,j}}, & k = j, \\ \dfrac{x_{i,j}-\alpha_i}{2x_{i,j}}, & k = j + Q_i/2, \\ 0, & \text{otherwise.} \end{cases}$$

This instantiates Eq. (1) with $\boldsymbol{a}_{i+1} = \mathbf{0}$ and $T_{i+1,h} = 0$ for $h < i$.

Whenever $M \mid Q_i/2$, both input positions and the output position have the same remainder modulo M. Therefore, this partition gives $\boldsymbol{\pi}^{(\boldsymbol{b})}_{i+1} = T^{(\boldsymbol{b})}_{i+1,i}\boldsymbol{\pi}^{(\boldsymbol{b})}_i$ as required by Eq. (2).

BaseFold. BaseFold [30] uses public parameters $\theta_{h,j} \neq \theta'_{h,j}$ to combine recursively encoded vectors. In the Boolean-evaluation representation, its combining matrix is

$$K^{\mathsf{BF}}_{h,j} = \begin{pmatrix} 1-\theta_{h,j} & \theta_{h,j} \\ 1-\theta'_{h,j} & \theta'_{h,j} \end{pmatrix}. \tag{3}$$

During opening, $Q_{i+1} = Q_i/2$, and each output entry is obtained by interpolating a pair of entries from $\boldsymbol{\pi}_i$. For the pair $(j, j+Q_i/2)$, abbreviate its encoding parameters by θ, θ'. The transition matrix is

$$T_{i+1,i}(\tau_{i+1})[j,k] = \begin{cases} \dfrac{\theta'-\alpha_i}{\theta'-\theta}, & k = j, \\ \dfrac{\alpha_i-\theta}{\theta'-\theta}, & k = j + Q_i/2, \\ 0, & \text{otherwise,} \end{cases}$$

where τ_{i+1} includes the folding challenge α_i. Thus, Eq. (1) holds with $\boldsymbol{a}_{i+1} = \mathbf{0}$ and $T_{i+1,h} = 0$ for $h < i$.

Whenever $M \mid Q_i/2$, both input and output positions have the same remainder modulo M. Consequently, defining $T^{(\boldsymbol{b})}_{i+1,i} := T_{i+1,i}[S^{(\boldsymbol{b})}_{i+1}, S^{(\boldsymbol{b})}_i]$, we have $\boldsymbol{\pi}^{(\boldsymbol{b})}_{i+1} = T^{(\boldsymbol{b})}_{i+1,i}\boldsymbol{\pi}^{(\boldsymbol{b})}_i$ as required by Eq. (2). For $D = MN$, the partition-preserving phase may stop after $d-m$ folds, leaving an oracle of length cM and a residual polynomial of size M.

DeepFold. We express the Reed-Solomon encoding in DeepFold [12] using the same multilinear representation as in the FRI example. For a Boolean evaluation vector representing f, the initial oracle satisfies $\boldsymbol{\pi}_0[j] = f(\boldsymbol{\Gamma}_d(x_{0,j}))$ and the recursive combining matrix is $K^{\mathsf{DF}}_{h,j} = K^{\mathsf{RS}}_{h,j}$. We note that there is an extra evaluation in DeepFold's commitment besides the Merkle root, which is $f(\boldsymbol{\Gamma}_d(\beta))$ at a random point β.

During opening, define $f_i(\boldsymbol{X}) := f(\alpha_0, \ldots, \alpha_{i-1}, \boldsymbol{X})$. The oracle at level i is $\boldsymbol{\pi}_i[j] = f_i(\boldsymbol{\Gamma}_{d-i}(x_{i,j}))$ where $Q_i = c2^{d-i}$. The next polynomial is $f_{i+1}(\boldsymbol{X}) = f_i(\alpha_i, \boldsymbol{X})$. Therefore, its oracle is obtained by the same interpolation between positions j and $j + Q_i/2$ as in FRI.

Write $F_{\beta,i}$ for the transition matrix in the FRI example with folding challenge β. DeepFold instantiates Eq. (1) with $T_{i+1,i}(\tau_{i+1}) = F_{\alpha_i,i}$, where $\boldsymbol{a}_{i+1} = \mathbf{0}$ and $T_{i+1,h} = 0$ when $h < i$. Whenever $M \mid Q_i/2$, the input and output positions have the same remainder modulo M. Therefore, $\boldsymbol{\pi}^{(\boldsymbol{b})}_{i+1} = F^{(\boldsymbol{b})}_{\alpha_i,i}\boldsymbol{\pi}^{(\boldsymbol{b})}_i$ as required by Eq. (2).

Ligerito. We consider an instantiation of Ligerito [20] using the Reed-Solomon encoders from the FRI example. At iteration i, let $A_i = 2^{\mu_i}$ and $B_i = 2^{\nu_i}$. Write the current multilinear polynomial as $f_i(\boldsymbol{u}, \boldsymbol{x})$, where $\boldsymbol{u}$ has ν_i variables and $\boldsymbol{x}$ has μ_i variables. Its Boolean evaluation table is arranged as a matrix $U_i \in \mathbb{F}^{A_i \times B_i}$ with $U_i[\boldsymbol{a}, \boldsymbol{u}] := f_i(\boldsymbol{u}, \boldsymbol{a})$ for $\boldsymbol{a} \in B_{\mu_i}$ and $\boldsymbol{u} \in B_{\nu_i}$. Encoding each column gives $X_i[j, \boldsymbol{u}] = f_i(\boldsymbol{u}, \boldsymbol{\Gamma}_{\mu_i}(x_{i,j}))$ where $x_{i,j} := \zeta_{\mu_i,j}$. Equivalently, $X_i = G_i U_i \in \mathbb{F}^{cA_i \times B_i}$, where G_i is the generator matrix of the Reed-Solomon code. We set $\boldsymbol{\pi}_i = \mathrm{vec}(X_i)$ and $Q_i = cA_iB_i$, where vec stacks columns to make a vector. Independent column encoding fits the recursive commitment structure by using $K^{\mathsf{RS}}_{h,j}$ within each column and $K_{h,j} = I_2$ above the column-encoding depth.

Let $\boldsymbol{r}_i \in \mathbb{F}^{\nu_i}$ be the partial-evaluation challenges, and define $\boldsymbol{w}_i[\boldsymbol{u}] := \widetilde{\mathrm{eq}}(\boldsymbol{u}, \boldsymbol{r}_i)$. The reduced table $\boldsymbol{y}_i := U_i\boldsymbol{w}_i$ represents the polynomial $f_i(\boldsymbol{r}_i, \boldsymbol{x})$. By linearity,

$$(X_i\boldsymbol{w}_i)[j] = f_i\big(\boldsymbol{r}_i, \boldsymbol{\Gamma}_{\mu_i}(x_{i,j})\big) = (G_i\boldsymbol{y}_i)[j].$$

Thus, combining the entries of each encoded row produces an encoding of the partially evaluated polynomial.

The next iteration reshapes $\boldsymbol{y}_i$ into U_{i+1}, where $A_i = A_{i+1}B_{i+1}$. In polynomial notation, this amounts to splitting the remaining variables into two blocks $f_{i+1}(\boldsymbol{v}, \boldsymbol{x}) := f_i(\boldsymbol{r}_i, \boldsymbol{v}, \boldsymbol{x})$. The next encoded matrix therefore satisfies $X_{i+1}[j, \boldsymbol{v}] = f_{i+1}(\boldsymbol{v}, \boldsymbol{\Gamma}_{\mu_{i+1}}(x_{i+1,j}))$.

For the selected nested encoders, inverting the top ν_{i+1} encoding layers recovers the encodings of the restrictions indexed by $\boldsymbol{v} \in B_{\nu_{i+1}}$. These inversions define a public linear map $\mathcal{U}_i$ satisfying $\mathcal{U}_i G_i = I_{B_{i+1}} \otimes G_{i+1}$. Consequently, the complete oracle transition is $T_{i+1,i}(\tau_{i+1}) := \mathcal{U}_i(\boldsymbol{w}_i^\top \otimes I_{cA_i})$ such that $\boldsymbol{\pi}_{i+1} = T_{i+1,i}(\tau_{i+1})\boldsymbol{\pi}_i$, with $\boldsymbol{a}_{i+1} = \mathbf{0}$ and $T_{i+1,h} = 0$ for $h < i$.

Each encoded row forms a Merkle leaf. We partition the oracle by complete rows:

$$S^{(\boldsymbol{b})}_i := \left\{ j + s\,cA_i : \begin{array}{l} 0 \le j < cA_i,\ 0 \le s < B_i, \\ j \bmod M = b \end{array} \right\}.$$

Writing $q := cA_{i+1}$, the entries of next-oracle row j depend only on preceding rows indexed by $j, j+q, \ldots, j+(B_{i+1}-1)q$. Whenever $M \mid q$, all these rows have the same remainder modulo M. Therefore, defining $T^{(\boldsymbol{b})}_{i+1,i} := T_{i+1,i}[S^{(\boldsymbol{b})}_{i+1}, S^{(\boldsymbol{b})}_i]$ we obtain $\boldsymbol{\pi}^{(\boldsymbol{b})}_{i+1} = T^{(\boldsymbol{b})}_{i+1,i}\boldsymbol{\pi}^{(\boldsymbol{b})}_i$ as required by Eq. (2).

PolyFRIM. Consider the rolling batch FRI of PolyFRIM [31]. Its recursive encoder uses $K^{\mathsf{PF}}_{h,j} = K^{\mathsf{RS}}_{h,j}$.

For an evaluation point $\boldsymbol{z} = (z_1, \ldots, z_d)$, define $f_i(\boldsymbol{X}) := f(z_1, \ldots, z_i, \boldsymbol{X})$ and $f_{i+1}(\boldsymbol{X}) = f_i(z_{i+1}, \boldsymbol{X})$. The oracle is $\boldsymbol{A}_i[j] := f_i\big(\boldsymbol{\Gamma}_{d-i}(x_{i,j})\big)$ For the rolling batch FRI oracle chain, set $h_0 = f$ and, at nonterminal levels, define additional polynomials $g_{i+1}(\boldsymbol{X}) := h_i(\alpha_{i+1}, \boldsymbol{X})$ and $h_{i+1}(\boldsymbol{X}) := g_{i+1}(\boldsymbol{X}) + \alpha^2_{i+1} f_{i+1}(\boldsymbol{X})$, where α_{i+1} is a verifier challenge. Let $\boldsymbol{P}_i$ and $\boldsymbol{V}_i$ be the evaluations of g_i and h_i, respectively, at the same point $\boldsymbol{\Gamma}_{d-i}(x_{i,j})$.

Using the matrix $F_{\beta,i}$ defined above, these polynomial restrictions and combinations satisfy $\boldsymbol{A}_{i+1} = F_{z_{i+1},i}\boldsymbol{A}_i$, $\boldsymbol{P}_{i+1} = F_{\alpha_{i+1},i}\boldsymbol{V}_i$, and $\boldsymbol{V}_{i+1} = \boldsymbol{P}_{i+1} + \alpha^2_{i+1}\boldsymbol{A}_{i+1}$.

The evaluation folding transition has one nonzero matrix $F_{z_{i+1},i}$ and zero offset. For the rolling chain in PolyFRIM, $\boldsymbol{P}_1 = F_{\alpha_1,0}\boldsymbol{A}_0$. For $i \geq 1$, eliminating $\boldsymbol{V}_i$ yields $\boldsymbol{P}_{i+1} = F_{\alpha_{i+1},i}\boldsymbol{P}_i + \alpha_i^2 F_{\alpha_{i+1},i}\boldsymbol{A}_i$. If $\boldsymbol{\pi}_k = \boldsymbol{P}_{i+1}$, $\boldsymbol{\pi}_p = \boldsymbol{P}_i$, and $\boldsymbol{\pi}_a = \boldsymbol{A}_i$, then $T_{k,p} = F_{\alpha_{i+1},i}$, $T_{k,a} = \alpha_i^2 F_{\alpha_{i+1},i}$ and $\boldsymbol{a}_k = \boldsymbol{0}$, and all other transition matrices are zero.

At each level, we apply the same cyclic partition to both the folding of polynomials and rolling-FRI oracles, assigning position j in each oracle to segment $j \bmod M$. Whenever $M \mid Q_i/2$, $\boldsymbol{A}_{i+1}^{(\boldsymbol{b})} = F_{z_{i+1},i}^{(\boldsymbol{b})}\boldsymbol{A}_i^{(\boldsymbol{b})}$ and $\boldsymbol{P}_{i+1}^{(\boldsymbol{b})} = F_{\alpha_{i+1},i}^{(\boldsymbol{b})}\boldsymbol{P}_i^{(\boldsymbol{b})} + \alpha_i^2 F_{\alpha_{i+1},i}^{(\boldsymbol{b})}\boldsymbol{A}_i^{(\boldsymbol{b})}$, which instantiates Eq. (2).

4. `UltraFold`: A Framework for Partition-Compatible Code-Based PCSs

We now construct a distributed PCS protocol for the partition-compatible code-based PCSs described in Section 3. The construction combines a recursive encoding schedule inspired by Bailey's FFT [5], an *interleaved* Merkle leaf layout, and the segmentwise oracle transitions of the underlying PCS. During encoding, we use a single round of all-to-all communication to exchange data among workers in a specific way such that all subsequent computations do not require further communication. Afterwards, workers generate subsequent oracle segments and local authentication paths without exchanging oracle entries with one another, while the coordinator aggregates short messages and completes the remaining terminal computation. We also incorporate *packing* to further optimize computation and proof size: when several instances have aligned oracle computations and queries, their corresponding leaves share one Merkle authentication path.

4.1. Setting and Partition Compatibility

Participants and input. Let $D = 2^d$, $M = 2^m$, and $N = D/M = 2^n$, with $n = d - m$. We consider workers $\{\mathcal{P}_{\boldsymbol{b}}\}_{\boldsymbol{b}\in B_m}$, a coordinator $\mathcal{C}$, and a verifier $\mathcal{V}$. Workers communicate directly during the encoding redistribution, while the verifier communicates with the coordinator. Worker $\mathcal{P}_{\boldsymbol{b}}$ initially holds

$$\boldsymbol{t}^{(\boldsymbol{b})} := \big(\boldsymbol{t}[\boldsymbol{a}\|\boldsymbol{b}]\big)_{\boldsymbol{a}\in B_n} = \big(f(\boldsymbol{a},\boldsymbol{b})\big)_{\boldsymbol{a}\in B_n}.$$

It retains this polynomial data, separately from its encoded-oracle segments, for the evaluation protocol. We write $\boldsymbol{\pi}_0 = E\boldsymbol{t}$ for the initial oracle and index subsequent oracles as $\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_R$.

Assigning segments. Recall that the partition $\{S_i^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_m}$ is the cyclic partition where $S_i^{(\boldsymbol{b})} := \{b + Mt : 0 \leq t < Q_i/M\}$. Worker $\mathcal{P}_{\boldsymbol{b}}$ holds $\boldsymbol{\pi}_i^{(\boldsymbol{b})} := \boldsymbol{\pi}_i|_{S_i^{(\boldsymbol{b})}}$ throughout the distributed phase. Partition compatibility guarantees that this segment can be computed from earlier segments with the same label and the public transcript, as in Eq. (2). All field elements in a Merkle leaf are assigned to the same worker.

4.2. Distributed Encoding and Merkle Commitment

In a naive distribution attempt, each worker $\mathcal{P}_{\boldsymbol{b}}$ encodes its own vector segment and sends $\mathsf{Enc}(\boldsymbol{t}^{(\boldsymbol{b})})$ to $\mathcal{C}$. Then the coordinator continues to compute the global codeword $\boldsymbol{\pi}_0 = \mathsf{Enc}(\boldsymbol{t})$ using the received intermediate results. Finally, the coordinator sends $\boldsymbol{\pi}_d$ back to workers to build local Merkle subtrees, and aggregates the subtree roots into a single Merkle root. However, this scheme would have each worker send to $\mathcal{C}$ and receive from $\mathcal{C}$ a total of $O(c\cdot N)$ elements, and would require $\mathcal{C}$ to perform the encoding, leading to both heavy communication and heavy computation for the coordinator as the coordinator here essentially has the asymptotic cost of performing a non-distributed encoding. Furthermore, the canonical Merkle leaf order makes the subsequent opening expensive: both folding and Merkle openings involve siblings stored on many workers, forcing additional cross-worker communication.

We remove the coordinator bottleneck by (1) replacing the centralized upload/download with a single all-to-all communication *only among workers*, (2) finishing all encoding computation *entirely locally* at each worker, and (3) building the Merkle tree in a specific *interleaved ordering* of the leaves so that the folding and Merkle openings in the distributed IOPP protocol remain local. Our protocol proceeds in four stages.

Stage A: local partial encoding. The encoding order is aligned with the input partition so that each segment $\boldsymbol{t}^{(\boldsymbol{b})} \in \mathbb{F}^{2^n}$ is the input to a depth-n recursive subproblem of Enc_d. Worker $\mathcal{P}_{\boldsymbol{b}}$ computes this encoding locally using the recursive encoding rule from Section 3.1 and get $\boldsymbol{p}^{(\boldsymbol{b})} := \mathsf{Enc}_n(\boldsymbol{t}^{(\boldsymbol{b})}) \in \mathbb{F}^{q_n} = \mathbb{F}^{cN}$. Specifically, for each local recursive encoding on a vector $\boldsymbol{u} \in \mathbb{F}^{2^h}$ with $1 \leq h \leq n$, the worker splits $\boldsymbol{u}$ into $\boldsymbol{u}_\ell$ and $\boldsymbol{u}_r$, and computes $\boldsymbol{\ell} := \mathsf{Enc}_{h-1}(\boldsymbol{u}_\ell)$ and $\boldsymbol{r} := \mathsf{Enc}_{h-1}(\boldsymbol{u}_r)$. It then combines their corresponding entries using the public matrices $K_{h,j}$:

$$\begin{pmatrix} \mathsf{Enc}_h(\boldsymbol{u})[j] \\ \mathsf{Enc}_h(\boldsymbol{u})[j+q_{h-1}] \end{pmatrix} = K_{h,j}\begin{pmatrix} \boldsymbol{\ell}[j] \\ \boldsymbol{r}[j] \end{pmatrix}.$$

The recursion terminates at Enc_0. All inputs to these computations belong to the same worker, so the n local combining layers require no communication. The resulting vectors $\{\boldsymbol{p}^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_m}$ are intermediate encodings and will be redistributed in Stage B.

Stage B: one all-to-all transpose. The remaining m encoding layers combine corresponding entries from the intermediate vectors $\{\boldsymbol{p}^{(\boldsymbol{b})}\}_{\boldsymbol{b}\in B_m}$. We redistribute these values so that the entries at each intermediate position are held by a single worker.

Let $q_n := cN$, with $c = 2^\gamma$. Here we require $M \mid q_n$, equivalently $n + \gamma \geq m$. Define $U_{\boldsymbol{w}} := \{w + Ms : 0 \leq s < q_n/M\} = S_0^{(\boldsymbol{w})} \cap [0, q_n)$, where w is the integer represented by $\boldsymbol{w} \in B_m$. Then each intermediate entry at position k is assigned to the worker whose label represents $k \bmod M$, and each set $U_{\boldsymbol{w}}$ contains exactly $q_n/M = cN/M$ positions.

For every destination worker $\mathcal{P}_{\boldsymbol{w}}$, worker $\mathcal{P}_{\boldsymbol{b}}$ forms the vector $(\boldsymbol{p}^{(\boldsymbol{b})}[k])_{k\in U_{\boldsymbol{w}}}$, with entries ordered by increasing k. It sends these packets in one all-to-all exchange, retaining the packet with $\boldsymbol{w} = \boldsymbol{b}$ locally. After the exchange, worker $\mathcal{P}_{\boldsymbol{w}}$ stores $A_{\boldsymbol{w}}[k, \boldsymbol{b}] := \boldsymbol{p}^{(\boldsymbol{b})}[k]$ for $k \in U_{\boldsymbol{w}}$ and $\boldsymbol{b} \in B_m$. Therefore, for each assigned index k, worker $\mathcal{P}_{\boldsymbol{w}}$ holds the entries at position k from all M intermediate vectors. These are precisely the inputs needed to perform the remaining recursive combinations for that index locally in Stage C. Each worker holds $M(q_n/M) = cN$ field elements after the exchange.

Stage C: local encoding completion. Each worker applies the remaining m encoding layers to its received values using the public combining matrices $K_{h,j}$ from Section 3.1. In layer $t \in [1, m]$, write the second index of $A_{\boldsymbol{w}}$ as $\boldsymbol{u}\|\beta\|\boldsymbol{v}$, with $\boldsymbol{u} \in B_{m-t}$, $\beta \in \{0, 1\}$, and $\boldsymbol{v} \in B_{t-1}$. For each assigned $k \in U_{\boldsymbol{w}}$, let v be the integer represented by $\boldsymbol{v}$, set $j = vq_n + k$, and update

$$\begin{pmatrix} A_{\boldsymbol{w}}[k, \boldsymbol{u}\|0\|\boldsymbol{v}] \\ A_{\boldsymbol{w}}[k, \boldsymbol{u}\|1\|\boldsymbol{v}] \end{pmatrix} \leftarrow K_{n+t,j} \begin{pmatrix} A_{\boldsymbol{w}}[k, \boldsymbol{u}\|0\|\boldsymbol{v}] \\ A_{\boldsymbol{w}}[k, \boldsymbol{u}\|1\|\boldsymbol{v}] \end{pmatrix}.$$

Both inputs are already local, so these operations require no communication.

The final values satisfy $A_{\boldsymbol{w}}[k, \boldsymbol{b}'] = \boldsymbol{\pi}_0[b'q_n + k]$ for $k \in U_{\boldsymbol{w}}$ and $\boldsymbol{b}' \in B_m$, where b' is the integer represented by $\boldsymbol{b}'$. Since $M \mid q_n$, every resulting output index satisfies $(b'q_n + k) \bmod M = k \bmod M = w$. Therefore, worker $\mathcal{P}_{\boldsymbol{w}}$ holds precisely $\boldsymbol{\pi}_0^{(\boldsymbol{w})} = \boldsymbol{\pi}_0|_{S_0^{(\boldsymbol{w})}}$.

Stage D: interleaved Merkle commitment. Each worker $\mathcal{P}_{\boldsymbol{b}}$ orders the entries in its segment $S_0^{(\boldsymbol{b})}$ by increasing logical index and builds a Merkle subtree over $(\boldsymbol{\pi}_0[b + Mt])_{t=0}^{Q_0/M-1}$ where b is the integer represented by $\boldsymbol{b}$. Then each worker sends its subtree root $\mathsf{rt}_0^{(\boldsymbol{b})}$ to $\mathcal{C}$ and retains its oracle segment and Merkle subtree. The coordinator constructs a top tree over the subtree roots in increasing order of b and publishes the global root rt_0.

Thus, the leaves of the global tree are ordered first by worker label and then by their position within the worker's subtree. The oracle index $j = b + Mt$ corresponds to leaf position $\sigma_0(j) = b(Q_0/M) + t$. The verifier queries oracle indices and uses σ_0 to determine the corresponding Merkle leaf positions. For each opening, the worker supplies the authentication path within its subtree, and $\mathcal{C}$ appends the path from that subtree root to rt_0.

Packing several instances. We also apply the construction to $L = 2^\ell$ polynomials of the same size by grouping their corresponding leaves into tuples, as in packed distributed FRI [25] and PipFRI [14]. For a single polynomial, choose $0 \le \ell \le n$, write $f(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{y})$ with variable counts $\ell, n-\ell, m$, and define $f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{y}) := f(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{y})$ for $\boldsymbol{u} \in B_\ell$. Each instance has size $D' := D/L$, and each worker holds an instance of size $N' := N/L$ from every instance. In particular, the total input held by a worker remains $LN' = N$.

Apply Stages A to C to every instance with D', N' in place of D, N, retaining the input-order alignment required by Stage A. Position j belongs to the worker whose label represents $j \bmod M$. In Stage B, combine all instances' messages for each destination into the same all-to-all communication. The encoding parameters should satisfy $M \mid cN'$, equivalently $M^2 \mid cD/L$.

If $\boldsymbol{\pi}_{\boldsymbol{u},i}$ is the oracle for instance $\boldsymbol{u}$ at level i, we pack the entries into one Merkle leaf as follows: $\widetilde{\pi}_i[j] := \big(\boldsymbol{\pi}_{\boldsymbol{u},i}[j]\big)_{\boldsymbol{u}\in B_\ell}$. The packed leaf at index j belongs to worker $\mathcal{P}_{\boldsymbol{b}}$, where $b = j \bmod M$, since this worker holds the entry at index j from every fragment oracle.

Each worker orders its packed leaves by increasing logical index and builds one Merkle subtree. The coordinator combines the subtree roots in increasing order of worker labels to obtain one global root. Initially, each fragment oracle has cD' entries, so the packed tree contains cD' leaves, each holding L field elements. As in Stage D, logical packed-leaf index $j = b + Mt$ corresponds to physical leaf position $\widetilde{\sigma}_0(j) = b(cD'/M) + t$ for $0 \le t < cD'/M$.

4.3. Distributed Opening

Given an evaluation point $\boldsymbol{z}$ and claimed value v, the coordinator and workers follow the message order, challenge distribution, and verification checks of the underlying PCS. The coordinator forwards public challenges and the required short transcript messages to the workers. We describe the common procedures for oracle generation, authentication, packing, and terminal computation below. The evaluation reductions and final consistency checks of individual PCSs are specified in Section 5.

Local oracle transitions. Before the terminal stage, worker $\mathcal{P}_{\boldsymbol{b}}$ computes each new oracle segment as $\boldsymbol{\pi}_i^{(\boldsymbol{b})} = \boldsymbol{a}_i^{(\boldsymbol{b})}(\tau_i) + \sum_{h=0}^{i-1} T_{i,h}^{(\boldsymbol{b})}(\tau_i)\boldsymbol{\pi}_h^{(\boldsymbol{b})}$. Every required oracle entry is already held by that worker. After computing the new segment, the worker builds its local Merkle subtree and sends its root to $\mathcal{C}$. The coordinator combines the subroots into rt_i, sends the prescribed commitment message to $\mathcal{V}$, and retains the top tree. Workers retain earlier oracle segments and subtrees for subsequent transitions and authentication queries. Thus, oracle generation and local Merkle-tree construction require no worker-to-worker communication during this phase.

Other prover messages. The remaining opening messages and computation vary for different instantiations, and will be specified and analyzed in Section 5.

Packed opening. Packing is used when the underlying PCS supports opening multiple polynomials with shared challenges and query positions. Workers apply the oracle transitions to each instance and pack the resulting entries at the same index into one leaf. Each worker builds one Merkle subtree over its packed leaves using the ordering in Stage D, and the coordinator combines the subtree roots into one global root. An opening of a packed leaf authenticates the entries of all L instances using only one Merkle path.

For the instances defined in Section 4.2, let $\boldsymbol{z} = (\boldsymbol{z}_u, \boldsymbol{z}_x, \boldsymbol{z}_y)$ and set $w_{\boldsymbol{u}} := \widetilde{\mathrm{eq}}(\boldsymbol{u}, \boldsymbol{z}_u)$. Then $f(\boldsymbol{z}) = \sum_{\boldsymbol{u}\in B_\ell} w_{\boldsymbol{u}} f_{\boldsymbol{u}}(\boldsymbol{z}_x, \boldsymbol{z}_y)$. The opening protocol establishes that the instance evaluations are consistent with the packed commitment and that their weighted sum equals the claimed

value v. It does so by verifying the instance evaluations individually.

Small-instance completion. At the specified terminal stage, the workers send their remaining states to $\mathcal{C}$. The coordinator aggregates this state and performs the remaining prover computations and commitments locally. The total transfer is bounded by the terminal-state bound specified by the instantiation.

Query handling. For each queried leaf of a distributed oracle, the coordinator $\mathcal{C}$ requests the leaf and its local Merkle authentication path from the worker that stores it. The coordinator appends the top-tree path from that worker's subtree root to the global root. The verifier checks the combined path at the physical leaf position determined by the public ordering for that oracle, as described in Stage D. For packed oracles, this ordering is defined over the packed leaves, and each opening reveals the complete tuple. The verifier performs the componentwise checks. For oracles committed by the coordinator, $\mathcal{C}$ supplies the complete Merkle openings itself.

4.4. Efficiency

With constant-cost combining matrices, the explicit encoder uses $O(cN \log D)$ field operations and one exchange of $O(cN)$ field elements per worker. The coordinator handles $O(M)$ subtree roots per distributed oracle and the terminal state of size $H(D, M)$, in addition to short-message aggregation and query responses. Opening arithmetic is determined by the instantiation's local transitions and auxiliary computations; partition compatibility alone does not imply a linear-time opening algorithm. For an oracle with J_i leaves, a distributed authentication path has $\log(J_i/M) + \log M = \log J_i$ hashes in the equal-size case. Consequently, distributing a fixed protocol adds no multiplicative worker-count factor to its proof size. Concrete opening and packing costs are given with the corresponding instantiations.

5. Instantiations

In this section, we instantiate `UltraFold` with BaseFold [30], FRI [6], and DeepFold [12]. For each construction, we explain how its evaluation protocol combines with the distributed encoding, oracle transitions, and Merkle authentication described in Section 4.

5.1. BaseFold

We instantiate `UltraFold` with BaseFold [30], combining its recursive encoder and oracle folds with a distributed sumcheck. The commitment and authentication procedures follow Section 4, and we detail how to combine the sumcheck messages during opening. We describe one construction that also incorporates the packing extension, with $L = 1$ recovering the unpacked protocol.

Parameters. Recall that $D = 2^d$, $M = 2^m$, and $N = D/M = 2^n$. For the packing parameter $L = 2^\ell$ with $0 \le \ell \le n$, define $D' := D/L = 2^{d'}$, $N' := N/L = 2^{n'}$, and $d' = d - \ell = n' + m$. We take $d' \ge 1$ and $c = 2^\gamma$, and require $M \mid cN'$. Each polynomial fragment $f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{y}) := f(\boldsymbol{u}, \boldsymbol{x}, \boldsymbol{y})$, indexed by $\boldsymbol{u} \in B_\ell$, has d' variables and D' Boolean evaluations. Worker $\mathcal{P}_{\boldsymbol{b}}$ initially holds $f_{\boldsymbol{u}}(\cdot, \boldsymbol{b})$ for every fragment, which is N field elements in total.

Protocol 1 (`UltraFold` commitment instantiated with BaseFold)**.**

Input: Worker $\mathcal{P}_{\boldsymbol{b}}$ holds the Boolean tables of $f_{\boldsymbol{u}}(\cdot, \boldsymbol{b})$ for all $\boldsymbol{u} \in B_\ell$.
Output: A packed global Merkle root $\widetilde{\mathsf{rt}}_0$.
Protocol:

1) Apply Stages A–C of Section 4.2 to every polynomial $f_{\boldsymbol{u}}$ ($\boldsymbol{u} \in B_\ell$). This produces $\boldsymbol{\pi}_{\boldsymbol{u},0} = \mathsf{Enc}_{d'}(\boldsymbol{t}_{\boldsymbol{u}})$, with worker $\mathcal{P}_{\boldsymbol{b}}$ holding the entries at indices congruent to b modulo M.
2) Form the packed leaves $\widetilde{\pi}_0[j] := \big(\boldsymbol{\pi}_{\boldsymbol{u},0}[j]\big)_{\boldsymbol{u} \in B_\ell}$ for $0 \le j < cD'$, and apply Stage D to obtain $\widetilde{\mathsf{rt}}_0$. Workers keep their encoded segments and Merkle subtrees, and $\mathcal{C}$ keeps the top tree.

Protocol 2 (`UltraFold` opening instantiated with BaseFold)**.**

Input: $(\widetilde{\mathsf{rt}}_0, \boldsymbol{z}, v)$ and the prover state from Protocol 1, including the original Boolean tables of $f_{\boldsymbol{u}}(\cdot, \boldsymbol{b})$ held by each worker $\mathcal{P}_{\boldsymbol{b}}$.
Notation: Write $\boldsymbol{z} = (\boldsymbol{z}_u, \boldsymbol{z}_x, \boldsymbol{z}_y)$. The encoder's restriction order is $(\boldsymbol{y}, \boldsymbol{x})$, and we use this same order for sumcheck, with $\boldsymbol{z}' := (\boldsymbol{z}_y, \boldsymbol{z}_x)$. At level $i \in [0, d']$, let $Q'_i := c2^{d'-i}$ and $\widetilde{\pi}_i[j] := (\boldsymbol{\pi}_{\boldsymbol{u},i}[j])_{\boldsymbol{u} \in B_\ell}$. Thus, $\widetilde{\pi}_i$ has Q'_i packed leaves, each containing L field elements.
Protocol:

1) **Preparation.** Set $w_{\boldsymbol{u}} := \tilde{eq}(\boldsymbol{u}, \boldsymbol{z}_u)$. Each worker $\mathcal{P}_{\boldsymbol{b}}$ forms and keeps the evaluation table of
$$g^{(\boldsymbol{b})}(\boldsymbol{x}) := \sum_{\boldsymbol{u} \in B_\ell} w_{\boldsymbol{u}} f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{b}).$$
These are the slices of $g(\boldsymbol{x}, \boldsymbol{y}) := \sum_{\boldsymbol{u} \in B_\ell} w_{\boldsymbol{u}} f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{y})$, and the opening claim is $g(\boldsymbol{z}_x, \boldsymbol{z}_y) = v$.
Each worker evaluates $a_{\boldsymbol{b}} := g^{(\boldsymbol{b})}(\boldsymbol{z}_x)$ and sends $a_{\boldsymbol{b}}$ to $\mathcal{C}$. The coordinator interprets $(a_{\boldsymbol{b}})_{\boldsymbol{b} \in B_m}$ as the evaluation table of
$$a(\boldsymbol{y}) := \sum_{\boldsymbol{b} \in B_m} a_{\boldsymbol{b}} \tilde{eq}(\boldsymbol{b}, \boldsymbol{y}) = g(\boldsymbol{z}_x, \boldsymbol{y}).$$
Set $v_0 := v$.
2) **Interleaved sumcheck and folding.**
For $i = 0, \ldots, d' - 1$, perform:
a) Compute the sumcheck message.
If $i < m$, $\mathcal{C}$ computes the next $\boldsymbol{y}$-round polynomial from its retained table of a as

$h_{i+1}(X) := \sum_{\boldsymbol{b}\in B_{m-i-1}} a(r_1,\dots,r_i,X,\boldsymbol{b}) \cdot \tilde{eq}((r_1,\dots,r_i,X,\boldsymbol{b}),\boldsymbol{z}_y)$.
If $i \geq m$, set $t := i - m$ and $\boldsymbol{r}_y := (r_1,\dots,r_m)$. Each worker $\mathcal{P}_{\boldsymbol{b}}$ computes the next local $\boldsymbol{x}$-round polynomial as $h_{t+1}^{(\boldsymbol{b})}(X) := \sum_{\boldsymbol{a}\in B_{n'-t-1}} g^{(\boldsymbol{b})}(r_{m+1},\dots,r_i,X,\boldsymbol{a}) \cdot \tilde{eq}((r_{m+1},\dots,r_i,X,\boldsymbol{a}),\boldsymbol{z}_x)$ and sends its coefficients to $\mathcal{C}$. The coordinator aggregates $h_{i+1}(X) := \tilde{eq}(\boldsymbol{r}_y,\boldsymbol{z}_y)\sum_{\boldsymbol{b}\in B_m} \tilde{eq}(\boldsymbol{b},\boldsymbol{r}_y) h_{t+1}^{(\boldsymbol{b})}(X)$.
In either case, $\mathcal{C}$ sends the coefficients of h_{i+1} to $\mathcal{V}$.
b) Receive the challenge and update polynomial tables. The verifier checks $\deg h_{i+1} \leq 2$ and $h_{i+1}(0) + h_{i+1}(1) = v_i$. It then samples $r_{i+1} \xleftarrow{\$} \mathbb{F}$ and sets $v_{i+1} := h_{i+1}(r_{i+1})$. The coordinator forwards r_{i+1} to the workers whenever $i < n'$ or $i \geq m$, so they can perform the required oracle folds or polynomial-table updates.
If $i < m$, $\mathcal{C}$ updates its current evaluation table of a in the next $\boldsymbol{y}$-coordinate challenge step. Otherwise, each worker updates its current evaluation tables in the next $\boldsymbol{x}$-coordinate challenge step. Each update replaces each corresponding pair of table entries (t_0,t_1) with $(1-r_{i+1})t_0 + r_{i+1}t_1$.
c) Fold and commit the encoded oracles. For every fragment $\boldsymbol{u} \in B_\ell$ and $j \in [0, Q'_{i+1})$, compute

$$\begin{aligned}\boldsymbol{\pi}_{\boldsymbol{u},i+1}[j] := \frac{\theta' - r_{i+1}}{\theta' - \theta}\boldsymbol{\pi}_{\boldsymbol{u},i}[j] \\ + \frac{r_{i+1} - \theta}{\theta' - \theta}\boldsymbol{\pi}_{\boldsymbol{u},i}[j + Q'_{i+1}],\end{aligned} \tag{4}$$

where θ, θ' are the BaseFold encoding parameters of this pair.
Commit to the resulting packed oracle with root $\widetilde{\mathsf{rt}}_{i+1}$ before the next challenge. For $i < n'$, workers compute their segments locally, since $M \mid Q'_{i+1}$, and send their Merkle subtree roots to $\mathcal{C}$, which constructs the top tree. For $i \geq n'$, $\mathcal{C}$ computes the folds and Merkle trees locally.

3) **Terminal check.** Reveal the final-layer oracle and verify its root against $\widetilde{\mathsf{rt}}_{d'}$. For each $\boldsymbol{u}$, recover $s_{\boldsymbol{u}} \in \mathbb{F}$ from its bottom column and check $\boldsymbol{\pi}_{\boldsymbol{u},d'} = \mathsf{Enc}_0(s_{\boldsymbol{u}})$.
Let $\boldsymbol{r}_y := (r_1,\dots,r_m)$, $\boldsymbol{r}_x := (r_{m+1},\dots,r_{d'})$, and $\boldsymbol{r} := (\boldsymbol{r}_y,\boldsymbol{r}_x)$. Check $v_{d'} = \tilde{eq}(\boldsymbol{r},\boldsymbol{z}')\sum_{\boldsymbol{u}\in B_\ell} w_{\boldsymbol{u}} s_{\boldsymbol{u}}$.
4) **Authenticated fold checks.** If $d' > 0$, after all roots and terminal values are fixed, repeat the following q times. The verifier samples $\mu \xleftarrow{\$} [0, Q'_1)$ and, for each $i = 0,\dots,d'-1$, sets $j_i := \mu \bmod Q'_{i+1}$. Open the packed leaves at j_i and $j_i + Q'_{i+1}$ in level i, and at j_i in level $i+1$. Use the query procedure of Section 4.3, verify the authentication paths using the public leaf-position maps, and check Eq. (4) for every packed component. Previously disclosed bottom entries need not be opened again. Accept only if all checks pass.

Completeness and knowledge soundness. We use the code-distance, field-size, and query-count conditions required by BaseFold's PCS knowledge-soundness analysis [30], together with the same oracle and Merkle tree assumptions. For a polynomial number of packed polynomials, `UltraFold` inherits BaseFold's knowledge-soundness guarantee.

Lemma 1 (Completeness of `UltraFold`)**.** *Assume that the workers hold a correct segmentation of the multilinear polynomial $f(\boldsymbol{u},\boldsymbol{x},\boldsymbol{y})$ and that $\widetilde{\mathsf{rt}}_0$ is produced by an honest execution of Protocol 1. For every evaluation point $\boldsymbol{z}$ and claimed value $v = f(\boldsymbol{z})$, an honest execution of Protocol 2 is accepted with probability* 1.

Lemma 2 (Knowledge soundness of `UltraFold`)**.** *Assume the BaseFold parameters and commitment compilation satisfy the conditions above. For any PPT adversary $P^\star$ controlling $\mathcal{C}$ and all workers that makes the verifier accept an opening instance $(\widetilde{\mathsf{rt}}_0,\boldsymbol{z},v)$ with non-negligible probability, there exists an efficient extractor $\mathcal{E}_{\mathrm{pack}}^{P^\star}$ that outputs a d-variate multilinear polynomial $\widehat{f}(\boldsymbol{u},\boldsymbol{x},\boldsymbol{y})$ satisfying the following properties, except with negligible probability:*

1) ***Consistency with the packed commitment.*** *For every $\boldsymbol{u} \in B_\ell$, define $\widehat{f}_{\boldsymbol{u}}(\boldsymbol{x},\boldsymbol{y}) := \widehat{f}(\boldsymbol{u},\boldsymbol{x},\boldsymbol{y})$, and let $\widehat{\boldsymbol{t}}_{\boldsymbol{u}}$ be its Boolean evaluation table in the encoder's prescribed order. The $\boldsymbol{u}$-th projection $\boldsymbol{\pi}_{\boldsymbol{u},0}$ of the initial packed oracle is close to $\mathsf{Enc}_{d'}(\widehat{\boldsymbol{t}}_{\boldsymbol{u}})$ under the same proximity notion and parameters as in the BaseFold PCS analysis.*
2) ***Correct opening.*** *$\widehat{f}(\boldsymbol{z}_u,\boldsymbol{z}_x,\boldsymbol{z}_y) = v$.*

The extractor uses the same extraction access as BaseFold, with running time polynomial in the instance size and the inverse acceptance probability.

The proofs are given in Appendix B. They combine a simulation of the distributed execution, consistency of the packed BaseFold, and extraction of the polynomial instances from their disclosed terminal evaluations.

Prover computation. Each worker performs $O(cN\log(MN'))$ field operations and $O(cN')$ hashes during commitment; it performs $O(cN)$ field operations and $O(cN')$ hashes during opening.

The coordinator performs $O(M)$ hashes during commitment; it performs $O(cLM)$ field operations and $O(cM + M\log N')$ hashes during opening.

Communication. During commitment, each worker sends and receives $O(cN)$ field elements, and sends $O(1)$ hash digests; the coordinator receives $O(M)$ hash digests.

During opening, the amortized communication per worker is $O(\lambda/M(\log N'\log(cN') + L\log N') + cL + logN')$; the communication of the coordinator is $O(\lambda(\log N'\log(cN') + L\log N') + M(cL + \log N'))$.

Proof size and verifier time. The proof size is $O(\lambda L\log(MN'))$ field elements and $O(\lambda\log(MN')\log(cMN'))$ hash digests. The verifier

performs is $O(\lambda L \log(MN'))$ field operations and $O(\lambda \log(MN') \log(cMN'))$ hashes.

Effect of packing. Packing reduces the initial number of Merkle leaves from cD to cD/L and the folding depth from d to $d' = d - \ell$. It reduces the number of hashes while preserving the $O(cN)$ worker opening time. The tradeoff is that the proof now contains $O(L \log(MN'))$ more field operations. In practice, the choice of L therefore depends on the concrete field and hash costs, and the desired proof size. We detail the effects of L on proof size in Fig. 2.

5.2. FRI

The FRI-based multilinear PCS considered here is a special case of BaseFold's general construction [30], instantiated with Reed–Solomon codes. Using the multiplicative-domain encoder from Section 3.3, we set $\theta_{h,j} = \zeta_{h,j}$ and $\theta'_{h,j} = -\zeta_{h,j}$, which is then exactly the standard FRI folding [6]. Incorporating these folds with the same sumcheck reduction and packing gives the distributed PCS protocol.

5.3. DeepFold

DeepFold [12] uses Reed-Solomon encoding and pairwise oracle folding, while reducing evaluation claims through linear polynomial messages. We first describe the instantiation without packing, setting $L = 1$, $d' = d$, and $n' = n$. The encoding parameters and domains are those of the FRI instantiation, and we retain the requirement $M \mid cN$. To follow the encoder's variable order, write $\bar{f}(\boldsymbol{y}, \boldsymbol{x}) := f(\boldsymbol{x}, \boldsymbol{y})$ and $\boldsymbol{z}' := (\boldsymbol{z}_y, \boldsymbol{z}_x)$. Let $\boldsymbol{\Gamma}_h$ be the evaluation map defined in Section 3.3.

Distributed evaluation messages. Workers retain their polynomial evaluation tables throughout opening. For a public point $\boldsymbol{a} = (\boldsymbol{a}_y, \boldsymbol{a}_x) \in \mathbb{F}^d$, they compute

$$\bar{f}(\boldsymbol{a}) = \sum_{\boldsymbol{b} \in B_m} \tilde{eq}(\boldsymbol{b}, \boldsymbol{a}_y) f(\boldsymbol{a}_x, \boldsymbol{b}). \tag{5}$$

Each worker evaluates its local polynomial, sends its weighted contribution, and $\mathcal{C}$ sums the contributions. Similarly, a linear polynomial

$$p_{i,\boldsymbol{w}}(X) := \bar{f}(r_1, \ldots, r_i, X, \boldsymbol{w}) \tag{6}$$

is derived by applying (5) at $X = 0$ and $X = 1$ and interpolating.

Protocol 3 (`UltraFold` commitment instantiated with DeepFold)**.**
Input: Worker $\mathcal{P}_{\boldsymbol{b}}$ holds the Boolean table of $f(\cdot, \boldsymbol{b})$.
Output: A commitment $C = (\mathsf{rt}_0, \eta, \nu)$.
Protocol:
1) **Encode and commit.** Apply Stages A–D of Section 4.2 using the Reed–Solomon encoder. This produces the oracle $\boldsymbol{\pi}_0[j] = \bar{f}(\boldsymbol{\Gamma}_d(\zeta_{d,j}))$ and its global Merkle root rt_0, which $\mathcal{C}$ sends to $\mathcal{V}$. Workers retain their evaluation tables, encoding segments, and Merkle subtrees.
2) **Commitment evaluation.** After receiving rt_0, $\mathcal{V}$ samples $\eta \xleftarrow{\$} \mathbb{F}$, and $\mathcal{C}$ forwards it to the workers. Using (5), the workers and $\mathcal{C}$ compute $\nu := \bar{f}\big(\boldsymbol{\Gamma}_d(\eta)\big)$. The coordinator sends ν to $\mathcal{V}$. The commitment is $C = (\mathsf{rt}_0, \eta, \nu)$.

Protocol 4 (`UltraFold` opening instantiated with DeepFold)**.**
Input: $(C, \boldsymbol{z}, v)$ and the prover state from Protocol 3.
Notation: At level i, let $Q_i := c2^{d-i}$ and $f_i(\boldsymbol{X}) := \bar{f}(r_1, \ldots, r_i, \boldsymbol{X})$. The list $\mathcal{C}_i$ contains evaluation claims $(\boldsymbol{a}, t)$ intended to assert $f_i(\boldsymbol{a}) = t$.
Protocol:
1) **Preparation.** Initialize $\mathcal{C}_0 := \big((\boldsymbol{z}', v), (\boldsymbol{\Gamma}_d(\eta), \nu)\big)$.
2) **Evaluation reduction and oracle folding.** For $i = 0, \ldots, d-1$, set $h := d - i$ and perform:
a) Receive the DEEP challenge. After rt_i is fixed, $\mathcal{V}$ samples $\alpha_{i+1} \xleftarrow{\$} \mathbb{F}$. The coordinator forwards it to the workers. Define the set of suffix points

$$\mathcal{W}_i := \big\{(a_2, \ldots, a_h) : (\boldsymbol{a}, t) \in \mathcal{C}_i\big\} \cup \big\{\boldsymbol{\Gamma}_{h-1}(\alpha_{i+1}^2)\big\}.$$

b) Send linear messages. For each distinct $\boldsymbol{w} \in \mathcal{W}_i$, the workers and $\mathcal{C}$ compute $p_{i,\boldsymbol{w}}(X) := \bar{f}(r_1, \ldots, r_i, X, \boldsymbol{w})$ using the distributed evaluation procedure in Eq. (6). The coordinator sends its coefficients to $\mathcal{V}$. The verifier checks $\deg p_{i,\boldsymbol{w}} \le 1$ and, for every $(\boldsymbol{a}, t) \in \mathcal{C}_i$, checks $p_{i,(a_2,\ldots,a_h)}(a_1) = t$.
c) Receive the folding challenge. After the linear messages are fixed, $\mathcal{V}$ samples $r_{i+1} \xleftarrow{\$} \mathbb{F}$, and $\mathcal{C}$ forwards it to the workers. Replace the claim list by

$$\mathcal{C}_{i+1} := \big((\boldsymbol{w}, p_{i,\boldsymbol{w}}(r_{i+1}))\big)_{\boldsymbol{w} \in \mathcal{W}_i}.$$

d) Fold and commit. Let $x_{i,j} := \zeta_{d-i,j}$. Compute

$$\boldsymbol{\pi}_{i+1}[j] := \frac{x_{i,j} + r_{i+1}}{2x_{i,j}} \boldsymbol{\pi}_i[j] + \frac{x_{i,j} - r_{i+1}}{2x_{i,j}} \boldsymbol{\pi}_i[j + Q_{i+1}]$$

for $0 \le j < Q_{i+1}$. For $i < n$, workers compute their segments locally; for $i \ge n$, $\mathcal{C}$ computes the folded oracle. If $i+1 < d$, authenticate the resulting oracle using Section 4.3 and send rt_{i+1} before the next DEEP challenge.
When $n < d$, immediately after n folds, workers transfer the residual oracle $\boldsymbol{\pi}_n$ to $\mathcal{C}$, totaling cM field elements. When $n = 0$, this transfer occurs before the first round. Workers keep their original polynomial tables and update the evaluation table for the folded polynomial.

3) **Terminal check.** The coordinator obtains and sends the final scalar $s = \bar{f}(r_1, \ldots, r_d)$. The final oracle is interpreted as the constant vector $\boldsymbol{\pi}_d = \mathsf{Enc}_0(s) = (s, \ldots, s)$, without an additional Merkle commitment. The verifier checks $t = s$ for every $(\cdot, t) \in \mathcal{C}_d$.
4) **Authenticated fold checks.** After all roots and the terminal value are fixed, repeat the following q times. The verifier samples $\mu \xleftarrow{\$} [0, Q_1)$ and sets $j_i := \mu \bmod Q_{i+1}$ for each $i = 0, \ldots, d-1$. Open the entries at j_i and $j_i + Q_{i+1}$ in level i, and the entry at j_i in level $i+1$ when $i+1 < d$. Use the query procedure of Section 4.3, verify all authentication paths, and check the folding equation above. At the final level, use s as the folded value. Accept only if all checks pass.

Packing. For L polynomial instances, use the parameters d', n' and the packed Merkle leaves from the BaseFold instantiation (Section 5.1). Run the protocols with shared DEEP challenges, folding challenges, and query positions. The commitment retains a separate value $\nu_{\boldsymbol{u}} := \bar{f}_{\boldsymbol{u}}\big(\boldsymbol{\Gamma}_{d'}(\eta)\big)$ with $\bar{f}_{\boldsymbol{u}}(\boldsymbol{y}, \boldsymbol{x}) := f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{y})$ for every polynomial instance $f_{\boldsymbol{u}}$. Before the opening challenges, the prover supplies the instance evaluations $v_{\boldsymbol{u}}$, and the verifier checks $v = \sum_{\boldsymbol{u} \in B_\ell} \tilde{eq}(\boldsymbol{u}, \boldsymbol{z}_u) v_{\boldsymbol{u}}$. Each instance is opened at $\boldsymbol{z}'$ with claimed value $v_{\boldsymbol{u}}$. Corresponding query responses share one packed Merkle path, similar to the BaseFold instantiation.

Security. The construction preserves DeepFold's algebraic checks and challenges, with the public Merkle leaf permutation and internal worker aggregation supplied by the framework. We therefore preserve the assumptions of the security analysis of DeepFold [12, Section 4.1 and Appendices B–C]. For the packed extension, the soundness guarantees are combined across the L polynomial instances with the corresponding union-bound soundness loss, similar to the proof of Lemma 4 in Appendix B.

6. Implementation and Evaluation

We implement `UltraFold` instantiated with BaseFold using Rust and Arkworks 0.5 [3]. We run our benchmarks on a cluster consisting of nodes with multiple Intel Xeon Platinum 8562Y+ CPUs. We assign each distributed worker a single core, and use a maximum of 256 workers. Throughout this section, `UltraFold` denotes the instantiation with BaseFold in Section 5.1. Our code is available here.

6.1. Parameters

We use the Goldilocks field $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$, as in other related works [14], [28], [29]. The number of sub-polynomials L is set to 64. We use BaseFold with blow-up factor $c = 8$, i.e., code rate $\rho := 1/c = 1/8$. Our target security is $\lambda = 100$. We repeat the BaseFold IOPP query phase for q independent repetitions. In our experiments, we choose $q = \lceil \lambda / \log_2 c \rceil = 34$ as the number of query repetitions. We detail the choice of q in Appendix C.

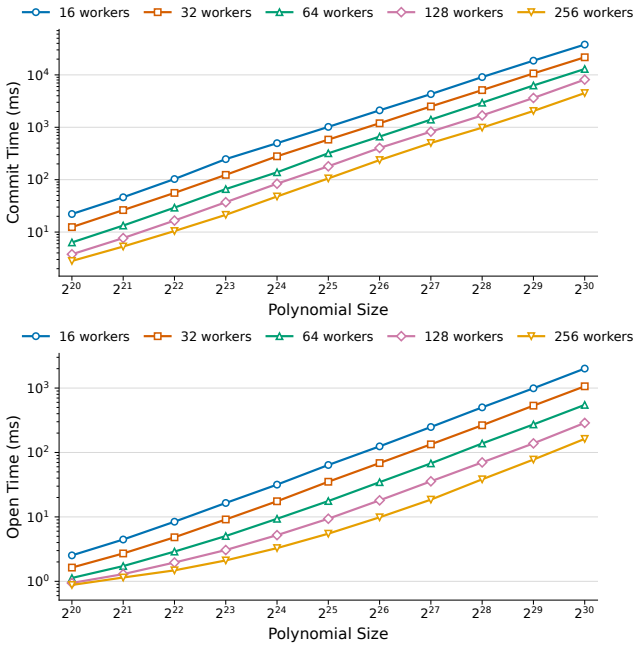

Figure 1. Commit time (top) and open time (bottom) for `UltraFold` across worker counts and polynomial sizes, including communication costs.

6.2. Performance Evaluation

Prover time. We benchmark the end-to-end prover computation time of `UltraFold` for committing to and opening polynomials, including communication latency. The runtime is the average of 10 runs, with 1 warmup. As shown in Fig. 1, `UltraFold` scales well with the number of workers. In particular, with 256 workers, we can commit to and open size-2^{30} polynomials in 4.66 seconds total (4.50 seconds for commit and 0.16 seconds for open).

Proof size and verifier time. We measure the proof size and verifier runtime for a single polynomial evaluation. The commitment is always a single Merkle root, and thus has a fixed size of 32 bytes. When the packing parameter L is fixed, both the proof size and verifier time scale only with the polynomial size and are independent of the number of workers. We use 8 threads for the verifier. Table 2 reports results across polynomial sizes. Even at size 2^{30}, `UltraFold` remains succinct, with 281 KB proofs and millisecond-scale verification.

Fixing the polynomial size to 2^{22}, we study how the proof size varies with the packing parameter L, ranging from 1 to 2^{10}. As shown in Fig. 2, proof size is minimized at moderate packing (around $L \in \{2^5, 2^6\}$) and increases for both very small and very large L. Small L provides limited Merkle-path savings, while overly large L inflates the field element number per opening.

Polynomial Size	Proof Size	Verifier Time
2^{20}	104 KB	0.435 ms
2^{21}	117 KB	0.520 ms
2^{22}	128 KB	0.628 ms
2^{23}	145 KB	0.773 ms
2^{24}	163 KB	0.697 ms
2^{25}	181 KB	0.859 ms
2^{26}	197 KB	0.700 ms
2^{27}	216 KB	0.863 ms
2^{28}	236 KB	0.882 ms
2^{29}	258 KB	0.856 ms
2^{30}	281 KB	0.956 ms

TABLE 2. PROOF SIZE AND EIGHT-THREAD VERIFIER TIME FOR ULTRAFOLD ACROSS POLYNOMIAL SIZES, WITH $L = 2^6$.

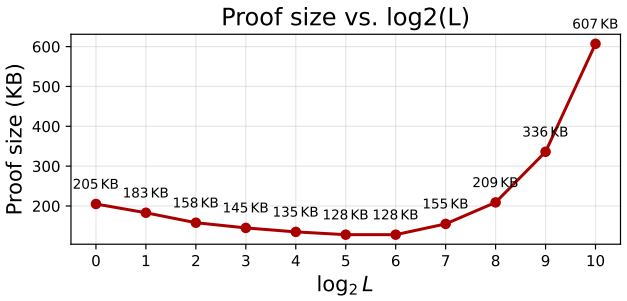

Figure 2. Proof size versus the packing parameter L (x-axis shows $\log_2 L$) for a polynomial of size 2^{22} with 16 workers.

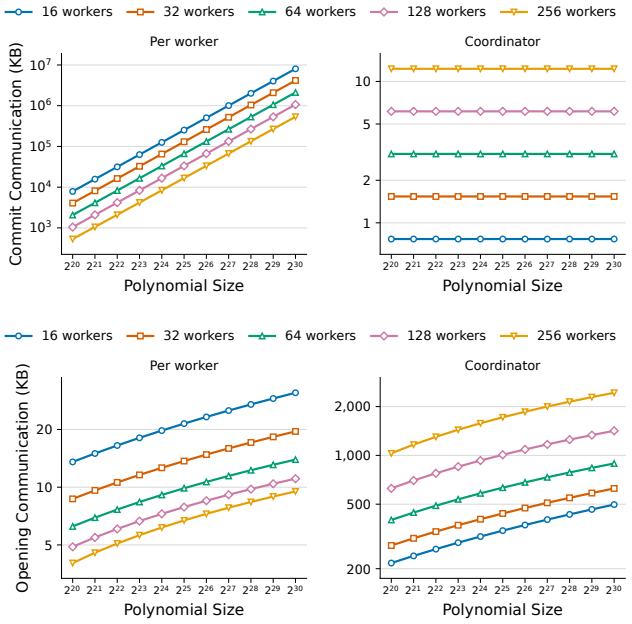

Figure 3. Commit communication (top) and open communication (bottom) for UltraFold for each worker and the coordinator across worker counts and polynomial sizes.

Communication. We measure the communication costs of UltraFold during commit and open separately. As shown in Fig. 3, during commitment, communication increases when there are fewer workers or the size of the polynomial increases; during commitment, the communication of the coordinator is light. During opening, the communication of each worker is extremely light, below 50 KB for 16 workers with a polynomial of size 2^{30}, because UltraFold makes most computation local within each worker during opening.

6.3. Comparison with Prior Works

We compare UltraFold's performance with deVirgo [25], DePipFRI [14], and Veloz-RS [29]. We did not benchmark FRIttata [27] and HyperFond [28], which have worse prover time than Veloz-RS (see Fig. 7 of [29]). To make a fair comparison, we use code rate $\rho = \frac{1}{8}$ consistent across these schemes and ran each scheme using the same hardware, with communication time counted. We also choose $M = 16$ workers to make sure the proof sizes of DePipFRI and Veloz-RS are optimal.

We compare commit time, open time, proof size, total communication during commit, and total communication during opening, and detail the results in Fig. 4. UltraFold has the fastest commit time and open time across all polynomial sizes. Specifically, at $D = 2^{30}$ with 16 workers, UltraFold commits to a polynomial in 37.91 s and opens it in 2.01 s, while Veloz-RS takes 50.65 s and 9.78 s for commit and open, respectively. UltraFold also has the smallest proof sizes. At $D = 2^{30}$, UltraFold produces a 281 KB proof, compared with 487.3 KB for DePipFRI and 492.5 KB for Veloz-RS.

Regarding total communication, UltraFold has almost the same total communication as Veloz-RS and DePipFRI during commit, which is approximately 64,425 MB when the polynomial size is 2^{30}. During opening, UltraFold has substantially less total communication than the other schemes. At $D = 2^{30}$ with 16 workers, the total opening communication is 0.50 MB for UltraFold, 4,027 MB for DePipFRI, and 5,131 MB for Veloz-RS.

In summary, with 16 workers, UltraFold achieves the lowest commit and opening times and the smallest proofs at every polynomial size. Its commit communication is nearly identical to that of DePipFRI and Veloz-RS and about one quarter of DeVirgo's over their shared range. Its opening communication is the lowest throughout the comparison and remains below 0.5 MB even at $D = 2^{30}$. This comparison demonstrates that even with fewer distributed workers, UltraFold has better practical performance than the other distributed code-based PCS schemes.

7. Related Works

DeVirgo. DeVirgo [25] is the first efficient transparent distributed ZKP in this line of work. At a high level, DeVirgo parallelizes proof generation by distributing the GKR-based polynomial IOP across multiple workers and by sharding the FRI polynomial commitment scheme at the input layer. With n machines, this reduces the prover's computational cost per machine by a factor of $1/n$. However, DeVirgo

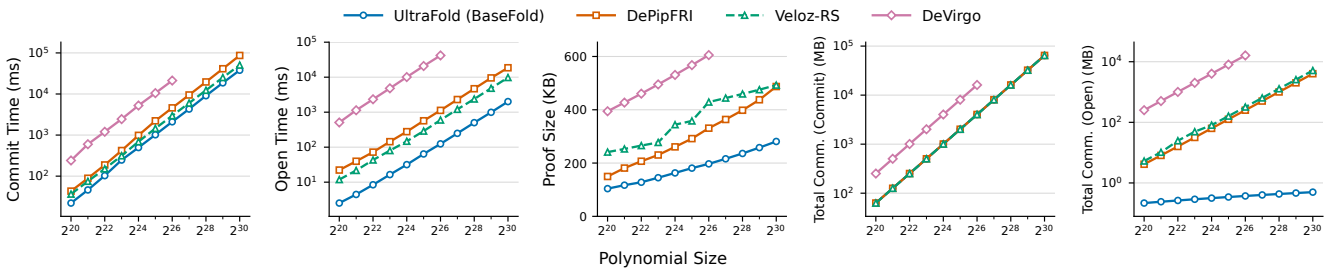

Figure 4. Comparison between `UltraFold` and DePipFRI, Veloz-RS, and DeVirgo, fixing 16 workers with varying polynomial sizes.

fundamentally relies on a data-parallel circuit structure: its distribution techniques do not extend to general circuits, and no distributed SNARK construction for general circuits is currently known within this framework.

FRIttata. FRIttata [27] develops a distributed FRI-based PCS with the goal of reducing communication costs. Its core technical technique is the *Fold-and-Batch* pattern (see also the design note [19]), in which each worker first commits locally to intermediate oracle(s), after which a coordinator aggregates these oracles to perform openings. While this design substantially reduces coordinator-side data transfer, the resulting proof size grows linearly with the number of workers. Consequently, when the number of workers is large (e.g., hundreds), the proof size becomes impractical.

PipFRI. Li et al. [14] introduce a "PCS-improved-from-PCS" transformation (PIP) that aims to improve the prover efficiency of the FRI PCS. Their key technique is to partition a large multilinear polynomial into independently handled polynomial fragments, where the encoding of the polynomial fragments can be committed using a single Merkle tree, saving prover time to build Merkle oracles. They achieve distributed proving by assigning each polynomial fragment to a separate worker. When the number of polynomial fragments is moderate ($O(\log N)$, or 64 or 128), they achieve better prover time compared with DeVirgo while maintaining the same proof size. However, distributed PipFRI suffers from two limitations. Firstly, their protocol requires the workers to exchange codes when building the sub-oracles during opening, which incurs substantial extra communication overhead during opening. Secondly, the distribution can not be at an arbitrary scale, since the distribution has to rely on the number of polynomial fragments, which cannot be set too high, or otherwise proof size and verifier time will degrade. `UltraFold`, in contrast, solves these problems.

Veloz. Veloz [29] distributes the Ligerito PCS, with instantiations using Reed-Solomon codes. It arranges the polynomial into a matrix and assigns rows to workers for local encoding. However, during opening, workers must repeatedly exchange intermediate values to aggregate and redistribute the reduced polynomial between iterations. Moreover, its optimized proof size is achieved under $M = O(\log D)$ workers. Similar to DePipFRI, supporting more workers requires increasing the number of rows, which also increases the proof size and verifier time. In contrast, `UltraFold` does not have these drawbacks.

HyperFond. HyperFond [28] is the closest work to ours in terms of cryptographic substrate: it distributes BaseFold's PCS by evenly dividing the coefficient vector so that each worker computes a partial BaseFold codeword locally. The commitment is therefore represented as multiple Merkle roots (one per worker). This design avoids global encoding communication, but it introduces two scaling costs: (i) the coordinator becomes a hub in the IOPP commit phase because workers send full folded vectors each round, and (ii) the proof size scales with the number of workers.

Distributed Brakedown. The recent distributed Brakedown work [16] targets the Brakedown code-based argument rather than BaseFold (or FRI), but it also tackles building and opening Merkle commitments distributedly. At a high level, their workers build Merkle subtrees over the polynomial slices they store. However, during opening, answering a single verifier query requires collecting Merkle paths from every worker. Consequently, while Merkle-tree construction can be distributed, the proof size (and the verifier time) scales linearly with the number of workers.

Distributed SNARKs. A line of work studies distributed proving for pairing-based SNARKs or SNARKs with trusted setup, which are not transparent and do not offer post-quantum security. We briefly mention the most relevant ones below. DIZK [24] and Hekaton [22] distribute Groth16 [11], which requires a circuit-specific trusted setup. Pianist [17] distributes KZG-based PLONK [9], Cirrus [23] and HyperPianist [13] distribute HyperPlonk [7] with pairing-based PCS, and Soloist [15] distributes Marlin [8] with bi-variate KZG, so all of them require a trusted setup and do not provide post-quantum security. Their techniques of distributing polynomial interactive oracle proofs are orthogonal to the transparent, post-quantum secure distributed PCS in this work, and can be combined to produce distributed SNARKs.

8. Conclusions and Future Directions

We have introduced `UltraFold`, a framework for distributing partition-compatible code-based PCSs while preserving their asymptotic proof size. We instantiate the framework with BaseFold, FRI, and DeepFold. Our BaseFold

implementation demonstrates practical prover performance and scalability with up to 256 single-core workers.

One future direction is robustness. In this work, the focus is on efficiency and scalability, and a natural next step is to add lightweight mechanisms for fault tolerance.

Acknowledgement

The authors would like to thank Ben Fisch for valuable discussions and feedback in the early stages of the project. They also thank Hadas Zeilberger for insightful guidance on distributing BaseFold and for clarifying key properties of the BaseFold protocol.

References

[1] Ceno: Non-uniform, segment and parallel risc-v zero-knowledge virtual machine. https://github.com/scroll-tech/ceno. Accessed: 2026-02-05.

[2] Deepprove: Zero-knowledge machine learning (zkml) inference. https://github.com/Lagrange-Labs/deep-prove. Accessed: 2026-02-05.

[3] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[4] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. Whir: Reed-solomon proximity testing with super-fast verification. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 214–243. Springer, 2025.

[5] David H Bailey. Ffts in external or hierarchical memory. *The journal of Supercomputing*, 4(1):23–35, 1990.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*, pages 14–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[7] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023.

[8] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 738–768. Springer, 2020.

[9] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[10] Alexander Golovnev, Jonathan Lee, Srinath TV Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and post-quantum snarks for r1cs. *IACR Cryptol. ePrint Arch.*, 2021:1043, 2021.

[11] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[12] Yanpei Guo, Xuanming Liu, Kexi Huang, Wenjie Qu, Tianyang Tao, and Jiaheng Zhang. Deepfold: Efficient multilinear polynomial commitment from reed-solomon code and its application to zero-knowledge proofs. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3497–3516, 2025.

[13] Chongrong Li, Yun Li, Pengfei Zhu, Wenjie Qu, and Jiaheng Zhang. Hyperpianist: Pianist with linear-time prover via fully distributed hyperplonk. *Cryptology ePrint Archive*, 2024.

[14] Weihan Li, Zongyang Zhang, Sherman SM Chow, Yanpei Guo, Boyuan Gao, Xuyang Song, Yi Deng, and Jianwei Liu. Shred-to-shine metamorphosis in polynomial commitment evolution. *Cryptology ePrint Archive*, 2025.

[15] Weihan Li, Zongyang Zhang, Yun Li, Pengfei Zhu, Cheng Hong, and Jianwei Liu. Soloist: Distributed snarks for rank-one constraint system. *Cryptology ePrint Archive*, 2025.

[16] Zesheng Li, Xinxuan Zhang, and Yi Deng. Transparent and post-quantum distributed snark with linear prover time. *Cryptology ePrint Archive*, 2025.

[17] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *Cryptology ePrint Archive*, 2023.

[18] Ralph C Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

[19] =nil;Research. On distributed fri-based proof generation. https://hackmd.io/@nil-research/rJ_NVyiRA. Accessed: 2025-12-09.

[20] Andrija Novakovic and Guillermo Angeris. Ligerito: A small and concretely fast polynomial commitment scheme. *Cryptology ePrint Archive*, 2025.

[21] Roy Rinberg, Alex Allain, Alessandro Chiesa, Miranda Christ, Rahul Dalal, Giacomo Fenzi, Ben Mayer, and Guy Weissenberg. Flashprove: an open-source library for provable inference. 2026.

[22] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. Hekaton: Horizontally-scalable zksnarks via proof aggregation. *Cryptology ePrint Archive*, 2024.

[23] Wenhao Wang, Fangyan Shi, Dani Vilardell, and Fan Zhang. Cirrus: Performant and accountable distributed snark. *Cryptology ePrint Archive*, 2024.

[24] 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, 2018.

[25] 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*, pages 3003–3017, 2022.

[26] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*, pages 299–328. Springer, 2022.

[27] Hua Xu, Mariana Gama, Emad Heydari Beni, and Jiayi Kang. Frittata: Distributed proof generation of fri-based snarks. *Cryptology ePrint Archive*, 2025.

[28] Yuanzhuo Yu, Mengling Liu, Yuncong Zhang, Shi-Feng Sun, Tianyi Ma, Man Ho Au, and Dawu Gu. Hyperfond: A transparent and post-quantum distributed snark with polylogarithmic communication. *Cryptology ePrint Archive*, 2025.

[29] Yuanzhuo Yu, Shi-Feng Sun, Yuncong Zhang, Chenhua Fan, Tianyi Ma, and Dawu Gu. Veloz: Efficient and flexible distribution framework for interleaved-code-based polynomial commitment scheme. *Cryptology ePrint Archive*, 2026.

[30] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: efficient field-agnostic polynomial commitment schemes from foldable codes. In *Annual International Cryptology Conference*, pages 138–169. Springer, 2024.

[31] Zongyang Zhang, Weihan Li, Yanpei Guo, Kexin Shi, Sherman SM Chow, Ximeng Liu, and Jin Dong. Fast {RS-IOP} multivariate polynomial commitments and verifiable secret sharing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3187–3204, 2024.

Appendix

1. Batch Opening

Batch opening is a technique [7], [9] for reducing the cost of proving and verifying *multiple* polynomial evaluation claims. In transparent, Merkle-based PCS settings (including BaseFold), the dominant practical cost of an opening is typically the IOPP query phase and the associated Merkle authentication paths, rather than the underlying field arithmetic. Batch opening reduces the number of separate openings by collapsing many evaluations into one global evaluation.

Assume there are $L = 2^\ell$ multilinear polynomials distributed across $M = 2^m$ workers. For each instance index $\boldsymbol{v} \in B_\ell$ and worker index $\boldsymbol{b} \in B_m$, worker $\mathcal{P}_{\boldsymbol{b}}$ holds a multilinear polynomial $f_{\boldsymbol{v}}^{(\boldsymbol{b})} \in \mathbb{F}_n^{\leq 1}[\boldsymbol{X}]$ (given by its evaluations on B_n). The global polynomial for instance $\boldsymbol{v}$ is the segmented multilinear polynomial

$$f_{\boldsymbol{v}}(\boldsymbol{x}, \boldsymbol{y}) := \sum_{\boldsymbol{b} \in B_m} f_{\boldsymbol{v}}^{(\boldsymbol{b})}(\boldsymbol{x}) \cdot \tilde{eq}(\boldsymbol{b}, \boldsymbol{y}) \in \mathbb{F}_{n+m}^{\leq 1}[\boldsymbol{X}, \boldsymbol{Y}].$$

Let $\{(\boldsymbol{\alpha_v}, \boldsymbol{\beta_v})\}_{\boldsymbol{v} \in B_\ell}$ be L evaluation points with $\boldsymbol{\alpha_v} \in \mathbb{F}^n$ and $\boldsymbol{\beta_v} \in \mathbb{F}^m$, and let $\{z_{\boldsymbol{v}}\}_{\boldsymbol{v} \in B_\ell} \subseteq \mathbb{F}$ be the claimed opening values. The goal is to prove, for all $\boldsymbol{v} \in B_\ell$, $f_{\boldsymbol{v}}(\boldsymbol{\alpha_v}, \boldsymbol{\beta_v}) \stackrel{?}{=} z_{\boldsymbol{v}}$.

We build on the distributed batch opening protocol in Cirrus [23]. The verifier samples a single random point $\boldsymbol{t} \xleftarrow{\$} \mathbb{F}^\ell$ and uses the multilinear Lagrange coefficients $\tilde{eq}(\boldsymbol{v}, \boldsymbol{t})$ to compress the L claims into one aggregated target value $s := \sum_{\boldsymbol{v} \in B_\ell} \tilde{eq}(\boldsymbol{v}, \boldsymbol{t}) \cdot z_{\boldsymbol{v}}$. The provers then prove a *single* SumCheck claim that is equivalent to $s = \sum_{\boldsymbol{v}} \tilde{eq}(\boldsymbol{v}, \boldsymbol{t}) \cdot f_{\boldsymbol{v}}(\boldsymbol{\alpha_v}, \boldsymbol{\beta_v})$, which can be combined with `UltraFold`.

Protocol 5 (Distributed batch opening)**.**

- $\mathcal{V}$ samples $\boldsymbol{t} \xleftarrow{\$} \mathbb{F}^\ell$ and sends $\boldsymbol{t}$ to $\mathcal{C}$ (who sends it to all workers).
- $\mathcal{V}$ computes the target $s := \sum_{\boldsymbol{v} \in B_\ell} \tilde{eq}(\boldsymbol{v}, \boldsymbol{t}) \cdot z_{\boldsymbol{v}}$.
- Each worker $\mathcal{P}_{\boldsymbol{b}}$ defines two multilinear polynomials over $(\boldsymbol{v}, \boldsymbol{x}) \in B_\ell \times B_n$: $g^{(\boldsymbol{b})}(\boldsymbol{v}, \boldsymbol{x}) := \tilde{eq}(\boldsymbol{v}, \boldsymbol{t}) \cdot f_{\boldsymbol{v}}^{(\boldsymbol{b})}(\boldsymbol{x})$ and $h^{(\boldsymbol{b})}(\boldsymbol{v}, \boldsymbol{x}) := \tilde{eq}(\boldsymbol{x}, \boldsymbol{\alpha_v}) \cdot \tilde{eq}(\boldsymbol{b}, \boldsymbol{\beta_v})$.
- The provers and $\mathcal{V}$ run a distributed SumCheck protocol for the claim

$$\sum_{\boldsymbol{b} \in B_m} \sum_{\boldsymbol{v} \in B_\ell} \sum_{\boldsymbol{x} \in B_n} g^{(\boldsymbol{b})}(\boldsymbol{v}, \boldsymbol{x}) \cdot h^{(\boldsymbol{b})}(\boldsymbol{v}, \boldsymbol{x}) \stackrel{?}{=} s,$$

2. Security of the BaseFold Instantiation of `UltraFold`

We first analyze the protocol with ideal oracle commitments and then apply the commitment compilation assumed in the main statement. Let $D' = 2^{d'}$ and $L = 2^\ell$. All encoded fragment tables use the variable order prescribed by the protocol. In the present instantiation, this order is $(\boldsymbol{y}, \boldsymbol{x})$, so we write

$$\boldsymbol{z}' = (\boldsymbol{z}_y, \boldsymbol{z}_x), \qquad \boldsymbol{r} = (\boldsymbol{r}_y, \boldsymbol{r}_x).$$

The notation $f_{\boldsymbol{u}}(\boldsymbol{x}, \boldsymbol{y})$ retains the displayed argument order. Let Acc denote the event that the verifier accepts.

We use BaseFold's PCS knowledge-soundness parameter regime. In particular, its proximity and code-distance conditions identify a unique nearby codeword for a sufficiently close oracle. The change from coefficient vectors to Boolean evaluation tables is an invertible change of basis and does not change the underlying codeword sets or their distances.

Lemma 3 (Centralized simulation)**.** *For every adversary controlling the coordinator and workers, there is a centralized prover producing the same verifier transcript in the packed protocol. The simulation preserves the verifier's acceptance probability and supports the same resets of the joint prover state.*

Proof. The centralized prover simulates the coordinator, all workers, and their internal messages. It forwards the same sumcheck polynomials, packed oracle commitments, terminal values, and query responses to the verifier, and delivers the verifier's challenges to the simulated parties in the prescribed order.

The public leaf-position maps identify the logical oracle entries authenticated by each Merkle leaf. For a distributed oracle, a worker's intra-subtree path followed by the coordinator's top-tree path is an authentication path in the global packed tree. Consequently, the distributed construction and the centralized simulation present identical authentication and algebraic checks to the verifier.

The redistribution during commitment and the oracle handoff during opening affect only where the prover computes and stores its data. They do not change the verifier's messages or predicates. Restoring the complete simulated state restores the coordinator and every worker, so the simulation also preserves the reset access used by the extractor. □

Lemma 4. *Fix the initial packed oracle and an opening prover whose acceptance probability is non-negligible. Under the assumed BaseFold PCS parameters, there are unique multilinear fragment polynomials $\{\widehat{f}_{\boldsymbol{u}}\}_{\boldsymbol{u} \in B_\ell}$ whose encodings are close to the corresponding initial oracle projections under BaseFold's proximity guarantee. Moreover,*

$$\Pr\Big[\mathsf{Acc} \ \wedge \ \exists \boldsymbol{u} \in B_\ell : s_{\boldsymbol{u}} \neq \widehat{f}_{\boldsymbol{u}}(\boldsymbol{r}_x, \boldsymbol{r}_y)\Big] \leq \mathrm{negl}(\lambda).$$

Here, $s_{\boldsymbol{u}}$ is the scalar authenticated by the bottom-code check for fragment $\boldsymbol{u}$.

Proof. Fix a fragment index $\boldsymbol{u}$. Project every packed oracle entry onto its $\boldsymbol{u}$-th component. The verifier's componentwise folding checks and bottom-code check then give a scalar BaseFold folding chain with the same challenges and query positions. This projection concerns oracle values; it does not require constructing a separate scalar Merkle commitment.

We apply the proximity and terminal-consistency argument from BaseFold's PCS analysis [30]. For completeness, we describe why this argument applies without an individual sumcheck for each fragment.

Write $C_i = \{\mathsf{Enc}_{d'-i}(\boldsymbol{v}) : \boldsymbol{v} \in \mathbb{F}^{2^{d'-i}}\}$ for $0 \le i \le d'$ and let δ be the coset-proximity threshold used by that analysis. Let Δ denote ordinary relative distance and Δ^* the corresponding coset distance. The parameters have a corresponding query-consistency threshold $\eta > 0$ such that

$$2\delta + \eta < \min_{0 \le i \le d'} \Delta_{C_i},$$

with negligible proximity error for the relevant BaseFold continuations and negligible repeated-query error $(1-\eta)^q$.

First, except with negligible probability of an accepting execution, each projected nonterminal oracle is within coset distance δ of its corresponding code. Indeed, condition on any transcript prefix ending with that oracle. If the oracle is farther from its code, acceptance of the full protocol implies acceptance of a BaseFold continuation on that oracle. The projected query indices are uniform over the corresponding smaller domains, so BaseFold's proximity bound applies to this continuation. Taking a union bound over the polynomially many fragments and levels preserves negligibility.

Second, fix the complete oracle chain before the query positions are sampled. If a single random query path fails the projected checks with probability greater than η, then all q independent query repetitions pass with probability at most $(1-\eta)^q$. Outside this negligible accepting event, every adjacent pair of projected oracles therefore satisfies

$$\Delta\big(\boldsymbol{\pi}_{\boldsymbol{u},i+1}, F_{r_{i+1},i}\boldsymbol{\pi}_{\boldsymbol{u},i}\big) \le \eta.$$

This follows because the query position at each level is uniform, and a failed folding equation at that level makes the complete query path fail.

Let $\boldsymbol{c}_{\boldsymbol{u},i} \in C_i$ be the nearby codeword at level i. At the bottom, set $\boldsymbol{c}_{\boldsymbol{u},d'} := \boldsymbol{\pi}_{\boldsymbol{u},d'} = \mathsf{Enc}_0(s_{\boldsymbol{u}})$, as required by the verifier's bottom-code check. Folding a pair can introduce an error only if that input pair contains an error, so

$$\Delta\big(F_{r_{i+1},i}\boldsymbol{\pi}_{\boldsymbol{u},i}, F_{r_{i+1},i}\boldsymbol{c}_{\boldsymbol{u},i}\big) \le \Delta^*(\boldsymbol{\pi}_{\boldsymbol{u},i}, \boldsymbol{c}_{\boldsymbol{u},i})\,.$$

The triangle inequality consequently gives

$$\Delta\big(\boldsymbol{c}_{\boldsymbol{u},i+1}, F_{r_{i+1},i}\boldsymbol{c}_{\boldsymbol{u},i}\big) \le 2\delta + \eta < \Delta_{C_{i+1}}.$$

Both vectors belong to C_{i+1}, so they must be equal.

It follows inductively that the nearby codewords form the exact folding chain of the polynomial underlying the initial nearby codeword. Since a BaseFold fold restricts the next variable to its challenge, the authenticated bottom scalar satisfies

$$s_{\boldsymbol{u}} = \widehat{f}_{\boldsymbol{u}}(\boldsymbol{r}_x, \boldsymbol{r}_y).$$

The initial nearby codeword is unique because ordinary distance is bounded by coset distance and $2\delta < \Delta_{C_0}$. Thus, $\widehat{f}_{\boldsymbol{u}}$ is fixed by the initial oracle before the opening challenges are sampled.

Finally, taking a union bound over the L fragment indices proves the claimed joint guarantee. □

Lemma 5 (Extraction of the polynomials)**.** *Under the conditions of Lemma 4, an extractor with access to the joint opening prover recovers all fragment polynomials $\{\widehat{f}_{\boldsymbol{u}}\}_{\boldsymbol{u}\in B_\ell}$ except with negligible probability. Its running time is polynomial in the instance size and the inverse acceptance probability.*

Proof. The extractor repeatedly restores the prover to the same state immediately before opening and runs the protocol with fresh verifier randomness. For each accepting execution, it records the challenge tuple $\boldsymbol{r}$ and all authenticated bottom scalars $(s_{\boldsymbol{u}})_{\boldsymbol{u}\in B_\ell}$.

For $\boldsymbol{r} \in \mathbb{F}^{d'}$, define its Lagrange evaluation vector

$$\boldsymbol{e}(\boldsymbol{r}) := \big(\tilde{eq}(\boldsymbol{a}, \boldsymbol{r})\big)_{\boldsymbol{a}\in B_{d'}} \in \mathbb{F}^{D'}.$$

The extractor retains an accepting execution only when its evaluation vector increases the span of the previously retained vectors. It continues until it has D' linearly independent vectors.

To bound the number of executions, let $W \subsetneq \mathbb{F}^{D'}$ be any proper span of previously retained vectors. Choose a nonzero vector $\boldsymbol{c}$ orthogonal to W. The polynomial $P_{\boldsymbol{c}}(\boldsymbol{R}) := \sum_{\boldsymbol{a}\in B_{d'}} \boldsymbol{c}[\boldsymbol{a}]\tilde{eq}(\boldsymbol{a}, \boldsymbol{R})$ is nonzero and multilinear. It is nonzero because its evaluations on the Boolean hypercube are exactly the entries of $\boldsymbol{c}$. If $\boldsymbol{e}(\boldsymbol{r}) \in W$, then $P_{\boldsymbol{c}}(\boldsymbol{r}) = 0$. Therefore, for a fresh uniform challenge tuple, $\Pr[\boldsymbol{e}(\boldsymbol{r}) \in W] \le \frac{d'}{|\mathbb{F}|}$.

Let ε be the prover's acceptance probability. A fresh execution both accepts and increases the current rank with probability at least $\varepsilon - \frac{d'}{|\mathbb{F}|}$. Under the assumed field-size conditions, this is inverse-polynomial whenever ε is non-negligible. Thus, the expected number of executions needed to obtain full rank is at most $\frac{D'}{\varepsilon - d'/|\mathbb{F}|}$. A polynomial cutoff based on an inverse-polynomial lower bound on ε gives a polynomial-time extractor with negligible failure probability.

Let $\boldsymbol{r}^{(1)}, \ldots, \boldsymbol{r}^{(D')}$ be the challenge tuples, and define $\mathsf{V}[k, \boldsymbol{a}] := \tilde{eq}(\boldsymbol{a}, \boldsymbol{r}^{(k)})$, $\mathsf{S}[k, \boldsymbol{u}] := s_{\boldsymbol{u}}^{(k)}$. By construction, $\mathsf{V} \in \mathbb{F}^{D'\times D'}$ is invertible. The extractor computes $\widehat{\mathsf{T}} := \mathsf{V}^{-1}\mathsf{S}$.

By Lemma 4, the probability that any of the polynomially many attempted executions is accepted with an incorrect terminal scalar is negligible. Outside this event, every retained scalar satisfies $s_{\boldsymbol{u}}^{(k)} = \widehat{f}_{\boldsymbol{u}}\big(\boldsymbol{r}_x^{(k)}, \boldsymbol{r}_y^{(k)}\big)$. Hence, the column $\widehat{\mathsf{T}}[\cdot, \boldsymbol{u}]$ is exactly the Boolean evaluation table of $\widehat{f}_{\boldsymbol{u}}$ in the encoder's prescribed variable order.

The same interpolation matrix recovers all L fragments. The sampling and linear algebra require polynomial time, and no efficient decoding algorithm for the BaseFold code is assumed. □

Proof of Lemma 1. Set $w_{\boldsymbol{u}} := \tilde{eq}(\boldsymbol{u}, \boldsymbol{z}_u)$, and $g(\boldsymbol{X}, \boldsymbol{Y}) := \sum_{\boldsymbol{u}\in B_\ell} w_{\boldsymbol{u}} f_{\boldsymbol{u}}(\boldsymbol{X}, \boldsymbol{Y})$. Multilinear interpolation gives

$$g(\boldsymbol{z}_x, \boldsymbol{z}_y) = f(\boldsymbol{z}_u, \boldsymbol{z}_x, \boldsymbol{z}_y) = v.$$

The sumcheck is therefore for the correct claim

$$v = \sum_{\substack{\boldsymbol{b}\in B_m \\ \boldsymbol{a}\in B_{n'}}} g(\boldsymbol{a}, \boldsymbol{b})\tilde{eq}(\boldsymbol{b}, \boldsymbol{z}_y)\tilde{eq}(\boldsymbol{a}, \boldsymbol{z}_x).$$

We verify that the distributed messages are the ordinary sumcheck messages in the order $(\boldsymbol{y}, \boldsymbol{x})$. The initial values sent to the coordinator satisfy $a_{\boldsymbol{b}} = g(\boldsymbol{z}_x, \boldsymbol{b})$. Their multilinear extension is therefore

$$a(\boldsymbol{Y}) = \sum_{\boldsymbol{b} \in B_m} a_{\boldsymbol{b}} \tilde{eq}(\boldsymbol{b}, \boldsymbol{Y}) = g(\boldsymbol{z}_x, \boldsymbol{Y}).$$

Summing over the Boolean $\boldsymbol{x}$-variables first shows that the coordinator's first m rounds are precisely the required sumcheck rounds over $\boldsymbol{y}$.

After these rounds, multilinearity gives

$$g(\boldsymbol{X}, \boldsymbol{r}_y) = \sum_{\boldsymbol{b} \in B_m} \tilde{eq}(\boldsymbol{b}, \boldsymbol{r}_y) g(\boldsymbol{X}, \boldsymbol{b}).$$

Consequently, the coordinator's weighted aggregation of the workers' remaining sumcheck messages, including the factor $\tilde{eq}(\boldsymbol{r}_y, \boldsymbol{z}_y)$, equals the ordinary sumcheck message in every subsequent round. All degree and round-consistency checks pass.

In parallel, correctness of the recursive encoder and its folding rule implies that every fragment oracle encodes the corresponding restriction of $f_{\boldsymbol{u}}$ in the same variable order. At the bottom, $\boldsymbol{\pi}_{\boldsymbol{u},d'} = \mathsf{Enc}_0\big(f_{\boldsymbol{u}}(\boldsymbol{r}_x, \boldsymbol{r}_y)\big)$. Thus, every componentwise fold check and bottom-code check passes. The honestly constructed Merkle trees also provide valid authentication paths.

Finally, the honest sumcheck terminal check is

$$\begin{aligned} v_{d'} &= \tilde{eq}(\boldsymbol{r}, \boldsymbol{z}') g(\boldsymbol{r}_x, \boldsymbol{r}_y) \\ &= \tilde{eq}(\boldsymbol{r}, \boldsymbol{z}') \sum_{\boldsymbol{u} \in B_\ell} w_{\boldsymbol{u}} s_{\boldsymbol{u}}, \end{aligned}$$

which is exactly the terminal check. This equality also holds when $\tilde{eq}(\boldsymbol{r}, \boldsymbol{z}') = 0$, because the verifier does not divide by this value. Therefore, the verifier accepts with probability 1. □

Proof of Lemma 2. By Lemma 3, we may treat the coordinator and all workers as one centralized adversarial prover. Fix its initial packed oracle and opening statement.

By Lemma 4, non-negligible acceptance implies the existence of fixed fragment polynomials $\{\widehat{f}_{\boldsymbol{u}}\}_{\boldsymbol{u} \in B_\ell}$ with the required per-fragment proximity guarantee. By Lemma 5, these polynomials can be efficiently extracted with the stated success guarantee.

Define

$$\widehat{f}(\boldsymbol{U}, \boldsymbol{X}, \boldsymbol{Y}) := \sum_{\boldsymbol{u} \in B_\ell} \tilde{eq}(\boldsymbol{u}, \boldsymbol{U}) \widehat{f}_{\boldsymbol{u}}(\boldsymbol{X}, \boldsymbol{Y}).$$

This polynomial is multilinear and satisfies $\widehat{f}(\boldsymbol{u}, \cdot, \cdot) = \widehat{f}_{\boldsymbol{u}}$ for every $\boldsymbol{u} \in B_\ell$. It therefore satisfies the consistency property in the lemma.

It remains to establish the claimed evaluation. Set $\widehat{g}(\boldsymbol{X}, \boldsymbol{Y}) := \sum_{\boldsymbol{u} \in B_\ell} w_{\boldsymbol{u}} \widehat{f}_{\boldsymbol{u}}(\boldsymbol{X}, \boldsymbol{Y})$ and $w_{\boldsymbol{u}} := \tilde{eq}(\boldsymbol{u}, \boldsymbol{z}_u)$. The initial oracle determines these polynomials before the opening challenges are sampled. Consider the fixed polynomial

$$\Phi(\boldsymbol{Y}, \boldsymbol{X}) := \widehat{g}(\boldsymbol{X}, \boldsymbol{Y}) \tilde{eq}(\boldsymbol{Y}, \boldsymbol{z}_y) \tilde{eq}(\boldsymbol{X}, \boldsymbol{z}_x).$$

It has degree at most two in each variable, and its Boolean sum is

$$\sum_{\substack{\boldsymbol{b} \in B_m \\ \boldsymbol{a} \in B_{n'}}} \Phi(\boldsymbol{b}, \boldsymbol{a}) = \widehat{g}(\boldsymbol{z}_x, \boldsymbol{z}_y) = \widehat{f}(\boldsymbol{z}).$$

Except with negligible probability of an accepting execution, Lemma 4 gives $s_{\boldsymbol{u}} = \widehat{f}_{\boldsymbol{u}}(\boldsymbol{r}_x, \boldsymbol{r}_y)$ for every fragment. The verifier's terminal check then implies

$$v_{d'} = \tilde{eq}(\boldsymbol{r}, \boldsymbol{z}') \widehat{g}(\boldsymbol{r}_x, \boldsymbol{r}_y) = \Phi(\boldsymbol{r}_y, \boldsymbol{r}_x).$$

Therefore, every such accepting execution also passes the ordinary sumcheck verifier for Φ, including its correct terminal evaluation.

If $\widehat{f}(\boldsymbol{z}) \neq v$, sumcheck soundness bounds this event by $2d'/|\mathbb{F}|$. Together with the negligible terminal-consistency error, this would make the packed prover's acceptance probability negligible, contradicting the hypothesis. Therefore, $\widehat{f}(\boldsymbol{z}) = v$.

Finally, apply the oracle-extraction and Merkle-compilation guarantee assumed for the selected BaseFold realization. The packed leaves and public leaf permutations change the authenticated data layout while preserving the logical oracle checks. The resulting realization inherits the additional errors and extraction losses of that compilation. Merkle binding ensures consistency of authenticated openings; the required extraction access is supplied by the compilation guarantee.

When $d' = 0$, the initial packed oracle is also the bottom oracle. The verifier directly checks every component's base-code membership and their weighted evaluation relation, so extraction and the claimed evaluation follow directly. This completes the proof. □

3. Choice of q

Here we discuss in detail our choice of q. As in FRI-style proximity tests, the soundness analysis is typically phrased in terms of a proximity parameter Δ (distance from the code) and a small slack ϵ (capturing lower-order terms coming from the correlated-agreement / list-decoding analysis and finite-field effects): for an oracle that is Δ-far from the target code, one query rejects with probability at least $\Delta - \epsilon$, so after q repetitions the overall soundness error is bounded by $(1 - \Delta + \epsilon)^q$ (see, e.g., the query-error bounds and discussion in DEEP-FRI [12]). For Reed-Solomon-based instantiations, WHIR [4] discusses the conjecture that the relevant (mutual) correlated-agreement / proximity-gap phenomenon holds essentially *up to capacity*, i.e., with proximity parameter $\Delta \approx 1 - \rho$ (more precisely, $1 - \rho - O(\epsilon)$), which would make the query-repetition behavior match the folklore $(1 - \Delta)^q$ heuristic [4]. Therefore, using the conjectured security, we have $q = \left\lceil \frac{\lambda}{\log_2(1/\rho)} \right\rceil = 34$.