Real Brakedown Performance

A Portable Scalar Baseline for Spielman-Code Polynomial Commitments

Jipeng Zhang

National University of Singapore, Singapore, jp-zhang@outlook.com

Abstract. Brakedown-style polynomial commitment schemes encode the rows of a coefficient matrix with a linear-time Spielman code and Merkle-commit the columns. New schemes are typically compared against the unoptimized LCPC and PolyCommit libraries. Conversely, the reported speedups of the AVX-512 encoder SIMD-Spielman over LCPC combine vector arithmetic with portable algorithmic and memory optimizations. We present RBP-LCPC, an optimized implementation of LCPC in portable stable Rust that integrates column-major batched encoding, right-to-left graph evaluation, lazy modular reduction and cache-aware slicing. Its incremental structure isolates the contribution of each optimization. On both x86-64 and AArch64, its compiled hot kernels contain no vector arithmetic. With LCPC's code and prime, its encodings, Merkle roots and proofs match those of LCPC bit-for-bit. On an AMD Zen 5 server with one thread and $N = 2^{16}$ to 2^{28} coefficients, RBP-LCPC encodes 8.14–9.06× faster than LCPC under its default build configuration. SIMD-Spielman, measured on the same server, remains 1.59–4.44× faster up to $N = 2^{26}$. Commit achieves a smaller speedup of 5.21–5.95× over the default LCPC configuration (4.48–5.95× across both default and tuned configurations), because column hashing then accounts for over a third of single-threaded commit time. Prove and verify are 1.83–7.61× and 2.65–5.36× faster than under the default LCPC configuration across thread counts and sizes, and single-threaded PCS setup is 1.31–1.44× faster. The accompanying artifact reproduces all baselines and measurements from fixed source revisions.

Keywords: Brakedown · Spielman codes · polynomial commitment schemes · zero-knowledge proofs · software performance · portable implementation

1 Introduction

Polynomial commitment schemes (PCSs) underpin succinct arguments, including succinct non-interactive arguments of knowledge (SNARKs) constructed from an interactive oracle proof and a PCS [Tha22]. Linear-code PCSs, such as Ligero [AHIV17], Brakedown [GLS+23] and Orion [XZS22], rely solely on collision-resistant hashing and require no trusted setup. The Brakedown commitment arranges the N coefficients of a multilinear polynomial into a $\sqrt{N} \times \sqrt{N}$ matrix. It encodes each row with a linear-time Spielman code [Spi96] and Merkle-commits the columns of the encoded matrix. Spielman codes operate over arbitrary fields, whereas Reed–Solomon codes require a field suited to the number-theoretic transform (NTT) or specialized evaluation methods [LCH14]. Encoding dominates the commitment cost: in Brakedown, it accounts for 60% to 80% of commitment time for $N = 2^{16}$ to 2^{28} [KDHS26a, Sect. 2.4].

Prior work typically benchmarks Brakedown encoding against the unoptimized Rust libraries LCPC [Wah21] and PolyCommit [CHL+24], which use neither SIMD nor optimizations for the memory bottleneck [KDHS26a]. A reported speedup over LCPC therefore conflates the contribution of a new technique with standard scalar optimization of the same Brakedown code.

Conversely, the fastest published encoder, SIMD-Spielman, encodes 12.8–17.2× faster than LCPC with one thread for $N = 2^{16}$ to 2^{26} on an AMD EPYC 9754 [KDHS26a]. It combines AVX-512 Integer Fused Multiply-Add (IFMA) arithmetic with right-to-left graph evaluation [HKPS25], lazy modular reduction, a column-major layout, last-level cache (LLC) slicing and multi-threading. Because its ablation measures each technique against its own AVX-512 baseline [KDHS26a, Sect. 4.1], it does not isolate how much of the speedup over LCPC requires SIMD.

We ask how fast a portable implementation of the same code can be without explicit SIMD, and how much of SIMD-Spielman's speedup it reproduces. To answer this question, we present RBP-LCPC (RBP: Real Brakedown Performance), a portable Rust implementation derived from LCPC that preserves its PCS interface and contains no vector arithmetic in its hot kernels. Structuring the implementation into incremental stages allows each optimization to be measured in isolation. We evaluate RBP-LCPC on an AMD Zen 5 server (Zen5), an Arm Neoverse-N1 server (N1) and an Arm Neoverse-V2 cloud virtual machine (V2). Our baselines are LCPC under its default build settings (LCPC-default), LCPC under our build settings and portable hash backend (LCPC-tuned), PolyCommit and SIMD-Spielman [KDHS26b] (Section 4). Throughout, T denotes the thread count, and a speedup is the ratio of baseline time to RBP-LCPC time, so values above 1 favor RBP-LCPC.

Contributions. This paper makes four contributions.

1. *A portable optimized Brakedown encoder in a full PCS* (Section 3). Under LCPC's code and prime, its encodings, Merkle roots and proofs match those of LCPC. Under SIMD-Spielman's parameters, it reproduces all 18 reference test-vector pairs.
2. *A measured baseline gap* (Section 5). With $T = 1$, RBP-LCPC encodes 8.14–9.06× faster than LCPC-default on Zen5 for $N = 2^{16}$ to 2^{28}. For $N \in \{2^{18}, 2^{22}, 2^{26}\}$, it is 3.14–3.24× faster on N1 and 6.11–6.22× faster on V2. Commit improves less because column hashing occupies more than a third of single-threaded commit time. With lazy linear combinations, batched hashing, deduplicated column openings and scalar challenge sampling, prove, verify and one-time PCS setup are also faster than in LCPC (Section 5.5).
3. *A separation of SIMD from portable optimization* (Section 5). Per edge-row multiply-accumulate at $T = 1$ ($N \in \{2^{20}, 2^{24}\}$), the portable optimizations on the SIMD-Spielman code close 64–68% of the logarithmic gap from LCPC-default on the LCPC code to SIMD-Spielman (63–66% from LCPC-tuned). With $T = 1$ and $N \leq 2^{26}$, SIMD-Spielman remains 1.59–4.44× faster.
4. *Reporting practices and a reproducible artifact* (Section 6, Code and Data Availability). The accompanying artifact runs the test suite before every benchmark and reproduces all measurements from fixed source revisions.

2 Background

2.1 Brakedown and Its Spielman Code

The Brakedown PCS [GLS+23] commits to a multilinear polynomial ϕ given by its N coefficients in the Lagrange basis. *Commit* arranges the coefficients in a matrix $U \in \mathbb{F}^{k \times k}$ with $k = \sqrt{N}$ and encodes every row with a linear code $\mathrm{Enc}\colon \mathbb{F}^k \to \mathbb{F}^n$ of inverse rate $\rho = n/k$. It then hashes every column of the encoded matrix $\hat{U} \in \mathbb{F}^{k \times n}$ and builds a Merkle tree over the column hashes [KDHS26a, Alg. 5]. *Prove* sends linear combinations of the rows of U and opens random columns of $\hat{U}$ with their Merkle paths. *Verify* encodes

the received combinations and checks them against the opened columns; we call this the *verifier-side encoding*. Orion [XZS22] uses the same structure with a matrix of 128 rows and $N/128$ columns. We call this layout the *Orion shape* and the square matrix the *Brakedown shape* [KDHS26a, Sect. 4.2.1]. We write h for the number of rows encoded together: k in the Brakedown shape and 128 in the Orion shape.

The code is a variant of Spielman's linear-time code [Spi96] built from sparse bipartite expander graphs $G = (L, R, E)$ [GLS+23]. Every left node has the same degree, and every edge (l, r) carries a nonzero random weight $w_{l,r} \in \mathbb{F}$ [KDHS26a, Sect. 2.2]. Evaluating G on $m \in \mathbb{F}^{|L|}$ yields $y \in \mathbb{F}^{|R|}$ with $y_r = \sum_{(l,r)\in E} w_{l,r}\, m_l$. The encoder is systematic and recursive [KDHS26a, Alg. 4]. Given a message m, it evaluates a graph G_1 on m, recursively encodes the result into z and evaluates a graph G_2 on z to produce y'. The resulting codeword is $(m \,\|\, z \,\|\, y')$. Below a threshold length, a base code terminates the recursion. The code is parameterized by the ratio $\alpha = |R|/|L|$ of G_1, a distance parameter β and the inverse rate ρ; its relative distance is $\delta = \beta/\rho$. All h rows share the same graphs, so encoding applies one sparse linear map to h vectors. We call one term $w_{l,r}\, m_l$ accumulated for a single row an *edge-row multiply-accumulate* (MAC).

A graph can be evaluated in two orders [KDHS26a, Alg. 2, 3]. *Left-to-right (L2R)* evaluation visits edges by left node and scatters updates $y_r \leftarrow y_r + w_{l,r}\, m_l$ across y. *Right-to-left (R2L)* evaluation, proposed by Hirner et al. [HKPS25], computes each y_r from its incoming edges. It writes every y_r once, gathers entries of m from random positions and requires the edges to be grouped by right node. Because R2L completes each sum in one place, the sum can remain unreduced until its final store. By contrast, L2R would require an unreduced accumulator for every right node simultaneously [KDHS26a, Sect. 3.1].

2.2 Two Code Instantiations

LCPC and SIMD-Spielman use the same parameters $\alpha = 0.238$, $\beta = 0.1205$ and $\rho = 1.72$, but instantiate the code differently. The *LCPC code* is LCPC's parameter set `SdigCode6`, which matches SIMD-Spielman's α, β and ρ (whereas LCPC's default is `SdigCode3`). Its graph degrees follow the formulas of [GLS+23] and vary with both the recursion level and N. Neighbors are distinct, and the recursion terminates in a Reed–Solomon (RS) base code of at most 20 elements. The *SIMD-Spielman code* follows Orion by fixing the left degrees of G_1 and G_2 to 10 and 20 at every level. It samples neighbors with replacement and terminates the recursion below 13 elements using the identity map. Both instantiations open the same number of columns. Because LCPC's soundness analysis covers only the degree formulas of the LCPC code, we use the SIMD-Spielman code solely for encoder comparisons against SIMD-Spielman.

2.3 Baseline Implementations

LCPC [Wah21] is a Rust library for polynomial commitments built from any linear code with sufficient minimum distance. Its `lcpc-2d` crate implements the generic PCS, while `lcpc-brakedown-pc` provides the Spielman encoding. We build on LCPC (version `f0e2116`), whose encoder exhibits three inefficiencies. First, it stores matrices in row-major order and encodes each row independently in a separate Rayon task, so every row traverses all graphs on its own. Second, it evaluates each graph as a sparse matrix–vector product using a compressed sparse column matrix from the `sprs` crate, which performs an L2R scatter. Third, every edge incurs a full Montgomery multiplication and a modular addition in `ff_derive`'s 127-bit field Ft127.

PolyCommit [CHL+24], the arkworks `poly-commit` library, provides a Brakedown encoding through `MultilinearBrakedown`; we measure only this encoding.

SIMD-Spielman [KDHS26a, KDHS26b] is a C++ Spielman encoder for x86-64 processors supporting AVX-512 IFMA, which fuses a 52-bit integer multiplication with a 64-bit

Table 1: Incremental stages of RBP-LCPC and counterparts in [KDHS26a]

Stage	Technique	Counterpart
`step/00-import`	Original LCPC (`f0e2116`)	–
`step/01-modernize`	Stable toolchain, current dependencies	–
`step/02-fields`	`LazyField` and three fields	Sect. 3.2.1
`step/03-column-major`	Column-major batched encoding	Sect. 3.3
`step/04-r2l`	R2L evaluation, run-time codes	Sect. 3.1
`step/05-lazy`	Lazy reduction per right node	Sect. 3.2.1
`step/06-blocking`	Multi-edge blocking, four edges	–
`step/07-slicing`	LLC slicing, OS cache detection	Sect. 3.4
`step/08-threads`	Expander-level thread pool	Sect. 3.5.3
`step/09-codegen`	Code-generation tuning	–

Note: Stage `step/09-codegen` includes refinement `cdebb9d`, restricting each worker's view to its output range. Final `HEAD` adds cached prepared weights, a lazy RS base case, threaded-encoder refinements and PCS-level optimizations (Section 3.6). IFMA arithmetic has no scalar counterpart (Section 3.5).

accumulation [KDHS26a, Sect. 2.5]. It packs eight consecutive rows into each AVX-512 vector [KDHS26a, Sect. 3.2]. Its implementation [KDHS26b] covers only the fields and the encoder; Merkle-tree hashing [KDHS26a, Sect. 5.2], prove and full verify are outside its scope.

3 Implementation

RBP-LCPC addresses the inefficiencies of LCPC (Section 2.3) in a single implementation incorporating every technique described in this section, with no compile-time option selecting a subset. Its run-time parameters are the field, the code instantiation, the slice count s and the thread count T. Built on LCPC (`f0e2116`), it is organized into incremental stages, each adding one optimization for per-technique attribution (Table 1); Code and Data Availability records the corresponding revision identifiers.

3.1 Constraints and Compatibility

RBP-LCPC is written in portable stable Rust without explicit SIMD. Its source code uses no `std::arch` intrinsics, `core::simd` types, `target_feature` attributes, inline assembly or architecture-specific code paths. Its dependencies are also restricted to scalar execution: the BLAKE3 library [ON] is compiled with its `pure` feature, graph sampling, Fiat–Shamir challenges and input generation use a portable scalar ChaCha20 generator (`ChaCha20Stream`), and `ppv-lite86` is restricted to its `no_simd` backend. To disable LLVM's loop and superword-level parallelism (SLP) vectorizers, we pass `-C no-vectorize-loops -C no-vectorize-slp`. On x86-64, compilation adds `-C target-cpu=native` to enable scalar extensions such as BMI2 (`mulx`) and ADX; on AArch64, it uses the generic target.

RBP-LCPC preserves the public PCS interface of LCPC. The types `SdigEncodingS`, `BrakedownCommit` and `BrakedownEvalProof`, as well as the `LcEncoding` trait, keep their original definitions. Three additional trait methods with backward-compatible defaults extend `LcEncoding`: `encode_rows` encodes all rows together, `collapse_rows` computes row-wise linear combinations, and `column_dots` computes column inner products. Their defaults preserve LCPC's original row-parallel and eager loops, leaving `lcpc-ligero-pc` unchanged. The committed matrix remains row-major, so commitment, proof and serializa-

Table 2: Fields of RBP-LCPC

Field	Modulus	Accumulator	D	Reduction
Mont127	127-bit prime q	320 bits	$\leq 2^{64}$	3-word REDC
Mers127	$2^{127}-1$	320 bits	$< 2^{66}$	$2^{127} \equiv 1$
Fp2M61	$p = 2^{61}-1$, i^2+1	2×128 bits	32	$2^{61} \equiv 1$, fold

Note: Every element occupies 16 bytes. Fp2M61 is $\mathbb{F}_p[\mathrm{i}]/(\mathrm{i}^2+1)$. For Mont127 and Mers127, D is an overflow bound and no fold occurs; REDC denotes Montgomery reduction. LCPC's field Ft127 has the prime q and runs Mont127's kernels.

tion formats match LCPC byte for byte. The only interface change is that the Brakedown encoding requires `F: LazyField` (Section 3.2), which any custom field must implement.

Stage `step/01-modernize` leaves all algorithms unchanged and preserves bit-exact outputs. It selects stable Rust 1.99.1, compiles with `opt-level=3`, a single codegen unit and full link-time optimization (LTO), and replaces LCPC's nightly-only benchmarks with a stable benchmark driver. External dependencies are updated to current releases; because `rand` 0.9 altered the `Uniform<usize>` sampler used for graph generation and column opening, `lcpc-2d` re-implements the `rand` 0.8 sampler. Twelve reference vectors generated by unmodified LCPC verify the coefficients, encoded rows, Merkle root, serialized proof and verification result. They cover Ft127 and LCPC's 63-bit test field `Ft63`, two code parameter sets and three matrix shapes.

3.2 Lazy-Accumulator Fields

Stage `step/02-fields` introduces the `lcpc-fields` crate, whose `LazyField` trait separates graph traversal from modular reduction. A field defines an unreduced accumulator type `Acc` and a bound D on the number of products that one accumulator may hold. Before accumulation, `prepare_weight` converts an edge weight into MAC-ready form, and `mac` adds $w \cdot x$ to an accumulator without reduction. When D products are reached, `fold` restores accumulator capacity, and `reduce` returns the canonical field element. All arithmetic uses native `u64` and `u128` integers (Table 2).

The `LazyField` trait is our portable counterpart to SIMD-Spielman's "unpacked" formats [KDHS26a, Sect. 3.2.1]. IFMA multiplies 52-bit limbs, so SIMD-Spielman splits elements and weights into three limbs in advance. By contrast, a scalar $64 \times 64 \to 128$-bit multiplication is exact, so RBP-LCPC stores field elements in two 64-bit words and the accumulators of Mont127 and Mers127 in five. Mont127 stores an element x in Montgomery form $x \cdot 2^{128} \bmod q$ and prepares a weight w as $w \cdot 2^{192} \bmod q$. Each accumulated product therefore carries the factor 2^{320}; `reduce` divides the sum by 2^{192} via a word-by-word REDC over three 64-bit words, returning the result in Montgomery form. Using $2^{127} \equiv 1$, Mers127 reduces its accumulator by splitting it at bits 127 and 254 and summing the parts. Fp2M61, the field of SIMD-Spielman's test vectors, holds its real and imaginary sums in two `u128` values and negates each weight's imaginary part modulo p so that both sums remain unsigned.

3.3 The Encoding Kernel

Stage `step/03-column-major` encodes all h rows in one batched pass rather than performing one sparse matrix–vector product per row and graph. With s slices (Section 3.4), a batch is one slice of $h_s \approx h/s$ rows; without slicing, $h_s = h$. The batch is stored column-major, so codeword position j of row i lies at index $jh_s + i$. Each edge (l, r) then updates an entire output column from an input column with unit stride, loading its weight once per column rather than once per element. This matches SIMD-Spielman's

Algorithm 1 R2L lazy kernel with multi-edge blocking

Input: graph G; input columns $X[l]$ of height h_s; right-node range $[r_0, r_1)$; zero scratch column A; fold bound D; block factor B
Output: output columns $Y[r]$ for $r_0 \le r < r_1$

for $r \leftarrow r_0, \ldots, r_1 - 1$ **do**
  $(l_1, w_1), \ldots, (l_d, w_d) \leftarrow$ incoming edges of r ▷ CSR, sampling order
  $c \leftarrow 0$ ▷ products since the last fold
  for each chunk J of B consecutive edges **do** ▷ then d mod B chunks of one
    if $c + |J| > D$ **then**
      $A[i] \leftarrow \mathsf{fold}(A[i])$ for all $i < h_s$; $c \leftarrow 1$
    end if
    $w'_e \leftarrow \mathsf{prepare_weight}(w_e)$ for all $e \in J$ ▷ once per edge
    for $i \leftarrow 0, \ldots, h_s - 1$ **do**
      $a \leftarrow A[i]$
      $a \leftarrow \mathsf{mac}(a, w'_e, X[l_e][i])$ for all $e \in J$ ▷ no reduction
      $A[i] \leftarrow a$
    end for
    $c \leftarrow c + |J|$
  end for
  for $i \leftarrow 0, \ldots, h_s - 1$ **do**
    $Y[r][i] \leftarrow \mathsf{reduce}(A[i])$; $A[i] \leftarrow 0$ ▷ one reduction per element
  end for
end for

layout [KDHS26a, Sect. 3.3] without packing eight rows into a vector. In scalar execution, the independent rows within a column expose instruction-level parallelism to the out-of-order core. Because the PCS retains a row-major matrix, `encode_rows` transposes the messages into a column-major scratch buffer and transposes the redundant part back using cache-tiled 16×16 blocks; the scratch buffer is retained and reused across calls.

Stage `step/04-r2l` replaces LCPC's L2R scatter with R2L evaluation [KDHS26a, Alg. 3]. Each output column $Y[r]$ is written once, input columns $X[l]$ are read-only, and rows form the unit-stride inner loop (Algorithm 1). The `R2lGraph` representation stores the incoming edges of every right node in compressed sparse row (CSR) format with 32-bit indices. During setup, a counting sort builds this CSR structure from each graph (sampled in LCPC's order) in $O(|L| + |R| + |E|)$ time. This stage also makes the code instantiation a run-time parameter, supporting both the LCPC code and the SIMD-Spielman code. The latter uses SIMD-Spielman's dimensions, degrees and base case, but samples its graphs with ChaCha20 instead of glibc's `random()`, so its encoding work equals that of SIMD-Spielman. Algorithm 1 evaluates a range $[r_0, r_1)$ of right nodes in blocks of B incoming edges; a single thread covers all of R, whereas multiple threads partition it (Section 3.4).

Stage `step/05-lazy` introduces lazy modular reduction into the R2L kernel. For each right node r, it accumulates the unreduced products of all incoming edges into one accumulator per row, performing one reduction per output element rather than one per edge and row. In the SIMD-Spielman code, the mean in-degree of a right node is about 42 in G_1 and 26 in G_2, matching the "up to 42× fewer modular reductions" of [KDHS26a, Sect. 3.2.1]. Because R2L completes one output column before the next, the only auxiliary memory is one scratch column A of h_s accumulators. Rather than preparing weights on every slice, the encoder caches `prepare_weight` for every graph edge once per code instance. In the LCPC code, it also materializes the Reed–Solomon base case as a dense Vandermonde `R2lGraph` with cached prepared weights, so both expander levels and the base case execute through the same lazy kernel (`encode_column_block_prepared`) without

eager Horner steps. The kernel is generic over `LazyField` and handles every field, both codes and any $h_s \geq 1$; verifier-side encoding invokes the same path.

Multi-edge blocking (stage `step/06-blocking`) has no counterpart in SIMD-Spielman. Under its column-major layout, SIMD-Spielman traverses an output column once per incoming edge [KDHS26a, Sect. 3.3], loading and storing every row's accumulator on each pass. RBP-LCPC instead processes $B = 4$ edges per pass (Algorithm 1). Each row accumulator is loaded once, updated with four MACs and stored once, cutting scratch-column traffic by a factor of B. The block factor B is a compile-time constant chosen empirically on a separate development machine. Because a fold leaves one product in the accumulator and a blocked pass adds B more, the blocked loop requires $D > B$. All three fields satisfy this bound; any field with $D \leq B$ falls back to $B = 1$.

3.4 Slicing and Threads

Stage `step/07-slicing` adopts SIMD-Spielman's cache-aware slicing [KDHS26a, Sect. 3.4]. It splits the matrix horizontally into s slices of about h/s rows and encodes one slice at a time in a reused buffer. The slice count is computed from a cache model rather than a per-CPU table. Without slicing, the input of the first graph holds N elements of 16 bytes. The model sets the adaptive slice count s_a to the smallest power of two at least $\lceil 16N/(0.5 \cdot \mathrm{LLC}) \rceil$, capped so that each slice retains at least 16 rows. The LLC size is queried from Linux sysfs, falling back to POSIX `sysconf` and then to a 16 MiB default. SIMD-Spielman instead selects a slice table at compile time [KDHS26b], although Krieger et al. [KDHS26a, Sect. 3.4] note that the operating system can supply the LLC size [Ell20]. An optional auto-tuner times candidates around s_a, but the measurements use s_a.

Stage `step/08-threads` replaces LCPC's row-level Rayon parallelism with the expander-level parallelism of [KDHS26a, Sect. 3.5.3], avoiding the LLC contention caused by independent concurrent row encodings [KDHS26a, Sect. 3.5.2]. All T workers cooperate on the same slice and graph (including the Reed–Solomon base-case graph), each evaluating a contiguous range $[r_0, r_1)$ of right nodes (Algorithm 1). Worker ranges are balanced by cumulative edge and reduction cost rather than node count alone, preventing stragglers on uneven in-degrees. Because left-node columns are shared read-only, each worker has a private scratch column, and output ranges are disjoint, the kernels need no locks or atomic operations. Workers also parallelize the tiled transposes and synchronize between graph levels using a sense-reversing barrier that spins for up to 64 iterations before yielding. The thread pool persists across encodings, serializes concurrent callers cleanly, and binds each worker to a dedicated core via `sched_setaffinity`. Commitment initialization, column hashing, Merkle-tree construction and proof-generation inner products use Rayon parallelism.

3.5 Code-Generation Tuning

Stage `step/09-codegen` leaves the algorithms unchanged and refines the hot kernels in portable Rust so that LLVM emits more compact scalar code. The 320-bit accumulations in Mont127 and Mers127 use `u64::carrying_add`, which LLVM lowers to a single add-with-carry chain without materializing carry bits. The Mont127 reduction is fully unrolled in local variables, and its final conditional subtraction compiles to a conditional move. Iteration with `chunks_exact` removes bounds checks from the inner loops. Fp2M61 retains the four-multiplication schoolbook product because the three-multiplication Karatsuba variant requires additional callee-saved registers. In place of IFMA, widening `u128` multiplications lower to `mulx` on x86-64 (with BMI2) and to `mul` and `umulh` on AArch64.

3.6 PCS-Level Optimizations

Beyond the encoder, RBP-LCPC applies four portable scalar optimizations across commitment, proving, verification and setup while keeping all outputs bit-exact with LCPC.

Single-pass commitment initialization and batched hashing. Commitment allocates the padded coefficient and codeword matrices and initializes each in one parallel pass over rows, avoiding redundant zero-filling. Column hashing processes blocks of 32 consecutive columns per Rayon task so each row element is read sequentially; each column streams through a 1 KiB stack buffer (one BLAKE3 chunk) before updating the digest, amortizing per-element hashing overhead. Internal Merkle-tree nodes hash each pair of child digests from a single stack buffer.

Lazy linear combinations in prove and verify. Proving and verification route their field inner products through `LazyField`. In `prove`, `collapse_rows_lazy` combines the first low-degree test and the evaluation tensor in one parallel pass over the coefficient matrix, blocking four rows across 256-column L1-resident tiles and reducing once per output entry. In `verify`, `dot_lazy` accumulates column inner products without intermediate reduction and applies a single constant-factor normalization after reduction; all degree-test and evaluation polynomials are encoded together in one batched call to `encode_rows`.

Tiled deduplicated column openings. Because column indices are sampled with replacement, `prove` deduplicates the requested columns, gathers each unique column once from the row-major matrix in 8×64 row–column tiles, and clones repeated openings while the gathered data remain warm in cache. `verify` records the first occurrence of each opened column index and skips redundant inner products and Merkle-path checks for identical repeats, leaving acceptance and rejection behavior unchanged.

Scalar graph and challenge sampling. Graph sampling, Fiat–Shamir challenge expansion and benchmark input generation use `ChaCha20Stream`, a portable scalar ChaCha20 generator operating on 16 `u32` registers with a four-block buffer that reproduces the word stream of `rand_chacha::ChaCha20Rng`. Graph construction avoids cloning neighbor arrays during CSR transposition.

4 Evaluation Setup

4.1 Platforms

The main measurements use two node classes of a shared Slurm cluster (Table 3). The x86-64 machine Zen5 has two AMD EPYC 9755 processors (Zen 5, 128 cores per socket), 1 TB of memory and AVX-512 IFMA support. Each job owns the eight physical cores of one L3 domain and runs one thread per core. The AArch64 machine N1 is a 128-core Ampere Neoverse-N1 node with 1 TB of memory, of which a job uses eight cores. Zen5 runs the `performance` frequency governor, and each N1 job requests it. Both node classes are shared with other users. On Zen5, frequency boost cannot be disabled and hardware performance counters are unavailable, so we report wall-clock times only.

A second AArch64 platform, V2, is a Google Cloud `c4a-highmem-16` virtual machine with 16 Neoverse-V2 cores, of which the benchmarks use eight. It was measured in one archived session at an earlier RBP-LCPC revision (`375a6e1`), prior to the cached prepared weights, lazy Reed–Solomon base case, threaded-encoder refinements and PCS-level optimizations of the main revision (`bd2718c`); we include it as an additional AArch64 data point without re-measurement.

Table 3: Benchmark platforms and build configurations

Platforms				
Attribute	Zen5 (x86-64)		N1 (AArch64)	V2 (AArch64)
Node class	xcnh		xcng	c4a-highmem-16 VM
CPU	AMD EPYC 9755 128-Core Processor		Neoverse-N1	Neoverse-V2
Cores used	8		8	8
L1d / L2 per core	48 KiB / 1 MiB		64 KiB / 1 MiB	64 KiB / 2 MiB
L3 per domain	32 MiB		not reported	80 MiB
Memory	1003 GiB		1005 GiB	126 GiB
AVX-512 IFMA	yes		no	no
Governor	performance		performance	not reported
OS	Ubuntu 26.04 LTS		Ubuntu 24.04.5 LTS	Debian GNU/Linux 12 (bookworm)
Kernel	7.0.0-28-generic		6.8.0-139-generic	6.1.0-53-cloud-arm64
glibc	2.43		2.39	2.36
rustc / LLVM	1.99.1 / 23.1.1		1.99.1 / 23.1.1	1.99.1 / 23.1.1
g++	15.2.0		13.3.0	12.2.0
Sessions	2		1	1
Implementations, x86-64 builds				
Implementation	Code	Fields	BLAKE3	Build
RBP-LCPC	both	Mont127, Mers127, Fp2M61	portable	fat LTO, `native`, vectorizers off
LCPC-default	LCPC code	Ft127	SIMD dispatch	default profile, no flags, vectorizers on
LCPC-tuned	LCPC code	Ft127	portable	fat LTO, `native`, vectorizers off
PolyCommit	own, shorter	Mont127	none	default profile, no flags, vectorizers on
SIMD-Spielman	SIMD-Spielman code	Mont127, Mers127, Fp2M61	none	`-march=native -O3`, AVX-512

Note: Platform values are copied from the manifests of the selected runs. "both": LCPC code and SIMD-Spielman code. RBP-LCPC also runs Ft127 in `step/01-modernize` and the PCS field check (Section 5.5). "vectorizers off": LLVM loop and SLP vectorizers disabled. "not reported": absent from the manifests.

4.2 Baselines

We evaluate the final implementation of RBP-LCPC and each incremental stage of Section 3 using the same benchmark driver.

LCPC is the unmodified reference implementation (`f0e2116`) [Wah21], invoked through a minimal benchmark driver whose parallel row loop matches LCPC's original `commit` procedure. Its two baseline variants differ only in their *build configuration*: the Cargo optimization profile and the compiler flags. LCPC-default uses LCPC's default settings: neither full LTO nor a single codegen unit, and empty `RUSTFLAGS`. Under these settings, LLVM's auto-vectorizers remain enabled and BLAKE3 selects a SIMD backend at run time. LCPC-tuned applies RBP-LCPC's optimization profile, compiler flags and portable

BLAKE3 backend [OANWO20, ON] as a conservative baseline; disassembly verification covers RBP-LCPC's hot kernels only (Section 4.4). Two additional LCPC configurations cross the two build settings with the two BLAKE3 backends to isolate BLAKE3's run-time SIMD dispatch from the compiler flags (Section 5.5).

PolyCommit is the `MultilinearBrakedown` encoding from arkworks `poly-commit` (`1eaec50`) [CHL+24] over the same prime. A minimal change selects a square matrix shape and sets `sec_param = 80`, because the default of 128 is unattainable over a 127-bit prime. This parameter affects only the number of opened columns. PolyCommit produces a shorter codeword: at $N = 2^{20}$ it encodes a 1024×1558 matrix, against 1024×1762 for RBP-LCPC and LCPC.

SIMD-Spielman is version `7b399db` [KDHS26b], modified only in its benchmark driver to support core affinity, per-sample timing output and slice-count overrides. It is compiled with its published flags (`-march=native -O3 -funroll-loops` and its AVX-512 options) and uses its built-in EPYC slice table without retuning for Zen5. Its multi-threaded encoder adds a spin-waiting coordinator thread to its T workers. On eight cores, we therefore also evaluate seven workers plus the coordinator. The best-of-T latency is the minimum across that configuration, $T = 8$ and $T = 1$.

The encoding time measures `encode_matrix`, which invokes `LcEncoding::encode_rows` in RBP-LCPC and the corresponding matrix-encoding routine in each baseline. For RBP-LCPC, it includes both tiled transposes between the row-major commitment matrix and the reused column-major scratch buffer. By contrast, SIMD-Spielman's timed region starts from input already in its packed column layout, and the LCPC driver excludes allocation and input copying; both choices favor the baselines. Setup is measured separately and decomposed into one-time PCS setup (graph sampling, CSR construction, slice choice and encoder initialization), verifier initialization and benchmark input generation (ChaCha20 coefficient sampling and row-buffer initialization). At the PCS level, we time commit, its encoding and hashing components, prove and verify.

4.3 Workloads and Statistics

Comparisons with LCPC and PolyCommit use the LCPC code, with RBP-LCPC over Mont127 and LCPC over Ft127, which shares the same prime. Comparisons with SIMD-Spielman and between fields use the SIMD-Spielman code over Mont127, Mers127 and Fp2M61. The measurements cover:

- prover encoding, $\log_2 N \in \{16, 18, \ldots, 28\}$, $T \in \{1, 8\}$, both shapes for the SIMD-Spielman code, with the Orion shape up to 2^{26} for SIMD-Spielman and 2^{28} for RBP-LCPC;
- thread scaling, $T \in \{1, 2, 4, 8\}$, $\log_2 N \in \{20, 24, 26, 28\}$;
- incremental stages from `step/01-modernize` to the final implementation (`HEAD` in Table 6), $\log_2 N \in \{16, 20, 24\}$, $T = 1$, both codes, with the SIMD-Spielman code over Mers127 from `step/04-r2l`;
- slice-count evaluation, $s \in \{1, 2, 4, \ldots, 256\}$, $\log_2 N \in \{20, 24, 26\}$, $T \in \{1, 8\}$;
- commit, prove and verify, $\log_2 N \in \{20, 24, 28\}$, $T \in \{1, 8\}$, and prove and verify over Ft127 at $\log_2 N \in \{20, 24\}$;
- setup and peak memory, $\log_2 N \in \{20, 24, 28\}$, $T = 1$, and field arithmetic microbenchmarks;
- verifier-side encoding of $v \in \{1, 2\}$ vectors at $T = 1$;
- AArch64 portability on N1 and V2 for RBP-LCPC and both LCPC configurations, $\log_2 N \in \{18, 22, 26\}$, $T \in \{1, 8\}$.

Every encoding and PCS configuration performs one warm-up and five timed runs. Each run encodes fresh input from a fixed seed without flushing caches, measuring steady-state performance. Single-threaded runs are bound to a single core. Runs with T threads use the first T of the job's eight cores, while SIMD-Spielman receives $T + 1$ cores (at most eight) for its coordinator. Each *session* executes the complete measurement series once on a single node, organized into groups whose configurations run in random order. The Zen5 data come from two sessions on two different nodes, and the N1 and V2 data from one session each.

Each table cell is the mean over sessions of the per-session median, and the artifact CSV files also record the range over all timed runs. A speedup is the ratio of two such means. We call RBP-LCPC *faster* than a baseline only if, in every session that measures both, its slowest run is faster than the fastest baseline run. A cell is *divergent* († in the tables) when its two session medians differ by more than 5% (Code and Data Availability). Five runs per session are too few to estimate confidence intervals reliably, so we report none. Time per MAC (ns/MAC) is the wall-clock time divided by the number of edge-row MACs.

4.4 Correctness and Scalar-Code Checks

Three test-only reference oracles verify the encoder output. First, a test module compiles unmodified LCPC (`f0e2116`) alongside RBP-LCPC in the same executable. It compares the original Ft127 against RBP-LCPC's Ft127 and Mont127 for `SdigCode3` and `SdigCode6`, three matrix shapes and three graph seeds. Each test checks the encoded matrix element by element, the Merkle root, the serialized evaluation proof byte for byte and proof verification. Second, the SIMD-Spielman artifact provides 18 input–output test vectors for Mont127, Mers127 and Fp2M61, $N \in \{2^{12}, 2^{14}, 2^{16}\}$ and both shapes [KDHS26b]. RBP-LCPC matches all 18 vectors bit-for-bit under the SIMD-Spielman code when supplied with identical expander graphs. Third, a direct reference encoder implements Algorithms 1–4 of [KDHS26a] one row at a time, using a re-implementation of glibc's `random()` verified against the C library. Its L2R-eager, R2L-eager and R2L-lazy variants match all 18 vectors bit-for-bit.

Further tests validate field arithmetic against the `num-bigint` big-integer library and test overflow bounds with all inputs and weights set to the largest field element on graphs of in-degree 100. They also confirm that batching, slicing and threading preserve the output. Before any benchmark runs, the test suite of RBP-LCPC or of each incremental stage and the disassembly verification below are executed. Each baseline is likewise verified against its reference vectors and test suite, and no timing is taken from a build whose tests fail.

The disassembly checker `check_codegen.py` inspects every benchmark executable. It classifies each instruction in the hot kernels `encode_column_block_prepared` (and `encode_column_block` and `reed_solomon_column_block` in the earlier V2 build), the lazy linear-combination kernels `collapse_rows_lazy` and `dot_lazy`, and the standalone `mac_column` and `reduce_column` routines (into all of which field arithmetic is inlined). Loads, stores, register copies and zeroing idioms on vector registers are classified as *vector moves*, which are reported and permitted. Any other instruction operating on a vector register, including an unrecognized opcode, is classified as *vector arithmetic* and fails verification. This analysis establishes one guarantee on both architectures: *the hot kernels contain no vector arithmetic, and the compiler uses vector registers only for zeroing and copy stores.* Four caveats apply. First, the transpose routine `transpose_into` runs within the timed encoding but is not inspected. Second, although compile-time features disable run-time dispatch to BLAKE3's SSE2 and SSE4.1 modules linked into the x86-64 executable, the benchmark driver does not assert the active backend at run time. Third, glibc's `memcpy` and `memset` may invoke AVX2 or AVX-512 implementations selected at load

Table 4: Prover encoding on Zen5

		RBP-LCPC	Baseline time (speedup)		
T	$\log_2 N$	time	LCPC-default	LCPC-tuned	PolyCommit
1	16	4.15 ms	37.36 ms (8.99×)	31.70 ms (7.63×)	16.40 ms (3.95×)
1	18	14.95 ms	135.43 ms (9.06×)	114.17 ms (7.63×)	66.19 ms (4.43×)
1	20	57.38 ms	503.11 ms (8.77×)	425.29 ms (7.41×)	324.96 ms (5.66×)
1	22	210.88 ms	1831.43 ms (8.68×)	1546.50 ms (7.33×)	1222.35 ms (5.80×)
1	24	850.50 ms	7074.26 ms (8.32×)	5976.86 ms (7.03×)	4610.19 ms (5.42×)
1	26	3267.25 ms	27175.93 ms (8.32×)	23008.51 ms (7.04×)	17947.83 ms (5.49×)
1	28	13359.35 ms	108806.41 ms (8.14×)	92031.71 ms (6.89×)	69325.43 ms (5.19×)
8	16	0.55 ms	4.77 ms (8.70×)	4.07 ms (7.41×)	2.13 ms (3.88×)
8	18	1.92 ms	17.23 ms (8.99×)	14.59 ms (7.61×)	8.44 ms (4.40×)
8	20	7.95 ms	63.66 ms (8.00×)	54.17 ms (6.81×)	41.46 ms (5.21×)
8	22	28.52 ms	231.12 ms (8.10×)	196.39 ms (6.89×)	153.70 ms (5.39×)
8	24	111.78 ms	894.02 ms (8.00×)	763.74 ms (6.83×)	586.10 ms (5.24×)
8	26	449.47 ms	3421.92 ms (7.61×)	2909.21 ms (6.47×)	2282.09 ms (5.08×)
8	28	1896.04 ms	13602.56 ms (7.17×)	11557.14 ms (6.10×)	8611.58 ms (4.54×)

Note: Format: time (speedup = baseline time / RBP-LCPC time). RBP-LCPC and LCPC run the LCPC code (RBP-LCPC over Mont127, LCPC over Ft127, same prime); PolyCommit runs its own shorter code over Mont127. All 42 speedups are significant.

time. Fourth, stage `step/01-modernize` predates the hot kernels, so the check locates none there and establishes no bound. All correctness tests and disassembly checks for the measured executables passed (CSV T0 and CSV T11 in the artifact).

5 Results

5.1 Speedup over LCPC and PolyCommit

On the same code and prime, RBP-LCPC encodes 8.14–9.06× faster than LCPC-default at $T = 1$ on Zen5 for $N = 2^{16}$ to 2^{28} (Table 4).

At $N = 2^{28}$, one encoding takes 13.4 s with RBP-LCPC and 109 s with LCPC-default, a speedup of 8.14×. Against LCPC-tuned, the speedup at $T = 1$ is 6.89–7.63×; against PolyCommit, it is 3.95–5.80×. In all 42 comparisons in the table, RBP-LCPC's slowest run is faster than the baseline's fastest run in every session (Section 4).

With $T = 8$, the speedup over LCPC-default falls to 7.17–8.99×, because LCPC's row-parallel encoder scales better than RBP-LCPC's expander-level parallelism (Section 5.4). The build configuration explains only a small part of the gap: LCPC-tuned is 1.18–1.19× faster than LCPC-default at $T = 1$.

The speedup transfers to both AArch64 platforms at $N \in \{2^{18}, 2^{22}, 2^{26}\}$ (Table 5). Against LCPC-default at $T = 1$, RBP-LCPC is 3.14–3.24× faster on N1 and 6.11–6.22× faster on V2 (measured at the earlier revision `375a6e1`); at $T = 8$, the speedups are 3.08–3.49× (N1) and 5.48–5.88× (V2). Against LCPC-tuned, the speedups are 2.84–2.93× (N1) and 5.16–5.26× (V2) at $T = 1$, and 2.74–2.82× (N1) and 4.63–4.97× (V2) at $T = 8$.

Stage `step/01-modernize` introduces no algorithmic changes, yet it encodes 1.20–1.21× faster than LCPC-tuned at $T = 1$. The two measurements use different benchmark drivers:

Table 5: Prover encoding on N1 and V2

		RBP-LCPC	Baseline time (speedup)	
T	$\log_2 N$	time	LCPC-default	LCPC-tuned
N1 (Neoverse-N1)				
1	18	88.38 ms	286.12 ms (3.24×)	258.99 ms (2.93×)
	22	1202.32 ms	3852.84 ms (3.20×)	3478.62 ms (2.89×)
	26	18197.66 ms	57213.72 ms (3.14×)	51663.11 ms (2.84×)
8	18	11.86 ms	41.40 ms (3.49×)	32.53 ms (2.74×)
	22	154.72 ms	481.85 ms (3.11×)	436.33 ms (2.82×)
	26	2324.02 ms	7156.90 ms (3.08×)	6462.43 ms (2.78×)
V2 (Neoverse-V2, 375a6e1)				
1	18	22.26 ms	138.35 ms (6.22×)	117.01 ms (5.26×)
	22	299.05 ms	1856.94 ms (6.21×)	1571.07 ms (5.25×)
	26	4508.46 ms	27533.26 ms (6.11×)	23267.14 ms (5.16×)
8	18	3.17 ms	17.38 ms (5.48×)	14.69 ms (4.63×)
	22	41.97 ms	232.57 ms (5.54×)	196.60 ms (4.68×)
	26	585.88 ms	3442.36 ms (5.88×)	2910.08 ms (4.97×)

Note: LCPC code; RBP-LCPC over Mont127, LCPC over Ft127. Format: time (speedup = LCPC time / RBP-LCPC time). Generic AArch64 builds; V2 at `375a6e1`.

the driver for the incremental stages invokes the row encoder sequentially, whereas the LCPC driver executes a single-threaded Rayon loop. This stage also updates the `ff` and `rand` dependencies. Because we did not isolate the cause, a factor of about 1.21 remains outside the per-technique attribution in Section 5.2.

5.2 Contribution of Each Technique

Lazy reduction contributes the largest single factor, 5.88× at $N = 2^{20}$ in the LCPC code (Table 6).

In the LCPC code at $N = 2^{20}$, encoding latency falls from 352 ms at the first stage to 56.8 ms in the final implementation (`HEAD`), a cumulative speedup of 6.21×. Code-generation tuning contributes 1.31×, and the final cached-weight, lazy Reed–Solomon base-case, tiled-transpose and reusable-scratch refinements add a further 1.09×. Column-major batching and R2L evaluation have little effect on latency by themselves, but they are prerequisites for lazy reduction (Section 3).

Stage `step/02-fields` increases latency by a factor of 1.49. It replaces Ft127 with Mont127, whose eager multiplication is 1.36× slower (Table 7). This difference accounts for only part of the slowdown; the remainder is unexplained by our measurements. The incremental speedups of the remaining stages at $N = 2^{20}$ lie between 0.99 and 1.09; at $N = 2^{24}$, slicing yields an incremental speedup of 1.06× at $T = 1$ (Section 5.4 covers $T = 8$).

In the SIMD-Spielman code over Mers127, the cumulative speedup from `step/04-r2l` to the final implementation is 2.12×, of which lazy reduction contributes 1.53×. We attribute the smaller lazy-reduction gain to the lower cost of Mers127 reduction, which requires no multiplication, though this cannot be verified directly without hardware counters (Section 4.1). As in any cumulative breakdown, these factors depend on the order

Table 6: Encoding latency per tagged commit on Zen5

	LCPC code				SIMD-Spielman code, Mers127		
Tag	2^{20} (ms)	step	cum.	2^{24} (ms)	2^{20} (ms)	step	cum.
`01 modernize`	352.38	base	base	4989.89	–	–	–
`02 fields`	523.49	0.67	0.67	7383.31	–	–	–
`03 column-major`	478.89	1.09	0.74	6809.43	–	–	–
`04 R2L`	484.53	0.99	0.73	6869.36	85.70	base	base
`05 lazy`	82.38	5.88	4.28	1317.47	56.16	1.53	1.53
`06 blocking`	80.11	1.03	4.40	1213.16	54.46	1.03	1.57
`07 slicing`	80.48	1.00	4.38	1148.11	54.47	1.00	1.57
`08 threads`	81.31	0.99	4.33	1142.06	56.10	0.97	1.53
`09 codegen`	62.05	1.31	5.68	880.04	41.35	1.36	2.07
`HEAD`	56.78	1.09	6.21	853.23	40.39	1.02	2.12

Note: $T = 1$. Step = previous tag's time / this tag's time; cum. = first measured tag's time / this tag's time; below 1 is a slowdown. Tag 01 runs Ft127, later tags Mont127. The SIMD-Spielman code needs tag 04 (–).

in which stages are applied.

With lazy reduction, the structure of a 127-bit modulus has little effect on encoding time. The lazy Mont127 MAC costs 0.92× as much as the Mers127 MAC (Table 7), even though eager Mont127 multiplication is 2.17× slower. Because MACs dominate encoding, the two fields encode within 10.1% of each other, with Mers127 slightly faster due to its cheaper reduction. Fp2M61 encodes 1.25× faster than Mont127, but it is a 122-bit field rather than a 127-bit one.

Table 7: Field arithmetic and encoding cost on Zen5

		Operation (ns)			Encoding (ms)		
Field	Bits	Mul	MAC	Reduce	2^{20}	2^{24}	ns/MAC
Mont127	127	9.31	1.73	3.53	45.26	732.00	1.83
Mers127	127	4.30	1.89	0.61	40.72	658.03	1.65
Fp2M61	122	4.10	1.39	0.48	36.20	584.89	1.46
Ft127	127	6.86	1.73	3.54	–	–	–

Note: Encoding: SIMD-Spielman code, $T = 1$; ns/MAC at 2^{20}. Nanoseconds are wall-clock measurements. Ft127 is LCPC's field, a test oracle without a SIMD-Spielman code encoder (–).

5.3 Distance to AVX-512

With $T = 1$ and $N \leq 2^{26}$, SIMD-Spielman is 1.59–4.44× faster than RBP-LCPC on Zen5 (Table 8).

At $N = 2^{20}$ with Mont127, RBP-LCPC takes 45.3 ms and SIMD-Spielman 14.0 ms. Across all N, the RBP-LCPC-to-SIMD-Spielman time ratio is 1.51–3.99× for Mont127, 1.43–4.44× for Mers127 and 0.79–2.87× for Fp2M61. For Mont127 and Mers127, this gap narrows as N grows and reaches its minimum at $N = 2^{28}$; for Fp2M61 at $N = 2^{28}$,

Table 8: Prover encoding: RBP-LCPC vs. SIMD-Spielman on Zen5

		RBP-LCPC / SIMD-Spielman in ms (speedup)		
T	$\log_2 N$	Mont127	Mers127	Fp2M61
Brakedown shape ($k \times k$)				
1	16	2.71 / 0.68 (3.99×)	2.44 / 0.55 (4.44×)	2.15 / 0.75 (2.87×)
	18	10.88 / 2.89 (3.76×)	9.75 / 2.39 (4.08×)	8.59 / 4.35 (1.97×)
	20	45.26 / 13.99 (3.24×)	40.72 / 12.56 (3.24×)	36.20 / 22.57 (1.60×)
	22	181.15 / 59.16 (3.06×)	162.49 / 52.75 (3.08×)	144.14 / 68.07 (2.12×)
	24	732.00 / 266.03 (2.75×)	658.03 / 244.14 (2.70×)	584.89 / 294.74 (1.98×)
	26	2977.84 / 1317.94 (2.26×)	2673.66 / 1223.93 (2.18×)	2365.65 / 1492.28 (1.59×)
	28	12250.57 / 8096.69 (1.51×)	10978.57 / 7653.66 (1.43×)	9834.38 / 12504.94† (0.79×)
8	16	0.39 / 0.18 (2.17×)	0.38 / 0.15 (2.53×)	0.33 / 0.25 (1.32×)
	18	1.40 / 0.72 (1.94×)	1.33 / 0.67 (1.99×)	1.19 / 0.89 (1.34×)
	20	6.45 / 3.37 (1.91×)	6.00 / 3.20 (1.88×)	5.47 / 3.48 (1.57×)
	22	25.04 / 11.80 (2.12×)	22.79 / 11.25 (2.03×)	20.69 / 13.59 (1.52×)
	24	99.90 / 53.17 (1.88×)	91.46 / 48.67 (1.88×)	83.61 / 53.38 (1.57×)
	26	415.64 / 258.00 (1.61×)	380.86 / 233.70 (1.63×)	348.20 / 232.10 (1.50×)
	28	1754.21 / 2803.66 (0.63×)	1628.34 / 2795.47 (0.58×)	1492.32 / 1870.34 (0.80×)
Orion shape (128 rows)				
1	16	2.63 / 0.72 (3.65×)	2.35 / 0.60 (3.92×)	2.06 / 0.82 (2.51×)
	18	10.91 / 3.41 (3.20×)	9.71 / 2.94 (3.30×)	8.56 / 3.70 (2.31×)
	20	46.47 / 20.31 (2.29×)	41.67 / 18.50 (2.25×)	36.88 / 22.95 (1.61×)
	22	207.44 / 116.34 (1.78×)	182.70 / 114.76 (1.59×)	170.29 / 167.80 (1.01×)
	24	1084.29 / 760.58 (1.43×)	991.95 / 737.91 (1.34×)	931.10 / 1035.43 (0.90×)
	26	6749.85 / 5107.92 (1.32×)	6321.66 / 5011.11 (1.26×)	6097.38 / 6534.87 (0.93×)
	28	31679.41 / o	29379.77 / o	27707.36 / o
8	16	0.35 / 0.18 (1.94×)	0.34 / 0.16 (2.12×)	0.31 / 0.23 (1.35×)
	18	1.37 / 0.72 (1.90×)	1.29 / 0.63 (2.05×)	1.16 / 0.80 (1.45×)
	20	6.49 / 4.56 (1.42×)	5.95 / 3.60 (1.65×)	5.43 / 3.50 (1.55×)
	22	29.83 / 39.75 (0.75×)	27.59 / 36.22 (0.76×)	25.96 / 22.66 (1.15×)
	24	149.71 / 239.38 (0.63×)	139.88 / 235.12 (0.59×)	133.41 / 154.89 (0.86×)
	26	963.62 / 1222.04 (0.79×)	926.20 / 1212.11 (0.76×)	917.03 / 949.85 (0.97×)
	28	4933.03 / o	5000.64† / o	4707.93 / o

Note: Format: RBP-LCPC ms / SIMD-Spielman ms (ratio = RBP-LCPC time / SIMD-Spielman time; above 1, SIMD-Spielman is faster). Both run the SIMD-Spielman code. $T = 8$: RBP-LCPC with 8 threads against the best of SIMD-Spielman's $T = 1$, $T = 8$ and 7-worker runs. o: not measured. †: the time differs by more than 5% between sessions.

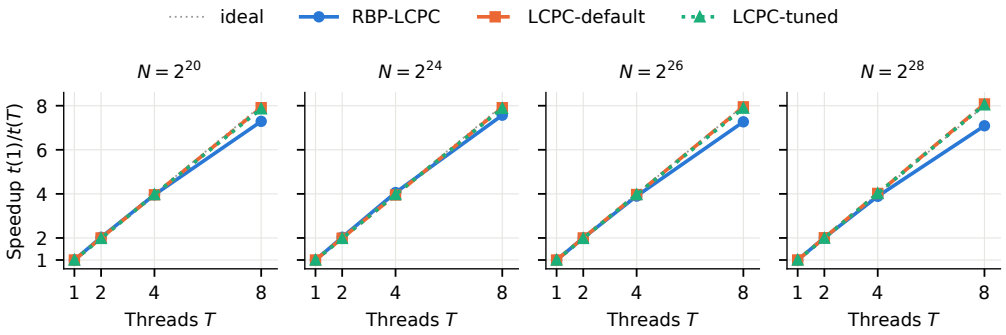


Figure 1: Thread scaling of the prover encoding on Zen5, LCPC code

RBP-LCPC is 1.27× faster. We attribute this narrowing to memory traffic limiting both encoders at large N and reducing the benefit of wide arithmetic.

On eight cores, the best SIMD-Spielman configuration is 1.32–2.53× faster than RBP-LCPC ($T = 8$) for $N \leq 2^{26}$; at $N = 2^{28}$, RBP-LCPC is 1.25–1.72× faster. There, SIMD-Spielman's best eight-core time is 1.27× the eight-thread time reported by Krieger et al. on another processor [KDHS26a], reproduced across both measurement sessions on two nodes. Because SIMD-Spielman's slice table is untuned for Zen5 and its encoder adds a coordinator thread, a retuned SIMD-Spielman may be faster.

In the Orion shape at $T = 1$, SIMD-Spielman's advantage narrows with N; over Fp2M61, the two encoders match at $N = 2^{22}$ and RBP-LCPC is faster at $N = 2^{24}$ and 2^{26}. Krieger et al. [KDHS26a] attribute SIMD-Spielman's weaker Orion performance to the short columns of the 128-row matrix, which restrict slicing.

Because comparisons with LCPC and SIMD-Spielman use different code instantiations, we normalize all three implementations by edge-row MAC count. The two codes differ in work: at $N = 2^{20}$, RBP-LCPC takes 1.27× as long on the LCPC code as on the SIMD-Spielman code. At $N = 2^{20}$ and $T = 1$, LCPC-default requires 15.43 ns/MAC, RBP-LCPC requires 1.83 ns/MAC, and SIMD-Spielman requires 0.57 ns/MAC. On a logarithmic scale, portable optimization thus closes 64% of the gap from LCPC-default to SIMD-Spielman at $N = 2^{20}$ and 68% at $N = 2^{24}$.

Per MAC, SIMD-Spielman remains 3.24× faster at $N = 2^{20}$. AVX-512 IFMA updates the accumulators of eight rows in one instruction [KDHS26a]. RBP-LCPC updates one row per scalar MAC, and each 128-bit product requires multiple scalar multiplications. Krieger et al. attribute the gap between the eight-lane vector width and their measured speedups to the higher memory pressure of vector code [KDHS26a].

5.4 Thread and Cache Scaling

With $T = 8$, RBP-LCPC reaches a parallel efficiency of 0.89–0.95, against 0.98–1.01 for both LCPC configurations.

In Fig. 1, at $T = 8$, RBP-LCPC scales by 7.09–7.57× over $T = 1$, versus 7.86–8.08× for both LCPC configurations. Balanced worker splits, tiled transposes and reused scratch buffers keep eight-core efficiency high, but LCPC assigns independent rows to threads whereas RBP-LCPC's workers share each graph level, synchronize at barriers between levels and share the LLC. We attribute the remaining efficiency gap to barriers and shared memory bandwidth.

Slicing yields modest gains at $T = 1$ and larger ones under multi-threading (Table 9): the best slice count is at most 1.04× faster than $s = 1$ at $T = 1$, but up to 1.43× faster at $T = 8$. The adaptive count s_a is within 7.3% of optimal in 6 of 6 configurations; the

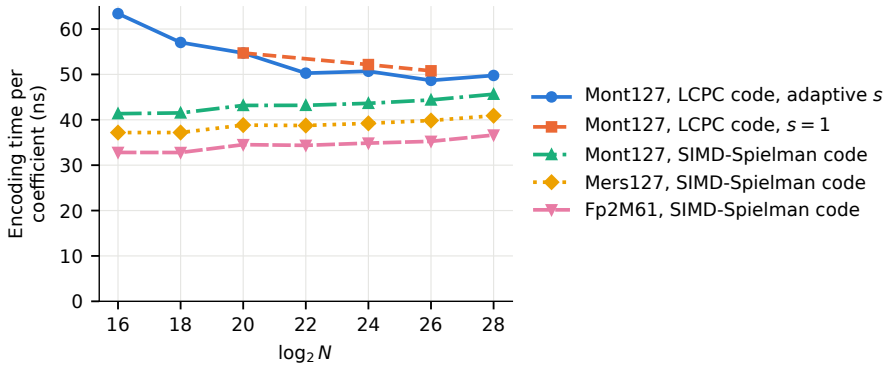


Figure 2: Single-threaded encoding time per coefficient on Zen5

largest gap occurs at $N = 2^{20}$ and $T = 8$, where $s_a = 1$. At $N \geq 2^{24}$ and $T = 1$ (Fig. 2), omitting slicing in the LCPC code slows encoding by 1.04× relative to the best slice count.

Table 9: Slice-count sweep on Zen5

		$s = 1$	Best		Adaptive			
$\log_2 N$	T	ms	s^*	ms	s_a	ms	gap	speedup
20	1	57.38	8	56.33	1	57.38	1.9%	1.02
20	8	7.91	8	7.37	1	7.91	7.3%	1.07
24	1	875.23	32	842.43	16	849.55	0.8%	1.04
24	8	142.44	32	110.19	16	111.86	1.5%	1.29
26	1	3408.59	128	3273.39	64	3273.41	0.0%	1.04
26	8	630.91	128	440.36	64	449.24	2.0%	1.43

Note: LCPC code, Mont127, $s \in \{1, 2, 4, \ldots, 256\}$. Gap = (adaptive time − best time) / best time. Speedup = time at $s = 1$ / best time.

5.5 PCS-Level Cost

At $T = 1$, commit is 4.48–5.95× faster than in either LCPC configuration (Table 10), less than encoding alone because column hashing takes a large share of commit time.

At $N = 2^{28}$, commit takes 23.8 s with RBP-LCPC and 125 s with LCPC-default. At $T = 1$, hashing accounts for 33.8–34.8% of RBP-LCPC's commit time, versus 9.0–10.7% for LCPC-default. With the same portable BLAKE3 backend, batched leaf and Merkle hashing make RBP-LCPC's hashing 1.20–1.25× faster than LCPC-tuned at $T = 1$. At $T = 8$, the commit speedup is 4.07–5.09×, with hashing accounting for 26.5–28.2% of commit time. Further encoder gains thus have a diminishing effect on commit; larger speedups require faster hashing.

Two additional LCPC configurations (CSV T6 in the artifact) vary the hash backend under a fixed build configuration. Portable BLAKE3 takes 0.79–0.81 of the hash time of BLAKE3 with run-time SIMD dispatch, and commit time changes by a factor of 1.00–1.05. On Zen5, the scalar-only constraint thus imposes no commit-time overhead on RBP-LCPC.

Prove and verify are faster in RBP-LCPC across all measured sizes and thread counts, while proof sizes remain identical (Table 10). Across T and N, RBP-LCPC proves 1.83–7.61× faster than LCPC-default (4.38–7.61× at $T = 1$) and 2.01–8.45× faster than LCPC-tuned.

Table 10: PCS operations on Zen5

T	$\log_2 N$	Implementation	Commit (speedup)	Hash	Prove (speedup)	Verify (speedup)
1	20	RBP-LCPC (Mont127)	92.41 ms	33.8%	10.36 ms	38.48 ms
		RBP-LCPC (Ft127)	91.47 ms	33.2%	10.41 ms	112.47 ms
		LCPC-default	549.74 ms (5.95×)	9.0%	45.38 ms (4.38×)	206.26 ms (5.36×)
		LCPC-tuned	476.43 ms (5.16×)	8.1%	51.78 ms (5.00×)	206.12 ms (5.36×)
1	24	RBP-LCPC (Mont127)	1525.19 ms	33.9%	110.55 ms	262.97 ms
		RBP-LCPC (Ft127)	1510.35 ms	33.6%	108.98 ms	365.56 ms
		LCPC-default	7944.02 ms (5.21×)	10.2%	617.53 ms (5.59×)	817.51 ms (3.11×)
		LCPC-tuned	6826.04 ms (4.48×)	9.1%	723.21 ms (6.54×)	812.19 ms (3.09×)
1	28	RBP-LCPC (Mont127)	23797.32 ms	34.8%	1547.07 ms	1237.07 ms
		LCPC-default	124678.14 ms (5.24×)	10.7%	11772.94 ms (7.61×)	3277.25 ms (2.65×)
		LCPC-tuned	106980.18 ms (4.50×)	9.7%	13076.86 ms (8.45×)	3247.98 ms (2.63×)
8	20	RBP-LCPC (Mont127)	14.35 ms	28.2%	4.76 ms	5.46 ms
		RBP-LCPC (Ft127)	14.23 ms	27.5%	4.70 ms	14.54 ms
		LCPC-default	72.99 ms (5.09×)	8.7%	8.71 ms (1.83×)	27.72 ms (5.08×)
		LCPC-tuned	65.19 ms (4.54×)	7.6%	9.58 ms (2.01×)	27.43 ms (5.02×)
8	24	RBP-LCPC (Mont127)	249.68 ms	26.5%	32.67 ms	34.94 ms
		RBP-LCPC (Ft127)	248.21 ms	26.2%	32.65 ms	48.19 ms
		LCPC-default	1158.99 ms (4.64×)	8.8%	122.06 ms (3.74×)	108.90 ms (3.12×)
		LCPC-tuned	1015.55 ms (4.07×)	7.8%	131.44 ms (4.02×)	107.59 ms (3.08×)
8	28	RBP-LCPC (Mont127)	3757.24 ms	27.7%	295.81 ms	161.48 ms
		LCPC-default	17396.94 ms (4.63×)	9.5%	1829.75 ms (6.19×)	435.12 ms (2.69×)
		LCPC-tuned	15348.57 ms (4.09×)	8.4%	1985.60† ms (6.71×)	427.17 ms (2.65×)

Note: LCPC code; LCPC runs Ft127. Baseline cells show time (speedup = baseline time / RBP-LCPC over Mont127 time). Hash = hash time / commit time.
†: the time differs by more than 5% between sessions.

It verifies 2.65–5.36× faster than LCPC-default and 2.63–5.36× faster than LCPC-tuned. Verifier-side encoding remains at most 0.33% of verify time (CSV T7). The prove and verify gains stem from lazy-reduction linear combinations, tiled deduplicated column openings, batched leaf hashing and batched verifier encoding (Section 3.6).

Running RBP-LCPC over Ft127 at $N \in \{2^{20}, 2^{24}\}$ (Table 10) confirms that these speedups do not depend on changing the field type. Because Ft127 delegates lazy accumulation to Mont127's kernels, commit and prove times match Mont127 closely: prove is within 0.99–1.00× of Mont127 (1.85–5.67× faster than LCPC-default). Verify over Ft127 remains 1.83–2.26× faster than LCPC-default, though 1.38–2.92× slower than over Mont127. RBP-LCPC defaults to Mont127 because it implements all arithmetic in safe Rust within `lcpc-fields`, whereas Ft127 resides in `lcpc-test-fields`, derives eager arithmetic via `ff_derive` and delegates lazy accumulation to Mont127 through `unsafe` transmutation.

Separating one-time PCS setup from benchmark input generation shows that RBP-LCPC's PCS setup is 1.31–1.44× faster than LCPC-default (1.18–1.30× faster than LCPC-tuned) at $T = 1$ (Table 11), and verifier initialization is 1.48–1.53× faster (CSV T8). PCS setup takes at most 0.03× of a single encoding (versus 0.00× for LCPC) and is paid once

Table 11: One-time setup and peak memory on Zen5

	PCS setup			Input gen.		Peak RSS (MB)	
$\log_2 N$	RBP-LCPC	/ encode	LCPC-default (speedup)	RBP-LCPC	LCPC-default	RBP-LCPC	LCPC-default
20	1.69 ms	0.03	2.22 ms (1.31×)	30.42 ms	18.21 ms	362.1	330.7
24	5.44 ms	0.01	7.49 ms (1.38×)	505.25 ms	403.02 ms	2378.6	2346.6
28	20.20 ms	0.00	29.14 ms (1.44×)	8147.19 ms	6550.12 ms	34434.6	34186.2

Note: $T = 1$, LCPC code. PCS setup = graph sampling and encoder initialization (speedup = LCPC-default/ RBP-LCPC); verifier initialization (CSV T8) is comparable. Input gen. = benchmark coefficient sampling and row-buffer initialization. Peak resident set size (RSS) = `VmHWM`, maximum over sessions.

per code instance. By contrast, benchmark input generation at $T = 1$ takes 1.24–1.67× the time of LCPC-default because RBP-LCPC samples polynomial coefficients with its portable scalar ChaCha20 generator whereas LCPC-default uses `rand_chacha`'s SIMD backend; because input generation dominates the combined harness setup, total setup is 1.23–1.48× that of LCPC-default. Peak resident set size (RSS) differs by 0.7–9.5% between the two implementations (0.7% at $N = 2^{28}$).

6 Discussion

Our measurements motivate four practices for evaluations comparing a new PCS or SNARK with Brakedown.

(i) *Compare against an optimized portable baseline across all PCS operations.* Speedups over LCPC-default include the 8.14–9.06× single-threaded encoding gap (Section 5.1), as well as 1.83–7.61× for prove, 2.65–5.36× for verify and 1.31–1.44× for one-time PCS setup (Section 5.5).

(ii) *Report the build profile, compiler flags and hash backend.* On the same LCPC source, build settings change encoding time by 1.18–1.19×, and the BLAKE3 backend changes hashing time by about one fifth (Sections 5.1 and 5.5).

(iii) *Report encoding and hashing separately.* Once encoding is optimized, hashing accounts for more than a third of single-threaded commit time (Section 5.5). Reporting only total commit time obscures which component a new technique improves.

(iv) *Report the thread count and code instantiation.* Speedups over LCPC's row-parallel encoder diminish with T, and different code instantiations perform different amounts of work. Time per edge-row MAC enables fair comparison across codes (Section 5.3).

7 Related Work

Hash- and code-based PCSs. Unlike pairing-based commitments such as KZG [KZG10], hash-based PCSs built from linear codes have a transparent setup [Tha22]. One family relies on code foldability for polylogarithmic proofs and verification. FRI [BSBHR18] folds RS codewords over smooth subgroups, and STIR [ACFY24] reduces FRI's query complexity by shifting evaluation domains to improve the code rate across rounds. WHIR [ACFY25] tests constrained RS codes to support multilinear queries with sub-millisecond verification. Sumcheck-coupled folding yields BaseFold [ZCF24] over field-agnostic foldable codes and DeepFold [GLH+25] over RS codes. Over binary tower fields, Binius [DP25] and FRI-Binius [DP26] adapt interleaved and FRI-based commitments, while Ligerito [NA25] recursively folds matrix–vector claims with partial sumchecks. A second family follows

Ligero [AHIV17] and Brakedown [GLS+23] by encoding rows of a coefficient matrix with a fast code, trading $O(\sqrt{N})$ proof size for linear or near-linear prover time. Brakedown obtains a field-agnostic linear-time prover from a Spielman-style code (its RS variant is Shockwave). Orion [XZS22] composes a Spielman code with proof recursion, and den Hollander and Slamanig [dS25] restore its soundness; the field Fp2M61 comes from the Orion reference software [Sun]. Subsequent schemes instantiate this template with expand-accumulate codes [BFK+24], quasi-Abelian codes [LLX+26], RS codes [CCC+25] or Spielman codes [YZRM24]. A third family applies code switching to combine fast encoding with polylogarithmic verification. Blaze [BCF+25] switches from repeat-accumulate-accumulate codes over binary fields, BrakingBase [NST25] from Brakedown codes, TensorSwitch [BFRW26] from tensor codes, Quasar [JLX+26] from quasi-Abelian codes, and SwitchFold [CYY26] recursively across arbitrary linear codes.

Accelerating code-based commitments. Hirner et al. [HKPS25] introduce R2L evaluation and accelerate Orion on an FPGA with high-bandwidth memory. With 64 threads, SIMD-Spielman reaches up to 4.3× the FPGA's throughput for $N \leq 2^{24}$ [KDHS26a]. Its authors note that the verifier, which must also encode, often lacks access to such accelerators. For RS codes, they scale published Intel HEXL NTT latencies [BKS+21] from 50-bit to 128-bit moduli and estimate an overhead of 1.15× to 1.44× for Spielman encoding. RBP-LCPC is closest to SIMD-Spielman but uses portable scalar code inside a complete PCS. Because it extends LCPC [Wah21], it can be checked for bit-exact agreement with one of SIMD-Spielman's baselines.

CPU optimization, vectorization and hashing. Optimizing core cryptographic primitives on general-purpose CPUs accelerates full protocols, as ENG25519 [ZHZ+24] demonstrates for TLS 1.3 handshakes via X25519 and Ed25519. For zero-knowledge proofs, MamaBearZKP [ZGL+26] co-designs prime fields and proving stacks for high-throughput execution on modern CPUs, complementing our focus on portable scalar encoding over existing fields. Vector extensions further accelerate homomorphic encryption [BKS+21], Dilithium [XHL+24], SIKE [CFGR22], SPHINCS+ [ZRW25], multi-scalar multiplication [JPL+25] and NTTs on RISC-V [RRG+25]. Achieving vectorized performance across architectures requires distinct backends per instruction set, as Zhang and Zhang [ZZ26] illustrate for Falcon-Sign across SSE2, AVX2, AVX-512F, NEON and RVV. Dedicated hardware also exists for NTTs [CYY+21] and Falcon signing [OZZ+24]. At the instruction-set level, VAXEN [ZCLZ26] proposes an AVX extension for zero-knowledge proofs and post-quantum cryptography. Even with existing AVX-512 extensions, Krieger et al. [KDHS26a] note that observed speedups stay well below the 8-lane width [JPL+25, CFGR22, GBB20]. Vectorized Merkle hashing, such as the `hashtree` library [Off25], is directly relevant to RBP-LCPC, where hashing takes more than a third of the single-threaded commit time. For slicing, the LLC size can be read from the operating system [Ell20]. Under non-uniform cache access, local LLC slices have lower latency than remote ones [FRMK19, RHT+25], an effect RBP-LCPC does not model.

Benchmarking and compilers. Frameworks such as eBACS [BL] and pqm4 [KRSS19] standardize measurement procedures for cryptographic primitives. Mytkowicz et al. [MDHS09] and Hoefler and Belli [HB15] document how an uncontrolled setup or an incomplete report distorts a comparison. Recent hash-based PCS papers, including BaseFold [ZCF24], expand-accumulate SNARKs [BFK+24], WHIR [ACFY25], Blaze [BCF+25], Quasar [JLX+26] and SwitchFold [CYY26], evaluate against Brakedown, making its baseline implementation relevant beyond Brakedown itself. Maleki et al. [MGG+11] show that compilers auto-vectorize only a fraction of the loops that can be vectorized by hand. CryptOpt [KEG+23] reports that field arithmetic found by program search outperforms compiler output.

8 Conclusion

RBP-LCPC implements portable scalar Brakedown encoding and PCS operations within LCPC and matches LCPC bit-for-bit over its Spielman code and prime. It encodes several times faster than LCPC on both x86-64 and AArch64, and also accelerates commit, prove, verify and one-time PCS setup. On a logarithmic scale, it closes most of the gap from LCPC to the AVX-512 IFMA encoder SIMD-Spielman, which remains faster up to $N = 2^{26}$ (Section 5). Benchmarks against Brakedown should therefore include an optimized portable baseline and report each baseline's build settings, hash backend and thread count (Section 6). Future work includes accelerating portable Merkle hashing, scaling beyond a single L3 domain and attributing the remaining gap to AVX-512 with hardware performance counters.

Code and Data Availability

RBP-LCPC is derived from LCPC [Wah21]. It retains the Apache-2.0 license, original copyright notices and full revision history; a `NOTICE` file records every modified, removed or added file. The bundled FFT implementation is adapted from the `fffft` crate [Wah20]. The artifact repository at `https://github.com/Ji-Peng/Real-Brakedown-Performance` provides the complete RBP-LCPC implementation (primary campaign at commit `bd2718c`; archived V2 session at earlier revision `375a6e1`), its test suite, reference oracles and reproduction scripts for all baselines and measurements. It also contains the raw measurement records and analysis scripts that regenerate every table and figure in this paper.

Reproduction workflow and checks. Scripts under `reproduce/` execute each implementation, the full evaluation suite and the Slurm jobs. The incremental stage tags `step/00-import` to `step/09-codegen` map to commits `2badf67`, `b05b33f`, `6ab6bb8`, `eae56d1`, `650763a`, `25c2248`, `643d38b`, `8567b53`, `285c01c` and `9f15bc2`. Running `python3 scripts/analyze.py` with `--selection results/selection.json` and `--check` regenerates every CSV, table and figure from raw records; `paper/scripts/paper_tables.py` then formats the paper tables and derived ratios with their formulas and source cells. The file `results/selection.json` lists selected sessions and all exclusion or omission reasons. The `--check` mode rejects duplicate or mixed cell records, thread counts exceeding assigned cores, unselected or incomplete runs, hot-kernel vector arithmetic, missing rows, empty or non-numeric cells and non-compiling tables.

Sessions and provenance. Primary executables on Zen5 and N1 (collected on 2026-11-22) are compiled from RBP-LCPC commit `bd2718c7698c` (with at most uncommitted cluster first-run records outside any Rust, Cargo or toolchain file). The archived V2 session (collected on 2026-10-23) is compiled from the earlier revision `375a6e18355a` (with uncommitted documentation and script edits only), prior to the PCS-level and threaded-encoder optimizations. Each benchmark writes raw JSON records and an environment manifest (CPU, cache hierarchy, OS, kernel, compilers, flags and source commits, plus Slurm job, hostname, load and frequency on the cluster). Each CSV header records the selection file's SHA-256 and each session's output directory, source commit and host.

Divergent and omitted cells. Of 580 two-session cells, 9 diverge by over 5% (`session_divergence.csv`), including 1 RBP-LCPC cell; the largest difference is 39.8% overall and 8.0% for RBP-LCPC, while 36 one-session cells lack a range. All 30 omitted cells (`todo_cells.csv`) are SIMD-Spielman Orion-shape cells at $N = 2^{28}$, where single-threaded graph generation exceeds 90 minutes per process.

Acknowledgements

We thank Florian Krieger and Sujoy Sinha Roy for helpful discussions. We also thank the authors of LCPC and of SIMD-Spielman for publishing their code, which made the comparisons in this paper possible.

AI-Tool Disclosure

Generative AI tools assisted with software implementation and with drafting and editing this manuscript. The authors verified all code, experiments, data and text, and take full responsibility for the contents of this work.

References

[ACFY24] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. STIR: Reed-Solomon proximity testing with fewer queries. In *Advances in Cryptology – CRYPTO 2024*, LNCS, pages 380–413. Springer, 2024.

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. WHIR: Reed-Solomon proximity testing with super-fast verification. In *Advances in Cryptology – EUROCRYPT 2025*, LNCS, pages 214–243. Springer, 2025.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS 2017*, pages 2087–2104. ACM, 2017.

[BCF+25] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D. Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. In *EUROCRYPT 2025*, LNCS, pages 123–152. Springer, 2025.

[BFK+24] Alexander R. Block, Zhiyong Fang, Jonathan Katz, Justin Thaler, Hendrik Waldner, and Yupeng Zhang. Field-agnostic SNARKs from expand-accumulate codes. In *Advances in Cryptology – CRYPTO 2024*, LNCS, pages 276–307. Springer, 2024. Cryptology ePrint Archive, Paper 2024/1871.

[BFRW26] Benedikt Bünz, Giacomo Fenzi, Ron D. Rothblum, and William Wang. TensorSwitch: Nearly optimal polynomial commitments from tensor codes. In *Advances in Cryptology – CRYPTO 2026*, LNCS, pages 297–329. Springer Nature Switzerland, 2026.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, pages 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, pages 14:1–14:17, 2018.

[CCC+25] Ignacio Cascudo, Anamaria Costache, Daniele Cozzo, Dario Fiore, Antonio Guimarães, and Eduardo Soria-Vazquez. Verifiable computation for approximate homomorphic encryption schemes. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025*, pages 643–677, Cham, 2025. Springer Nature Switzerland.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):41–68, February 2022.

[CHL+24] Alessandro Chiesa, Yuncong Hu, William Lin, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Poly-commit: A Rust library for polynomial commitments. https://github.com/arkworks-rs/poly-commit/tree/master/, 2024. arkworks; this paper benchmarks commit 1eaec50.

[CYY+21] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. CFNTT: Scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):94–126, November 2021.

[CYY26] Mingshu Cong, Tsz Hon Yuen, and Siu-Ming Yiu. SwitchFold: Code-agnostic succinct polynomial commitments via recursive code switching. Cryptology ePrint Archive, Paper 2026/1489, 2026.

[DP25] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In *Advances in Cryptology – EUROCRYPT 2025*, Lecture Notes in Computer Science, pages 93–122. Springer Nature Switzerland, 2025. Cryptology ePrint Archive, Paper 2023/1784.

[DP26] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. In *Advances in Cryptology – EUROCRYPT 2026*, volume 16547 of *LNCS*, pages 3–32. Springer, 2026.

[dS25] Thomas den Hollander and Daniel Slamanig. A crack in the firmament: Restoring soundness of the Orion proof system and more. In *Advances in Cryptology – ASIACRYPT 2025*, Lecture Notes in Computer Science, pages 396–426. Springer Nature Singapore, 2025.

[Ell20] Daniel Ellis. Obtaining CPU cache size. https://medium.com/cemac/obtaining-cpu-cache-size-bafd9607e61a, 2020. Accessed on December 2025.

[FRMK19] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the most out of last level cache in Intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[GBB20] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for AVX-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR '20, pages 25–35, New York, NY, USA, 2020. Association for Computing Machinery.

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

[GLS+23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023 – 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II*, volume 14082 of *Lecture Notes in Computer Science*, pages 193–226. Springer, 2023. Cryptology ePrint Archive, Paper 2021/1043.

[HB15] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In *SC 2015*. ACM, 2015.

[HKPS25] Florian Hirner, Florian Krieger, Constantin Piber, and Sujoy Sinha Roy. Accelerating hash-based polynomial commitment schemes with linear prover time. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):341–385, September 2025.

[JLX+26] Yuhao Jia, Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. Quasar: A field-agnostic polynomial commitment scheme with polylogarithmic verification from quasi-abelian codes. Cryptology ePrint Archive, Paper 2026/1839, 2026.

[JPL+25] Rui Jiang, Cong Peng, Min Luo, Rongmao Chen, and Debiao He. SIMDMSM: SIMD-accelerated multi-scalar multiplication framework for zkSNARKs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):681–704, March 2025.

[KDHS26a] Florian Krieger, Christian Dobrouschek, Florian Hirner, and Sujoy Sinha Roy. High-performance SIMD software for Spielman codes in zero-knowledge proofs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):167–191, 2026.

[KDHS26b] Florian Krieger, Christian Dobrouschek, Florian Hirner, and Sujoy Sinha Roy. SIMD-Spielman-for-ZKPs (source code). https://github.com/flokrieger/SIMD-Spielman-for-ZKPs, 2026. SIMD-Spielman-for-ZKPs; this paper uses commit 7b399db (MIT license).

[KEG+23] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. CryptOpt: Verified compilation with randomized program search for cryptographic primitives. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1268–1292, 2023.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2019/844, 2019.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. *Lecture Notes in Computer Science*, 6477:177–194, 2010.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to Reed-Solomon erasure codes. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 316–325, 2014.

[LLX+26] Zhe Li, Hongqing Liu, Chaoping Xing, Yizhou Yao, and Chen Yuan. More efficient SNARKs via quasi-abelian codes: Faster, smaller, and field-agnostic. Cryptology ePrint Archive, Paper 2026/939, 2026.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS 2009*, pages 265–276. ACM, 2009.

[MGG+11] Saeed Maleki, Yaoqing Gao, María J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *PACT 2011*, pages 372–382. IEEE, 2011.

[NA25] Andrija Novakovic and Guillermo Angeris. Ligerito: A small and concretely fast polynomial commitment scheme. Cryptology ePrint Archive, Paper 2025/1187, 2025.

[NST25] Vineet Nair, Ashish Sharma, and Bhargav Thankey. BrakingBase – a linear prover, poly-logarithmic verifier, field agnostic polynomial commitment scheme. In *Advances in Cryptology – ASIACRYPT 2025*, LNCS, pages 427–455. Springer Nature Singapore, 2025.

[OANWO20] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3: One function, fast everywhere. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf, 2020.

[Off25] OffchainLabs. hashtree. https://github.com/OffchainLabs/hashtree, 2025.

[ON] Jack O'Connor and Samuel Neves. blake3: Rust crate, version 1.8.7. https://github.com/BLAKE3-team/BLAKE3. Built with features pure, no_sse2, no_sse41, no_avx2, no_avx512.

[OZZ+24] Yi Ouyang, Yihong Zhu, Wenping Zhu, Bohan Yang, Zirui Zhang, Hanning Wang, Qichao Tao, Min Zhu, Shaojun Wei, and Leibo Liu. FalconSign: An efficient and high-throughput hardware architecture for Falcon signature generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):203–226, December 2024.

[RHT+25] Mikka Rainer, Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Leon Trampert, Lukas Gerlach, and Michael Schwarz. Rapid reversing of non-linear CPU cache slice functions: Unlocking physical address leakage. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3497–3515, 2025.

[RRG+25] Tiago B. Rodrigues, Alexandre Rodrigues, Manuel Goulão, Pedro Tomás, and Leonel Sousa. Accelerating NTT with RISC-V vector extension for fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):711–736, September 2025.

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42:1723–1731, 1996.

[Sun] Sunblaze-ucb. sunblaze-ucb/orion. https://github.com/sunblaze-ucb/Orion. Accessed on October 2025.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security*, 4(2-4):117–660, 2022.

[Wah20] Riad S. Wahby. fffft: Rust crate, version 0.4.2. https://crates.io/crates/fffft, 2020. Apache-2.0; upstream LCPC's FFT dependency of lcpc-ligero-pc.

[Wah21] Riad S. Wahby. lcpc: Polynomial commitment scheme from any linear code with sufficient minimum distance. https://github.com/conroi/lcpc, 2021. Apache-2.0; this paper forks and benchmarks commit f0e2116.

[XHL+24] Runqing Xu, Debiao He, Min Luo, Cong Peng, and Xiangyong Zeng. Optimizing Dilithium implementation with AVX2/-512. *ACM Trans. Embed. Comput. Syst.*, 23(6), September 2024.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, volume 13510 of *LNCS*, pages 299–328. Springer, 2022. Cryptology ePrint Archive, Paper 2022/1010.

[YZRM24] Xiao Yang, Chengru Zhang, Mark Ryan, and Gao Meng. Multivariate multi-polynomial commitment and its applications. Cryptology ePrint Archive, Paper 2024/827, 2024.

[ZCF24] Hadas Zeilberger, Binyi Chen, and Ben Fisch. BaseFold: Efficient field-agnostic polynomial commitment schemes from foldable codes. In *CRYPTO 2024*, LNCS, pages 138–169. Springer, 2024.

[ZCLZ26] Jipeng Zhang, Hao Cheng, Tao Lu, and Jiaheng Zhang. VAXEN: A versatile AVX extension for zero-knowledge proof and post-quantum cryptography. In *Proceedings of the 59th IEEE/ACM International Symposium on Microarchitecture, MICRO 2026, Athens, Greece, October 31 – November 4, 2026*. IEEE, 2026.

[ZGL+26] Jipeng Zhang, Yanpei Guo, Tao Lu, Hao Cheng, and Jiaheng Zhang. MamaBearZKP: A holistic co-design of prime fields and proving stacks for high-throughput ZKP on modern CPUs. Cryptology ePrint Archive, Paper 2026/1698, 2026. Accepted to ACM CCS 2026.

[ZHZ+24] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024*, pages 6381–6398, Philadelphia, PA, USA, August 2024. USENIX Association.

[ZRW25] Yaoyun Zhou, Kavin Rajasekaran, and Qian Wang. Exploring parallel implementation of SPHINCS+ using advanced vector extensions (AVX) sets. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*, pages 1–8, 2025.

[ZZ26] Jipeng Zhang and Jiaheng Zhang. Vectorized Falcon-Sign implementations using SSE2, AVX2, AVX-512F, NEON, and RVV. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026:302–324, 2026.