# MamaBearZKP: A Holistic Co-design of Prime Fields and Proving Stacks for High-Throughput ZKP on Modern CPUs

Jipeng Zhang
School of Computing
National University of Singapore
Singapore, Singapore
jp-zhang@outlook.com

Yanpei Guo
School of Computing
National University of Singapore
Singapore, Singapore
guo.yanpei@u.nus.edu

Tao Lu
School of Computing
National University of Singapore
Singapore, Singapore
lutaocc2020@gmail.com

Hao Cheng
School of Cyber Science and Technology
Shandong University
Qingdao, China
hao.cheng@sdu.edu.cn

Jiaheng Zhang
School of Computing
National University of Singapore
Singapore, Singapore
jhzhang@nus.edu.sg

## Abstract

Sum-check and Fast Fourier Transforms (FFTs) dominate the computational cost of modern zero-knowledge proving systems, such as HyperPlonk (Eurocrypt 2023) and FRI-based schemes like DeepFold (USENIX Security 2025). Despite numerous optimizations, existing efforts remain fragmented across algorithmic, protocol, and implementation layers, leaving significant CPU performance potential untapped.

We present **MamaBearZKP**, a co-designed framework that bridges these layers to enable high-throughput ZK proving on modern CPUs. At its core, MamaBearZKP leverages MamaBear, a 49-bit prime field ($p = 2^{49} - 2^{34} + 1$), and introduces a systematic vectorization framework specifically tailored for the AVX-512IFMA execution model. By treating field arithmetic, protocol structure, and low-level hardware primitives as a unified optimization target, MamaBearZKP achieves unprecedented efficiency.

We instantiate our framework in a HyperPlonk–DeepFold prover and obtain single-thread speedups of up to 42×, 33×, 15×, 21×, and 21× for ZeroCheck, ProductCheck, DeepFold Commit, DeepFold Open, and end-to-end proof generation, respectively, compared to a Goldilocks-based baseline on the same platform. With 8-thread execution, the corresponding speedups increase to as much as 64×, 47×, 81×, 45×, and 45×. Across our end-to-end evaluations, MamaBearZKP also achieves up to 18× single-thread speedup over Plonky3, which already uses an AVX-512 BabyBear backend. Rather than resulting from isolated improvements, these gains arise from a *synergistic cascading* effect: the 49-bit field's headroom enables efficient lazy reduction, which paves the way for high-performance fused fold-and-evaluate kernels. The efficiency of these kernels facilitates a unified "stay-packed" dataflow throughout the HyperPlonk and DeepFold stacks; it is precisely this end-to-end dataflow that materializes the hardware throughput of AVX-512IFMA into realized performance gains.

## Keywords

zero-knowledge proofs, SIMD, AVX-512IFMA, sum-check protocol, polynomial commitment

## 1 Introduction

Zero-knowledge proofs (ZKPs) now underpin privacy-preserving payments, blockchain-oriented proving systems, and general-purpose zkVMs [9, 14, 53]. In practical deployments, however, the prover remains the dominant bottleneck. Modern proving systems are built around a small set of high-cost kernels. In polynomial interactive oracle proofs (PIOPs) such as HyperPlonk [21], ZeroCheck and ProductCheck are sum-check based. In FRI-style [10, 16] polynomial commitment schemes (PCS) such as DeepFold [29], fast Fourier transforms (FFTs) dominate the commitment phase. Because these components lie on the critical path of proof generation, their efficiency largely determines end-to-end prover throughput.

Despite this clear target, current optimization efforts are fragmented. Algorithmic and protocol-level improvements are typically developed in isolation from low-level implementation concerns. Meanwhile, implementation optimizations must inherit field choices, data layouts, and execution patterns that were not designed for modern SIMD hardware. This split is especially costly on contemporary CPUs with AVX-512IFMA support. Their 8-lane 52-bit integer multiply-accumulate datapath is well suited to the arithmetic intensity of prover kernels.

This fragmentation appears at multiple layers:

- **Field layer.** Common ZKP fields do not align with the AVX-512IFMA datapath. Goldilocks ($p = 2^{64} - 2^{32} + 1$) exceeds the 52-bit multiplier width, while BabyBear ($p = 2^{31} - 2^{27} + 1$) leaves much of it unused.
- **Protocol-kernel layer.** Standard prover kernels are not organized for vectorized execution. The usual sum-check fold order conflicts with SIMD execution flow and introduces extra permutation or rearrangement overhead, while naive FFT implementations process one layer at a time in a load-compute-store pattern that quickly becomes memory bound.
- **End-to-end proving layer.** Once these mismatches accumulate across arithmetic and kernel structure, the full prover fails to exploit the performance potential of contemporary CPUs.

**Table 1: Representative end-to-end prover times. Baseline is Goldilocks; both "This work" columns are MamaBear $\mathbb{F}_{p^3}$ with a 5-level FFT split. Rows are $\mu \in \{19, 21, 23\}$ of Table 6.**

| $\mu$ | Baseline | This work | Speed-up | This work Parallel | Speed-up |
|---|---|---|---|---|---|
| 19 | 4.34 s | 0.26 s | 17x | 97.3 ms | 45x |
| 21 | 18.61 s | 1.14 s | 16x | 0.44 s | 42x |
| 23 | 75.49 s | 6.37 s | 12x | 1.98 s | 38x |

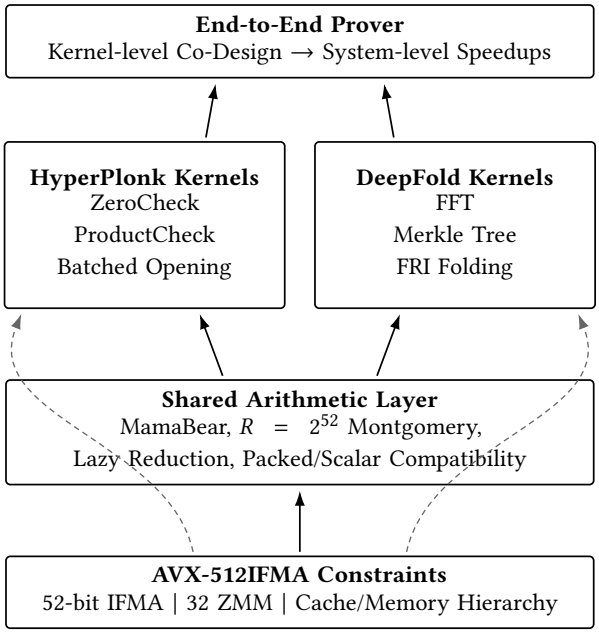



**Figure 1: Co-design path of MamaBearZKP: AVX-512IFMA constraints shape a shared arithmetic layer that enables HyperPlonk and DeepFold kernel families and yields end-to-end prover speedups.**

In this paper, we present **MamaBearZKP**, a co-designed framework for zero-knowledge proving on AVX-512IFMA CPUs. MamaBearZKP combines MamaBear, a 49-bit prime field with characteristic $p = 2^{49} - 2^{34} + 1$, with a systematic vectorization framework for the proving stack. Rather than optimizing arithmetic, protocol structure, and low-level kernels separately, MamaBearZKP treats field choice, arithmetic representation, protocol structure, data layout, and SIMD implementation as a co-designed optimization target. This yields a unified $R = 2^{52}$ Montgomery representation with lazy reduction, packed execution strategies for sum-check and ProductCheck, layer merging and sparse FFT pipelines for DeepFold, and a prover-wide stay-packed dataflow that minimizes conversion and memory overhead. The reason these layers must be designed together is that the gains are interdependent: the 49-bit field leaves enough room for lazy reduction within the AVX-512IFMA operand budget; that arithmetic regime makes fused fold-and-evaluate kernels practical across the dominant sum-check scans; those kernels, in turn, make it worthwhile to preserve packed dataflow across HyperPlonk, batched opening, and DeepFold rather than reverting to scalar form at protocol boundaries; and only this end-to-end packed execution converts AVX-512IFMA throughput into prover-level speedups. Optimizing any one layer in isolation leaves arithmetic, layout, or conversion mismatches that erase much of the system-level benefit.

Figure 1 summarizes the co-design path of MamaBearZKP: AVX-512IFMA hardware constraints motivate a shared arithmetic layer, which then enables both HyperPlonk and DeepFold kernels and ultimately translates into end-to-end prover gains. This joint focus is also empirically justified: in the Goldilocks baseline, we observe roughly comparable contributions from HyperPlonk and DeepFold to end-to-end proving time. Since the FFT-dominated DeepFold side scales as $O(N \log N)$, its share grows further with circuit size.

Table 1 previews representative end-to-end prover results and shows that the co-designed stack translates kernel-level gains into large system-level speedups across practical circuit sizes.

Our contributions are summarized as follows.

- **MamaBear field design (§3).** Existing ZKP fields either overflow the 52-bit IFMA multiplier or underuse it. We address this mismatch with MamaBear, a 49-bit prime field $p = 2^{49} - 2^{34} + 1$ tailored to AVX-512IFMA, together with a unified $R = 2^{52}$ Montgomery representation and a lazy-reduction discipline that propagates range invariants across the proving stack. This design enables zero-cost transitions between scalar and packed execution while providing a degree-3 extension field of size about 147 bits.
- **Co-designed proving kernels (§5).** Standard HyperPlonk kernels are poorly matched to SIMD execution because the memory layout is fundamentally misaligned with the instruction-level execution flow. We redesign these kernels for packed execution through fold reordering, fused fold-and-round evaluation, and a *per-wire* ProductCheck protocol with contiguous SIMD-friendly accesses. Together, these changes sustain packed execution through all but the last three sum-check rounds and eliminate the 25% padding overhead in ProductCheck.
- **Vectorized DeepFold PCS (§6).** Naive FFT-based PCS implementations quickly become memory bound and repeatedly cross packed/scalar boundaries. We therefore design a packed DeepFold strategy with FFT layer merging, sparse FFT techniques that exploit FRI zero-padding structure, and AVX-512 Merkle hashing, all organized under a prover-wide stay-packed dataflow principle. The key evidence is that layer merging reduces FFT memory accesses by 3×, allowing packed execution to translate into system-level gains rather than isolated kernel improvements.
- **System implementation and evaluation (§7).** The central question is whether a co-designed optimization target spanning field arithmetic, protocol structure, and implementation materially improves the full prover rather than just individual kernels. We answer this by implementing a complete HyperPlonk–DeepFold prover under a prover-wide stay-packed execution

strategy on an AMD Zen 5 processor with 8 physical cores. Relative to a Goldilocks-based baseline on the same platform, MamaBearZKP obtains single-thread speedups of up to 42×, 33×, 15×, 21×, and 21× for ZeroCheck, ProductCheck, DeepFold Commit, DeepFold Open, and end-to-end proof generation, respectively. With 8-thread execution, the corresponding speedups increase to as much as 64×, 47×, 81×, 45×, and 45×. Across the evaluated end-to-end settings, MamaBearZKP also achieves up to 18× single-thread speedup over Plonky3, which already uses an AVX-512 BabyBear backend.

Our implementation is available as an open-source artifact at https://github.com/Ji-Peng/MamaBearZKP-Artifact.

The rest of this paper is organized as follows. §2 reviews background on sum-check, HyperPlonk, DeepFold, and AVX-512IFMA. §3 presents the MamaBear field design. §4 introduces the framework-level stay-packed pipeline. §5 and §6 describe our vectorized HyperPlonk and DeepFold schemes, respectively. §7 reports experimental results. §8 surveys related work and §9 concludes with the methodology's adaptability and future work.

## 2 Preliminaries

### 2.1 Notation

Let $\mathbb{F}_p$ denote a prime field with characteristic $p$ and $\mathbb{F}_{p^k}$ its degree-$k$ extension. We write $\tilde{f}$ for the multilinear extension (MLE) of a function $f\colon \{0,1\}^\mu \to \mathbb{F}$. For a table $A$ of length $2^m$, $\widetilde{A}$ denotes its MLE and $\widetilde{A}(\mathbf{r})$ its evaluation at $\mathbf{r} \in \mathbb{F}^m$. For a vector $\mathbf{w} = (w_0, \ldots, w_{\ell-1})$, we write $\mathbf{w}_{<i} = (w_0, \ldots, w_{i-1})$ and $\mathbf{w}_{>i} = (w_{i+1}, \ldots, w_{\ell-1})$. The equality polynomial

$$\widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{x}) = \prod_{i=0}^{\mu-1} (w_i x_i + (1-w_i)(1-x_i))$$

interpolates the Boolean indicator of $\mathbf{x} = \mathbf{w}$. $\mathcal{L}_{U,\alpha}(X)$ denotes the Lagrange basis polynomial of $\alpha$ in a set $U$. We use $\mu$ for the number of variables and $N = 2^\mu$ for the circuit size.

### 2.2 Sum-check Protocol

The sum-check protocol [41] reduces the claim $C = \sum_{\mathbf{x}\in\{0,1\}^\mu} g(\mathbf{x})$ to one evaluation $g(\mathbf{r})$ through $\mu$ rounds. In round $i$, the prover sends a univariate polynomial

$$s_i(X_i) = \sum_{\mathbf{x}'\in\{0,1\}^{\mu-i-1}} g(r_0, \ldots, r_{i-1}, X_i, \mathbf{x}'),$$

and the verifier checks $s_i(0) + s_i(1) = C_i$ before sampling a fresh challenge $r_i$.

### 2.3 HyperPlonk

HyperPlonk [21] is a multilinear PIOP for structured circuits. A circuit with $N = 2^\mu$ gates is represented by wire tables $L, R, O\colon \{0,1\}^\mu \to \mathbb{F}$ and a selector table $S$ indicating whether a gate is additive or multiplicative.

*Gate constraint.* In this work, we restrict attention to circuits composed only of binary addition and multiplication gates. The gate polynomial is therefore

$$h(\mathbf{X}) = (1 - S(\mathbf{X}))\big(L(\mathbf{X}) + R(\mathbf{X})\big) + S(\mathbf{X}) \cdot L(\mathbf{X}) \cdot R(\mathbf{X}) - O(\mathbf{X}), \quad (1)$$

where $S(\mathbf{x}) = 0$ denotes an addition gate and $S(\mathbf{x}) = 1$ denotes a multiplication gate.

*ZeroCheck.* ZeroCheck reduces the claim $h(\mathbf{b}) = 0$ for all $\mathbf{b} \in \{0,1\}^\mu$ to a sum-check on

$$f(\mathbf{X}) = \widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{X}) \cdot h(\mathbf{X}),$$

where $\mathbf{w} \in \mathbb{F}^\mu$ is verifier-chosen.

*ProductCheck.* HyperPlonk's permutation argument reduces multiset equality to the ProductCheck relation

$$\prod_{\mathbf{x}\in\{0,1\}^m} f(\mathbf{x}) = \prod_{\mathbf{x}\in\{0,1\}^m} g(\mathbf{x}),$$

with

$$f(\mathbf{x}) = \alpha + w(\mathbf{x}) + \beta \cdot \mathrm{id}(\mathbf{x}), \qquad g(\mathbf{x}) = \alpha + w(\mathbf{x}) + \beta \cdot \sigma(\mathbf{x}),$$

where $w$ is the concatenation of the three wire tables padded from $3N$ to $4N = 2^{\mu+2}$. Following the DeepFold [29] implementation[1], we realize this check through a layered product argument: each round reduces a product-tree claim to a degree-3 sum-check over even/odd halves, and the terminal claims are discharged by PCS openings. Section 5 only needs this structure and the fact that the padded $4N$ layout is the baseline that our per-wire variant later removes. Here id is the canonical position embedding over the padded domain, whereas $\sigma$ is the copy-constraint permutation over the same slots. Concretely, $\mathrm{id}(\mathbf{x})$ labels the original slot of $w(\mathbf{x})$, while $\sigma(\mathbf{x})$ labels the slot to which that wire value is constrained. Before concatenation and padding, one can equivalently view the permutation side wirewise as three maps $(\sigma_L, \sigma_R, \sigma_O)$ on $\{0,1\}^\mu$; the corresponding preprocessing permutation polynomials are $P_0, P_1, P_2$ for the $L$, $R$, and $O$ wires, respectively. The global $\sigma$ is their padded concatenation, while the identity side has analogous trivial embeddings $(\mathrm{id}_L, \mathrm{id}_R, \mathrm{id}_O)$ and is therefore written simply as id.

*Batch opening.* ZeroCheck and ProductCheck leave a small set of multilinear evaluation claims. HyperPlonk batches them by a random linear combination over the claim index and reduces the whole set to one final degree-2 sum-check, followed by one PCS opening.

Appendix C gives a more detailed formal description of this baseline HyperPlonk flow, including the layered ProductCheck and the final batch-opening reduction.

### 2.4 DeepFold Polynomial Commitment

DeepFold [29] is a multilinear PCS built from FRI [10, 16] over Reed–Solomon codewords and can be viewed as a DEEP-enhanced refinement of BaseFold [63]. For a $\mu$-variate multilinear polynomial $\tilde{f}$, the commit phase evaluates its univariate twin on a working domain $L_0$ of size $K = \rho^{-1}2^\mu$ and Merkle-commits to the resulting codeword. The evaluation phase then performs $\mu$ fold rounds, maintains a small set of anchor claims induced by the query point and the DEEP challenge, and finally verifies $s$ FRI queries by checking collinearity and Merkle openings across the folded codewords.

For this paper, three facts matter. First, DeepFold is dominated by FFTs, codeword folds, and Merkle hashing, which makes it a natural target for SIMD and memory-layout co-design. Second, the split-FFT trick decomposes the round-0 FFT into $2^t$ smaller sub-FFTs and absorbs the first $t$ FRI folds into one fat round-0 commitment; this

[1] https://github.com/paulguoyanpei/DeepFold-Hyperplonk at commit 4148026.

**Table 2: Comparison of prime fields used in ZKP systems. "IFMA fit" indicates whether a single element fits in one 52-bit IFMA operand. "Headroom" denotes the number of additional bits available for deferred reduction. "2-adic" denotes the 2-adic valuation of $p-1$. The last column reports the common extension degree used to obtain an extension field in the ∼128-bit range; the number in parentheses is the resulting extension-field size in bits.**

| Field | Bits | IFMA fit | Head room | 2-adic | Degree |
|---|---|---|---|---|---|
| BabyBear | 31 | ✓ (waste) | 21 | 27 | 4 (∼124) |
| **MamaBear** | **49** | ✓ | **3** | **34** | **3** (∼**147**) |
| Goldilocks | 64 | ✗ | — | 32 | 2 (∼128) |

is the architectural lever exploited in §6, and Appendix J collects the fuller mechanism, soundness analysis, and parameter tradeoffs. Third, our evaluation fixes the FRI query number and code rate across backends and compares performance under that shared parameter regime.

## 2.5 AVX-512IFMA Instructions

AVX-512 exposes 32 architectural 512-bit ZMM registers, and AVX-512IFMA provides packed 52-bit integer fused multiply-add instructions over eight 64-bit lanes within those registers.

Concretely, `VPMADD52LUQ` and `VPMADD52HUQ` accumulate the low and high halves of a 52-bit×52-bit product inside one 512-bit vector. In this paper, these hardware facts drive the entire co-design: the 52-bit multiplier width determines the field representation in §3, while the 32-ZMM register budget constrains how aggressively we can fuse prover kernels before spills erase the benefit of packed execution.

# 3 MamaBear prime

The MamaBear prime and its arithmetic form the foundation of MamaBearZKP. Every optimization in subsequent sections relies on the properties established here. Throughout this paper, we restrict attention to proof systems over small fields, by which we mean that one base-field element fits in a 64-bit integer; systems built around elliptic-curve fields such as BN254 [8] or towers of binary fields such as Binius [24] fall outside our scope. Table 2 summarizes the field-design tradeoffs that motivate MamaBear relative to the main small-field alternatives considered in this paper.

## 3.1 Design Rationale

We seek a prime $p$ that simultaneously satisfies four constraints:

- **C1 IFMA fit.** $\lceil \log_2 p \rceil \le 52$ so that every field element fits in one IFMA operand slot.
- **C2 Reduction headroom.** Enough slack below 52 bits for *lazy reduction*: intermediate sums of several elements must remain representable without overflow.

**Algorithm 1:** Montgomery multiplication for MamaBear field using AVX-512IFMA

**Constants:** $p = 2^{49} - 2^{34} + 1$, $R = 2^{52}$, $p_{\text{inv}} = p^{-1} \bmod R$
**Input:** $a, b \in [0, R)^8$ with $R = 2^{52}$
**Output:** $r \equiv abR^{-1} \pmod p$ lane-wise; see Lemma D.2 for the bound

```
c0 ← VPMADD52LUQ(0, a, b);          // ab mod R
c1 ← VPMADD52HUQ(p, a, b);          // ⌊ab/R⌋ + p
m0 ← VPMADD52LUQ(0, c0, p_inv);     // c0 p^{-1} mod R
q  ← VPMADD52HUQ(0, m0, p);         // ⌊m0 p/R⌋
r  ← VSUB64(c1, q);
return r;
```

- **C3 Large smooth subgroup.** The multiplicative group order $p-1$ must have a large power-of-two factor $2^s$ for FFT of that scale.
- **C4 Extension-field size.** A degree-3 extension should yield an extension field of at least 128 bits.

*The MamaBear prime* $p = 2^{49} - 2^{34} + 1$ meets all four constraints. Its 49-bit representation leaves **3 bits of headroom** below the 52-bit IFMA boundary (**C1–C2**). The 2-adic valuation of $p-1$ is 34, supporting FFTs of size up to $2^{34}$—larger than Goldilocks' $2^{32}$ (**C3**). The sparse form (three nonzero coefficients) enables a fast reduction path using two shifts, one mask, and two add/sub operations (**C2**); Appendix D gives the corresponding lane-wise pseudocode and range proof in Algorithm 3. Finally, $|\mathbb{F}_{p^3}| \approx 2^{147}$, comfortably exceeding the 128-bit threshold in (**C4**). With all other parameters fixed, a larger field yields smaller soundness error and hence stronger security for HyperPlonk and FRI-based PCS [9, 12, 13, 21, 29, 30].

The name *MamaBear* follows the "Goldilocks and the Three Bears" tradition: at 49 bits, it sits between BabyBear (31) and Goldilocks (64)—"just right" for IFMA.

## 3.2 Unified Montgomery Representation

We adopt Montgomery form [42] with $R = 2^{52}$: the Montgomery image of $a \in \mathbb{F}_p$ is $\hat{a} = a \cdot R \bmod p$, stored as an unsigned 64-bit integer. The key advantage of $R = 2^{52}$ (matching the IFMA operand width) is that the **scalar type** and the **AVX-512 type** (holding 8 lanes) share the *same representation*. Converting between them is a zero-cost memory reinterpretation.

*Montgomery multiplication.* As shown in Algorithm 1, the AVX-512IFMA Montgomery kernel operates on 8-lane vectors $a, b \in [0, R)^8$; all interval bounds and congruences in the pseudocode are interpreted lane-wise.

## 3.3 Lazy Reduction Discipline

The 3-bit headroom ($49 + 3 = 52$) allows us to defer modular reduction: intermediate results may exceed $p$ as long as they stay below $2^{52}$ (or $2^{64}$ for additions before multiplication). We formalize this as a **Lazy Reduction trait** with the following primitives:

- lazy_add$(a, b)$: addition without reduction. If $a, b \in [0, kp)$, result $\in [0, 2kp)$.
- lazy_sub$(a, b)$: subtraction without reduction (caller ensures nonnegative).

- lazy_add_xp$(a, x)$: $a+xp$. Used before subtraction to keep values positive.
- con_sub_xp$(a, x)$: $\min(a,\ a - xp)$. Conditional subtraction that tightens the range by $xp$.
- reduce_fast$(a)$: modular reduction exploits the sparse form of $p$ as shown in Algorithm 3.
- reduce$(a)$: full reduction to $[0, p)$ using the first two primitives.

*Reduction strength hierarchy.* The cost ordering runs from reduce to reduce_fast to con_sub_xp, with the remaining lazy primitives cheaper still. Our framework always selects the *cheapest* operation that achieves the target range. For example, after MontMul with both inputs $< 2p$ and output $\in [0, 1.5p)$, a single con_sub_xp(1) suffices to reach $[0, p)$; calling reduce() would be wasteful.

## 3.4 Extension Field

We construct the degree-3 extension field $\mathbb{F}_{p^3} = \mathbb{F}_p[x]/(x^3 - x - 1)$, in which elements are identified by the triples $(a_0, a_1, a_2)$.

A critical optimization throughout our framework is **base-by-extension multiplication**: multiplying an extension element by a base element costs only 3 Montgomery multiplications in $\mathbb{F}_{p^3}$, compared to 6 for full extension multiplication using Karatsuba. We systematically prefer base-by-extension multiplication wherever the protocol structure allows one operand to remain in the base field.

# 4 Framework Overview

The unified scalar/AVX-512 representation of §3.2 enables a prover-wide design principle that cuts across the rest of the system: keep data in packed form whenever the next kernel can consume it directly. Figure 1 places this principle at the center of the overall co-design path: the shared arithmetic layer established in §3 must feed both the HyperPlonk and DeepFold kernels without repeatedly crossing scalar/packed boundaries. We refer to this cross-layer rule as the *stay-packed pipeline*. Its purpose is not to change any protocol, but to prevent layout conversion from erasing the arithmetic gains unlocked by MamaBear and AVX-512IFMA.

A simple example appears in $\mathbb{F}_{p^3}$. A scalar array naturally uses an array-of-structures (AoS) layout $[(a_{00}, a_{01}, a_{02}), (a_{10}, a_{11}, a_{12}), \ldots]$, where $(a_{i0}, a_{i1}, a_{i2})$ is one extension-field element, whereas the packed representation is a structure-of-arrays (SoA) layout

$$\begin{bmatrix} [a_{00} & a_{10} & \ldots & a_{70}], \\ [a_{01} & a_{11} & \ldots & a_{71}], \\ [a_{02} & a_{12} & \ldots & a_{72}] \end{bmatrix}.$$

For large arrays, the AoS/SoA conversion is memory-bound and scales with circuit size, so repeatedly crossing this boundary quickly erases SIMD gains.

The later sections therefore preserve packed dataflow across sum-check, ProductCheck, batched opening, FFT, and Merkle kernels whenever protocol boundaries permit, falling back to scalar form only at unavoidable tails or serialization points. In terms of the paper's organization, this section is the bridge between the field-specific arithmetic of §3 and the protocol-specific packed kernels of §5 and §6. This framework-level rule is what lets large component-level improvements survive into end-to-end proving with minimal loss, which is why it reappears in the evaluation discussion of §7.

# 5 Vectorized HyperPlonk

This section presents our SIMD-friendly designs for the two core HyperPlonk sub-protocols: ZeroCheck and ProductCheck. Both build on the MamaBear lazy reduction discipline (§3.3) and share the same high-level goal: keep the prover's table scans packed for as long as possible and restructure the protocol so that arithmetic savings translate into end-to-end throughput. Batched opening reuses the same packed sum-check primitives and the multithreading policy of §7.3 (Appendix K), so we do not give it a dedicated subsection.

## 5.1 Sum-check–SIMD Co-Design

*5.1.1 The Vectorization Barrier.* In the standard sum-check protocol, the prover folds variables in order $x_0, x_1, \ldots, x_{\mu-1}$. After round $i$, the bookkeeping table has $2^{\mu-i-1}$ entries. With 8-lane SIMD, the table drops below one packed block ($< 8$ entries) after round $\mu - 3$, forcing the last three rounds into scalar mode.

More critically, the standard fold direction interleaves even/odd entries ($T[2j]$ and $T[2j{+}1]$), which maps cleanly to adjacent memory but creates a data dependency pattern where adjacent lanes within a packed block contribute to the *same* output. This constrains SIMD performance in the fold kernel, as *cross-register* throughput typically exceeds that of *within-register* lane manipulation.

*5.1.2 Fold Reordering.* We reorder the sum-check variables so that the prover folds $x_3, x_4, \ldots, x_{\mu-1}$ first, then $x_0, x_1, x_2$ last:

$$\underbrace{x_3, x_4, \ldots, x_{\mu-1}}_{\text{packed rounds}}\ \underbrace{x_0, x_1, x_2}_{\text{scalar tail}}.$$

We lay out the bookkeeping table so that the eight assignments to $x_0, x_1, x_2$ occupy one packed block. For example, when $\mu = 4$, folding $x_3$ is an elementwise operation between the adjacent packed registers $[T[0], T[1], \ldots, T[7]]$ and $[T[8], T[9], \ldots, T[15]]$, producing one full packed block for the residual variables $x_0, x_1, x_2$. More generally, each packed round halves the number of packed blocks while keeping every surviving block full, so all $\mu - 3$ packed rounds keep every lane active.

*Correctness.* The sum-check protocol is agnostic to variable ordering: the verifier's challenges are applied in the corresponding reordered sequence, and the final evaluation point is a permutation of the standard one. Soundness and completeness are preserved; a formal proof is given in Appendix E.

*5.1.3 Fused Fold and Round Evaluation.* A naïve implementation treats the round-$i$ fold and the round-$(i{+}1)$ evaluation as two separate table passes. After receiving the verifier challenge $r_i$, it first materializes

$$T_{i+1}[j] = T_i[2j] + r_i \cdot (T_i[2j+1] - T_i[2j]),$$

and then, in the next round, scans $T_{i+1}$ again to accumulate the evaluations of the round-$(i{+}1)$ polynomial. Our implementation keeps the same protocol flow but changes this local schedule: after receiving $r_i$, it does not immediately fold. Instead, at the start of round $(i{+}1)$, it processes each quadruple $(T_i[4j], T_i[4j+1], T_i[4j+2], T_i[4j+3])$, forms

$$u_0 = T_i[4j] + r_i \cdot (T_i[4j+1] - T_i[4j]),$$

$$u_1 = T_i[4j+2] + r_i \cdot (T_i[4j+3] - T_i[4j+2]),$$

stores $(u_0, u_1)$ as the next bookkeeping values, and immediately uses the same pair to accumulate the round-$(i{+}1)$ polynomial.

Writing $M = |T_i|$ for the current bookkeeping-table length, the naïve schedule performs $M$ reads to fold, $M/2$ writes for $T_{i+1}$, and another $M/2$ reads for the next-round evaluation, for total table traffic $2M$. Our fused schedule performs only $M$ reads and $M/2$ writes, for total traffic $3M/2$, i.e., a 25% reduction and one fewer full scan over the folded table.

Because the prover sends the same round messages and receives the same verifier challenges in the same order, this optimization changes only the prover's local execution schedule and does not affect the sum-check protocol and its security.

*Gate polynomial rewriting.* Recall $h = (1{-}S)(L{+}R) + S \cdot L \cdot R - O$. We algebraically rewrite this as $h = (L{+}R) + S \cdot (L \cdot R - L - R) - O$, reducing the per-entry cost from 3 extension-field multiplications to 2 (one for $L \cdot R$ and one for the $S$-weighted term).

## 5.2 Prefix- and Suffix-Factoring of Eq-Poly

Both ZeroCheck and ProductCheck run eq-weighted sum-checks, so we apply two orthogonal factorizations to the equality polynomial and reuse the resulting packed kernel in both protocols.

*Prefix-factoring.* At round $i$, letting $\mathbf{x}$ denote the residual suffix variables, we write

$$s_i(X) = c_i(X) \cdot t_i(X), \qquad c_i(X) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<i}, \mathbf{r}_{<i}) \cdot \widetilde{\mathrm{eq}}(w_i, X),$$

$$t_i(X) = \sum_{\mathbf{x}} \widetilde{\mathrm{eq}}(\mathbf{w}_{>i}, \mathbf{x}) \cdot g(\mathbf{r}_{<i}, X, \mathbf{x}),$$

where $g$ is the protocol-specific gate. Following Gruen [28] and the subsequent use of the same degree-reduction idea in Twist and Shout [51], this peels the already fixed eq prefix out of the online interpolation target. In ZeroCheck it turns the degree-4 round polynomial into a degree-3 target; in ProductCheck it isolates the same linear eq prefix before the packed wire-product scan.

*Suffix-factoring.* Bagad et al. [5] further split the residual suffix as $\mathbf{x} = (\mathbf{x}_L, \mathbf{x}_R)$ and $\mathbf{w}_{>i} = (\mathbf{w}_L, \mathbf{w}_R)$. Concretely, if the residual suffix has $m = \mu - i - 1$ variables, we take

$$\mathbf{x}_R \in \{0,1\}^{\lfloor m/2 \rfloor}, \qquad \mathbf{x}_L \in \{0,1\}^{m - \lfloor m/2 \rfloor},$$

where $\mathbf{x}_R$ is the lower-index half scanned by the packed inner loop and $\mathbf{x}_L$ is the remaining higher-index block handled by the outer loop; $\mathbf{w}_R$ and $\mathbf{w}_L$ are split conformably. Then

$$\widetilde{\mathrm{eq}}(\mathbf{w}_{>i}, \mathbf{x}) = \widetilde{\mathrm{eq}}_L(\mathbf{w}_L, \mathbf{x}_L) \cdot \widetilde{\mathrm{eq}}_R(\mathbf{w}_R, \mathbf{x}_R).$$

Hence

$$t_i(X) = \sum_{\mathbf{x}_L} \widetilde{\mathrm{eq}}_L(\mathbf{w}_L, \mathbf{x}_L) \sum_{\mathbf{x}_R} \widetilde{\mathrm{eq}}_R(\mathbf{w}_R, \mathbf{x}_R) \cdot g(\mathbf{r}_{<i}, X, \mathbf{x}_L, \mathbf{x}_R).$$

The inner loop reuses the $\widetilde{\mathrm{eq}}_R$ table and the outer loop multiplies once by $\widetilde{\mathrm{eq}}_L$, which improves cache locality without changing the protocol. We use the same split in the packed rounds of both ZeroCheck and ProductCheck. Appendix F gives the time, space, cache, and soundness comparison with flat eq tables.

**Algorithm 2:** Protocol skeleton for the BDDT25-style ZeroCheck prover[5]

**Input:** Base-field tables $S, L, R, O : \{0,1\}^\mu \to \mathbb{F}_p$, challenge point $\mathbf{w} \in \mathbb{F}_{p^k}^\mu$, prefix length $\ell_0$

**Output:** Round messages $\{s_i(u)\}_{u \in \widehat{U}_3}$ for $i = 0, \ldots, \mu - 1$

**Interaction:** `send` means the prover sends a message to the verifier; `receive` means the prover receives a random challenge from the verifier

Precompute $A_i(\mathbf{v}, u)$ for all $i < \ell_0$, $\mathbf{v} \in U_3^i$, and $u \in \widehat{U}_3$ as in Algorithm 5;
Initialize $R_0 \leftarrow (1)$;
**for** $i \leftarrow 0$ **to** $\ell_0 - 1$ **do**
  $t_i(u) = \sum_{\mathbf{v} \in U_3^i} R_i[\mathbf{v}] \cdot A_i(\mathbf{v}, u)$ for all $u \in \widehat{U}_3$;
  $s_i(u) = c_i(u) \cdot t_i(u)$, send $\{s_i(u)\}_{u \in \widehat{U}_3}$, and receive $r_i$;
  $R_{i+1} \leftarrow R_i \otimes (\mathcal{L}_{U_3,\alpha}(r_i))_{\alpha \in U_3}$; `// Kronecker product`
`// For` $\ell_0$`-th round:`
For $\mathbf{x} \in \{0,1\}^{\mu - \ell_0}$, fold each $p_k \in \{S, L, R, O\}$ to $p_k(\mathbf{r}_{<\ell_0}, \mathbf{x}) = \sum_{\mathbf{y} \in \{0,1\}^{\ell_0}} \widetilde{\mathrm{eq}}(\mathbf{r}_{<\ell_0}, \mathbf{y}) \cdot p_k(\mathbf{y}, \mathbf{x})$;
Set $c_{\ell_0}(u) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<\ell_0}, \mathbf{r}_{<\ell_0}) \widetilde{\mathrm{eq}}(w_{\ell_0}, u)$ and compute $s_{\ell_0}(u) = c_{\ell_0}(u)\, t_{\ell_0}(u)$ for all $u \in \widehat{U}_3$ via (10);
send $\{s_{\ell_0}(u)\}_{u \in \widehat{U}_3}$ and receive $r_{\ell_0}$;
**for** $i \leftarrow \ell_0 + 1$ **to** $\mu - 1$ **do**
  Build $t_i(u)$ from $\widetilde{\mathrm{eq}}_L$, $\widetilde{\mathrm{eq}}_R$, and packed gate tables;
  Fuse round evaluation, fold, and table update; send $\{s_i(u)\}_{u \in \widehat{U}_3}$; receive $r_i$;

## 5.3 ZeroCheck: BDDT25-Style Sum-check

For ZeroCheck, we employ the optimized prover of Bagad et al. [5, Alg. 6 in Fig. 9] (hereafter BDDT25-style sum-check), as sketched in Algorithm 2; the matching verifier is given by Algorithm 4 in Appendix G. The two eq-factorizations of §5.2 reduce the round target from degree 4 to degree 3, and the round-sum identity removes one further transmitted point. As a result, only the three values at $\widehat{U}_3$ are exchanged per round. The verifier reconstructs $s_i(1)$ from the running claim and divides out the publicly known eq-prefix $c_i$ before interpolating. The first two ingredients follow the prefix- and suffix-factorings of §5.2; Appendix F elaborates on their global cost model, and Appendix G details the derivation of the ZeroCheck-specific accumulator tables $A_i(\mathbf{v}, u)$ and the transition-round formula at $i = \ell_0$.

We utilize the interpolation set $U_3 = \{\infty, 0, 1, 2\}$ per BDDT25, defining $q(\infty)$ as the $X^3$ coefficient of $q$. The prover thus emits evaluations for $\widehat{U}_3 = \{\infty, 0, 2\}$, leaving the value at $u = 1$ to be deduced by the verifier through the round relation. Together with the linear factor $c_i$, these values uniquely determine the degree-3 target $t_i$ on $U_3$. This choice streamlines the $u = \infty$ evaluation in our ZeroCheck gate, where only the cubic term $S \cdot L \cdot R$ remains non-vanishing, significantly simplifying the resulting expressions.

BDDT25 adds a third ingredient on top of those two factorizations: the first $\ell_0$ rounds are executed on the small-value grid $U_3$. In this paper, *small-value* refers to a base-field evaluation point, as opposed to an extension-field value. The core computation for the

first $\ell_0$ rounds therefore remains in $\mathbb{F}_p$, delaying the expensive $\mathbb{F}_{p^3}$ arithmetic until a later transition round.

While Bagad et al. [5] focus on Spartan-in-Jolt and Binius-style binary towers[2], our integration into HyperPlonk requires significant adaptation. First, we must transition from base-field tables $(S, L, R, O)$ to the extension field $\mathbb{F}_{p^3}$ while maintaining the stay-packed dataflow of §4. Second, our round kernel integrates the *eq*-factoring strategy (§5.2) with the fused fold-and-evaluation strategy (§5.1.3) and an optimized base-by-extension multiplication schedule. Detailed engineering trade-offs are deferred to Appendices F and G.

Our implementation follows a three-phase schedule. In rounds $i < \ell_0$, we precompute accumulator tables $A_i(\mathbf{v}, u)$ for $\mathbf{v} \in U_3^i$ and $u \in \widehat{U}_3$, then update a short tensor vector of Lagrange coefficients. ZeroCheck adds one more pruning rule in this phase: whenever the small-value prefix stays Boolean, the corresponding accumulator vanishes because $h(\mathbf{b}) = 0$ for all $\mathbf{b} \in \{0,1\}^\mu$. At the transition round $i = \ell_0$, we first fold each base witness table $p_k \in \{S, L, R, O\}$ into $p_k(\mathbf{r}_{<\ell_0}, \cdot)$, then evaluate $t_{\ell_0}(u)$ for $u \in \widehat{U}_3$ through the same eq factorization of §5.2. For rounds $i > \ell_0$, we combine the same split equality tables with the fused fold-and-evaluate kernel from §5.1.3 and keep the bookkeeping tables packed until the final three scalar rounds.

*Choice of $\ell_0$.* The prefix length trades two opposing costs against each other, exactly as analyzed by Bagad et al.: a larger $\ell_0$ inflates the $(d{+}1)^{\ell_0}$ prefix tables and their precompute, but postpones more $\mathbb{F}_{p^3}$ work. In our $\mathbb{F}_{p^3}$ prover the optimum is $\ell_0 = 2$, in the single-threaded and the 8-thread setting alike, and the implementation fixes it there for both. Sweeping $\ell_0 \in \{1, 2, 3\}$ over $\mu = 18, \ldots, 28$ puts $\ell_0 = 2$ ahead of both neighbours in every measured cell: $\ell_0 = 1$ leaves too much arithmetic in the extension field, while at $\ell_0 = 3$ the prefix precompute already outweighs what it defers. The general rule is that as the cost ratio $C_{\mathbb{F}_p}/C_{\mathbb{F}_{p^3}}$ decreases and avoiding extension-field arithmetic becomes more valuable, a larger $\ell_0$ becomes preferable.

## 5.4 Per-Wire Batched ProductCheck

*5.4.1 Motivation.* As described in §2.3 and §C, the baseline ProductCheck forms two tables $f$ and $g$ by concatenating the three wire tables and padding the result from $3N$ to $4N = 2^{\mu+2}$. Equivalently, the baseline product tree and its subsequent sum-check run on a $(\mu + 2)$-variate cube of length $4N$, even though only $3N$ entries carry wire data. The padded fourth block exists only to force the concatenation to a power-of-two length, so materializing it wastes **25%** of the product-tree and sum-check work.

Our per-wire prover keeps the same ProductCheck relation but decomposes these two sides into three wire-local tables

$$f_q(\mathbf{x}) = \alpha + w_q(\mathbf{x}) + \beta \cdot \mathrm{id}_q(\mathbf{x}), \qquad g_q(\mathbf{x}) = \alpha + w_q(\mathbf{x}) + \beta \cdot \sigma_q(\mathbf{x}),$$

for $q \in \{L, R, O\}$ and $\mathbf{x} \in \{0,1\}^\mu$, where $(w_L, w_R, w_O) = (L, R, O)$ and $\mathrm{id}_q, \sigma_q$ are the restrictions of the baseline identity and permutation embeddings to wire $q$. Thus, unlike the baseline $(\mu+2)$-variate cube, our prover works over three independent $\mu$-variate cubes of length $N$ and never materializes the artificial fourth block.

[2] https://github.com/ingonyama-zk/smallfield-super-sumcheck/tree/sb/eq-optimized

*5.4.2 Per-wire design.* We keep the claim notation of §2.3 and §C, but specialize the prover to the wire-local tables above. For each side $s \in \{f, g\}$ and each wire $q \in \{L, R, O\}$, define a product tree

$$T_0^{s,q} = s_q, \qquad T_{j+1}^{s,q}[k] = T_j^{s,q}[2k] \cdot T_j^{s,q}[2k+1].$$

Since each $T_0^{s,q}$ has length $2^\mu$, the top layer $T_{\mu-1}^{s,q}$ has two entries. The prover sends

$$\tau_{q,0}^{(s)} = T_{\mu-1}^{s,q}[0], \qquad \tau_{q,1}^{(s)} = T_{\mu-1}^{s,q}[1],$$

for all $q \in \{L, R, O\}$ and both sides $s \in \{f, g\}$, together with the aggregated products

$$\Pi^{(s)} = \prod_{q \in \{L,R,O\}} \tau_{q,0}^{(s)} \cdot \tau_{q,1}^{(s)}.$$

The verifier checks $\Pi^{(f)} = \Pi^{(g)}$, samples $r_0$ and $\lambda$, sets $\mathbf{p}_{\mu-1} = (r_0)$, and defines the wire weights

$$\chi_L(\lambda) = 1, \qquad \chi_R(\lambda) = \lambda, \qquad \chi_O(\lambda) = \lambda^2.$$

This is the only place where wire batching occurs: $\lambda$ compresses the three wire claims for a fixed side $s$, but the two sides $f$ and $g$ remain separate throughout the protocol. The initial claims are therefore

$$y_{\mu-1}^{(s)} = \sum_{q \in \{L,R,O\}} \chi_q(\lambda) \cdot \widetilde{T_{\mu-1}^{s,q}}(r_0), \qquad s \in \{f, g\}.$$

Each level-$j$ sum-check is again eq-weighted, so ProductCheck reuses the same two factorizations from §5.2: the fixed prefix is peeled into the linear factor $c_i(X)$, and the residual suffix is split into $\widetilde{\mathrm{eq}}_L$ and $\widetilde{\mathrm{eq}}_R$ tables for the packed inner/outer scan. This shared eq engine is what lets ZeroCheck and ProductCheck use the same SIMD sum-check infrastructure despite having different gates.

For each lower level $j = \mu-2, \mu-3, \ldots, 0$, the prover and verifier then run *two synchronized degree-3 sum-checks*, one for each side $s \in \{f, g\}$, on the relation

$$y_{j+1}^{(s)} = \sum_{\mathbf{x} \in \{0,1\}^{\mu-1-j}} \widetilde{\mathrm{eq}}(\mathbf{p}_{j+1}, \mathbf{x}) \sum_{q \in \{L,R,O\}} \chi_q(\lambda) \cdot \widetilde{T_j^{s,q,\mathrm{e}}}(\mathbf{x}) \cdot \widetilde{T_j^{s,q,\mathrm{o}}}(\mathbf{x}),$$

where $T_j^{s,q,\mathrm{e}}$ and $T_j^{s,q,\mathrm{o}}$ are the even/odd halves of $T_j^{s,q}$. After the sum-check at level $j$ returns a challenge vector $\mathbf{r}_j$, the prover reveals the six pairs

$$(v_{q,0}^{(s)}, v_{q,1}^{(s)}) = \big(\widetilde{T_j^{s,q,\mathrm{e}}}(\mathbf{r}_j),\ \widetilde{T_j^{s,q,\mathrm{o}}}(\mathbf{r}_j)\big),$$

the verifier samples a fresh scalar $\rho_j$, sets $\mathbf{p}_j = (\rho_j, \mathbf{r}_j)$, and updates

$$y_j^{(s)} = \sum_{q \in \{L,R,O\}} \chi_q(\lambda) \cdot \big(v_{q,0}^{(s)} + (v_{q,1}^{(s)} - v_{q,0}^{(s)}) \cdot \rho_j\big).$$

This is exactly the structure of our per-wire prover: the six wire-local arrays are folded in lockstep, the two sides share one challenge vector, and each packed gate accumulates $\sum_q \chi_q(\lambda) \cdot T_j^{s,q,\mathrm{e}} \cdot T_j^{s,q,\mathrm{o}}$ for a fixed side $s$.

*5.4.3 Correctness.* The per-wire protocol proves the same ProductCheck relation as §2.3 and §C; the only change is that the padded concatenation is replaced by three wire-local factors, which are then recombined via the random linear combination $\chi_q(\lambda)$. The batching variable $\lambda$ compresses the three wire claims on each side independently, while the two side claims $y_j^{(f)}$ and $y_j^{(g)}$ are preserved all the way to the PCS opening. The soundness analysis therefore follows from the usual ProductCheck argument plus the Schwartz–Zippel bound for the degree-2 polynomial in $\lambda$. From a cost perspective, removing the padded fourth block eliminates about 25% of the ProductCheck computation, since both the product tree and the subsequent sum-check scans now operate on $3N$ useful values rather than $4N$ padded ones. Because the prover no longer carries the artificial fourth block, the ProductCheck portion of the proof also becomes slightly smaller. Appendix H gives a detailed side-by-side comparison of the padded and per-wire ProductCheck designs, including prover work, SIMD-level effects, soundness, verifier cost, and proof size.

## 6 Vectorized DeepFold

This section describes our vectorized DeepFold PCS [29]. The commit phase is dominated by FFTs and Merkle hashing. Accordingly, §6.1, §6.2, and §6.3 present three orthogonal FFT optimizations: layer merging, sparse FFTs for zero-padded polynomials, and the split-FFT trick. Section 6.4 then discusses our high-throughput Merkle tree construction. The opening phase performs FRI folds, next-domain evaluations, and batched polynomial combination, all of which reuse the same packed primitives under the stay-packed pipeline of §4 and the selective multithreading policy of §7.3 (Appendix K).

### 6.1 FFT Layer Merging

A radix-2 DIF FFT applies butterflies of the form $(a, b) \mapsto (a+b,\ (a-b)\omega^k)$, where the twiddle exponent $k$ depends on the current layer and butterfly index.

*6.1.1 Motivation.* A standard radix-2 DIF FFT of size $N_{\text{fft}} = 2^L$ performs $L$ butterfly layers, each requiring a full load and store of the $N_{\text{fft}}$-element array. On modern CPUs, these repeated round-trips are a major contributor to FFT runtime.

Earlier vectorized NTT implementations for lattice cryptography, such as the AVX2 codebases [56, 57] of ML-KEM [19, 44] and ML-DSA [43], already exploit layer merging tailored to fixed size-256 NTTs. In contrast, DeepFold must support variable FFT sizes rather than relying on a fixed merging schedule.

*6.1.2 Generic Three-Stage Framework.* For $L \geq 6$, we express the forward DIF FFT as a three-stage pipeline controlled only by $L$:

(1) **Phase 1: no-shuffle merged prefix.** This is the high-performance core of the design. As long as the current butterfly stride is at least the SIMD width, we greedily merge three FFT layers at a time without any lane shuffles, so the kernel only performs packed butterflies and contiguous loads/stores.
(2) **Phase 2: alignment remainder.** If $L \bmod 3 \in \{1, 2\}$, we execute one 1-layer or 2-layer no-shuffle kernel as a transition. Its purpose is to leave exactly three FFT layers for the final stage.
(3) **Phase 3: specialized tail-3 kernel.** The last three layers are handled by a dedicated in-register shuffle network. This stage is carefully designed because it must simultaneously realize the dataflow of the final three FFT layers, emit an output layout friendly to Merkle tree construction, and align the round-0 FRI fold with contiguous memory reads instead of strided or gather accesses.

In Phase 1, one 3-layer no-shuffle kernel loads 8 ZMM registers (64 coefficients), executes 12 radix-2 butterflies entirely in registers, and stores the block once. This choice of three layers is dictated by the AVX-512IFMA execution model of §2.5: a 3-layer kernel needs 8 data registers, 7 twiddle registers, and only a few temporaries, so it fits within the available 32 ZMM registers without spills. A 4-layer kernel would require 16 data registers and 15 twiddle registers before temporaries, reintroducing memory traffic through spills and largely erasing the gain.

The merged schedule reduces memory traffic by up to 3×; more precisely, Appendix I shows that the reduction factor for an $L$-layer FFT is $L/\lceil L/3 \rceil$. Appendix I also gives the full control flow, the register-budget rationale, and the exact tail-shuffle network.

Phase 3 applies a fixed `shuffle4` → `shuffle2` → `shuffle1` network and outputs data in a *pair-major blocked* layout:

$$[a_0, \ldots, a_7,\ b_0, \ldots, b_7,\ a_8, \ldots, a_{15},\ b_8, \ldots, b_{15},\ \ldots]$$

where $(a_j, b_j)$ is the $j$-th evaluation pair $(f(\omega^j),\ f(\omega^{j+N_{\text{fft}}/2}))$. The shuffle network is co-designed so that this final stage simultaneously achieves three goals:

(1) it constructs exactly the dataflow required by the last three FFT layers once the butterfly stride falls below the SIMD width;
(2) it emits 16-field-element contiguous chunks (= 2 packed blocks), which are directly suitable as Merkle leaves; and
(3) it places each round-0 FRI evaluation pair $(a_j, b_j)$ in adjacent memory locations, so the fold reads contiguous blocks instead of relying on strided or gather loads.

### 6.2 Sparse FFT for Zero-Padded Polynomials

DeepFold uses a code rate $\rho = 1/R_{\text{fri}}$ (e.g., $R_{\text{fri}} = 8$), meaning the coefficient vector is zero-padded from length $M = 2^n$ to $N_{\text{fft}} = R_{\text{fri}} \cdot 2^n$. In a DIF FFT, the first $r = \log_2 R_{\text{fri}}$ layers are *degenerate*. At layer $t < r$ the butterfly stride is $m_t = N_{\text{fft}}/2^{t+1}$, and because

$$M = N_{\text{fft}}/2^r \leq N_{\text{fft}}/2^{t+1} = m_t,$$

every live coefficient still lies entirely in the "plus" half of its butterfly pair, while the "minus" half remains zero. Thus the DIF butterfly $(a, b) \mapsto (a + b,\ (a - b)\omega^k)$ always specializes to

$$(a, 0) \mapsto (a,\ a\omega^k),$$

which means these first $r$ layers do not mix different coefficients at all: they only copy each $a_j$ into more branches and accumulate deterministic twiddle phases.

*Chirp block decomposition.* Instead of executing $r$ degenerate layers, we directly construct $R_{\text{fri}}$ *chirp blocks*:

$$C_q[j] = a_j \cdot \omega^{\beta_q \cdot j}, \qquad \beta_q = \text{bitrev}_r(q), \qquad q \in \{0, \ldots, R_{\text{fri}} - 1\}.$$

The exponent $\beta_q$ records exactly which "take the copied branch" / "take the twiddled branch" decisions were made across the first $r$ layers; in DIF order these decisions appear in bit-reversed order,

hence $\beta_q = \mathrm{bitrev}_r(q)$. After this sparse prefix, the original zero-padded FFT has been reorganized into $R_{\mathrm{fri}}$ dense size-$M$ inputs, and the remaining layers are just ordinary dense FFTs on those chirp blocks. This saves roughly $0.5N_{\mathrm{fft}}$ Montgomery multiplications and $3N_{\mathrm{fft}}$ additions (at $r = 3$), and the $R_{\mathrm{fri}}$ blocks are naturally independent, enabling straightforward parallelization.

## 6.3 Parameterized Analysis of the Split FFT

§2.4 and §J.1 introduced the $t$-level split as an FFT decomposition whose soundness is analyzed in §J.2. The remaining question is: how does the choice of $t$ affect proof size?

This section focuses on the two primary conclusions driving our design choices, while deferring the full derivation, instantiated table, and scaling analysis to §J.3. Beyond the notation fixed in §2.4 ($\mu$, $\rho$, $K = \rho^{-1}2^{\mu}$, and the query count $s$), we use:

- $t$: number of FFT split levels ($0 \le t \le \mu$).
- $N_p$: total number of base-field polynomials committed in one batched open, partitioned across $J$ independent round-0 Merkle trees with counts $N_{p,1}, \ldots, N_{p,J}$ (so $N_p = \sum_j N_{p,j}$); after round 0 the FRI transcript collapses into a single shared trace.
- $h$: hash-digest size; $\mathrm{fe}_b$, $\mathrm{fe}_e$: base-field and extension-field element sizes (in bytes).

For our HyperPlonk–DeepFold instance $N_p = 7$ and $J = 2$: the preprocessed group carries $N_{p,1} = 4$ polynomials $\{S, P_0, P_1, P_2\}$ and the witness group carries $N_{p,2} = 3$ polynomials $\{L, R, O\}$; each group keeps its own round-0 tree but they share for round $i \ge 1$.

Lemma 6.1 (Arithmetic invariance of the split FFT). *For any admissible split depth $t$, the split replaces one size-$K$ FFT by $t$ chirp layers and $2^t$ independent size-$K/2^t$ sub-FFTs without changing the total radix-2 butterfly count:*

$$C_{fft}^{split}(K,t) = \tfrac{tK}{2} + 2^t\, C_{fft}(K/2^t) = \tfrac{K}{2}\log_2 K = C_{fft}(K). \tag{2}$$

Lemma 6.1 says that $t$ is not an arithmetic knob. Increasing $t$ preserves the total field-operation count, while exposing $2^t$ independent sub-FFTs and shrinking each sub-FFT's working set to $K/2^t$ elements. The split prefix reuses the same chirp-block kernel as §6.2.

*Merkle effect.* At commit time, the $t$-level split replaces the first $t$ round-$i$ trees of each group with one fat round-0 tree whose leaves contain the $2^{t+1}$ values needed to reconstruct those early rounds on demand. This removes roughly $J\sum_{i=1}^{t-1} K/2^{i+1} \approx JK/2$ hash invocations from the commit path, independently of $N_p$.

Lemma 6.2 (Per-query proof-size delta). *Let $\Delta P_q(t)$ be the per-query proof-size change of a $t$-level split relative to standard FRI. Then*

$$\Delta P_q(t) = (2^{t+1} - 2)N_p\, fe_b - S_q(K,t;J,h,fe_e), \tag{3}$$

*where $S_q(K,t;J,h,fe_e)$ is the per-query saving term from round-0 path shortening and absorbed trees, expanded in Appendix J.3. Hence the total proof-size change is $s \cdot \Delta P_q(t)$.*

Lemma 6.2 isolates the design asymmetry. The inflation term is exponential in $t$ and linear in $N_p$, whereas the saving term $S_q$ grows only linearly in $t$ and is independent of $N_p$. Therefore larger batched opens force the proof-size-optimal split depth downward even though the commit-side locality keeps improving.

**Table 3: Backends and field parameters used in the evaluation. The field-size column reports $\log_2 |\mathbb{F}_{p^d}|$ for the extension degree used by each backend.**

| Backend | Prime | Degree | Field Size | ISA |
|---|---|---|---|---|
| Goldilocks | $2^{64} - 2^{32} + 1$ | 2 | ~128 | x64 |
| BabyBear | $2^{31} - 2^{27} + 1$ | 4 | ~124 | x64 |
| Plonky3 | $2^{31} - 2^{27} + 1$ | 4 | ~124 | AVX-512 |
| MamaBear | $2^{49} - 2^{34} + 1$ | 3 | ~147 | AVX-512IFMA |

## 6.4 High-Throughput Merkle Tree

Merkle tree construction constitutes a non-trivial portion of the commit-phase overhead. We use the BLAKE3 library's AVX-512 backend, which hashes 16 independent inputs in parallel using one invocation. In our representation, each Merkle leaf stores 16 field elements contiguously, occupying 128 bytes in memory[3]. The pair-major blocked layout from §6.1 guarantees this contiguity and therefore avoids gather overhead.

# 7 Evaluation

## 7.1 Security Parameters and Baselines

*Shared security parameters.* Following the security analyses in prior work [9, 13, 21, 29, 30], we do not re-optimize the full HyperPlonk–DeepFold parameter space separately for each backend. Instead, we compare backends under a shared parameter regime and focus on the monotone effect of the field size: with all other parameters fixed, a larger field yields a smaller soundness error and therefore a larger security margin. Under the configurations in Table 3, MamaBear over $\mathbb{F}_{p^3}$ has field size about 147 bits, which is about 19 bits larger than Goldilocks over $\mathbb{F}_{p^2}$ and about 23 bits larger than BabyBear over $\mathbb{F}_{p^4}$. Across all evaluations we fix the DeepFold FRI code rate to 1/8 and hold the FRI query path at a common target: MamaBear uses 88 queries plus 16 grinding bits, while the Goldilocks and BabyBear baselines use 101 queries and no grinding, that codebase having no grinding step. At the rate-driven yield of ≈1.28 bits per query the two settings agree to within one bit (128.4 against 129.0).

*Goldilocks and BabyBear baselines.* Goldilocks follows the DeepFold [29] implementation[4] over $\mathbb{F}_{p^2}$. Our BabyBear baseline is obtained by replacing only the field backend in that same HyperPlonk–DeepFold codebase with Plonky3's BabyBear $\mathbb{F}_{p^4}$ arithmetic [47]. We include Goldilocks and BabyBear together because they represent the existing small-field design route under a common prover stack: Goldilocks as the widely used 64-bit small-prime baseline, and BabyBear as a smaller-prime alternative. Keeping the PIOP and PCS fixed while changing only the field backend isolates the effect of field choice more cleanly. Neither the Goldilocks nor the BabyBear implementation in this codebase provides multithreading support.

[3] Our in-memory representation stores each field element in an 8-byte word to preserve AVX-512-friendly alignment. By contrast, proof-size accounting uses the packed on-wire size of 7 bytes per field element.

[4] https://github.com/paulguoyanpei/DeepFold-Hyperplonk at commit 4148026.

**Table 4: ZeroCheck and ProductCheck time across circuit sizes $2^\mu$ for $\mu = 18$ to 28. Goldilocks and BabyBear serve as baselines. "M. vs. G./B." reports the MamaBear speedup over Goldilocks / BabyBear. "-Par" columns use 8 threads. "–" indicates data not reported due to out-of-memory (OOM).**

**(a) ZeroCheck time over $\mathbb{F}_{p^3}$ with $\ell_0$=2 (§5.3).**

| $\mu$ | Gold. $\mathbb{F}_{p^2}$ | Baby. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|
| | | | Time | M. vs. G./B. | Time | M. vs. G./B. |
| 18 | 0.23 s | 0.20 s | 5.5 ms | 42x/36x | 4.1 ms | 56x/49x |
| 19 | 0.46 s | 0.40 s | 11.6 ms | 40x/34x | 7.4 ms | 62x/54x |
| 20 | 0.91 s | 0.80 s | 23.8 ms | 38x/34x | 14.3 ms | 64x/56x |
| 21 | 1.64 s | 1.41 s | 48.3 ms | 34x/29x | 31.1 ms | 53x/45x |
| 22 | 3.36 s | 2.81 s | 0.12 s | 28x/23x | 83.8 ms | 40x/34x |
| 23 | 6.56 s | 5.66 s | 0.24 s | 27x/24x | 0.17 s | 39x/33x |
| 24 | 13.27 s | 11.46 s | 0.46 s | 29x/25x | 0.32 s | 41x/36x |
| 25 | 26.41 s | 22.52 s | 0.98 s | 27x/23x | 0.57 s | 46x/40x |
| 26 | 52.74 s | 45.72 s | 1.93 s | 27x/24x | 1.12 s | 47x/41x |
| 27 | 105.8 s | 91.00 s | 3.82 s | 28x/24x | 2.30 s | 46x/40x |
| 28 | 210.8 s | 183.8 s | 7.71 s | 27x/24x | 4.46 s | 47x/41x |

**(b) ProductCheck time using the per-wire design (§5.4) over $\mathbb{F}_{p^3}$.**

| $\mu$ | Gold. $\mathbb{F}_{p^2}$ | Baby. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|
| | | | Time | M. vs. G./B. | Time | M. vs. G./B. |
| 18 | 0.58 s | 0.49 s | 27.8 ms | 21x/18x | 18.3 ms | 32x/27x |
| 19 | 1.14 s | 1.01 s | 35.6 ms | 32x/28x | 24.3 ms | 47x/42x |
| 20 | 2.33 s | 2.01 s | 71.1 ms | 33x/28x | 63.3 ms | 37x/32x |
| 21 | 4.66 s | 4.17 s | 0.24 s | 19x/17x | 0.16 s | 29x/26x |
| 22 | 9.50 s | 8.24 s | 0.52 s | 18x/16x | 0.34 s | 28x/24x |
| 23 | 18.64 s | 15.91 s | 1.12 s | 17x/14x | 0.69 s | 27x/23x |
| 24 | 38.41 s | 34.07 s | 2.19 s | 18x/16x | 1.32 s | 29x/26x |
| 25 | 77.30 s | 67.21 s | 4.35 s | 18x/15x | 2.57 s | 30x/26x |
| 26 | 149.8 s | 132.5 s | 8.88 s | 17x/15x | 5.12 s | 29x/26x |
| 27 | 304.1 s | 266.0 s | 17.68 s | 17x/15x | 10.16 s | 30x/26x |
| 28 | – | – | 35.13 s | –/– | 20.42 s | –/– |

*Plonky3 baseline.* We report Plonky3 [47] separately only for end-to-end proving. For this evaluation, we use Plonky3 with its AVX-512 BabyBear $\mathbb{F}_{p^4}$ backend. Its PCS is similar to ours because both systems are FRI-based [10, 16], but its PIOP is different: Plonky3 uses a univariate AIR-STARK formulation [11] rather than our multilinear HyperPlonk PIOP. To reduce the influence of this PIOP mismatch, we add a HyperPlonk-style degree-3-gate AIR benchmark that matches our add/mul workload more closely. Unlike the Goldilocks and BabyBear HyperPlonk–DeepFold baselines above, Plonky3 provides multithreaded execution, so we report both single-threaded and parallel Plonky3 results. Pairing/KZG-based stacks (Espresso HyperPlonk[5] over BLS12-381 [7, 20] with multilinear KZG [34, 46]) and binary-tower systems such as Binius [24] occupy substantially different design spaces and are therefore not directly comparable.

## 7.2 Experimental Setup

*Hardware.* All experiments run on a Google Cloud instance `c4d-highmem-16` with an AMD EPYC 9B45 processor, 8 physical cores (16 logical CPUs), and 123 GiB memory.

*Software.* Our implementation is built upon the DeepFold [29] codebase, written in Rust, and evaluated on Ubuntu 24.04.3 LTS with `rustc 1.98.0-nightly`. We compile with `RUSTFLAGS="-C target-cpu=native"` to enable AVX-512IFMA.

*Circuit sizes.* All reported evaluations use size-$2^\mu$ random circuits composed of addition and multiplication gates, with $\mu$ ranging from 18 to 28 for the ZeroCheck and ProductCheck components and from 18 to 27 for DeepFold and the end-to-end prover. Entries marked '–' indicate configurations that were not reported, typically due to memory constraints; the Goldilocks and BabyBear baselines stop earlier than MamaBear for that reason. While our framework supports SHA256, AES128, BLAKE3, Keccak-$f$[1600], and Poseidon2 (see gate counts in Table 10 of Appendix L), we omit individual performance metrics for these primitives as prover latency is determined solely by the total circuit size. Incorporating lookup arguments and custom gates—which we leave for future work—is expected to reduce these circuit sizes substantially.

[5] https://github.com/EspressoSystems/hyperplonk.

## 7.3 Performance and Comparison

Unless stated otherwise, multithreaded MamaBear results are compared against the single-threaded baselines available in our codebase, and against multithreaded Plonky3 when parallel data is available; the latter is therefore the most directly comparable parallel point.

*Selective multithreading policy.* Thread-level parallelism in MamaBearZKP is intentionally selective rather than uniform. Many packed scans already drive a single AVX-512IFMA core close to the effective DRAM roofline, so we enable parallel execution only when the task granularity is large enough to amortize scheduling overhead and the per-task working set remains cache-resident. Protocol semantics impose the second constraint: sum-check and FRI folds remain serial across transcript rounds because each round depends on the previous Fiat–Shamir challenge, so parallelism appears only within round-local scans or across naturally independent jobs such as ProductCheck wire-local trees, split-FFT subproblems, and Merkle hashing. Appendix K records the main negative cases and threshold intuition.

Before turning to protocol-level timings, note that the arithmetic layer already favors MamaBear. Table 9 in Appendix L shows clear microbenchmark gains, so the larger component-level speedups below are not due to protocol restructuring alone.

*ZeroCheck.* Table 4a reports ZeroCheck performance. Our MamaBear implementation achieves **27–42**× speedup over Goldilocks in single-threaded mode, and **39–64**× with 8-thread parallelism.

**Table 5: DeepFold performance across circuit sizes $2^\mu$ for $\mu = 18$ to 27. Goldilocks and BabyBear serve as baselines. "M. vs. G./B." reports the MamaBear speedup over Goldilocks / BabyBear. "-Par" columns use 8 threads. Our MamaBear implementations use 5-level splitting FFT. "–" indicates data not reported due to out-of-memory (OOM).**

**(a) DeepFold-Commit time.**

| $\mu$ | Gold. $\mathbb{F}_{p^2}$ | Baby. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|
| | | | Time | M. vs. G./B. | Time | M. vs. G./B. |
| 18 | 0.65 s | 0.64 s | 45.1 ms | 14x/14x | 8.6 ms | 76x/74x |
| 19 | 1.46 s | 1.37 s | 0.10 s | 15x/14x | 18.1 ms | 81x/76x |
| 20 | 2.93 s | 2.83 s | 0.23 s | 13x/12x | 40.2 ms | 73x/70x |
| 21 | 6.08 s | 5.84 s | 0.51 s | 12x/11x | 0.10 s | 61x/58x |
| 22 | 12.23 s | 11.94 s | 1.19 s | 10x/10x | 0.23 s | 53x/52x |
| 23 | 24.82 s | 24.38 s | 3.19 s | 8x/8x | 0.53 s | 47x/46x |
| 24 | 50.32 s | 49.80 s | 7.50 s | 7x/7x | 1.18 s | 43x/42x |
| 25 | – | – | 17.01 s | –/– | 2.95 s | –/– |
| 26 | – | – | 40.03 s | –/– | 7.18 s | –/– |
| 27 | – | – | 92.34 s | –/– | 17.81 s | –/– |

**(b) DeepFold-Open time.**

| $\mu$ | Gold. $\mathbb{F}_{p^2}$ | Baby. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|
| | | | Time | M. vs. G./B. | Time | M. vs. G./B. |
| 18 | 0.59 s | 0.52 s | 31.5 ms | 19x/17x | 17.7 ms | 33x/29x |
| 19 | 1.27 s | 1.10 s | 63.9 ms | 20x/17x | 34.5 ms | 37x/32x |
| 20 | 2.70 s | 2.47 s | 0.13 s | 21x/19x | 68.8 ms | 39x/36x |
| 21 | 5.47 s | 4.96 s | 0.27 s | 20x/18x | 0.13 s | 42x/38x |
| 22 | 11.20 s | 10.21 s | 0.58 s | 19x/18x | 0.25 s | 45x/41x |
| 23 | 22.49 s | 20.46 s | 1.17 s | 19x/17x | 0.50 s | 45x/41x |
| 24 | 45.21 s | 40.88 s | 2.54 s | 18x/16x | 1.01 s | 45x/40x |
| 25 | – | – | 5.39 s | –/– | 2.10 s | –/– |
| 26 | – | – | 11.11 s | –/– | 4.16 s | –/– |
| 27 | – | – | 22.38 s | –/– | 8.37 s | –/– |

**Table 6: HyperPlonk–DeepFold prover time across circuit sizes $2^\mu$ for $\mu = 18$ to 27. Goldilocks, BabyBear, and Plonky3 serve as baselines. "M. vs. G./B./P." reports the MamaBear speedup over Goldilocks / BabyBear / Plonky3. "-Par" columns use 8 threads. Our MamaBear implementations use 5-level splitting FFT. "-" indicates data not reported due to out-of-memory (OOM).**

| $\mu$ | Goldilocks $\mathbb{F}_{p^2}$ | BabyBear $\mathbb{F}_{p^4}$ | Plonky3 B. $\mathbb{F}_{p^4}$ | Plonky3-Par B. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Time | M. vs. G./B./P. | Time | M. vs. G./B./P. |
| 18 | 2.07 s | 1.82 s | 1.82 s | 0.33 s | 0.10 s | 21x/18x/18x | 49.1 ms | 42x/37x/7x |
| 19 | 4.34 s | 3.88 s | 3.66 s | 0.67 s | 0.26 s | 17x/15x/14x | 97.3 ms | 45x/40x/7x |
| 20 | 8.84 s | 8.07 s | 7.47 s | 1.35 s | 0.52 s | 17x/16x/14x | 0.20 s | 44x/40x/7x |
| 21 | 18.61 s | 16.60 s | 15.48 s | 2.64 s | 1.14 s | 16x/15x/14x | 0.44 s | 42x/38x/6x |
| 22 | 37.30 s | 33.86 s | 30.27 s | 5.84 s | 2.72 s | 14x/12x/11x | 0.96 s | 39x/35x/6x |
| 23 | 75.49 s | 68.64 s | 61.86 s | 11.82 s | 6.37 s | 12x/11x/10x | 1.98 s | 38x/35x/6x |
| 24 | 152.8 s | 138.0 s | 128.3 s | 27.20 s | 14.37 s | 11x/10x/9x | 4.14 s | 37x/33x/7x |
| 25 | – | – | – | – | 31.76 s | –/–/– | 8.71 s | –/–/– |
| 26 | – | – | – | – | 68.77 s | –/–/– | 19.10 s | –/–/– |
| 27 | – | – | – | – | 144.1 s | –/–/– | – | –/–/– |

*ProductCheck.* Table 4b reports ProductCheck performance. Our per-wire ProductCheck over $\mathbb{F}_{p^3}$ achieves **17–33**× (serial) and **27–47**× (parallel) speedup over Goldilocks.

*DeepFold Commit.* Table 5a reports the commit performance. The commit phase (FFT + Merkle tree) shows serial speedups of **7–15**× and parallel speedups of **43–81**× over Goldilocks. The large parallel speedup reflects the high parallelism available in both the split-FFT (up to $2^t$ independent sub-FFTs) and Merkle tree construction.

*DeepFold Open.* Table 5b reports the DeepFold open performance. The open phase achieves serial speedups of **18–21**× and parallel speedups of **33–45**× over Goldilocks.

Taken together, these component-level gains reflect the co-design story of MamaBearZKP. The stay-packed pipeline of §4 removes repeated scalar/packed conversion, the protocol restructurings of §5 and §6 turn arithmetic advantages into cheaper kernels, and the selective multithreading policy above adds parallel throughput where the dependency structure permits. The speedups therefore come from co-design across representation, protocol structure, and implementation rather than from any single optimization.

Under single-threaded execution, ZeroCheck benefits the most. This is again a co-design effect: unlike ProductCheck, it starts from the base-field tables $S$, $L$, $R$, and $O$, so the first $\ell_0$ rounds of §5.3 remain on the small-value grid in $\mathbb{F}_p$ before the transition to $\mathbb{F}_{p^3}$. ProductCheck instead operates over extension-field values from the start and cannot exploit the same cheap base-field prefix.

Under multithreaded execution, the DeepFold components benefit more, especially in the commit phase. Here the protocol and implementation layers align particularly well: FFTs have higher arithmetic intensity than the streaming $O(N)$ sum-check scans, and

the split-FFT trick of §6.3 and Appendix J.3 exposes $2^t$ independent sub-FFTs. By contrast, sum-check remains constrained by round-to-round transcript dependencies, so its parallelism is largely limited to round-local scans.

*End-to-end proving.* Table 6 reports the full HyperPlonk–DeepFold prover time. Our single-threaded MamaBear prover achieves **11–21**× speedup over Goldilocks and **9–18**× over Plonky3. **Parallel MamaBear** (8 threads) achieves **37–45**× speedup over Goldilocks and **6–7**× over parallel Plonky3.

These end-to-end gains are the system-level manifestation of that co-design. By preserving packed dataflow across HyperPlonk rounds, batched opening, and DeepFold kernels, the stay-packed pipeline of §4 prevents scalar/packed conversion overhead from reappearing at component boundaries, allowing the component-level gains above to carry through to the full prover with little loss.

*Verifier, proof size, and memory.* Our verifier executes within 0.9–3.3 ms across the whole sweep, comparable to the Goldilocks and BabyBear baselines, while the Plonky3 verifier ranges from 11–18 ms. Proof-size comparisons (Table 11 in Appendix L) reflect our 5-level splitting FFT (§6.3): the MamaBear proof is about 3.3%–12.1% larger than Goldilocks at $\mu \in \{18, 19, 20\}$ but 2.9%–9.0% *smaller* at $\mu \in [21, 24]$, an advantage that grows with circuit size as analyzed in §J.3. For comparison, Appendix L also reports a 3-level split variant (Tables 12 and 13). It still achieves up to 39× proving-time speedup over Goldilocks in parallel (16× serial) while making the proof 21.0%–24.2% smaller over $\mu \in [18, 24]$. The stay-packed pipeline of §4 also lowers peak memory by about 4× relative to Goldilocks and 6× relative to Plonky3 (Table 14 in Appendix L); on our 123 GiB machine the other baselines run out of memory for $\mu \geq 25$, whereas MamaBearZKP still completes proving at $\mu = 27$. Those memory figures are the prover's own peak, taken from a dedicated binary that proves once. The benchmark harness needs more at the same $\mu$, since it holds the proving key and the witness across its timed samples, peaking near 110 GiB at $\mu = 27$; within the harness only the serial 5-level-split cell therefore fits at that size, which is why the remaining $\mu = 27$ entries of Table 6 are '–'.

## 8 Related Work

*Sum-check and multilinear proving.* HyperPlonk [21], Spartan [50], Hyrax [58], and the folding line Nova/HyperNova [35, 36] all live in the multilinear world, but they expose different bottlenecks: Hyrax and Nova-family systems are shaped more by group-based commitments and recursive compression, whereas HyperPlonk exposes long eq-weighted scans and a PCS boundary that are highly sensitive to field arithmetic and memory layout. On the algorithmic side, Gruen [28] and Twist and Shout [51] peel fixed equality factors out of the online round target, BDDT25 [5] combines prefix/suffix factorization with small-value rounds, and small-characteristic variants study binary-tower or other low-bit settings [6, 23, 24]. Lookup-oriented systems such as Lasso and Jolt [4, 49] reuse the same sum-check substrate for zkVM-style workloads, while Thaler's streaming line [55] optimizes memory rather than packed SIMD saturation. MamaBearZKP builds on this line, but at a different design-space point: we stay inside a transparent HyperPlonk–DeepFold stack and co-design the small-prime field, packed layout, and prover kernels together.

*Transparent FRI/STARK systems.* Within transparent arguments, Aurora and Virgo [15, 64] helped establish the post-quantum, hash-based line for R1CS and layered circuits, while Orion and Brakedown [27, 60] pushed the linear-prover, code-based direction further. Within transparent polynomial commitments, DeepFold and BaseFold [29, 63] are the closest multilinear baselines, while eth-STARK [9], Plonky2/3 [1, 47], RISC Zero [48], SP1 [53], Circle-STARK-based Stwo [31, 52], and Bitcoin-oriented ZeroSync [39] represent the broader engineering landscape over Goldilocks, BabyBear, Mersenne31, or Bitcoin-specific STARK deployments. Recent Reed–Solomon IOPP refinements such as STIR and WHIR [2, 3], together with adjacent FRI-derived constructions such as FRIDA [32] and code-based descendants [27, 37], improve query complexity, verifier work, availability proofs, or concrete prover throughput by changing the low-degree testing or code family itself. Our comparison point is different: rather than proposing a new RS-IOPP or a new transparent PCS, MamaBearZKP keeps the multilinear HyperPlonk–DeepFold architecture fixed and asks how far CPU-side co-design can go once the field/backend is matched to AVX-512IFMA.

*Pairing-based, Plonkish, and folding-based systems.* Marlin [22], Plonk and KZG [25, 34], Halo Infinite and the Halo2 ecosystem [18, 62], and related pairing- or IPA-based systems occupy a different point in the design space centered on univariate polynomials, universal setup or recursive accumulation, and larger scalar fields. The aggregation-oriented deployments such as SnarkPack [26] further optimize the verification footprint of Groth16-style proofs rather than the small-prime prover kernels targeted here. Conversely, multilinear commitments such as Hyrax [58], folding systems such as Nova and HyperNova [35, 36], and lattice-based PCS lines such as Greyhound [45] are closer on one axis of our problem setting but still optimize different resources: recursion, commitment algebra, or post-quantum lattice structure rather than SIMD-oriented field/layout co-design inside a transparent FRI stack. MamaBearZKP therefore complements this line rather than competing on the same proof-system objective.

*Hardware acceleration and prover engineering.* A large body of work accelerates fixed proving backends on particular devices. Intel HEXL targets AVX-512 polynomial-arithmetic kernels on CPUs [17]; cuZK, PipeZK, and related GPU/FPGA efforts target MSMs, Merkle trees, or end-to-end pipelines on specialized hardware [40, 65]; and DIZK [59] distributes proof generation across compute clusters. Engineering platforms such as ZPrize and Supranational's sppark push open benchmarks and reusable GPU primitives for ZK workloads [54, 66], while recent SIMD/MSM or code-based implementations refine specific transparent-proof kernels further [33, 37]. This line is complementary but not identical to ours: most prior work treats the field and protocol stack as fixed and then optimizes the kernels that remain, or moves proving to clusters and accelerators. MamaBearZKP instead moves one layer earlier in the stack by co-designing the prime field, arithmetic representation, protocol structure, and packed CPU dataflow around commodity AVX-512IFMA processors.

## 9 Conclusion

We presented MamaBearZKP, a co-designed framework for high-throughput zero-knowledge proving on AVX-512IFMA CPUs that combines the MamaBear prime field ($p = 2^{49} - 2^{34} + 1$) with packed prover kernels and data layouts jointly tuned to field arithmetic, protocol structure, and SIMD implementation. The end-to-end speedups arise from this cross-layer dependency: up to 21× over Goldilocks in single-threaded mode, 45× over Goldilocks with 8 threads, and 18× over Plonky3. The 49-bit field leaves enough room for lazy reduction under the AVX-512IFMA operand limit; that arithmetic regime supports fused packed kernels across the prover's critical scans; those kernels justify keeping the dataflow packed across HyperPlonk and DeepFold; and only this end-to-end packed execution converts hardware throughput into prover-level speedups.

*Adaptability and future work.* The same coupling generalizes beyond AVX-512IFMA and HyperPlonk–DeepFold: it should transfer to emerging SIMD instruction sets (ARM SVE2, RISC-V Vector) and to other proving stacks, including Spartan [50], STIR [2]/WHIR [3], and Brakedown [27]; stack-specific steps such as our per-wire ProductCheck remain protocol dependent. Extending the system beyond addition and multiplication to richer custom-gate support remains future work.

## Acknowledgments

This research is supported in part by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award MOE-T2EP20125-0015); by the National Research Foundation, Singapore, under the NRF Fellowship and under its CyberSG R&D Programme; and by the National Natural Science Foundation of China (Grant No. 62502275).

## References

[1] 0xPolygonZero. 2021. Plonky2. GitHub repository. https://github.com/0xPolygonZero/plonky2 Deprecated; superseded by Plonky3. Accessed 2026-04-30.

[2] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. 2024. STIR: Reed-Solomon Proximity Testing with Fewer Queries. In *Advances in Cryptology - CRYPTO 2024*. Springer, Cham, Switzerland, 380–413. https://eprint.iacr.org/2024/390

[3] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. 2025. WHIR: Reed-Solomon Proximity Testing with Super-Fast Verification. In *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV (Lecture Notes in Computer Science)*, Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, 214–243. doi:10.1007/978-3-031-91134-7_8

[4] Arasu Arun, Srinath Setty, and Justin Thaler. 2024. Jolt: SNARKs for Virtual Machines via Lookups. In *Advances in Cryptology - EUROCRYPT 2024*. Springer, Cham, Switzerland, 3–33. doi:10.1007/978-3-031-58751-1_1

[5] Suyash Bagad, Quang Dao, Yuval Domb, and Justin Thaler. 2025. Speeding Up Sum-Check Proving. *IACR Cryptology ePrint Archive* 2025 (2025), 1117. https://eprint.iacr.org/2025/1117

[6] Suyash Bagad, Yuval Domb, and Justin Thaler. 2024. The Sum-Check Protocol over Fields of Small Characteristic. *IACR Cryptology ePrint Archive* 2024 (2024), 1046. https://eprint.iacr.org/2024/1046

[7] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. 2002. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *Security in Communication Networks, Third International Conference, SCN 2002, Amalfi, Italy, September 11-13, 2002. Revised Papers (Lecture Notes in Computer Science)*, Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano (Eds.). Springer, 257–267. doi:10.1007/3-540-36413-7_19

[8] Paulo S. L. M. Barreto and Michael Naehrig. 2005. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Bart Preneel and Stafford E. Tavares (Eds.). Springer, 319–331. doi:10.1007/11693383_22

[9] Eli Ben-Sasson. 2021. ethSTARK Documentation. *IACR Cryptology ePrint Archive* 2021 (2021), 582. https://eprint.iacr.org/2021/582

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018) (LIPIcs, Vol. 107)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:17. doi:10.4230/LIPIcs.ICALP.2018.14

[11] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046

[12] Eli Ben-Sasson, Dan Carmon, Ulrich Haböck, Swastik Kopparty, and Shubhangi Saraf. 2026. On Proximity Gaps of Reed-Solomon Codes. In *Proceedings of the 58th Annual ACM Symposium on Theory of Computing, STOC 2026, Salt Lake City, UT, USA, June 22-26, 2026*, Aditya Bhaskara and Artur Czumaj (Eds.). ACM, 1157–1167. doi:10.1145/3798129.3800827

[13] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. 2020. Proximity Gaps for Reed-Solomon Codes. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, Sandy Irani (Ed.). IEEE, 900–909. doi:10.1109/FOCS46700.2020.00088

[14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 459–474.

[15] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, 103–128. doi:10.1007/978-3-030-17653-2_4

[16] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. 2020. DEEP-FRI: Sampling Outside the Box Improves Soundness. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, Seattle, Washington, USA, January 12-14, 2020 (LIPIcs)*, Thomas Vidick (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:32. doi:10.4230/LIPIcs.ITCS.2020.5

[17] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. 2021. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, New York, NY, USA, 57–62. doi:10.1145/3474366.3486926

[18] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. 2021. Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I (Lecture Notes in Computer Science)*, Tal Malkin and Chris Peikert (Eds.). Springer, 649–680. doi:10.1007/978-3-030-84242-0_23

[19] Joppe W. Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, Piscataway, NJ, USA, 353–367. doi:10.1109/EuroSP.2018.00032

[20] Sean Bowe. 2017. BLS12-381: New zk-SNARK Elliptic Curve Construction. Zcash Blog. https://web.archive.org/web/2017/https://electriccoin.co/blog/new-snark-curve/.

[21] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2023. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. Springer, 499–530. doi:10.1007/978-3-031-30617-4_17

[22] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. 2020. 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 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, 738–768. doi:10.1007/978-3-030-45721-1_26

[23] Benjamin E. Diamond and Jim Posen. 2024. Polylogarithmic Proofs for Multilinears over Binary Towers. *IACR Cryptology ePrint Archive* 2024 (2024), 504. https://eprint.iacr.org/2024/504

[24] Benjamin E. Diamond and Jim Posen. 2025. Succinct Arguments over Towers of Binary Fields. In *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV (Lecture Notes in Computer Science)*. Springer, 93–122. doi:10.1007/978-3-031-91134-7_4

[25] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge. *IACR Cryptology ePrint Archive* 2019 (2019), 953. https://eprint.

iacr.org/2019/953

[26] Nicolas Gailly, Mary Maller, and Anca Nitulescu. 2022. SnarkPack: Practical SNARK Aggregation. In *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers (Lecture Notes in Computer Science)*, Ittay Eyal and Juan A. Garay (Eds.). Springer, 203–229. doi:10.1007/978-3-031-18283-9_10

[27] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. 2023. Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II (Lecture Notes in Computer Science)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, 193–226. doi:10.1007/978-3-031-38545-2_7

[28] Angus Gruen. 2024. Some Improvements for the PIOP for ZeroCheck. *IACR Cryptol. ePrint Arch.* 2024 (2024), 108. https://eprint.iacr.org/2024/108

[29] Yanpei Guo, Xuanming Liu, Kexi Huang, Wenjie Qu, Tianyang Tao, and Jiaheng Zhang. 2025. DeepFold: Efficient Multilinear Polynomial Commitment from Reed-Solomon Code and Its Application to Zero-knowledge Proofs. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 3497–3516. https://www.usenix.org/conference/usenixsecurity25/presentation/guo-yanpei

[30] Ulrich Haböck. 2022. A Summary on the FRI Low Degree Test. *IACR Cryptology ePrint Archive* 2022 (2022), 1216. https://eprint.iacr.org/2022/1216

[31] Ulrich Haböck, David Levit, and Shahar Papini. 2024. Circle STARKs. *IACR Cryptology ePrint Archive* 2024 (2024), 278. https://eprint.iacr.org/2024/278

[32] Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. 2024. FRIDA: Data Availability Sampling from FRI. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VI (Lecture Notes in Computer Science)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, 289–324. doi:10.1007/978-3-031-68391-6_9

[33] Rui Jiang, Cong Peng, Min Luo, Rongmao Chen, and Debiao He. 2025. SimdMSM: SIMD-accelerated Multi-Scalar Multiplication Framework for zkSNARKs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2025, 2 (2025), 681–704. doi:10.46586/TCHES.V2025.I2.681-704

[34] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology – ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 177–194. doi:10.1007/978-3-642-17373-8_11

[35] Abhiram Kothapalli and Srinath Setty. 2024. HyperNova: Recursive Arguments for Customizable Constraint Systems. In *Advances in Cryptology - CRYPTO 2024*. Springer, Cham, Switzerland, 345–379. https://eprint.iacr.org/2023/573

[36] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV (Lecture Notes in Computer Science)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 359–388. doi:10.1007/978-3-031-15985-5_13

[37] Florian Krieger, Christian Dobrouschek, Florian Hirner, and Sujoy Sinha Roy. 2026. High-Performance SIMD Software for Spielman Codes in Zero-Knowledge Proofs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2026, 2 (2026), 167–191.

[38] Weihan Li, Zongyang Zhang, Sherman S. M. Chow, Yanpei Guo, Boyuan Gao, Xuyang Song, Yi Deng, and Jianwei Liu. 2025. Shred-to-Shine Metamorphosis of (Distributed) Polynomial Commitments. *IACR Cryptology ePrint Archive* 2025 (2025), 1354. https://eprint.iacr.org/2025/1354 Published elsewhere. Minor revision. USENIX Security 2026.

[39] Robin Linus, Lukas George, Max Gillet, and Tino Steffens. 2022. ZeroSync - STARK proofs for Bitcoin. GitHub repository. https://github.com/ZeroSync/ZeroSync Project citation metadata from the official CITATION.cff; accessed 2026-04-30.

[40] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2023. cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 3 (2023), 194–220. doi:10.46586/TCHES.V2023.I3.194-220

[41] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic Methods for Interactive Proof Systems. *J. ACM* 39, 4 (1992), 859–868. doi:10.1145/146585.146605

[42] Peter L. Montgomery. 1985. Modular Multiplication Without Trial Division. *Math. Comp.* 44, 170 (1985), 519–521. doi:10.2307/2007970

[43] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication 204. National Institute of Standards and Technology. doi:10.6028/NIST.FIPS.204

[44] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Federal Information Processing Standards Publication 203. National Institute of Standards and Technology. doi:10.6028/NIST.FIPS.203

[45] Ngoc Khanh Nguyen and Gregor Seiler. 2024. Greyhound: Fast Polynomial Commitments from Lattices. *IACR Cryptology ePrint Archive* 2024 (2024), 1293. doi:10.1007/978-3-031-68403-6_8

[46] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. 2013. Signatures of Correct Computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings (Lecture Notes in Computer Science)*, Amit Sahai (Ed.). Springer, 222–242. doi:10.1007/978-3-642-36594-2_13

[47] Plonky3 Contributors. 2026. Plonky3. GitHub repository. https://github.com/Plonky3/Plonky3 Accessed 2026-04-01.

[48] RISC Zero. 2026. RISC Zero. GitHub repository. https://github.com/risc0/risc0 Accessed 2026-04-30.

[49] Srinath Setty, Justin Thaler, and Riad S. Wahby. 2024. Unlocking the Lookup Singularity with Lasso. In *Advances in Cryptology - EUROCRYPT 2024*. Springer, Cham, Switzerland, 180–209. doi:10.1007/978-3-031-58751-1_7

[50] Srinath T. V. Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III (Lecture Notes in Computer Science)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, 704–737. doi:10.1007/978-3-030-56877-1_25

[51] Srinath T. V. Setty and Justin Thaler. 2025. Twist and Shout: Faster memory checking arguments via one-hot addressing and increments. *IACR Cryptol. ePrint Arch.* 2025 (2025), 105. https://eprint.iacr.org/2025/105

[52] StarkWare. 2026. Stwo. GitHub repository. https://github.com/starkware-libs/stwo Accessed 2026-04-30.

[53] Succinct Labs. 2024. SP1: A Performant, 100% Open-Source, Contributor-Friendly zkVM. GitHub repository. https://github.com/succinctlabs/sp1 Accessed 2026-04-01.

[54] Supranational. 2026. sppark. GitHub repository. https://github.com/supranational/sppark Accessed 2026-04-30.

[55] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Annual Cryptology Conference (CRYPTO)*. Springer, 71–89. doi:10.1007/978-3-642-40084-1_5

[56] The PQ-Crystals Team. 2024. CRYSTALS-Dilithium AVX2 implementation. https://github.com/pq-crystals/dilithium/tree/master/avx2. Accessed: 2026-04-01. Specifically `ntt.S` and `invntt.S` routines..

[57] The PQ-Crystals Team. 2024. CRYSTALS-Kyber AVX2 implementation. https://github.com/pq-crystals/kyber/tree/main/avx2. Accessed: 2026-04-01. Specifically `ntt.S` and `invntt.S` routines..

[58] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-Efficient zkSNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 926–943. doi:10.1109/SP.2018.00060

[59] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 675–692. https://www.usenix.org/conference/usenixsecurity18/presentation/wu

[60] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. Orion: Zero Knowledge Proof with Linear Prover Time. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV (Lecture Notes in Computer Science)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 299–328. doi:10.1007/978-3-031-15985-5_11

[61] Guangwu Xu, Yiran Jia, and Yanze Yang. 2024. Chinese Remainder Theorem Approach to Montgomery-Type Algorithms. arXiv:2402.00675 [cs.CR] doi:10.48550/arXiv.2402.00675

[62] Zcash Foundation. 2026. The halo2 Book. Project documentation. https://zcash.github.io/halo2/ Accessed 2026-04-30.

[63] Hadas Zeilberger, Binyi Chen, and Ben Fisch. 2024. BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part X (Lecture Notes in Computer Science)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, 138–169. doi:10.1007/978-3-031-68403-6_5

[64] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 859–876. doi:10.1109/SP40000.2020.00052

[65] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. 2021. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 416–428. doi:10.1109/ISCA52012.2021.00040

[66] ZPrize. 2023. ZPrize. Competition website. https://www.zprize.io/ Accessed 2026-04-30.

## A Open Science

All artifacts needed to reproduce our experimental results are provided:

- **Source code**: The complete Rust implementation of this work is available at: https://github.com/Ji-Peng/MamaBearZKP-Artifact.
- **Benchmark suite**: The benchmarks for all components (ZeroCheck, ProductCheck, DeepFold Commit/Open) and end-to-end HyperPlonk–DeepFold proving, with scripts to reproduce the main evaluation results in this paper.
- **Reference platform**: To reproduce the reported numbers as closely as possible, use the same platform and software stack: Google Cloud `c4d-highmem-16`, AMD EPYC 9B45, 8 physical cores (16 logical CPUs), 123 GiB memory, Ubuntu 24.04.3 LTS, and `rustc 1.98.0-nightly`. On a machine with less memory, cap the sweep with `BENCH_NV_MAX` rather than adding swap: a prover whose working set is partly on disk measures the disk, so a run that touches swap is not comparable to the reported timings. Capped rows are simply absent and render as '–', exactly as the out-of-memory cells in our own tables do.
- **Baseline implementations**: Goldilocks and BabyBear reference implementations within the same framework, enabling direct comparison.
- **Hardware requirements**: An x86-64 processor with AVX-512IFMA support (Intel Ice Lake+ or AMD Zen 4+) is required; see https://en.wikipedia.org/wiki/AVX-512#CPUs_with_AVX-512 for the current list of supporting CPUs.

## B Generative AI Usage

GPT-5.4 was used only for limited editorial assistance during manuscript preparation. Its use was restricted to grammar correction, spelling correction, sentence-level rephrasing, and style polishing.

The tool was not used to generate substantive technical content, including proofs, algorithms, experimental results, citations, or scientific claims. All accepted edits were manually reviewed by the authors before incorporation. The technical content, implementation details, and reported measurements were independently checked against the source files, codebase, and experiment outputs.

## C Detailed HyperPlonk Description

This appendix records the fuller formal background for HyperPlonk that is only sketched in §2.3. We focus on the three pieces used in the paper: the ZeroCheck, the layered ProductCheck, and the final batch-opening step.

*ZeroCheck.* ZeroCheck verifies the relation

$$\mathcal{R}_{\mathsf{ZC}} = \left\{ h : h(\mathbf{b}) = 0 \ \forall\, \mathbf{b} \in \{0,1\}^{\mu} \right\}.$$

HyperPlonk reduces this to a sum-check on $f(\mathbf{X}) = \widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{X}) \cdot h(\mathbf{X})$ at a random verifier point $\mathbf{w} \in \mathbb{F}^{\mu}$, since the sum equals $\tilde{h}(\mathbf{w})$.

*ProductCheck relation.* ProductCheck verifies the relation

$$\mathcal{R}_{\mathsf{PC}} = \left\{ (f, g) : \prod_{\mathbf{x} \in \{0,1\}^m} f(\mathbf{x}) = \prod_{\mathbf{x} \in \{0,1\}^m} g(\mathbf{x}) \right\}.$$

HyperPlonk invokes ProductCheck on the permutation polynomials

$$f(\mathbf{x}) = \alpha + w(\mathbf{x}) + \beta \cdot \mathrm{id}(\mathbf{x}), \qquad g(\mathbf{x}) = \alpha + w(\mathbf{x}) + \beta \cdot \sigma(\mathbf{x}),$$

where $w$ is formed by concatenating the three wire polynomials $L, R, O$ (each of length $N = 2^{\mu}$) into a table of length $3N$ and zero-padding to $2^{\mu+2} = 4N$, so $m = \mu + 2$; id is the identity embedding over the padded domain; $\sigma$ is the wiring permutation extended to the padded domain; and $\alpha, \beta \in \mathbb{F}$ are verifier-sampled random challenges. Equal products imply $w \circ \sigma = w$ as multisets with high probability.

Following the implementation strategy[6] used by the DeepFold prover stack [29], we realize ProductCheck as a layered product argument over $\{0,1\}^{\mu+2}$: for each side $s \in \{f, g\}$ the prover builds an $(\mu+2)$-level binary product tree $T_j^s$ with $T_0^s \equiv s$ and $T_j^s[k] = T_{j-1}^s[2k] \cdot T_{j-1}^s[2k{+}1]$. Starting from the top-layer claim $T_{\mu+1}^f[0] \cdot T_{\mu+1}^f[1] = T_{\mu+1}^g[0] \cdot T_{\mu+1}^g[1]$, the prover and verifier run, level by level from $j = \mu$ down to 0, a degree-3 sum-check that reduces a multilinear-extension claim on $T_{j+1}^s$ to one on the even/odd halves $T_j^{s,\mathrm{e}}, T_j^{s,\mathrm{o}}$, with the verifier sampling fresh challenges to fold the resulting two-side claims to the next level. After level 0, the residual claims are evaluations of $f$ and $g$ at a random point and are discharged via PCS openings.

*Batch opening.* ZeroCheck and ProductCheck leave the prover with $N_q$ residual evaluation claims of the form $\tilde{q}_i(\mathbf{z}_i) = y_i$ for $i \in \{0, \ldots, N_q - 1\}$, where each $q_i$ is viewed as a $\mu$-variate multilinear polynomial and each query point satisfies $\mathbf{z}_i \in \mathbb{F}^{\mu}$. Let $\ell = \lceil \log_2 N_q \rceil$, and write $\langle i \rangle \in \{0,1\}^{\ell}$ for the binary encoding of the claim index $i$. The batching idea of HyperPlonk [21, Sec. 3.8] reduces these claims to one final sum-check. The verifier samples a random index point $\mathbf{t} \in \mathbb{F}^{\ell}$, and the prover forms a random linear combination over the claim indices,

$$G(\mathbf{X}) = \sum_{i=0}^{N_q - 1} \widetilde{\mathrm{eq}}(\mathbf{t}, \langle i \rangle) \cdot \tilde{q}_i(\mathbf{X}),$$

so that proving all claims is equivalent to checking

$$\sum_{i=0}^{N_q - 1} \widetilde{\mathrm{eq}}(\mathbf{t}, \langle i \rangle) \big( \tilde{q}_i(\mathbf{z}_i) - y_i \big) = 0.$$

This identity is arithmetized as a degree-2 sum-check using an equality table for the points $\mathbf{z}_i$, and the terminal claim is a single evaluation $G(\mathbf{r})$ at one verifier-sampled random point $\mathbf{r} \in \mathbb{F}^{\mu}$. Thus the residual openings left by ZeroCheck and ProductCheck can be discharged by one final PCS opening.

## D MamaBear Arithmetic Kernels

To keep the main text focused, we collect two core arithmetic kernels here. Throughout this appendix, pseudocode is written in a lane-wise form. Under the AVX-512 execution model, each variable denotes a ZMM register, and scalar operators such as $\gg$, $\ll$, &, +, and − act element-wise on lanes. Named operations such as `VMADD52L`, `VMADD52H`, and `VSUB64` map directly to the corresponding AVX-512 instructions.

[6] https://github.com/paulguoyanpei/DeepFold-Hyperplonk at commit 4148026.

**Algorithm 3:** Fast reduction for MamaBear field

**Constants:** $p = 2^{49} - 2^{34} + 1$
**Input:** $x \in [0, 2^{64})$
**Output:** $y \equiv x \pmod{p}$; see Lemma D.1 for the bound

```
x_hi ← x ≫ 49;
x_lo ← x & (2^49 − 1);
t ← (x_hi ≪ 34) − x_hi;                    // x_hi(2^34 − 1)
y ← x_lo + t;
return y;
```

Lemma D.1. *Algorithm 3 returns a value $y$ such that $y \equiv x \pmod{p}$ and, more generally, $y < 2^{49} + x/2^{15}$. In particular, for $x \in [0, 2^{64})$ we have $y \in [0, 2^{50})$.*

Proof. Write $x = x_{\mathrm{lo}} + x_{\mathrm{hi}} 2^{49}$ with $x_{\mathrm{lo}} = x \bmod 2^{49}$ and $x_{\mathrm{hi}} = \lfloor x/2^{49} \rfloor$. Since $p = 2^{49} - 2^{34} + 1$, we have $2^{49} \equiv 2^{34} - 1 \pmod{p}$. Therefore the algorithm output satisfies

$$y = x_{\mathrm{lo}} + x_{\mathrm{hi}}(2^{34} - 1) \equiv x \pmod{p}.$$

For the range, note that $x_{\mathrm{lo}} < 2^{49}$ and $x_{\mathrm{hi}}(2^{34}-1) < x_{\mathrm{hi}} 2^{34} \le x/2^{15}$, so $y < 2^{49} + x/2^{15}$. If $x < 2^{64}$, then $x_{\mathrm{hi}} \le 2^{15} - 1$, and hence

$$y \le (2^{49} - 1) + (2^{15} - 1)(2^{34} - 1) < 2^{50}.$$

This is the bound used by reduce_fast in the main text. □

The key micro-optimization in Algorithm 1 is the second line: instead of computing $\lfloor ab/R \rfloor$ and then adding $p$ as a separate correction step, we seed `VMADD52H` with $p$ and obtain $c_1 = \lfloor ab/R \rfloor + p$ directly. This makes the addition of $p$ for the non-negative correction effectively free in instruction count.

The range analysis below is scalar and applies independently to each SIMD lane.

Lemma D.2. *Let $c = ab$, and suppose $c < mp^2$ for some $m \ge 0$. Algorithm 1 returns a value $r$ such that $r \equiv cR^{-1} \pmod{p}$ and*

$$0 \le r < \left(\frac{mp}{R} + 1\right) p.$$

*In particular, for $a, b \in [0, R)$ we have $r \in [0, R + p)$.*

Proof. Write $c = c_1 R + c_0$ with $c_0 \in [0, R)$ and $c_1 = \lfloor c/R \rfloor$. Let $m_0 \equiv c_0 p^{-1} \pmod{R}$. Then $m_0 p - c_0$ is divisible by $R$, and the CRT formulation of Montgomery-type algorithms [42, 61] gives

$$cR^{-1} \equiv c_1 - \left\lfloor \frac{m_0 p}{R} \right\rfloor \pmod{p}.$$

Algorithm 1 computes exactly

$$r = c_1 + p - \left\lfloor \frac{m_0 p}{R} \right\rfloor,$$

because `VMADD52H` is initialized with $p$ and therefore folds the usual $+p$ correction into the high-half accumulation without a separate vector add. Hence $r \equiv cR^{-1} \pmod{p}$.

For the range, $m_0 < R$ implies $\lfloor m_0 p/R \rfloor < p$, so $r \ge 0$. Also $c < mp^2$ implies $c_1 < mp^2/R$, and therefore

$$r < \frac{mp^2}{R} + p = \left(\frac{mp}{R} + 1\right) p.$$

For the common case $a, b \in [0, R)$, we have $c \le (R-1)^2$, hence $c_1 = \lfloor c/R \rfloor \le R - 2$, and so

$$r < (R - 2) + p < R + p.$$

This proves the stated bounds. □

# E Correctness and Soundness of Fold Reordering

This appendix gives a formal account of the fold-reordering optimization introduced in §5.1.2. The optimization changes only the order in which the prover eliminates variables; the message format and the verifier's checks are identical to the standard description in §2.2. Throughout, $f : \mathbb{F}^\mu \to \mathbb{F}$ has degree at most $d$ in each of $x_0, \ldots, x_{\mu-1}$ separately, with claimed sum $C = \sum_{\mathbf{b} \in \{0,1\}^\mu} f(\mathbf{b})$; $d$ is therefore also the degree of each round polynomial, and the SIMD width is $W = 2^w$ (with $w = 3$ in our AVX-512IFMA setting). We do *not* assume $f$ multilinear: the argument below is a variable relabeling and is degree-agnostic, which is what lets §5 and the system-level soundness accounting invoke it for the ZeroCheck sum-check, where $f = \widetilde{\mathrm{eq}} \cdot h$ has per-variable degree 4.

## E.1 Reordering as a Variable Relabeling

Recall from §5.1.2 that, after reordering, the prover folds variables in the cyclically shifted order

$$x_w, x_{w+1}, \ldots, x_{\mu-1}, x_0, x_1, \ldots, x_{w-1}.$$

Encode this as the permutation $\pi$ on $\{0, \ldots, \mu - 1\}$ with

$$\pi(i) = \begin{cases} i + w, & 0 \le i \le \mu - w - 1, \\ i - (\mu - w), & \mu - w \le i \le \mu - 1, \end{cases}$$

so that round $i$ of the reordered protocol eliminates the original variable $x_{\pi(i)}$. Define the relabeled polynomial $g : \mathbb{F}^\mu \to \mathbb{F}$ by

$$g(y_0, y_1, \ldots, y_{\mu-1}) \;=\; f\big(y_{\pi^{-1}(0)}, y_{\pi^{-1}(1)}, \ldots, y_{\pi^{-1}(\mu-1)}\big). \tag{4}$$

Equivalently, $g$ is obtained from $f$ by renaming $x_{\pi(i)}$ as $y_i$. Since $\pi$ is a bijection on $\{0,1\}^\mu$,

$$\sum_{\mathbf{b} \in \{0,1\}^\mu} g(\mathbf{b}) \;=\; \sum_{\mathbf{b} \in \{0,1\}^\mu} f(\mathbf{b}) \;=\; C, \tag{5}$$

and $g$ obeys the same per-variable degree bound $d$, since $\pi$ only permutes the variable names. In our SIMD prover, the only change between standard and reordered execution is the bookkeeping-table layout. The prover code always folds along the least significant bit of the vector index, so rearranging the table from *vector-friendly* to *normal* order automatically realizes $\pi$ without any control-flow change. All dependent objects (equality polynomial, verifier point, PCS opening point) are reordered consistently in the same data move. Recall the standard sum-check soundness bound on $g$ (degree-$d$ rounds, $\mu$ variables) [41]:

$$\epsilon_{\mathrm{std}} \;\le\; \frac{d\mu}{|\mathbb{F}|}. \tag{6}$$

## E.2 Equivalence Lemma

Lemma E.1. *For every $i \in \{0, \ldots, \mu - 1\}$ and every challenge sequence $r_0, \ldots, r_{i-1} \in \mathbb{F}$, the round-$i$ message of the reordered protocol on $f$ equals the round-$i$ message of the standard sum-check protocol on*

*$g$ at the same challenges. The final-round PCS query $f(\rho_0, \ldots, \rho_{\mu-1})$, where $\rho_j = r_{\pi^{-1}(j)}$, equals $g(r_0, \ldots, r_{\mu-1})$.*

Proof. Apply the substitution $y_k = z^{(i)}_{\pi(k)}$ from (4): the reordered round-$i$ assignment ($r_k$ for already-folded slots, $X$ for the current slot, summation variables for unfolded slots) becomes the standard round-$i$ assignment on $g$. The Boolean specialization at the final round gives $g(r_0, \ldots, r_{\mu-1}) = f(\rho_0, \ldots, \rho_{\mu-1})$. □

## E.3 Completeness and Soundness

Theorem E.2 (Correctness). *For every $f : \mathbb{F}^\mu \to \mathbb{F}$ of per-variable degree at most $d$ and every claim $C = \sum_{\mathbf{b}} f(\mathbf{b})$, the reordered sum-check protocol of §5.1.2 is complete: an honest prover convinces the verifier with probability* 1.

Proof. Immediate from Lemma E.1 and the completeness of standard sum-check on $g$. □

Theorem E.3 (Soundness). *For every $f : \mathbb{F}^\mu \to \mathbb{F}$ of per-variable degree at most $d$, every false claim $C \neq \sum_{\mathbf{b}} f(\mathbf{b})$, and every cheating prover $\mathcal{A}$ in the reordered protocol,*

$$\Pr\big[\textit{verifier accepts}\big] \;\le\; \frac{d\mu}{|\mathbb{F}|} \;=\; \epsilon_{\mathrm{std}}.$$

Proof. By Lemma E.1, every transcript of the reordered protocol on $f$ is, round-for-round, a transcript of the standard sum-check protocol on $g$ at the same challenges. Since $C \neq \sum_{\mathbf{b}} g(\mathbf{b})$ by (5), $\mathcal{A}$'s acceptance probability is bounded by the standard soundness on $g$, which is (6). □

# F Eq-Poly Factoring Across ZeroCheck and ProductCheck

This appendix records only the implementation-side consequences of the factorization formulas stated in §5.2. We therefore do not restate the algebraic rewriting of $s_i(X) = c_i(X)t_i(X)$ or $\widetilde{\mathrm{eq}} = \widetilde{\mathrm{eq}}_L \widetilde{\mathrm{eq}}_R$ here. Instead, we focus on the storage, cache, and security implications of using this shared eq engine in both ZeroCheck and ProductCheck. Appendix G specializes the same plan to the BDDT25-style ZeroCheck transition, while Appendix H isolates the orthogonal gain from removing the padded fourth wire block in ProductCheck.

## F.1 Storage, Computation, and Cache Behavior

Let $N_i = 2^m$ denote the size of the residual Boolean cube in one round. Flat eq handling occupies $\Theta(N_i)$ field elements per round; suffix-factoring stores one right table of size $2^{\lfloor m/2 \rfloor}$ and one geometrically growing left family, totaling $\Theta(\sqrt{N_i} \log N_i)$ across the whole sum-check while the hot inner loop still performs one linear scan over gate data. Table 7 summarizes the comparison along storage, build cost, cache working set, and protocol effect.

This distinction is substantial at the problem scales used in our implementation. For $m = 24$ and $\mathbb{F}_{p^3}$ arithmetic (24 B per element), one flat eq table occupies $2^{24} \cdot 24$ B = 384 MiB. The factored family is about $12 \cdot 2^{12} \cdot 24$ B $\approx 1.125$ MiB, over two orders of magnitude smaller. More importantly, the inner-loop right table is only $2^{12} \cdot 24$ B = 96 KiB, which fits in the private L2 cache of the target CPUs and is often partly retained in L1. The inner scan therefore streams only the actual gate data from DRAM, while the eq weights are reused from cache.

**Table 7: Flat eq handling versus the factored eq plan for one eq-weighted sum-check instance with residual size $N_i = 2^m$.**

| Metric | Flat eq | Factored eq |
|---|---|---|
| Eq-table storage | $\Theta(N_i)$ | $\Theta(\sqrt{N_i} \log N_i)$ |
| Eq-table build/fold work | $\Theta(N_i)$ | $\Theta(\sqrt{N_i} \log N_i)$ |
| Hot-loop eq working set | $\Theta(N_i)$ | $\Theta(\sqrt{N_i})$ |
| Memory locality | DRAM streaming | cache-resident $\widetilde{\mathrm{eq}}_R$ |
| Protocol soundness | unchanged | unchanged |

The same two-stage structure is also SIMD-friendly. The right table can be broadcast lane-wise in the packed inner loop, while the left factor is applied once per outer group. This avoids the gather-like access pattern that a flat eq table would induce and meshes naturally with the MamaBear lazy-reduction discipline: the inner loop accumulates with `lazy_add`, and the outer loop reduces only when multiplying by the left factor.

## F.2 Protocol Effect and Security

Both factorizations are purely algebraic rewritings of the same sum. Prefix-factoring only separates a verifier-known linear factor $c_i(X)$ from the residual target $t_i(X)$. Suffix-factoring only changes the parenthesization of the residual sum by writing the same equality polynomial as $\widetilde{\mathrm{eq}}_L \cdot \widetilde{\mathrm{eq}}_R$. Neither step introduces new randomness, changes the queried point, or alters the proof transcript format on its own.

Accordingly, the soundness error of the underlying sum-check is unchanged. The verifier still checks the same relation, and the prover still commits to the same tables. Any additional soundness term comes from an orthogonal protocol change, such as the $\lambda$-batching used by per-wire ProductCheck; that separate effect is analyzed in Appendix H. The factored eq plan itself is therefore a prover-side optimization scheme, shared by ZeroCheck and ProductCheck, with no standalone security penalty.

# G BDDT25-Style ZeroCheck Details

This appendix expands the BDDT25-style ZeroCheck used in §5.3. It specializes the generic eq-factorization plan of §5.2 and Appendix F to the ZeroCheck gate, derives the degree-3 round target used by Algorithm 2, gives the matching verifier in Algorithm 4, and details the precomputation of the accumulator tables $A_i(\mathbf{v}, u)$.

## G.1 From the Round Polynomial to the Degree-3 Target

Applying the prefix-factoring of §5.2, recall the HyperPlonk ZeroCheck polynomial

$$f(\mathbf{X}) = \widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{X}) \cdot h(\mathbf{X}),$$

where $h$ is the gate polynomial from (1). In round $i$, the prover sends

$$s_i(X) = \sum_{\mathbf{x} \in \{0,1\}^{\mu-i-1}} f(\mathbf{r}_{<i}, X, \mathbf{x}).$$

Factoring out the equality terms that depend only on the fixed prefix gives

$$s_i(X) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<i}, \mathbf{r}_{<i}) \cdot \widetilde{\mathrm{eq}}(w_i, X) \sum_{\mathbf{x} \in \{0,1\}^{\mu-i-1}} \widetilde{\mathrm{eq}}(\mathbf{w}_{>i}, \mathbf{x}) \cdot h(\mathbf{r}_{<i}, X, \mathbf{x}). \tag{7}$$

Thus $s_i(X) = c_i(X) \cdot t_i(X)$ with

$$c_i(X) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<i}, \mathbf{r}_{<i}) \cdot \widetilde{\mathrm{eq}}(w_i, X)$$

and

$$t_i(X) = \sum_{\mathbf{x} \in \{0,1\}^{\mu-i-1}} \widetilde{\mathrm{eq}}(\mathbf{w}_{>i}, \mathbf{x}) \cdot h(\mathbf{r}_{<i}, X, \mathbf{x}). \tag{8}$$

Since $h$ is cubic in the witness values, $t_i$ has degree 3 even though $s_i$ has degree 4.

For the rounds where the split-equality optimization is active, namely $i < \lfloor \mu/2 \rfloor - 1$, apply the suffix-factoring of Appendix F and write the residual suffix as $\mathbf{x} = (\mathbf{x}_L, \mathbf{x}_R)$ with

$$\mathbf{x}_R \in \{0,1\}^{\lfloor \mu/2 \rfloor}, \qquad \mathbf{x}_L \in \{0,1\}^{\mu-i-1-\lfloor \mu/2 \rfloor},$$

and split the verifier point as

$$\mathbf{w}_R = (w_{i+1}, \ldots, w_{i+\lfloor \mu/2 \rfloor}), \qquad \mathbf{w}_L = (w_{i+\lfloor \mu/2 \rfloor+1}, \ldots, w_{\mu-1}).$$

Then (8) becomes

$$t_i(X) = \sum_{\mathbf{x}_L} \widetilde{\mathrm{eq}}_L(\mathbf{w}_L, \mathbf{x}_L) \sum_{\mathbf{x}_R} \widetilde{\mathrm{eq}}_R(\mathbf{w}_R, \mathbf{x}_R) h(\mathbf{r}_{<i}, X, \mathbf{x}_L, \mathbf{x}_R). \tag{9}$$

When $i \geq \lfloor \mu/2 \rfloor - 1$, the left block disappears and only the right equality table remains.

The correspondence with Algorithm 2 is direct. The first loop computes $t_i(u)$ for $u \in \widehat{U}_3$ by evaluating (9) through precomputed tables; the transition line materializes the packed extension-field witness tables needed to evaluate (8) after the verifier challenges leave the base field; and the final loop uses the same equation with the split equality tables and the fused fold kernel from §5.1.3. At the transition round $i = \ell_0$ each witness table $p_k \in \{S, L, R, O\}$ is partially folded over the first $\ell_0$ variables, and the round polynomial specializes to

$$s_{\ell_0}(u) = c_{\ell_0}(u) \cdot t_{\ell_0}(u), \qquad c_{\ell_0}(u) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<\ell_0}, \mathbf{r}_{<\ell_0}) \cdot \widetilde{\mathrm{eq}}(w_{\ell_0}, u), \tag{10}$$

with $t_{\ell_0}(u)$ given by (9) at $i = \ell_0$ on the folded tables; the partially folded values are reused across all $u \in \widehat{U}_3$.

## G.2 Verifier Algorithm

The two eq-factorizations of §5.2 reduce the natural ZeroCheck round polynomial from degree 4 to the degree-3 target $t_i$ of (8), and the round-sum identity removes one further transmitted value. The prover therefore sends only $|\widehat{U}_3| = 3$ values per round (Algorithm 2). Algorithm 4 is the matching verifier: each round it (i) reconstructs the missing value $s_i(1)$ from the running round identity, (ii) divides out the publicly computable eq-prefix $c_i(X)$ to expose $t_i$ on all four points of $U_3$, (iii) interpolates $t_i$ at the freshly sampled challenge $r_i$, and (iv) updates a single running eq-prefix $\eta$ that doubles as both the next-round factor and the closing $\widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{r})$ used in the final PCS check. Compared with the textbook degree-$d$ sum-check verifier of §2.2, the only changes are these recovery and interpolation steps; the per-round consistency check $s_i(0) + s_i(1) = y$ and the closing $f(\mathbf{r})$ test are structurally identical.

**Algorithm 4:** Verifier for the BDDT25-style ZeroCheck (companion to Algorithm 2)

**Input:** Challenge point $\mathbf{w} \in \mathbb{F}_{p^k}^{\mu}$ that defines the ZeroCheck reduction; PCS commitments to the witness tables $S, L, R, O$

**Output:** accept or reject

**Interaction:** `receive` means the verifier receives a round message from the prover; `send` means the verifier sends a fresh random challenge to the prover

Initialize the running claim $y \leftarrow 0$ and the running eq-prefix $\eta \leftarrow 1$;

**for** $i \leftarrow 0$ **to** $\mu - 1$ **do**

- `receive` $\{s_i(u)\}_{u \in \widehat{U}_3}$, where $\widehat{U}_3 = \{\infty, 0, 2\}$;
- Form $c_i(u) = \eta \cdot \widetilde{\mathrm{eq}}(w_i, u)$ for $u \in U_3$, where $\widetilde{\mathrm{eq}}(w_i, X) = (1-w_i) + (2w_i-1)X$ and $\widetilde{\mathrm{eq}}(w_i, \infty) = 2w_i-1$ is its leading coefficient under the BDDT25 convention;
- Reconstruct $s_i(1) = y - s_i(0)$ from the round identity $s_i(0) + s_i(1) = y$;
- Recover the degree-3 target $t_i(u) = s_i(u)/c_i(u)$ for $u \in U_3$, with $u = \infty$ interpreted as the ratio of leading coefficients; `// 4 values pin down` $t_i$
- `send` a fresh challenge $r_i \in \mathbb{F}_{p^k}$ to the prover;
- Interpolate $t_i(r_i)$ as the unique degree-3 univariate that matches $\{t_i(u)\}_{u \in U_3}$;
- Update $\eta \leftarrow \eta \cdot \widetilde{\mathrm{eq}}(w_i, r_i)$ and $y \leftarrow \eta \cdot t_i(r_i)$; `//` $y = s_i(r_i)$

`// Final consistency check via the PCS`

Open $S(\mathbf{r}), L(\mathbf{r}), R(\mathbf{r}), O(\mathbf{r})$ at $\mathbf{r} = (r_0, \ldots, r_{\mu-1})$, and form $h(\mathbf{r}) = \big(L(\mathbf{r}) + R(\mathbf{r})\big) + S(\mathbf{r}) \cdot \big(L(\mathbf{r})R(\mathbf{r}) - L(\mathbf{r}) - R(\mathbf{r})\big) - O(\mathbf{r})$ using the gate rewrite of §5.1.3;

**accept** iff $y = \eta \cdot h(\mathbf{r})$, equivalently $y = \widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{r}) \cdot h(\mathbf{r})$; otherwise **reject**;

*Why the running prefix is enough.* At the end of round $i$, the standard sum-check fold requires the next claim to be $s_i(r_i) = c_i(r_i) \cdot t_i(r_i)$. Since $c_i(r_i) = \widetilde{\mathrm{eq}}(\mathbf{w}_{<i}, \mathbf{r}_{<i}) \cdot \widetilde{\mathrm{eq}}(w_i, r_i)$, the multiplicative update $\eta \leftarrow \eta \cdot \widetilde{\mathrm{eq}}(w_i, r_i)$ followed by $y \leftarrow \eta \cdot t_i(r_i)$ realizes that fold without ever materializing $c_i$ as a separate quantity. After the final round, $\eta = \widetilde{\mathrm{eq}}(\mathbf{w}, \mathbf{r})$, so the closing check $y = \eta \cdot h(\mathbf{r})$ is the standard end-of-protocol test $s_{\mu-1}(r_{\mu-1}) = f(\mathbf{r})$ for the ZeroCheck polynomial $f = \widetilde{\mathrm{eq}}(\mathbf{w}, \cdot) \cdot h$. The verifier therefore tracks only one extra extension-field scalar across the loop, on top of the standard running claim $y$.

*Soundness.* Provided $c_i(u) \neq 0$ for every $u \in U_3$, the four values $\{t_i(u)\}_{u \in U_3}$ uniquely pin down the degree-3 univariate $t_i$, the verifier-side division by $c_i$ is information-preserving, and each round inherits the textbook degree-3 sum-check error $3/|\mathbb{F}_{p^k}|$, totalling $3\mu/|\mathbb{F}_{p^k}|$. The proviso is not free and must be charged separately: $c_i$ is linear, so it has one root, and the bad event is that this root falls in the four-point grid $U_3$; a union bound over the

**Algorithm 5:** Precomputing the small-value tables $A_i(\mathbf{v}, u)$

**Input:** Base-field tables $S, L, R, O : \{0,1\}^\mu \to \mathbb{F}_p$, verifier point $\mathbf{w}$, prefix limit $\ell_0$
**Output:** $A_i(\mathbf{v}, u)$ for all $i < \ell_0$, $\mathbf{v} \in U_3^i$, and $u \in \widehat{U}_3$

**for** $i \leftarrow 0$ **to** $\ell_0 - 1$ **do**
  Split the residual suffix as in (9) and build the tables $\widetilde{\mathrm{eq}}_L(\mathbf{w}_L, \cdot)$ and $\widetilde{\mathrm{eq}}_R(\mathbf{w}_R, \cdot)$;
  **foreach** $\mathbf{v} \in U_3^i$ **do**
    **foreach** $u \in \widehat{U}_3$ **do**
      **if** $(\mathbf{v}, u) \in \{0,1\}^{i+1}$ **then**
        skip the entry, since ZeroCheck gives $h(\mathbf{v}, u, \mathbf{x}_L, \mathbf{x}_R) = 0$ for all Boolean prefixes;
      Compute $A_i(\mathbf{v}, u)$ using (11);

grid and over the $\mu$ rounds bounds it by $4\mu/|\mathbb{F}_{p^k}|$ over the random draw of $\mathbf{w}$. The system-level accounting therefore carries it as a distinct summand of $\epsilon_{\mathrm{ZC}}$; it cannot be absorbed into the $\mu/|\mathbb{F}_{p^k}|$ of the ZeroCheck-to-sum-check reduction plus the $3\mu/|\mathbb{F}_{p^k}|$ of the per-round degree test, because that budget is already fully allocated. The correct total is $\epsilon_{\mathrm{ZC}} \le 8\mu/|\mathbb{F}_{p^k}|$, not $4\mu/|\mathbb{F}_{p^k}|$. The final PCS check contributes the binding error $\epsilon_{\mathrm{PCS}}$ in the usual way; this matches the BDDT25 baseline.

## G.3 Accumulator Tables and Their Precomputation

For $i < \ell_0$, the small-value rounds represent $t_i$ through its values on the grid $U_3 = \{\infty, 0, 1, 2\}$. Define the accumulator table

$$A_i(\mathbf{v}, u) := \sum_{\mathbf{x}_L} \widetilde{\mathrm{eq}}_L(\mathbf{w}_L, \mathbf{x}_L) \sum_{\mathbf{x}_R} \widetilde{\mathrm{eq}}_R(\mathbf{w}_R, \mathbf{x}_R) h(\mathbf{v}, u, \mathbf{x}_L, \mathbf{x}_R), \quad (11)$$

for $\mathbf{v} \in U_3^i$ and $u \in \widehat{U}_3$. If $R_i$ denotes the tensor-product vector of Lagrange coefficients accumulated from the first $i$ verifier challenges, then

$$t_i(u) = \sum_{\mathbf{v} \in U_3^i} R_i[\mathbf{v}] \cdot A_i(\mathbf{v}, u). \quad (12)$$

This is exactly the quantity used in the first loop of Algorithm 2.

At the transition round $i = \ell_0$, the prover must replace the four base-field tables by their partially folded extension-field versions. For each witness table $p_j \in \{S, L, R, O\}$,

$$p_j(\mathbf{r}_{<\ell_0}, \mathbf{x}) = \sum_{\mathbf{b} \in \{0,1\}^{\ell_0}} \widetilde{\mathrm{eq}}(\mathbf{r}_{<\ell_0}, \mathbf{b}) \cdot p_j(\mathbf{b}, \mathbf{x}), \quad (13)$$

which is the materialization step shown between the two loops of Algorithm 2. Equation (10) then evaluates $h(\mathbf{r}_{<\ell_0}, u, \mathbf{x}_L, \mathbf{x}_R)$ from these folded tables for each $u \in \widehat{U}_3$. After this point, the later rounds operate on packed tables over $\mathbb{F}_{p^3}$.

## G.4 Choosing the Prefix Length

BDDT25 analyzes $\ell_0$ as a trade-off between the prefix table size (growing as $(d+1)^{\ell_0}$) and the extension-field work deferred to later rounds. Our measurements refine that trade-off for AVX-512IFMA over $\mathbb{F}_{p^3}$: sweeping $\ell_0 \in \{1, 2, 3\}$ across $\mu = 18, \ldots, 28$ selects $\ell_0 = 2$ in every measured cell, single-threaded and 8-thread alike, and the prover fixes $\ell_0 = 2$ on both paths. The 8-thread prover gains more from the move $\ell_0 = 1 \to 2$ than the single-threaded one does, which is consistent with the post-transition packed tables becoming small enough to sit in each core's private cache. We use $\ell_0$ itself as a tuning knob for balancing prefix-table work against extension-field bandwidth; Appendix K discusses the parallelization rules.

# H Padded vs. Per-Wire ProductCheck

This appendix compares the padded baseline ProductCheck from Appendix C with the per-wire variant from §5.4. We keep the notation of the main text. The circuit has $N = 2^\mu$ gates; the baseline ProductCheck works over two padded tables $f, g : \{0,1\}^{\mu+2} \to \mathbb{F}$ of length $4N$; and the per-wire variant keeps the six wire-local tables $f_q, g_q : \{0,1\}^\mu \to \mathbb{F}$ for $q \in \{L, R, O\}$ and batches the three wire claims with one random scalar $\lambda$. Both protocols keep the two sides $f$ and $g$ separate. They also reuse the prefix- and suffix-factored eq handling of §5.2 and Appendix F, and they use the same degree-3 sum-check structure with the same three transmitted hat-point values $\widehat{U}_3 = \{\infty, 0, 2\}$ per side and per round.

*Computation reduction.* The padded baseline scans tables of length $4N$, while the per-wire protocol scans only $3N$ useful values. The product-tree storage drops correspondingly from $16N - 4$ to $12N - 12$ field elements, an asymptotic 25% saving before any low-level optimization is applied.

*SIMD-level synergy.* The implementation gain is larger than a bare 25% table reduction because the per-wire layout is also a better fit for packed execution. The padded baseline concatenates three wires and one dummy block into one length-$4N$ vector, so early product-tree levels and sum-check scans cross wire and padding boundaries. The per-wire protocol instead keeps six wire-local arrays in lockstep, and the per-wire pair-adjacent arrays match the packed even/odd kernel and avoid the cross-wire mixed layout of the padded baseline. Under the MSB-first layout used by our packed kernels, the even/odd split is a zero-copy prefix/suffix split for each wire. Moreover, the $\lambda$-batching is absorbed before the hot inner loop: one packed block of the per-wire gate performs 24 packed extension multiplications for three wire pairs, whereas the padded baseline performs 8 for one wire pair, so the cost per effective wire pair remains the same. The extra batching work is therefore lower-order, while the 25% reduction in useful table size and the improved memory layout translate directly into higher SIMD utilization.

*Soundness accounting.* Both schemes incur the same shared permutation-compression term $3N/|\mathbb{F}|$ from Schwartz–Zippel on the degree-$3N$ univariate $\Delta_\beta(\alpha)$ that arises after fixing the verifier challenges $(\alpha, \beta)$. The padded baseline contributes $3(\mu+1)(\mu+2)/(2|\mathbb{F}|)$ from $\mu+1$ degree-3 sum-check instances with variable counts $1, \ldots, \mu+1$. The per-wire variant contributes $3\mu(\mu-1)/(2|\mathbb{F}|)$ from $\mu-1$ instances with variable counts $1, \ldots, \mu-1$, plus a $\lambda$-batching term $2(\mu-1)/|\mathbb{F}|$ from Schwartz–Zippel on the degree-2 batching polynomial $q(\lambda) = e_L + \lambda e_R + \lambda^2 e_O$. Subtracting yields a $(4\mu+5)/|\mathbb{F}|$ tightening; both bounds remain $\Theta(N/|\mathbb{F}|)$ asymptotically.

*Proof-size effect.* With $|\widehat{U}_3| = 3$ per side per round, summing over all rounds, top-level transcript values, and final evaluations gives

the closed-form difference

$$\Delta P = P_{\text{pw}} - P_{\text{pad}} = -(4\mu + 14).$$

The ProductCheck proof is therefore always smaller in the per-wire protocol. For our parameter $\mu = 22$, this is a reduction of 102 extension-field elements (1752 → 1650, a 5.8% reduction). The reason is structural: round messages scale quadratically in $\mu$, whereas the additional top values and final evaluations scale only linearly.

*Verifier work.* The per-wire verifier performs a slightly heavier check inside each surviving instance because each final claim contains three wire products instead of one, but it also removes two full product-tree levels and thus two sum-check instances. In our setting these two effects nearly cancel, so verifier cost stays roughly neutral while prover work and ProductCheck proof size both decrease.

*Summary.* The padded baseline wastes one quarter of the ProductCheck work on a dummy fourth block. The per-wire design removes that protocol-level inefficiency, preserves the same degree and message format, keeps the same asymptotic soundness order $\Theta(N/|\mathbb{F}|)$, tightens the explicit bound by removing two padded sum-check instances, shrinks the ProductCheck transcript, and aligns the data layout with the packed even/odd kernel so that the arithmetic saving is amplified by SIMD-friendly execution.

# I Generic AVX-512 Layer-Merging FFT Details

This appendix expands the layer-merging framework summarized in §6.1. The main text keeps only the design rule and the memory argument; here we record the register-budget rationale and the tail-shuffle network that preserves packed execution through the last three FFT layers.

## I.1 Three-Stage Pipeline and Register Budget

We organize the forward DIF FFT for $N_{\text{fft}} = 2^L$ as a three-stage pipeline (prefix of 3-layer no-shuffle kernels, optional 1- or 2-layer remainder, and a fixed 3-layer in-register tail), keyed only on $L$ mod 3 (§6.1). In the prefix stage, one 3-layer kernel loads 8 packed registers (64 coefficients) from stride-spaced locations, performs 12 radix-2 butterflies entirely in registers, and stores once; while the stride is at least 8, no lane permutation is needed.

A 3-layer kernel uses 8 data and 7 twiddle ZMMs, comfortably within the 32-register budget of §2.5; 4 layers would need 16 data and 15 twiddle ZMMs before temporaries, forcing spills and undoing the bandwidth win that layer merging is meant to capture. This is the only reason we cap merge granularity at three layers. The same constraint is what distinguishes our framework from fixed-size 256-point NTT schedules such as the AVX2 Kyber/Dilithium codebases: those implementations hard-wire one stage partition, whereas our prover must choose a uniform merge granularity across many values of $L$.

## I.2 Tail-3 Shuffle Network

The last three FFT layers have strides 4, 2, and 1, all below the SIMD width. At that point, plain stride-based no-shuffle merging is no longer possible. We instead process each 64-coefficient block as four independent 16-point tails and apply the fixed network

$$\texttt{shuffle4} \rightarrow \texttt{shuffle2} \rightarrow \texttt{shuffle1},$$

where $\texttt{shuffle}_k(A, B)$ deinterleaves two ZMM lanes at stride $k$ (the standard radix-2 reorder for stride $k$), aligning the inputs for one packed butterfly per layer. The terminal $\texttt{shuffle}_1$ produces the pair-major blocked output used by the Merkle tree and by DeepFold round 0.

## I.3 Memory Accounting

Let one scalar memory operation denote one 64-bit load or store. A naive layer-by-layer FFT performs one full read and one full write per layer, so its total data traffic is

$$\text{Mem}_{\text{naive}}(L) = 2N_{\text{fft}}L.$$

In the merged schedule, each 3-layer pass costs only one full read and one full write, and the entire transform uses exactly $\lceil L/3 \rceil$ such passes. Therefore

$$\text{Mem}_{\text{merged}}(L) = 2N_{\text{fft}}\lceil L/3 \rceil.$$

The ratio is

$$\frac{\text{Mem}_{\text{naive}}(L)}{\text{Mem}_{\text{merged}}(L)} = \frac{L}{\lceil L/3 \rceil},$$

which is essentially 3 across the prover-relevant range of $L$. This memory reduction, rather than any change in arithmetic complexity, is the main reason layer merging materially improves the DeepFold FFT throughput on modern CPUs.

# J Split FFT Trick: Mechanics, Soundness, and Trade-offs

This appendix collects the fuller mechanism, transcript-level soundness reduction, and parameter trade-offs of the split FFT trick summarized in §2.4 and §6.3.

## J.1 Mechanics of the Split FFT Trick

We first restore the fuller description of the split FFT trick that is only summarized in §2.4. The purpose of the trick is to reinterpret the round-0 FFT and the first few FRI folds as one combined transformation, so that more parallelism is exposed without changing the statement being proved.

*Algebraic decomposition.* Iterating the standard even/odd split of the univariate twin $f(X) = f_0(X^2) + Xf_1(X^2)$ for $t$ levels rewrites one size-$|L_0|$ FFT over $L_0$ as $2^t$ independent FFTs of size $|L_0|/2^t$ over $L_t$, plus $t$ linear recombination layers. This decomposition is the source of the extra parallelism used in §6. It is also consistent with the *shred-to-shine* interpretation of polynomial-commitment design [38].

*Relation to the first $t$ FRI folds.* In standard FRI, the verifier queries a pair $(x, -x)$ on round $i$ and checks that the next-round value is obtained by the fold rule

$$f^{(i+1)}(x^2) = \text{fold}_{r_{i+1}}\big(f^{(i)}(x), f^{(i)}(-x)\big).$$

After the algebraic decomposition above, the first $t$ folds are no longer conceptually separate from the round-0 FFT. For a fixed

point $y \in L_t$, the entire chain of values needed to recover

$$f^{(0)}(\pm x),\ f^{(1)}(\pm x^2),\ \ldots,\ f^{(t-1)}(\pm x^{2^{t-1}}),\ f^{(t)}(y)$$

is determined by the $2^{t+1}$ round-0 codeword values on the corresponding $x$-coset in $L_0$ together with its negation. Thus the first $t$ FRI rounds can be absorbed into the round-0 commitment: instead of materializing and Merkle-committing every intermediate codeword $f^{(1)}, \ldots, f^{(t-1)}$, the prover commits once to a fat round-0 tree whose leaves already contain the local data needed to reconstruct those folds.

*Transcript interpretation.* Under this split view, the prover still commits to the same round-0 codeword $f^{(0)}|_{L_0}$, but the leaf layout changes. Each fat leaf stores the $2^{t+1}$ values on one round-0 coset block, so that after the verifier samples $r_1, \ldots, r_t$, both parties can locally derive the absorbed rounds from that one leaf. The transcript therefore changes only in bookkeeping:

- standard FRI sends $rt_0, rt_1, \ldots, rt_{t-1}, rt_t, \ldots, rt_{\mu-1}$;
- the split variant sends one fat root $rt_0^{\text{fat}}$, then $rt_t, \ldots, rt_{\mu-1}$.

The omitted roots are not lost semantically. They become deterministic functions of the fat round-0 opening and the already sampled fold challenges. This is why the split can improve locality and proof-size trade-offs without changing the underlying FRI statement.

*What this subsection does not cover.* This subsection explains the mechanism of the split. §J.2 proves that absorbing these rounds preserves FRI soundness, and §J.3 analyzes when the split is worthwhile from the proof-size and throughput perspectives.

## J.2 Soundness of the Split FFT Trick

We show that the $t$-level split FFT variant of FRI introduced in §2.4 inherits the soundness bound of standard FRI on the same domain, code rate, and query count. Throughout, $\mu$ denotes the number of FRI rounds, $\rho$ the code rate, $L_i$ the round-$i$ evaluation domain, $K = |L_0|$ the initial codeword length, $s$ the number of verifier queries, $r_i$ the Fiat–Shamir fold challenge of round $i$, $\beta_i$ the query point at round $i$, $rt_i$ the Merkle root of the round-$i$ codeword, and $f^{(i)}$ the round-$i$ folded codeword. We follow the Fiat–Shamir convention in the random-oracle model (ROM).

*Protocols.* Let $\Pi_{\text{std}}$ denote the standard FRI commit-and-query protocol:

(1) Commit phase: the prover sends $rt_0, rt_1, \ldots, rt_{\mu-1}$, where $rt_i$ is a pair-leaf Merkle tree over the codeword $f^{(i)}$ on $L_i$; challenges $r_{i+1}$ are derived by Fiat–Shamir from the transcript up through $rt_i$.
(2) Query phase: for each of $s$ query indices $\beta_0 \in L_0$ (with $\beta_i = \beta_{i-1}^2$), the prover opens $f^{(i)}(\pm\beta_i)$ against $rt_i$ for $i = 0, \ldots, \mu-1$. The verifier checks, for every $i$, that

$$f^{(i+1)}(\beta_{i+1}) = \text{fold}_{r_{i+1}}\big(f^{(i)}(\beta_i), f^{(i)}(-\beta_i)\big).$$

Let $\Pi_{\text{split}}^{(t)}$ denote the $t$-level split variant, differing from $\Pi_{\text{std}}$ as follows. In the commit phase the prover sends

$$(rt_0^{\text{fat}}, rt_t, rt_{t+1}, \ldots, rt_{\mu-1}),$$

where $rt_0^{\text{fat}}$ is a Merkle tree whose leaves each hash a coset of $2^{t+1}$ values of $f^{(0)}$ on $L_0$ (an $x$-coset of size $2^t$ together with its negation); the intermediate trees $rt_1, \ldots, rt_{t-1}$ are omitted. Challenges $r_1, \ldots, r_t$ are derived from the transcript up through $rt_0^{\text{fat}}$, and $r_{i+1}$ from the transcript up through $rt_i$ for $i \geq t$. In the query phase, for each query $\beta_0 \in L_0$ the prover opens exactly one fat leaf containing $f^{(0)}$ on the relevant coset; the verifier locally computes $f^{(1)}(\pm\beta_1), \ldots, f^{(t)}(\pm\beta_t)$ from that leaf by applying the FRI fold rule with $r_1, \ldots, r_t$, and proceeds with the standard checks from round $t$ onwards.

Theorem J.1. *Let $\text{Adv}_{\Pi}(\mathcal{A})$ denote the probability that an adversary $\mathcal{A}$ makes a verifier of $\Pi$ accept a false claim (soundness advantage). For every $t \in \{0, 1, \ldots, \mu\}$ and every PPT adversary $\mathcal{A}$,*

$$\text{Adv}_{\Pi_{\text{split}}^{(t)}}(\mathcal{A}) \ \leq\ \text{Adv}_{\Pi_{\text{std}}}(\mathcal{B}) + \text{negl}(\lambda)$$

*for an explicit reduction $\mathcal{B}$ with essentially the same running time. In particular, both the provable (Johnson-regime) bound and the conjectured (capacity-regime) bound of $\Pi_{\text{std}}$ carry over to $\Pi_{\text{split}}^{(t)}$ unchanged.*

Proof sketch. We give a transcript-level ROM reduction. Given $\mathcal{A}$ against $\Pi_{\text{split}}^{(t)}$, $\mathcal{B}$ runs $\mathcal{A}$ internally and (i) extracts a unique round-0 codeword $\vec{v}$ from $\mathcal{A}$'s fat-leaf openings via Merkle binding, (ii) reconstructs the standard-FRI commitment $(rt_0^{\text{std}}, \ldots, rt_{t-1}^{\text{std}}, rt_t, \ldots, rt_{\mu-1})$ by applying honest folds $\vec{v}^{(i)} = \text{FriFold}_{r_i}(\vec{v}^{(i-1)})$ to $\vec{v}$ and copying $\mathcal{A}$'s remaining roots verbatim, programming the random oracle on $rt_1^{\text{std}}, \ldots, rt_{t-1}^{\text{std}}$ to keep the Fiat–Shamir challenges $r_1, \ldots, r_\mu$ aligned (negligible loss in $\mathcal{A}$'s oracle-query count), (iii) for each verifier query, derives the pairs $\vec{v}^{(i)}(\pm\beta_i)$ for $i < t$ from the fat leaf via local fold and authenticates them against the honestly constructed $rt_i^{\text{std}}$, and forwards $\mathcal{A}$'s round-$\geq t$ openings verbatim. Every standard fold check holds by construction of $\vec{v}^{(i)}$, and the boundary check at $i = t$ coincides with the split verifier's boundary between $rt_0^{\text{fat}}$ and $rt_t$. Hence $\mathcal{B}$ accepts whenever $\mathcal{A}$ does, up to the negligible oracle-programming loss. □

*Remark* 1 (Strict narrowing of the attack surface.). The reduction above is in fact lossy toward the split: in $\Pi_{\text{std}}$ a cheating prover may submit intermediate trees $rt_1, \ldots, rt_{t-1}$ that are *not* honest folds of $rt_0$ yet still pass the $s$ query-time spot checks, whereas in $\Pi_{\text{split}}^{(t)}$ the intermediate codewords are determined bit-for-bit by the fat leaves. Therefore every cheating strategy in $\Pi_{\text{split}}^{(t)}$ is also a cheating strategy in $\Pi_{\text{std}}$, but not conversely, and the split can only tighten soundness. In particular, standard conjectured and Johnson-regime FRI bounds continue to apply to $\Pi_{\text{split}}^{(t)}$ for every admissible $t$ [9, 13, 30].

*Remark* 2 (Independence from the DEEP layer.). The argument above treats FRI as a low-degree test on $f^{(0)}$. In DeepFold, the outer DEEP challenges $(\alpha, c)$ in §2.4 and the anchor-set updates in the eval phase are orthogonal to the split: they act on $\vec{v}^{(0)}$ and its folds as abstract codewords, independent of whether the first $t$ rounds are absorbed. The security reduction of DeepFold therefore composes with Theorem J.1 with no additional cost.

## J.3 Parameterized Split-FFT Trade-offs

This subsection expands the design-space analysis summarized in §6.3. The split depth $t$ trades off commit-time FFT locality, Merkle hashing, and round-0 query bytes. The main text keeps only the decision rules; this subsection records the derivation and concrete numbers for our HyperPlonk–DeepFold setting.

*FFT arithmetic.* The split rewrites one size-$K$ FFT as $t$ chirp layers ($K/2$ butterflies each) plus $2^t$ independent size-$K/2^t$ sub-FFTs, leaving the total butterfly count at $\frac{K}{2}\log_2 K = C_{\text{fft}}(K)$ (Lemma 6.1). The arithmetic count is invariant, but the working set of each sub-FFT shrinks to $K/2^t$ elements and the scheduler gains $2^t$ independent jobs.

*Merkle construction.* Standard FRI builds one Merkle tree per round $i = 0, \ldots, \mu-1$, of height $\log_2 K - i$. The split absorbs rounds $0, \ldots, t-1$ into one fat round-0 tree per group:

- Round-0 fat tree: $K/2^{t+1}$ leaves of $2^{t+1}N_{p,j}\,\mathrm{fe}_b$ bytes each, with height $\log_2 K - t - 1$.
- Rounds $1, \ldots, t-1$: eliminated from the transcript; their values are reconstructed locally from the fat leaves.
- Rounds $t, \ldots, \mu-1$: unchanged.

The number of avoided commit-phase hash invocations is

$$J\sum_{i=1}^{t-1}\frac{K}{2^{i+1}} \approx \frac{JK}{2},$$

independent of $N_p$. For $J = 2$ and $t = 5$, this is about $2.0\times10^6$ fewer leaf/internal hashes at $K = 2^{21}$ and about $1.26\times10^8$ at $K = 2^{27}$.

*Per-query proof size.* Counting the $J$ round-0 trees and the shared rounds $i \geq 1$ separately, the per-query proof-size delta is

$$\Delta P_q(t) = \underbrace{(2^{t+1}-2)N_p\,\mathrm{fe}_b}_{\text{fat-leaf inflation}} - \underbrace{S_q(K,t;J,h,\mathrm{fe}_e)}_{\text{round-0 path shortening + absorbed trees}}, \tag{14}$$

with $S_q = Jth + \sum_{i=1}^{t-1}\left[2\,\mathrm{fe}_e + (\log_2 K - i - 1)h\right]$ (Lemma 6.2).

Three consequences matter for our design. First, increasing $\mu$ shifts the proof-size optimum upward because each absorbed round removes a longer authentication path. Under our query count $s = 88$, the optimum is $t^* = 3$ at $\mu = 18$ (about 64.1 KB saved) but $t^* = 4$ at $\mu = 24$ (about 100.9 KB saved). Second, this same effect makes the production choice $t = 5$ look different across the two code lengths: at $\mu = 18$ it incurs a ∼31.5 KB penalty, whereas at $\mu = 24$ it still shrinks the proof by about 34.5 KB. Third, $t = 6$ is already beyond the useful regime for $N_p = 7$ in both cases: once the fat-leaf term overtakes the accumulated path savings, the proof grows quickly.

*Scaling with $N_p$.* $S_q$ is independent of $N_p$ but grows with $\log_2 K$, whereas the inflation term is linear in $N_p$ and independent of $K$. Hence larger batched opens force the proof-size-optimal split depth downward, while larger code lengths at fixed rate push it upward. Table 8 shows both trends: moving from $\mu = 18$ to $\mu = 24$ shifts the optimum from $t^* = 3$ to $t^* = 4$, whereas keeping $N_p = 7$ still rules out $t = 6$.

**Table 8: Proof-size delta of a $t$-level split versus $t = 0$ for the HyperPlonk–DeepFold open at the two code lengths used in our evaluation. Shared parameters: $\rho = 1/8$, $s = 88$, $h = 32$ B, $\mathrm{fe}_b = 7$ B, $\mathrm{fe}_e = 14$ B, $N_p = 7$, $J = 2$. The upper block uses $\mu = 18$ ($\log_2 K = 21$), and the lower block uses $\mu = 24$ ($\log_2 K = 27$). Negative values mean the split shrinks the proof.**

| $t$ | Fat-leaf (B/q) | Savings (B/q) | $\Delta P_q$ (B) | $s\cdot\Delta P_q$ (KB) |
|---|---|---|---|---|
| $\mu = 18, \log_2 K = 21$ | | | | |
| 1 | 98 | 64 | +34 | +2.9 |
| 2 | 294 | 764 | −470 | −40.4 |
| 3 | 686 | 1432 | −746 | −64.1 |
| 4 | 1470 | 2068 | −598 | −51.4 |
| 5 | 3038 | 2672 | +366 | +31.5 |
| 6 | 6174 | 3244 | +2930 | +251.8 |
| $\mu = 24, \log_2 K = 27$ | | | | |
| 1 | 98 | 64 | +34 | +2.9 |
| 2 | 294 | 956 | −662 | −56.9 |
| 3 | 686 | 1816 | −1130 | −97.1 |
| 4 | 1470 | 2644 | −1174 | −100.9 |
| 5 | 3038 | 3440 | −402 | −34.5 |
| 6 | 6174 | 4204 | +1970 | +169.3 |

## K Additional Parallelization Notes

The main text states only the multithreading principles that consistently improve prover throughput. This appendix records the main failure modes underlying those rules.

The first failure mode is bandwidth saturation. Some packed $\mathbb{F}_{p^3}$ ZeroCheck and fold kernels already run close to the single-core DRAM roofline, so replacing a serial scan by thread-parallel work can increase contention without improving useful arithmetic throughput. This is why ZeroCheck parallelism is applied only after the packed tables become short enough to be cache-friendly, and why the prefix length $\ell_0$ is tuned jointly with the thread count.

The second failure mode is protocol-level serialization. In ProductCheck, batched opening, and DeepFold open, each transcript round depends on the previous challenge, so only round-internal scans can be parallelized; the round sequence itself must remain serial. When a fold reads one table and writes the next, separate scratch buffers are required to preserve the same mathematical state transition as the serial prover. Very small tails remain serial because task-management overhead dominates their arithmetic work.

The third failure mode is over-aggregation. In DeepFold commit, independent sub-FFTs and Merkle hashing scale well, but bundling too much work into one task can exceed the effective per-core cache share and push temporary data back to DRAM. The practical rule is therefore to parallelize across naturally independent jobs while keeping each task's working set cache-resident. The same principle explains the size thresholds used throughout the prover: multithreading is enabled only when the task granularity is large enough to amortize scheduling and synchronization overhead.

**Table 9: Field arithmetic microbenchmark: time (ms) for $10^8$ operations on the $\mathbb{F}_{p^d}$ extension used by each backend (configuration matches Table 3). For our MamaBear addition we report the `lazy_add()` variant from §3.3. "M. vs. G./B./P." is the MamaBear speedup over Goldilocks / BabyBear / Plonky3.**

| Op | Gold. | Baby. | Plonky3 | MamaBear | M. vs. G./B./P. |
|---|---|---|---|---|---|
| Add | 52.88 | 21.65 | 5.75 | 2.27 | 23.3x/9.5x/2.5x |
| Mul | 367.40 | 341.00 | 82.78 | 67.92 | 5.4x/5.0x/1.2x |
| reduce_fast | – | – | – | 6.87 | – |

# L Additional HyperPlonk–DeepFold Evaluation Results

This appendix collects evaluation material that is omitted from the main text for space reasons. It complements §7 with field-arithmetic microbenchmarks, representative primitive circuit sizes, proof-size data, and peak-memory data.

## L.1 Field Arithmetic Microbenchmarks

Table 9 isolates the arithmetic layer by measuring representative field operations on the extension field used by each backend. These results are not an end-to-end comparison on their own, but they help explain why the protocol-level kernels in §7 benefit strongly from the MamaBear field backend.

## L.2 Circuit Sizes of Supported Primitives

**Table 10: Representative circuit sizes.**

| Primitive | Operation | Circuit Size |
|---|---|---|
| SHA256 | 1 block | $2^{18.20}$ |
| AES128 | 1 call | $2^{16.42}$ |
| BLAKE3 | 1 permutation | $2^{17.19}$ |
| Keccak-f[1600] | 1 permutation | $2^{19.16}$ |
| Poseidon2 | 1 permutation | $2^{12.38}$ |

Table 10 reports representative gate counts for SHA256, AES128, BLAKE3, Keccak-f[1600], and Poseidon2 under the current add/mul-gate circuit model. For Poseidon2, we use width 16 and S-box $x^{11}$.

## L.3 Proof Size

Table 11 reports the proof size across circuit sizes and the three security levels for each backend.

**Table 11: HyperPlonk–DeepFold proof size (in KiB) across circuit sizes $2^\mu$ for $\mu = 18$ to 27. The MamaBear configuration uses the 5-level FFT split described in §6.3. "-" indicates data not reported due to out-of-memory (OOM).**

| $\mu$ | Goldilocks | BabyBear | Plonky3 | MamaBear | M. vs. G. |
|---|---|---|---|---|---|
| 18 | 395.11 | 389.43 | 722.88 | 442.87 | +12.1% |
| 19 | 447.72 | 441.07 | 793.58 | 475.82 | +6.3% |
| 20 | 498.41 | 489.98 | 867.83 | 514.99 | +3.3% |
| 21 | 565.80 | 552.34 | 945.07 | 549.26 | -2.9% |
| 22 | 620.20 | 624.84 | 1025.42 | 600.92 | -3.1% |
| 23 | 687.34 | 685.85 | 1109.15 | 654.54 | -4.8% |
| 24 | 756.78 | 753.73 | 1196.36 | 688.51 | -9.0% |
| 25 | – | – | – | 747.54 | – |
| 26 | – | – | – | 792.62 | – |
| 27 | – | – | – | 848.94 | – |

## L.4 Three-Level Split Variant

For comparison with the 5-level split used in the main evaluation, Tables 12 and 13 report the end-to-end proving time and proof size of a 3-level split configuration.

**Table 13: HyperPlonk–DeepFold proof size (in KiB) across circuit sizes $2^\mu$ for $\mu = 18$ to 27. The MamaBear configuration uses the 3-level FFT split described in §6.3. "-" indicates data not reported due to out-of-memory (OOM).**

| $\mu$ | Goldilocks | BabyBear | Plonky3 | MamaBear | M. vs. G. |
|---|---|---|---|---|---|
| 18 | 395.11 | 389.43 | 722.88 | 312.13 | -21.0% |
| 19 | 447.72 | 441.07 | 793.58 | 344.33 | -23.1% |
| 20 | 498.41 | 489.98 | 867.83 | 388.73 | -22.0% |
| 21 | 565.80 | 552.34 | 945.07 | 428.80 | -24.2% |
| 22 | 620.20 | 624.84 | 1025.42 | 484.25 | -21.9% |
| 23 | 687.34 | 685.85 | 1109.15 | 529.12 | -23.0% |
| 24 | 756.78 | 753.73 | 1196.36 | 590.07 | -22.0% |
| 25 | – | – | – | 645.42 | – |
| 26 | – | – | – | 699.30 | – |
| 27 | – | – | – | 774.31 | – |

## L.5 Peak Memory

Table 14 reports the prover peak memory.

**Table 12: HyperPlonk–DeepFold prover time across circuit sizes $2^\mu$ for $\mu = 18$ to 27. Goldilocks, BabyBear, and Plonky3 serve as baselines. "M. vs. G./B./P." reports the MamaBear speedup over Goldilocks / BabyBear / Plonky3. "-Par" columns use 8 threads. Our MamaBear implementations use 3-level splitting FFT. "-" indicates data not reported due to out-of-memory (OOM).**

| $\mu$ | Goldilocks $\mathbb{F}_{p^2}$ | BabyBear $\mathbb{F}_{p^4}$ | Plonky3 B. $\mathbb{F}_{p^4}$ | Plonky3-Par B. $\mathbb{F}_{p^4}$ | MamaBear $\mathbb{F}_{p^3}$ | | MamaBear-Par $\mathbb{F}_{p^3}$ | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Time | M. vs. G./B./P. | Time | M. vs. G./B./P. |
| 18 | 2.07 s | 1.82 s | 1.82 s | 0.33 s | 0.13 s | 16x/14x/14x | 53.1 ms | 39x/34x/6x |
| 19 | 4.34 s | 3.88 s | 3.66 s | 0.67 s | 0.28 s | 16x/14x/13x | 0.11 s | 39x/35x/6x |
| 20 | 8.84 s | 8.07 s | 7.47 s | 1.35 s | 0.64 s | 14x/13x/12x | 0.23 s | 38x/35x/6x |
| 21 | 18.61 s | 16.60 s | 15.48 s | 2.64 s | 1.36 s | 14x/12x/11x | 0.52 s | 36x/32x/5x |
| 22 | 37.30 s | 33.86 s | 30.27 s | 5.84 s | 3.51 s | 11x/10x/9x | 1.11 s | 34x/31x/5x |
| 23 | 75.49 s | 68.64 s | 61.86 s | 11.82 s | 7.96 s | 9x/9x/8x | 2.36 s | 32x/29x/5x |
| 24 | 152.8 s | 138.0 s | 128.3 s | 27.20 s | 17.25 s | 9x/8x/7x | 5.09 s | 30x/27x/5x |
| 25 | – | – | – | – | 37.68 s | –/–/– | 11.41 s | –/–/– |
| 26 | – | – | – | – | 81.48 s | –/–/– | 24.05 s | –/–/– |
| 27 | – | – | – | – | – | –/–/– | – | –/–/– |

**Table 14: HyperPlonk–DeepFold prover peak memory (in GiB) across circuit sizes $2^\mu$ for $\mu = 18$ to 27. "M. vs. G./B./P." reports the MamaBear memory reduction factor over Goldilocks / BabyBear / Plonky3. The MamaBear configuration uses the 5-level FFT split described in §6.3. "-" indicates data not reported due to out-of-memory (OOM).**

| $\mu$ | Gold. | Baby. | Plonky3 | MamaBear | | MamaBear-Par | |
|---|---|---|---|---|---|---|---|
| | | | | Mem. | M. vs. G./B./P. | Mem. | M. vs. G./B./P. |
| 18 | 0.56 | 0.52 | 0.72 | 0.13 | 4x/4x/6x | 0.17 | 3x/3x/4x |
| 19 | 1.11 | 1.03 | 1.44 | 0.25 | 4x/4x/6x | 0.33 | 3x/3x/4x |
| 20 | 2.23 | 2.06 | 2.87 | 0.50 | 4x/4x/6x | 0.66 | 3x/3x/4x |
| 21 | 4.45 | 4.12 | 5.74 | 1.00 | 4x/4x/6x | 1.32 | 3x/3x/4x |
| 22 | 8.91 | 8.24 | 11.47 | 2.00 | 4x/4x/6x | 2.66 | 3x/3x/4x |
| 23 | 17.81 | 16.47 | 22.94 | 4.00 | 4x/4x/6x | 5.31 | 3x/3x/4x |
| 24 | 35.63 | 32.94 | 45.88 | 8.00 | 4x/4x/6x | 10.63 | 3x/3x/4x |
| 25 | – | – | – | 16.00 | –/–/– | 21.25 | –/–/– |
| 26 | – | – | – | 32.00 | –/–/– | 42.50 | –/–/– |
| 27 | – | – | – | 64.00 | –/–/– | 85.00 | –/–/– |