AVXPoS: Reducing Consensus Verification Cost in the Ethereum Proof-of-Stake Client

Ganqin Liu[1], Hao Cheng[1,2], Georgios Fotiadis[3], Jipeng Zhang[4], and Chen Qian[1,2]*

[1]*School of Cyber Science and Technology, Shandong University, Qingdao, China*
[2]*State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China*
[3]*Digital Security and Trusted Computing Group, Ubitech Ltd., Athens, Greece*
[4]*National University of Singapore, Singapore*

ganqin.liu@mail.sdu.edu.cn hao.cheng@sdu.edu.cn gfotiadis@ubitech.eu
jp-zhang@outlook.com qian.chen@sdu.edu.cn

Abstract

Ethereum Proof-of-Stake (PoS) clients must verify large volumes of Boneh–Lynn–Shacham (BLS) signatures for attestations, sync-committee messages, and other consensus-critical objects within fixed slot deadlines. This recurring cost competes with state transition, fork choice, and message propagation for client CPU time, so reducing it increases the verification headroom available under bursty load. Prior cryptographic-engineering work has shown that SIMD can substantially accelerate BLS verification kernels, but these gains do not automatically survive client software boundaries, runtime scheduling, and irregular verification ranges.

We present AVXPoS, a client-aware batching framework for BLS verification in the Prysm Ethereum PoS client. AVXPoS treats batched verification as a client-level systems problem: it preserves Prysm's verification semantics while reorganizing protocol-shaped requests into native batched states that expose SIMD parallelism across API, worker, and native-backend boundaries. We instantiate AVXPoS with an AVX-512 backend for BLS12-381, combining Go-side range formation with C-side width-adaptive dispatch. On a resource-constrained two-core Intel host, AVXPoS gains 1.44–2.10× over Prysm's production `blst` backend at selected small batch sizes that bracket the post-aggregation $p50/p95/p99$ batch-size quantiles of an all-subnets steady-state mainnet stress trace, and reaches up to 3.18× in controlled capacity sweeps. On a 16-core AMD host, a production checkpoint-backfill verifier at Ethereum's 128-block request cap improves by 1.81×. Cross-platform results indicate that the relative speedup depends in part on Prysm's worker budget, because worker partitioning determines how much SIMD parallelism remains within each native range.

1 Introduction

Ethereum PoS consensus clients continuously ingest, validate, and propagate signed protocol traffic from a large validator set under fixed slot deadlines. They must perform this work while also participating in fork choice, block processing, state transition, and attestation handling [23]. Signature verification is therefore a recurring client-software cost rather than only a backend implementation detail: time spent on it reduces the headroom available for other consensus-critical work under bursty load.

Ethereum relies on the Boneh–Lynn–Shacham (BLS) signature scheme [13] over the BLS12-381 pairing-friendly elliptic curve [6, 15] because BLS supports compact, non-interactive aggregation of validator signatures. This saves network and storage bandwidth, but shifts substantial verification work to the receiving node. On hot paths such as gossip validation, block processing, and epoch transitions, a production client performs point decompression, hash-to-curve [43], subgroup membership tests [11, 32], randomized or aggregate verification [3], and pairing checks [42] before accepting the corresponding messages or blocks.

Motivation. The repeated attention given by the Ethereum community to faster verification of multiple BLS signatures and secure batch verification techniques highlights that verification performance is a practical systems bottleneck for beacon-node operation, rather than a purely theoretical optimization concern [3, 17]. Faster verification therefore increases the timing margin available to the client, rather than merely improving an isolated library microbenchmark.

Prior cryptographic-engineering work accelerates BLS12-381 arithmetic, pairing computation, and signature verification with SIMD. Single-instance kernels often fill SIMD lanes by permuting values between formula-specific layouts, which raises the instruction cost of each arithmetic formula [18, 33]. Cross-instance kernels can reduce this layout overhead when each native call retains enough independent work. A production client, however, reaches these kernels only after selecting an API, collecting requests, partitioning work across runtime workers, and translating data across language and library boundaries. Crucially, client threads and cross-instance SIMD draw from the same pool of independent verification work: partitioning a logical batch across more workers can

*Corresponding author.

leave each native call too narrow to fill the vector backend. The systems problem is therefore not only how to vectorize BLS arithmetic, but how to preserve enough independent verification instances after API selection and worker partitioning for those kernels to reduce complete client-side verification cost.

Challenges. AVXPoS must satisfy two constraints. First, because production clients such as Prysm partition verification work across Go workers before invoking the native BLS backend, a large logical request can reach the backend as several narrow ranges, reducing SIMD occupancy even when abundant work exists at the client level. Conversely, forcing full-width SIMD execution on irregular ranges wastes lanes or creates expensive fallback paths. The backend must exploit the range shape that survives client scheduling rather than assume ideal full batches.

Second, recovering this parallelism must not change the client behavior that formed those ranges. Prysm already batches verification requests before the native backend boundary, so introducing another collection stage would trade cryptographic throughput against additional queueing latency. Likewise, regrouping work must not change randomized-batch coefficients, failure propagation, fallback behavior, or fault isolation. The integration must therefore recover SIMD parallelism from already-scheduled work while preserving the client's existing batching and verification contract.

Empirical motivation. In an all-subnets steady-state mainnet stress trace (Section 6.2), batch sizes entering Prysm's equal-message aggregation have $p50/p95/p99$ quantiles of $12/85/162$, whereas those reaching the native VerifyMultipleSignatures boundary retain only $1/3/7$. This contraction shows that Prysm's client pipeline can limit native SIMD width even when incoming work is abundant. Worker partitioning imposes a second limit: the same $n = 32$ call gains $1.39\times$ under a 32-thread budget and $3.39\times$ under a four-thread budget, both over Prysm's production `blst` backend on the same host (Section 6.4). Client-side aggregation and worker partitioning therefore determine how much cross-instance parallelism reaches the native backend.

Contributions. We present AVXPoS, a client-aware batching framework instantiated in Prysm, a production Ethereum PoS consensus client implemented primarily in Go [35]. We make the following contributions:

- We identify and empirically characterize client-side aggregation and worker partitioning as determinants of the cross-instance SIMD width that reaches a production BLS backend, and design a client-aware batching framework that preserves Prysm's existing batching, fallback, and verification semantics while recovering SIMD width below the worker boundary.
- We turn prior AVX-512 BLS12-381 building blocks into staged multi-instance verification operators with stage-local width selection, allowing irregular client-formed ranges to retain SIMD efficiency rather than requiring ideal full-width batches.
- We evaluate AVXPoS from native verification kernels through complete Prysm verification APIs on a commodity two-core Intel Ice Lake host and a 16-core AMD Zen 5 host. At small batch sizes that bracket the post-aggregation quantiles of an all-subnets steady-state mainnet stress trace, AVXPoS gains 1.44–2.10× over Prysm's production backend, reaches up to 3.18× in a controlled multiple-signature capacity sweep, and improves the mainnet checkpoint-backfill verifier by 1.81× at the protocol's 128-block request cap. Cross-platform worker experiments show that the native range width surviving client partitioning strongly influences the realized speedup.

2 Background

2.1 BLS signature verification

Ethereum instantiates the BLS signature scheme [13] over the BLS12-381 pairing-friendly curve [15] in the *minimal-public-key-size* mode. Public keys are elements of $\mathbb{G}_1 \subset E_1(\mathbb{F}_p)$ and signatures are elements of $\mathbb{G}_2 \subset E_2(\mathbb{F}_{p^2})$, where $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/\langle i^2+1\rangle$. Compressed public keys and signatures occupy 48 and 96 bytes, respectively [11]. Verification relies on the optimal ate pairing [42] $e : \mathbb{G}_2 \times \mathbb{G}_1 \to \mathbb{G}_T$ over the BLS12-381 curve (see e.g., [33] for details).

For a message M, public key $\mathrm{pk} \in \mathbb{G}_1$, and signature $\sigma \in \mathbb{G}_2$, verification checks

$$e(\sigma, P_1) \stackrel{?}{=} e(H(M), \mathrm{pk}),$$

where P_1 is the generator of $\mathbb{G}_1$ and H is the standardized hash-to-curve map [25]. The pairing consists of a Miller loop followed by final exponentiation [42]. Rather than evaluating the two pairings independently, a verifier combines their Miller-loop outputs and performs one final exponentiation [26, 42].

From an implementation perspective, this equation defines a multi-stage verification pipeline rather than a single pairing call. Inputs received in compressed form must be decompressed and checked for subgroup membership; the message is mapped to $\mathbb{G}_2$ through hash-to-curve; and the resulting points enter the pairing computation. These stages appear in production libraries such as `blst` [41] and constitute the verification work that AVXPoS later reorganizes across independent instances. Subgroup checks are required to reject points outside the intended prime-order subgroups [11, 32].

Aggregation. BLS supports non-interactive aggregation of signatures. Let $\sigma_1, \ldots, \sigma_N \in \mathbb{G}_2$ be signatures on messages $M_1, \ldots, M_N$ under public keys $\mathrm{pk}_1, \ldots, \mathrm{pk}_N \in \mathbb{G}_1$. Their aggregate $\sigma_{\mathrm{agg}} = \sum_{i=1}^{N} \sigma_i$ satisfies

$$e(\sigma_{\mathrm{agg}}, P_1) \stackrel{?}{=} \prod_{i=1}^{N} e(H(M_i), \mathrm{pk}_i).$$

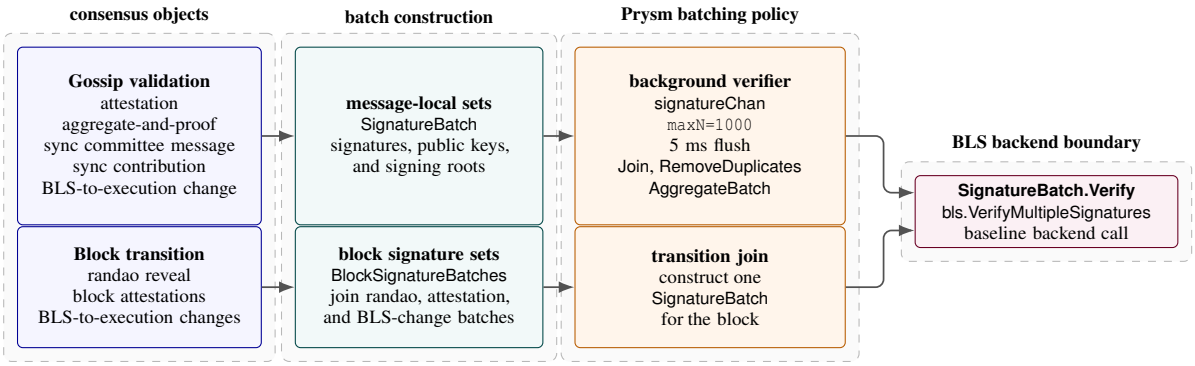


Figure 1: Prysm's consensus-layer path from Ethereum messages to SignatureBatch verification.

Table 1: Consensus-facing verification paths used throughout the paper. Names follow Prysm's `crypto/bls` API; Verify is the single-signature Signature.Verify method.

Path	Input shape	Representative use	AVXPoS target
Verify	one key/message/signature	one-shot checks, diagnosis	narrow pairing path
FastAggregateVerify	many keys, one message	sync committee, aggregates	key aggregation, final check
VerifyMultipleSignatures	many keys and messages	gossip, transition, backfill	ranges, multi-instance pairing

The product form is important computationally: the verifier can accumulate multiple Miller-loop outputs and complete the equation with a single final exponentiation instead of one final exponentiation per pairing.

Same-message aggregation. When all signatures authenticate the same message M, bilinearity reduces the right-hand side to the equivalent equation:

$$e(\sigma_{\text{agg}}, P_1) \stackrel{?}{=} e\left(H(M), \sum_{i=1}^{N} \text{pk}_i\right).$$

Ethereum exposes this form through FastAggregateVerify [11]. Its batchable work therefore lies primarily in public-key aggregation; once the keys have been reduced to one aggregate public key, only one pairing equation remains.

Randomized batch verification. For signatures on independent messages, Ethereum also uses randomized batch verification [3, 17]. Given signatures σ_i, public keys pk_i, messages M_i, and random coefficients r_i, the verifier checks

$$e\left(\sum_{i=1}^{n} [r_i]\sigma_i, P_1\right) \stackrel{?}{=} \prod_{i=1}^{n} e(H(M_i), [r_i]\text{pk}_i).$$

This form retains per-item work, including hash-to-curve and scalar multiplication, but combines the randomized signature terms and the per-message Miller-loop outputs into one final verification equation. Consequently, it exposes substantially more cross-instance work than the same-message case while still allowing the pairing results to share one final exponentiation. This randomized form underlies the VerifyMultipleSignatures-style path used later in the paper.

2.2 AVX-512

AVX-512 is a family of x86 SIMD extensions with 512-bit vector registers, mask-based predication, and permutation support [28]. Of particular relevance to multi-precision cryptography is AVX-512IFMA, whose 52-bit fused multiply-add instructions have been used in high-performance finite-field and elliptic-curve implementations [18, 19, 28].

AVX-512 can exploit either cross-instance parallelism across independent field elements or curve points, or intra-instance parallelism within one operation. The former is most efficient when enough independent inputs reach the backend; irregular ranges instead require narrower schedules or leave lanes unused. In a consensus client, batching and worker partitioning can reduce this cross-instance width before backend execution.

2.3 Ethereum consensus-layer verification

Ethereum consensus clients continuously verify BLS signatures under a fixed slot schedule, including attestations, aggregate-and-proof objects, and sync-committee messages [23]. Verification therefore competes with networking, state transition, and fork choice for client CPU time; reducing it increases the headroom available within the consensus cadence.

For AVXPoS, the relevant workload is characterized by two verification shapes (Table 1). Same-message aggregates, including aggregated attestations and sync-committee signatures, reduce to public-key aggregation followed by FastAggregateVerify. Mixed-message checks are grouped into SignatureBatch objects and verified through Prysm's VerifyMul-

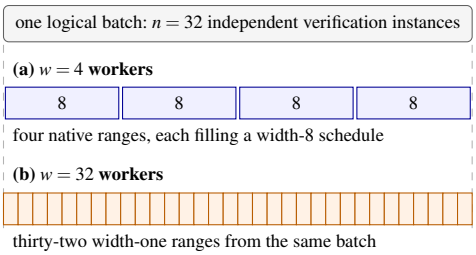


Figure 2: Client worker partitioning and SIMD execution draw from the same logical batch; more worker ranges can reduce the cross-instance width visible to the native backend.

tipleSignatures interface for randomized batch verification. This path covers gossip batches, block-transition signature sets, deposits, and other heterogeneous consensus checks. Single-signature verification remains for one-shot checks and fallback diagnosis, but is not the main high-volume batch path [35].

2.4 Prysm implementation

Prysm is an Ethereum consensus client written primarily in Go [35]. Importantly, it already performs batching and threading before entering the native BLS backend. As shown in Figure 1, gossip validation constructs message-local SignatureBatch sets, buffers them on `signatureChan`, and flushes the buffer when either `batch-verifier-limit` is reached (1000 by default) or a 5 ms timer expires. Prysm then joins buffered sets, removes duplicates, aggregates equal-message entries, and invokes VerifyMultipleSignatures. Block-transition checks construct the same abstraction through BlockSignatureBatches. The `blst` binding further derives its worker count from `runtime.GOMAXPROCS` [35, 41].

The baseline is therefore already a batched, multi-thread-aware production workflow rather than a sequence of isolated scalar verifications. This is the key attachment point for AVXPoS: Prysm retains protocol control, batch formation, and worker scheduling, while AVXPoS operates below the native backend boundary to recover SIMD parallelism from the worker ranges that remain after client-side scheduling.

Prysm's default backend, `blst`, is itself a highly optimized BLS12-381 implementation with production-grade subgroup checks, hash-to-curve, scalar arithmetic, and pairing-state support [27, 36, 38, 41, 43]. AVXPoS therefore targets the organization of complete verification work rather than relying on an unoptimized cryptographic baseline.

3 Framework

Section 1 identified client worker partitioning as the mechanism that limits the cross-instance width reaching a native BLS backend.

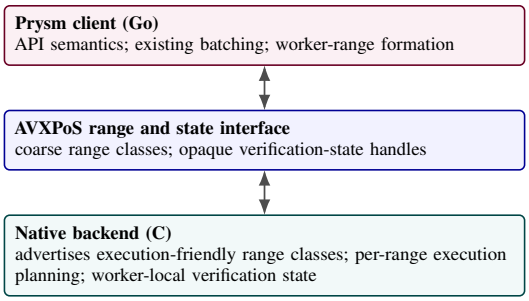


Figure 3: Ownership split in AVXPoS. The client keeps API semantics, its existing batching, and worker-range formation; the native backend advertises execution-friendly range classes and owns per-range execution planning and worker-local verification state.

Parallelism after client partitioning. A logical request of n independent verification instances does not reach the backend as one unit: the client runtime partitions it into worker ranges $R_1, \ldots, R_w$, and each range becomes a separate native call, so that call sees $|R_i|$ instances, about n/w for an even partition, rather than n. Thread-level concurrency and cross-instance SIMD consume the same independent instances, so raising w trades one against the other: Figure 2 contrasts four ranges wide enough to fill a cross-instance schedule with thirty-two single-instance ranges formed from the same batch of $n = 32$. AVXPoS therefore separates client-side range formation from native execution planning.

Separation of responsibilities. The client retains API semantics, the batching it already performs, and the formation of worker ranges. The backend receives one range at a time and owns how that range is executed and what state it produces (Figure 3). Range size is the main scheduling information exchanged across this boundary: the backend advertises coarse range classes it executes efficiently, and the client forms ranges accordingly. The interface therefore carries coarse scheduling preferences without exposing SIMD layouts or stage-specific execution plans; consensus code handles no native verification state, and the backend no protocol objects or worker policy.

Preserving verification semantics. Two invariants constrain the design. First, AVXPoS reorganizes only work that an existing verification call has already admitted: it adds no collection or queueing stage, changes neither call membership nor randomized-batch coefficients, and leaves caller-visible verification and fallback behavior unchanged. Second, worker-local state may be accumulated and merged inside the backend, but the global verification equation is closed at the verification boundary the client already had. Section 5.2 details how these invariants are preserved for randomized verification and failure handling.

Range-local execution and global reduction. AVXPoS does not change the verification equations of Section 2.1;

it changes when the global result is materialized. Each worker processes its range independently against worker-local state that stays mergeable until the original final verification. The two verification shapes of Section 2.3 use different mergeable state: key-aggregation state for FastAggregateVerify and pairing state for VerifyMultipleSignatures. Three operations form the reduction interface: *commit* makes a worker-local state eligible for reduction, *merge* combines committed states, and *finalize* closes the original verification equation.

Section 4 instantiates this framework with a width-adaptive AVX-512 backend, and Section 5 describes its integration into Prysm.

4 Implementation

The AVX-512 instantiation turns the batchable work preserved by AVXPoS into staged multi-instance BLS verification. It builds on prior vectorized BLS12-381 subroutines [18, 33], but extends them from isolated field, curve, pairing, or single-signature operations to complete verification stages that operate on client-formed ranges.

The backend has three layers. Low-level kernels implement finite-field and elliptic-curve operations over the vector layouts and execution widths of the AVX-512 backend. Mid-level routines compose these kernels into verification stages, including point decompression, subgroup checks, hash-to-curve, scalar multiplication, and Miller loop evaluation, and select an execution plan for each active range. High-level state objects then combine these stages into the accumulator and pairing-context interfaces consumed by the Go integration. The following sections describe these layers from vector representation and formula kernels, through verification-stage dispatch, to mergeable native state.

4.1 Low-level vector layout and formulas

SIMD layout. The backend inherits the BLS12-381 representation of [18]: an $\mathbb{F}_p$ element is stored as eight 48-bit Montgomery limbs, matching `vpmadd52` IFMA arithmetic. We use two principal layouts. In 8×1, each register holds the same-indexed limb of eight independent elements; in 4×2, four elements each occupy two lanes to expose intra-formula parallelism. The backend favors 8×1 when enough independent work is available and uses 4×2 or remapped narrower layouts when dependencies limit cross-instance occupancy.

Formula kernels. The low-level kernels cover elliptic-curve point operations, pairing line computations and evaluations, and tower-field arithmetic. Width-8 point kernels use inter-instance formulas, while narrower paths reuse the parallel elliptic-curve-operation constructions of [33]; width-one E_1 doubling is selected at build time between the vector and scalar routines because the faster choice is host-dependent (Appendix D.10). Pairing line kernels use formulas from the Explicit-Formulas Database (EFD) [8] at widths 8 and 4 and the vectorized forms of [18] at width 2, while the tower-field layer provides width-specific squaring and sparse-line multiplication. The exact formulas for elliptic curve operations are summarized in Appendix A.

A recurring design choice is whether a narrow operation should remain in a 4×2 layout or be remapped into an 8×1 schedule. The E_2 doubling example in Figure 4 illustrates this tradeoff: two width-2 doublings are embedded into one full-width schedule by remapping their operands, preserving the underlying formula while aligning independent limb products for the higher-throughput 8×1 layout.

Lazy reduction. Following [33], curve and line kernels defer conditional corrections whenever the enlarged intermediate range remains valid for the next Montgomery multiplication. At the tower-field level we use the double-length technique of [9]. Wider multi-instance schedules expose more independent arithmetic and therefore more opportunities to delay these reductions.

4.2 Mid-level verification kernels

The low-level formulas above are assembled into verification-stage kernels for point decompression, scalar multiplication, subgroup checks, hash-to-curve, and Miller loop evaluation. This layer also owns width dispatch. Fixed-width kernels expose width-$8/4/2/1$ interfaces, while a stage-specific selector maps each active count to either a padded wider kernel, a composition of narrower calls, or a scalar fallback. The profitable choice depends on the stage rather than on the active count alone: for example, a six-item range may use one width-8 call or a width-4 plus width-2 composition. Measured tail schedules are reported in Appendix D.11.

Point decompression. Point decompression is dominated by square-root extraction through a fixed addition chain of squarings and multiplications. AVXPoS evaluates this chain in lockstep across independent inputs at widths 8 and 4, including Montgomery conversion and sign correction. Because the dependency-heavy chain offers too little useful parallelism at narrower widths, width-2 and width-1 decompression use the scalar path.

Scalar multiplication. We provide batched $\mathbb{G}_1$ and $\mathbb{G}_2$ scalar-multiplication kernels at all execution widths using the point addition and doubling kernels of Section 4.1. For the 64-bit random scalars used in randomized batch verification [3], we use batched wNAF [29] with width-specific window parameters selected by the sweep in Appendix D.10.

At each wNAF window, Booth recoding [14] produces the signed digit for every active lane, determining which precomputed multiple is added in that iteration. Rather than performing lane-dependent indexed lookups, the kernel scans the precomputation table row by row and uses AVX-512 masks to select the matching multiple independently in each lane; a

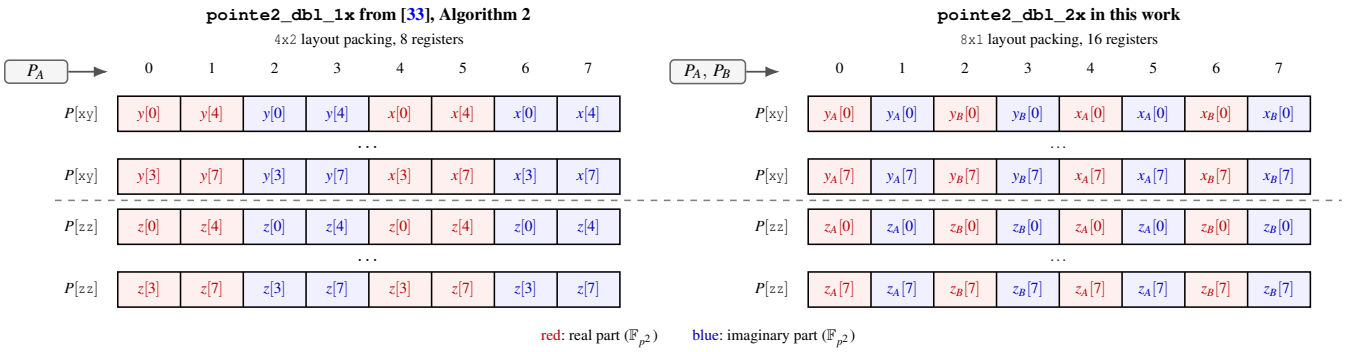


Figure 4: Example of input-layout remapping from `pointe2_dbl_1x` to `pointe2_dbl_2x`. Both kernels implement the same E_2 point doubling interface, but the width-2 version changes the AVX-512 packing of the operands so that two independent point doublings can reuse the same formula under a doubled logical input layout.

second mask applies conditional negation for negative digits. This lets lanes follow different wNAF digits without data-dependent table addresses or control flow.

Subgroup membership testing. We implement Scott's endomorphism-based subgroup tests for $\mathbb{G}_1$ and $\mathbb{G}_2$ [38], including the twisted-Frobenius $\mathbb{G}_2$ endomorphism of [16]. Their fixed-scalar relations are evaluated with the batched elliptic-curve kernels of Section 4.1. Each routine returns a per-lane validity mask; failed active lanes and the point at infinity are rejected before the corresponding range can update committed verification state.

Hash-to-curve. Ethereum PoS hashes messages to $\mathbb{G}_2$ using the RFC 9380 Wahby–Boneh construction [25, 43], as implemented in `blst`. The path consists of byte-oriented preprocessing, `map-to-isogenous-curve`, the isogeny map, and cofactor clearing. The implementation of [33] vectorizes the isogeny and cofactor-clearing stages, but leaves `map-to-isogenous-curve` scalar. This stage is difficult to vectorize within a single message: each message produces only two map inputs, while the dependency-heavy square-root addition chains provide little intra-instance parallelism.

AVXPoS exposes cross-message parallelism instead. If message m_j produces map inputs $u_{j,0}$ and $u_{j,1}$, the backend collects the $u_{j,k}$ values from multiple messages and treats them as independent map instances. These inputs are packed into width-8/4 kernels for `map-to-isogenous-curve`; the resulting points are then regrouped by message and added pairwise before continuing through the isogeny and cofactor-clearing stages. The latter stages reuse the vector layouts and batched point kernels of Section 4.1. This cross-message organization supplies the independent map instances that are absent within a single hash-to-$\mathbb{G}_2$ operation.

Miller-loop. AVXPoS follows the BLS12-381 optimal ate Miller loop and reuses the width-specific point, line, and tower-field kernels of Section 4.1. For a product-of-pairings equation, the backend can accumulate several pairing terms under one loop schedule, as shown in Algorithm 1.

Algorithm 1: Accumulating multi-pairing Miller loop (`Miller_loop_n`). Line values use the BLS12-381 denominator-eliminated representation.

Input: $\{(P_j, Q_j)\}_{j=1}^{N}$ with $P_j \in \mathbb{G}_1$, $Q_j \in \mathbb{G}_2$; the bits $|z| = (b_\ell, \ldots, b_0)$ of the BLS12-381 loop parameter.
Output: Denominator-eliminated Miller accumulator $f \in \mathbb{F}_{p^{12}}^{*}$.

$f \leftarrow 1$; $T_j \leftarrow Q_j$ for $1 \le j \le N$
for $i \leftarrow \ell - 1$ **down to** 0 **do**
  $f \leftarrow f^2$ // one shared $\mathbb{F}_{p^{12}}$ squaring
  for $j \leftarrow 1$ **to** N **do**
    $(T_j, L_j) \leftarrow \mathsf{LineDouble}(T_j, P_j)$
  $f \leftarrow f \cdot \prod_{j=1}^{N} L_j$
  if $b_i = 1$ **then**
    for $j \leftarrow 1$ **to** N **do**
      $(T_j, L_j) \leftarrow \mathsf{LineAdd}(T_j, Q_j, P_j)$
    $f \leftarrow f \cdot \prod_{j=1}^{N} L_j$
if $z < 0$ **then**
  $f \leftarrow \overline{f}$ // negative-loop conjugation
return f

The backend provides two Miller-loop execution modes. In the *accumulating multi-pairing* mode, several (P_j, Q_j) terms share one running Miller state. Their line contributions are multiplied into that state at each iteration, so the accumulated chunk performs one $\mathbb{F}_{p^{12}}$ squaring per loop iteration regardless of the number of pairing terms. This mode matches the product-of-pairings formulation [26] and is used when a verification range is absorbed into a pairing context.

In the *SIMD-parallel* mode, lanes evaluate independent pairing terms of one verification equation under shared control flow while retaining separate Miller-loop states. This mode is used for the two pairing terms of a single-signature check and for the corresponding final check of FastAggregateVerify after public-key aggregation. The lane-local outputs are then reduced with width-specific $\mathbb{F}_{p^{12}}$ multiplication schedules to obtain one Miller product for that equation.

Algorithm 2: C-side range absorption for randomized batch verification.

Input: Pairing context C; affine public keys $P[0:n) \subset \mathbb{G}_1$; compressed signatures $\hat{\sigma}[0:n)$; messages $m[0:n)$; scalars $r[0:n)$.
Output: Updated $C.f \in \mathbb{F}^*_{p^{12}}$ and C.AggrSign, or reject.

foreach *Miller-loop chunk* $R = [i, i+t) \subseteq [0,n)$ **do**
  $b \leftarrow 0$
  allocate $H[t] \subset \mathbb{G}_2$, $P_J[t] \subset \mathbb{G}_1$, $P_A[t] \subset \mathbb{G}_1$
  while $b < t$ **do**
    $q \leftarrow \min(8, t-b)$
    $o \leftarrow i+b$
    $\sigma[0:q] \leftarrow E_2\mathsf{UncompressSelect}(\hat{\sigma}[o:o+q], q)$
    reject if $E_2\mathsf{SubgroupCheckSelect}(\sigma[0:q], q)$ fails
    $S[0:q] \leftarrow E_2\mathsf{ScalarMulSelect}(\sigma[0:q], r[o:o+q], q)$
    $C.\mathsf{AggrSign} \leftarrow C.\mathsf{AggrSign} + \mathsf{PointSum}(S[0:q])$
    $H[b:b+q] \leftarrow \mathsf{HashTo}\mathbb{G}_2\mathsf{Select}(m[o:o+q], q)$
    $P_J[b:b+q] \leftarrow E_1\mathsf{ScalarMulSelect}(P[o:o+q], r[o:o+q], q)$
    $b \leftarrow b+q$
  $P_A[0:t] \leftarrow E_1\mathsf{BatchToAffine}(P_J[0:t])$
  $C.f \leftarrow C.f \cdot \mathsf{BatchMillerLoopSelect}(H[0:t], P_A[0:t], t)$
return ok

This local reduction is distinct from the later pairing-context merge. The former combines lane-local Miller states produced by one SIMD invocation; the latter combines committed worker or chunk states into the global verification state. Width-specific Miller-loop kernels share the same loop structure but use different line and $\mathbb{F}_{p^{12}}$ kernels, with measured schedules for irregular active counts reported in Appendix D.12.

After all Miller states have been reduced, AVXPoS performs one final exponentiation using the optimized single-instance implementation of [18].

4.3 High-level state composition

The top C layer packages the mid-level verification stages into two opaque state objects consumed by the Go bridge. A *point accumulator* represents the worker-local $\mathbb{G}_1$ aggregate used by FastAggregateVerify, while a *pairing context* stores the mergeable state required by VerifyMultipleSignatures, including the accumulated Miller product and randomized signature aggregate. These objects hide SIMD layouts and kernel choices from Go and expose only a small state-management interface for update, commit, merge, finalization, and destruction.

Point accumulators for FastAggregateVerify. The FastAggregateVerify path accumulates affine $\mathbb{G}_1$ public keys into a worker-local projective point. Worker states are combined by projective addition, and only the final merged aggregate is converted back to affine form for the pairing check. This keeps the intermediate representation local to the native backend while allowing Go to reduce opaque worker states.

Pairing contexts for VerifyMultipleSignatures. The randomized batch-verification path uses the fused range routine in Algorithm 2. Each worker range is processed as one or more Miller-loop chunks. Within a chunk, signature validation and randomization, hash-to-curve, and randomized public-key multiplication are performed in subbatches, after which the public-key results are converted to affine form together and absorbed into one Miller product.

The chunk-level affine conversion uses batch inversion (Montgomery's trick), replacing multiple field inversions with one inversion and linear-time multiplications [34]. The resulting affine public keys and hashed message points then enter the accumulating Miller-loop path of Section 4.2. Thus the pairing context retains only the state needed to combine completed chunks rather than exposing stage-local SIMD choices to the integration layer.

Commit, merge, and finalization. Commit materializes the worker-local pairing state into a form that can be combined with other completed workers. Merge multiplies committed Miller products and adds their randomized signature aggregates. Finalization then evaluates the remaining generator–aggregate-signature Miller term for $(\sigma_{\mathrm{agg}}, P_1)$ and applies one final exponentiation to the complete verification equation.

Point accumulators follow the same reduction pattern at a simpler state level: worker-local projective aggregates are merged first, and the resulting public key is materialized in affine form only when the final FastAggregateVerify pairing input is required.

5 Integration

AVXPoS is inserted below Prysm's existing Go verification APIs, leaving batch construction, API selection, and caller-visible failure handling unchanged.

Figure 5 shows the integration after Prysm reaches the VerifyMultipleSignatures backend boundary introduced in Section 2.4. The Go adapter materializes contiguous buffers for compressed signatures, affine public keys, messages, and random scalars. Backend-provided coarse range classes guide worker-range formation, and each worker owns one opaque pairing context while executing the C range-absorption primitive of Algorithm 2. Go then reduces the committed contexts and invokes finalization on the resulting native state. Batch construction and verbose diagnostics remain entirely above this boundary.

5.1 API boundary

Consensus code continues to call Prysm's existing verification interfaces, with AVXPoS installed below them as an alternative backend. We evaluate Verify, FastAggregateVerify, and VerifyMultipleSignatures (Table 1); AggregateVerify remains available for compatibility and specification tests [11], but is not a main beacon-node consensus path.

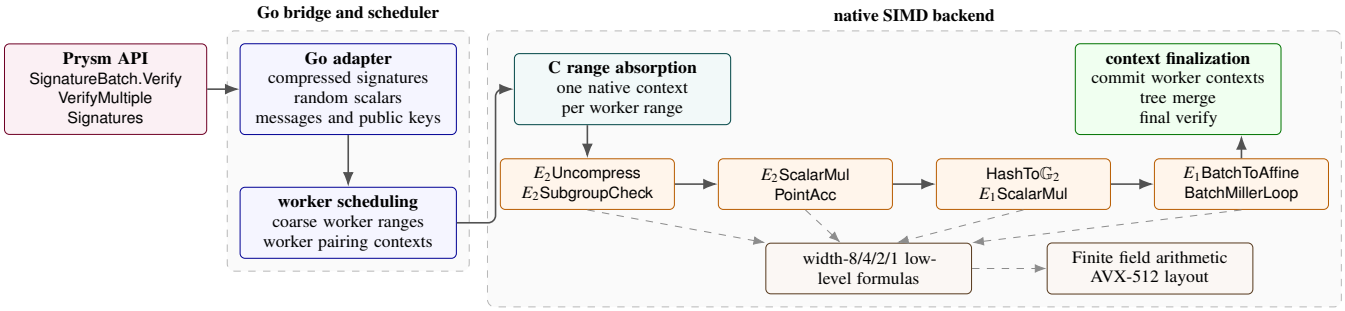


Figure 5: AVXPoS integration path for Prysm's VerifyMultipleSignatures backend call.

Verify. The single-signature path remains a one-shot verification call and does not batch independent Prysm requests. AVXPoS hashes the message to $\mathbb{G}_2$ and maps the two pairing terms

$$e(H(M), \mathrm{pk}) \quad \text{and} \quad e(\sigma, P_1)$$

to the SIMD-parallel Miller-loop mode of Section 4.2. Their lane-local Miller states are reduced to one product before a single final exponentiation closes the verification equation.

FastAggregateVerify. FastAggregateVerify is the same-message aggregate path. The integration layer partitions the public-key set into worker ranges, accumulates affine $\mathbb{G}_1$ keys into worker-local native point accumulators, and merges those states into one aggregate public key. The batchable work is therefore concentrated in public-key aggregation; after this reduction, the two pairing terms of the final verification equation use the SIMD-parallel Miller-loop mode.

VerifyMultipleSignatures. VerifyMultipleSignatures is the mixed-message backend used by SignatureBatch. Prysm samples one 64-bit random scalar per item following [17] and replaces an all-zero draw with a nonzero value; AVXPoS passes these coefficients unchanged. Each worker range enters one C range-absorption call that performs signature validation and randomization, hash-to-$\mathbb{G}_2$, randomized public-key multiplication, and Miller-loop accumulation. Go then merges the committed pairing contexts and invokes finalization on the resulting native state, as shown in Figure 5.

5.2 Security and semantic preservation

AVXPoS operates after Prysm has selected the verification API and admitted the corresponding validator keys. It changes the organization of native execution, but not the caller's trust assumptions, validation rules, or failure policy. The relevant questions are therefore whether randomized-batch soundness, failure handling, key-admission assumptions, and timing discipline are preserved across the new backend boundary.

Threat model and randomized-batch soundness. Inputs to the evaluated verification paths are public and may be adversarially chosen: gossip peers supply signatures, messages, and public keys to SignatureBatch, while block proposers supply the signature sets checked during state transition. Verification uses no secret key material. Randomized batch verification instead requires each coefficient r_i to be fresh and unpredictable when the adversary fixes its inputs [3, 17]. Prysm samples these coefficients after the request is formed; AVXPoS passes them unchanged and neither reorders nor reuses them across calls or worker ranges. The cancellation attack motivating this requirement is summarized in Appendix C.1.

Failure handling and denial of service. A rejected worker range is discarded before its state can be committed or merged, and the failure is returned through Prysm's existing diagnostic path. AVXPoS adds neither a native bad-lane search nor a new verification rule, so an invalid batch enters the same VerifyVerbosely fallback used by the original backend. Consequently, a well-formed invalid signature can still trigger Prysm's $O(n)$ per-item diagnostic scan. Vectorization changes the constant cost of this path, but not its asymptotic behavior or admission and peer-scoring policy; AVXPoS therefore makes no new denial-of-service claim. The inherited failure path is analyzed in Appendix C.2 and measured in Appendix D.5.

Rogue-key handling and proof of possession. AVXPoS preserves Prysm's existing distinction between proof-of-possession-based FastAggregateVerify and randomized VerifyMultipleSignatures. Validator keys entering the same-message path have already passed Ethereum's deposit proof-of-possession process, whereas mixed-message batches retain Prysm's per-item randomization [11, 17, 23]. AVXPoS does not admit validators or replace these checks. The optional raw-public-key aggregation entry point only decompresses and subgroup-checks supplied keys; its additional API details are given in Appendix C.3.

Constant-time discipline. Although the randomized coefficients are not long-lived protocol secrets, the backend conservatively treats coefficient-dependent scalar processing as secret-dependent. Signed-digit selection therefore uses table scans and masks rather than coefficient-dependent branches or memory addresses. Batch size, range boundaries, API dispatch, and error handling remain public control flow. Static branch auditing and statistical timing tests are reported in Section 6.

5.3 Bridge and backend selection

Prysm's Go objects do not directly match the contiguous representation expected by the native backend. The bridge materializes affine $\mathbb{G}_1$ public keys, compressed $\mathbb{G}_2$ signatures, messages, and random scalars into contiguous buffers before entering C. Validator public keys are normally already resident in decoded affine form, so public-key decoding remains outside the range absorption primitive of Algorithm 2; the optional compressed-key entry point is discussed in Section 5.2.

Memory ownership. Transient input buffers remain Go-owned and are reused through `sync.Pool`, while pairing contexts and point accumulators are allocated in C and exposed to Go only as opaque handles. Native states never retain Go memory: cgo calls are synchronous, and input buffers are recycled only after the corresponding call returns [4]. Thus the bridge passes temporary contiguous data into C while keeping persistent verification state entirely on the native side.

Backend selection. AVXPoS is selected only when the AVX backend is requested, the `linux/amd64` native bridge is available, and runtime feature checks confirm the required ISA support and operating-system-enabled vector state. The bridge is compiled against an explicit instruction set rather than `-march=native`, avoiding build-host-specific instruction selection. If any requirement is missing, `PRYSM_BLS_BACKEND=avx` resolves to Prysm's existing `blst` backend before any AVXPoS native entry point is invoked. The same binary can therefore run on heterogeneous hosts, while consensus code remains unaware of AVX-specific dispatch.

5.4 Worker scheduling

Both aggregate paths use worker-local native state followed by a tree reduction, but choose different range policies because their local costs differ. Each worker returns one opaque handle, and Go merges adjacent handles in rounds until one remains, avoiding a serial reduction tail without exposing native layouts.

FastAggregateVerify. Workers in the FastAggregateVerify path accumulate affine public keys into worker-local point accumulators. Because this stage performs relatively light point aggregation before one final pairing check, additional workers are useful only when each receives enough keys to amortize scheduling, cgo, and merge overhead. We therefore enable another worker only when at least 16 public keys are available for it: two width-8 batches provide the operands for one full batched point-accumulator addition rather than leaving the worker with only a single batch to materialize. Additional keys are distributed in width-aligned chunks. The evaluated configuration caps this path at four workers; the worker-sensitivity study underlying this tuning is reported in Appendix D.1.

VerifyMultipleSignatures. VerifyMultipleSignatures workers produce committed pairing contexts rather than point accumulators. Go rounds the per-worker target to backend-provided range classes $1, 2, 4, 8, 12, 16, 24, 32$, followed by larger multiples of eight. These classes reflect the measured tail schedules in Appendix D.11 and Appendix D.12, allowing Go to avoid inefficient worker-range shapes without exposing stage-local SIMD plans. The C backend remains responsible for the concrete decomposition and execution of each range.

The workers are ordinary Go goroutines and are not pinned to operating-system threads; `runtime.GOMAXPROCS(0)` determines the available worker budget [5]. A VerifyMultipleSignatures worker enters one blocking cgo call that absorbs its entire range, whereas FastAggregateVerify workers issue repeated accumulator updates against worker-local native state. End-to-end measurements therefore include goroutine scheduling, cgo transitions, and tree reduction.

Worker parallelism and vector occupancy. Because worker partitioning fixes the range width visible to the native backend (Section 3), Prysm's default $\min(\texttt{GOMAXPROCS}, n)$ policy reduces every range to one item whenever $n \leq \texttt{GOMAXPROCS}$, eliminating cross-instance SIMD within those native calls.

AVXPoS therefore exposes an optional worker cap that preserves at least two items per range in the measured small-batch regime. We do not enable this cap by default because the best worker/SIMD tradeoff depends on both platform and batch size; its measured effect is reported in Table 5.

5.5 Native-state lifecycle

The two native state types from Section 4.3 follow the same ownership discipline. A point accumulator remains worker-local until the public-key reduction produces one affine aggregate, while a pairing context must be committed before it can participate in reduction. Failed workers discard and free their local state; successful merges consume and free the right-hand handle. Final exponentiation is deferred until the global point-accumulator or pairing-context reduction completes, so no worker performs the final verification step independently.

6 Evaluation

Goals and setup. The evaluation asks five questions, one per subsection: whether the range-oriented native backend is effective before any client overhead (Section 6.1); what batch sizes production Prysm actually exposes (Section 6.2); whether the gains survive Prysm's public API boundary at those sizes (Section 6.3); whether client worker partitioning determines native SIMD occupancy and the realized speedup (Section 6.4); and whether a production recovery path improves (Section 6.5). Section 6.6 states the deployment scope of these results. Here, "end-to-end" denotes one complete Prysm verification API invocation through the selected native

backend and back; it excludes network transfer, state transition, fork choice, and the remainder of a slot. Unless identified as a secondary-platform result, experiments use an Intel Core i3-1005G1 (Ice Lake) with two physical cores and four hardware threads, running Ubuntu 22.04.5, GCC 13.3, and Go 1.25.8 with `GOMAXPROCS=4`; a commodity part is taken as the reference platform because meeting the slot deadline on such a host is what lowers the hardware floor for participation. A 16-core Zen 5 host is the secondary platform (Appendix D.7); the two differ mainly in thread budget. Native C measurements are cycles per verified item; Go-level measurements are end-to-end latency in `ms/op` with turbo enabled, the configuration a deployed node uses, and include bridge and scheduling overhead (the frequency settings are stated in Appendix **??**). Kernel measurements, worker sweeps, workload traces, and the secondary-platform results are reported in Appendix D.

Validation. We validate the backend with Prysm's BLS correctness scripts and by cross-checking AVXPoS against `blst` on the evaluated verification paths. For timing behavior, a static branch audit finds only public control flow (batch-size dispatch, fixed loop bounds, parsing checks, pointer guards, error handling), and a `dudect` [37] test of the randomized-scalar windowing path, treated conservatively as secret-dependent in Section 5.2, showed no detectable leakage. These checks are not a formal constant-time proof.

6.1 Native verification performance

Table 2: C-level API performance comparison on the reference platform. "Scalar C" is the `blst` scalar baseline.

		Cycles per verified item			
Operation	n	Scalar C	Liu et al. [33]	AVXPoS	Speedup vs. scalar
single verify	1	4 131 062	2 437 399	2 361 518	1.75×
aggregation	2	2 392 586	1 571 572	1 272 421	1.88×
	3	2 322 413	1 541 623	1 234 237	1.88×
	4	2 284 104	1 526 168	972 891	2.35×
	5	2 271 267	1 517 774	1 150 768	1.97×
	6	2 258 109	1 510 886	1 036 650	2.18×
	7	2 255 190	1 508 229	938 392	2.40×
	8	2 238 449	1 533 564	820 697	2.73×

We first ask how much headroom the range-oriented native backend recovers before Go scheduling, bridge conversion, and tree reduction are included. Table 2 compares three C-level APIs: a scalar C baseline, the API built from the single-instance-oriented kernels of Liu et al. [33], and AVXPoS. The single-verify row isolates the pairing check that remains after FastAggregateVerify's public-key aggregation; the aggregation rows measure the range-absorption path of Algorithm 2, the hot native path for VerifyMultipleSignatures.

The gain peaks at $n = 8$, where the range fills one width-8 kernel: 2.73× over the scalar baseline and about 1.87× over the single-instance-oriented backend of [33], showing the additional benefit of AVXPoS's range-oriented, multi-instance organization on top of already-vectorized building blocks. Irregular sizes still gain 1.88–2.40× because the width-adaptive selectors compose narrower kernels instead of padding every range to eight, so the native mechanism does not require ideal width-8 batches. Whether it helps a deployed client depends on the ranges Prysm actually forms, which we characterize next.

6.2 Production workload shape

In the all-subnets stress trace of Appendix D.3, the steady-state gossip quantiles are $p50/p95/p99 = 12/85/162$ items per batch before Prysm's equal-message aggregation, but only $1/3/7$ at the native VerifyMultipleSignatures (VMS) boundary after it. The small post-aggregation sizes are a product of the client's batching policy rather than of scarce input: Prysm's 5 ms flush timer triggers nearly every batch, so the client exposes only small post-aggregation VMS calls despite much larger pre-aggregation input. Client-side batching and aggregation policy, not the volume of arriving signatures, thus determine how much independent verification work survives to the native backend.

This finding fixes the operating points for the rest of the evaluation: the end-to-end sweep below reports $n = 1, 3, 4, 6, 8$ because these sizes bracket the post-aggregation quantiles. Larger VMS calls also arise during transient recovery and checkpoint backfill: Section 6.3 uses the recovery observations to motivate its large-batch capacity point, and Section 6.5 evaluates checkpoint backfill directly.

A policy-level opportunity. The flush interval is itself a potential tuning knob. Prysm currently flushes the gossip verification queue after 5 ms, and the trace above shows that this policy leaves only small post-aggregation VerifyMultipleSignatures batches at the native boundary. A longer collection interval could expose more inter-instance parallelism and lower verification CPU cost per item, but only by adding queueing delay before verification begins. We leave this tradeoff unchanged in AVXPoS: all headline results use Prysm's existing batching policy. Determining whether a different interval improves whole-client behavior requires verifier CPU cost and consensus-message latency to be measured jointly, and is left to future work.

6.3 Prysm-boundary end-to-end performance

AVXPoS preserves useful gains at the small post-aggregation sizes that dominate steady-state operation, and the gain grows as more cross-instance work survives worker partitioning. Table 3 and Figure 8 report the evaluated Prysm paths; VMS is Prysm's generic SignatureBatch backend for gossip batch verification, block-transition signature sets, and deposit and pending-deposit validation. At the selected $n = 1, 3, 4, 6, 8$ points that bracket the observed post-aggregation quantiles

Table 3: Multi-threaded end-to-end latency of AVXPoS and Prysm's `blst` backend on evaluated verification paths. Each entry is the median of five three-second repetitions. The VMS sweep includes small sizes that bracket the post-aggregation quantiles $p50/p95/p99 = 1/3/7$ of the all-subnets stress trace (Section 6.2), as well as the capacity point; sizes above 32 are plotted in Figure 8.

Operation	Inputs	Latency (ms) AVXPoS	Prysm/`blst`	Speedup
Verify	1	0.803	1.117	1.39×
FastAggregateVerify	512	0.938	1.418	1.51×
VerifyMultipleSignatures	1	1.078	1.649	1.53×
	3	1.615	2.327	1.44×
	4	1.676	2.447	1.46×
	6	1.776	3.547	2.00×
	8	1.844	3.868	2.10×
	16	2.587	6.699	2.59×
	32	4.285	12.443	2.90×
	1000	106.159	337.308	3.18×

$p50/p95/p99 = 1/3/7$, AVXPoS gains 1.44–2.10×. The modest gains at $n \leq 4$ reflect the limited cross-instance work available at these sizes and are consistent with substantial non-batched and boundary overheads, as suggested by the $n = 1$ rows; from $n = 6$ upward the range rounding of Section 5.4 gives every native range at least two items (three width-2 ranges at $n = 6$), so no call runs at width one, and the gain grows with batch size to 2.90× at $n = 32$ and above 2.5× across the extended sweep of Figure 8.

Large VMS calls also arise without changing Prysm's batching logic: five controlled four-slot recovery episodes under default subnet subscriptions produce pending-path VMS calls with a median size of 871 and a maximum of 874 (Appendix D.4), and checkpoint backfill is measured in Section 6.5. We therefore use $n = 1000$ as a nearby controlled capacity point, where AVXPoS reaches 3.18×, rather than as a claim about frequent steady-state gossip.

On the other evaluated paths, FastAggregateVerify at $n = 512$ represents the direct sync-committee aggregate workload and gains 1.51×, and the single-signature Verify path gains 1.39×: it exposes no cross-signature batching opportunity, and the measured interval includes the full Prysm boundary rather than an isolated C kernel.

The inherited failure path is measured separately. With one well-formed invalid signature at $n = 874$, AVXPoS reduces complete verbose diagnosis from 1379.6 to 890.4 ms, but Prysm still performs all 874 single-signature checks (Appendix D.5); this is failure-cost characterization, not a denial-of-service mitigation (Section 5.2).

6.4 Worker partitioning and SIMD occupancy

The size dependence above still ties batch size to range width through one platform's worker policy. This subsection tests the mechanism of Section 3 directly: client threads and SIMD lanes consume the same pool of independent verification instances, so the client's partition fixes the native range width, and hence the SIMD occupancy, of every native call.

A controlled experiment varies only the worker partition. The same $n = 32$ VMS call becomes four width-8 ranges on the four-thread reference host and gains 2.90×, but becomes 32 width-one ranges on the 32-thread Zen 5 host and gains only 1.39×. Restricting the Zen 5 host to four threads, with hardware, binaries, and benchmark otherwise unchanged, raises its gain to 3.39× (Table 4; setup details in Appendix D.7). This controlled change identifies worker partitioning as a major contributor to the cross-platform speedup gap. Table 4 also shows AVXPoS substituting SIMD width for thread count: at every measured size, AVXPoS restricted to four threads is at least as fast as `blst` with all 32 (1.00–1.46×). FastAggregateVerify responds differently: its dominant work is public-key aggregation followed by one pairing check, so it reaches its best latency with only a few workers, and additional workers mainly add scheduling and merge overhead. This asymmetry is why the framework keeps API paths, worker ranges, and width selection separate.

Table 4: Effect of the client thread budget on the Zen 5 host. Only `GOMAXPROCS` changes; latencies are medians of five three-second repetitions in ms. The 32-thread columns reproduce Table 12 within about 1%.

n	32 threads `blst`	32 threads AVXPoS	32 threads Sp.	8 thr. Sp.	4 threads `blst`	4 threads AVXPoS	4 threads Sp.
1	0.905	0.658	1.37×	1.38×	0.905	0.672	1.35×
2	0.941	0.683	1.38×	1.40×	0.940	0.690	1.36×
3	0.966	0.702	1.38×	1.40×	0.951	0.711	1.34×
4	1.001	0.700	1.43×	1.41×	1.303	0.710	1.83×
6	1.064	0.739	1.44×	1.44×	1.347	0.840	1.60×
8	1.235	0.881	1.40×	1.66×	1.728	0.844	2.05×
12	1.285	0.920	1.40×	1.65×	2.159	1.027	2.10×
16	1.361	0.980	1.39×	1.99×	2.900	1.035	2.80×
24	1.467	1.072	1.37×	2.07×	3.780	1.466	2.58×
32	1.685	1.216	1.39×	2.40×	5.011	1.476	3.39×

Table 5: Batch-aware worker-cap ablation at fixed thread budget. "Default" is Prysm's min(`GOMAXPROCS`, n) policy, used by all other results in this paper; "capped" keeps at least two items in every native range and is opt-in. Retaining at least two items per range recovers 6.9–13.3% of verification-path latency at the measured small-batch points. Medians of five three-second repetitions, in ms.

Platform	n	Default workers	Default ms	Capped workers	Capped ms	Recovered
Reference, 4 threads	2	2	1.315	1	1.219	7.3%
	3	3	1.613	2	1.407	12.8%
	4	4	1.675	2	1.466	12.5%
Zen 5, 32 threads	8	8	0.889	4	0.828	6.9%
	32	32	1.239	16	1.074	13.3%

Batch-aware worker cap (ablation). Table 5 holds the thread budget fixed and varies only the AVX worker count, so the machine and the `blst` baseline do not move while the native range width does; batch-aware worker selection recovers additional latency but is not required for the headline AVXPoS results. The optimum is not the widest possible range: at $n = 4$, two workers of two items beat both four workers of one item and one worker of four (1.466 against 1.675 and 1.645 ms), so the best worker count balances thread-level concurrency against SIMD occupancy rather than maximizing either. Because that balance moves with batch size and platform, the cap remains opt-in (Section 5.4); full worker sweeps and the trace connection are reported in Appendix D.1.

6.5 Production checkpoint backfill

Checkpoint backfill provides a production test of whether AVXPoS's API-level gains survive in a complete client path: after checkpoint sync places a node near the current head, Prysm's backfill service requests consecutive historical ranges and verifies one proposer signature, with a distinct signing root, per returned block. This naturally exercises VMS at larger sizes without changing the client's batching policy. The experiment and the mainnet traces require a synchronized node and were therefore collected on the Zen 5 secondary platform, where the 32-thread worker budget narrows native ranges and reduces the relative AVXPoS gain at small and medium batch sizes (Section 6.4, Table 4). We checkpoint-synced a fresh mainnet database and, for each valid request size (32, 64, and 128), ran `blst` and AVXPoS on clones derived from that same checkpoint with an identical backfill origin and two backfill workers. The traced interval wraps the complete backfill verifier, that is SignatureBatch construction plus its final VMS call, and excludes network download and database insertion. Because live peers serve each configuration independently, we do not read downloaded blocks per second as a backend comparison, and instead report verification time normalized by successfully verified blocks.

Table 6: Mainnet checkpoint-backfill verifier cost. Batch size is the configured request cap and the observed $p50$; "batches/blocks" reports the numbers of valid traced batches and proposer blocks; times are median microseconds per block after normalizing each batch by its actual size.

Batch size		Batches / blocks		μs per block		
Config.	Actual	`blst`	AVXPoS	`blst`	AVXPoS	Speedup
32	32	49/1559	34/1083	65.56	49.68	1.32×
64	64	25/1590	32/2036	48.17	30.38	1.59×
128	127	16/2036	28/3562	36.29	20.07	1.81×

Across the 184 valid batches and 11,866 proposer blocks of Table 6, AVXPoS improves the median complete-verifier cost by 1.32–1.81×, reaching 1.81× at the protocol's request cap of 128, with the advantage increasing as fixed batch-construction cost is amortized and more distinct-message verification work reaches the backend. The aggregate-mean speedups, computed as total verifier time divided by total verified blocks, are 1.22×, 1.66×, and 1.98×, respectively.

The largest production point is 128 rather than 512 or 1000 because the mainnet Deneb preset limits a BeaconBlocksByRange response to 128 blocks [22]. We confirmed this boundary by attempting a 512-slot range: conforming peers produced no valid backfill verifier batch, so we exclude its retries, and the still larger 1000-slot configuration, rather than presenting them as workload. Backfill therefore establishes a real recovery path at $n \leq 128$, while the larger sizes remain the capacity points of Section 6.3, relevant to applications whose own bulk-verification protocols admit larger collections.

6.6 Deployment scope and limitations

Intel documents that sustained AVX-512 can select a lower turbo-frequency level [30], so we measure that behavior rather than assume it away: Appendix D.6 reports the effective frequency of both backends and repeats the comparison with turbo disabled, and the gains persist in both configurations. The Zen 5 secondary platform executes AVX-512 on a full-width 512-bit datapath and applies no large fixed frequency offset [2,31]; its absolute latency is substantially lower, but its relative gain on small batches is smaller because 32 threads split those batches into single-item ranges before the backend sees them (Section 6.4). Results on hosts with narrower or throttled AVX-512 will differ again.

The framework boundaries are not specific to one vector width, but the arithmetic implementation evaluated here is: only the AVX-512IFMA backend has measured kernel and Prysm results. AVX2, ARM SVE/SVE2 [40], and RISC-V V [1] could retain the same API-path, range, native-state, and width-selection organization, but require new kernels and remain future work. On an incompatible host, a runtime CPUID and OS-vector-state check selects the portable `blst` backend before any AVXPoS native code executes, allowing heterogeneous-fleet deployment without accelerating those hosts. Appendix B maps a plausible placement of the same boundary in a Rust client, but does not implement or evaluate that integration.

7 Related work

Relation to prior BLS vectorization work. The closest BLS12-381 optimization efforts are the AVX-512 optimal-ate-pairing implementation of Cheng et al. [18] and the SIMD BLS-signature implementation of Liu et al. [33]. AVXPoS reuses their basic-field, extension-field, curve, and single-instance pairing techniques as building blocks, but changes the optimization boundary. Prior work targets individual pairings, standalone signature operations, or single verification

instances; AVXPoS instead treats the client-formed verification range as the unit of optimization. This distinction matters because Prysm's batching and worker partitioning determine both how much independent verification work reaches the native backend and the SIMD width available within each native call. Section 6.1 quantifies the difference between the two optimization boundaries: at $n = 8$, AVXPoS's range-absorption path is about $1.87\times$ faster than the API built from the kernels of Liu et al. (Table 2).

Systems integration of SIMD cryptographic acceleration. Several systems papers study how optimized cryptographic implementations retain their gains when exposed through production software. OpenSSLNTRU integrates batch post-quantum key generation into TLS with small library changes and no application changes [7]; ENG25519 exposes optimized X25519/Ed25519 through an OpenSSL engine and evaluates TLS and DNS-over-TLS benefits [45]; and optimized ML-KEM work studies how AVX-512 affects TLS 1.3 handshakes [46]. AVXPoS addresses a different integration constraint. Prysm already batches verification work and partitions it across runtime workers before the native cryptographic backend runs, so the systems boundary itself determines how much SIMD parallelism remains available. AVXPoS therefore must reconcile client threading with vector occupancy rather than only expose a faster primitive through an existing interface.

SIMD batch cryptographic pipelines. Throughput-oriented cryptographic software often maps multiple independent operations to a fixed vector layout, as in AVX-512 batching for isogeny-based group actions and SIKE [19, 20] and AVX2/AVX-512 batch EdDSA verification [21, 44]. These systems demonstrate the throughput benefit of full-width homogeneous batches. Ethereum verification is less regular at the native boundary: Prysm first divides a logical batch across Go workers, so each resulting range may contain a sub-width or otherwise SIMD-unfriendly number of inputs. AVXPoS therefore combines coarse client-side range formation with C-side width-$8/4/2/1$ selection, rather than assuming that every native call arrives as one full SIMD batch.

BLS batch verification and products of pairings. Prior work on BLS batch verification establishes the algebraic and security basis for combining multiple verification terms into one equation, including product-of-pairings evaluation and randomized batch verification techniques [3,17,26]. Ethereum applies these mechanisms on consensus-facing paths, including same-message aggregate verification and randomized verification of distinct-message batches [23]. AVXPoS is orthogonal to these algebraic techniques: it does not change the verification equations, randomized coefficients, or Prysm's failure semantics. Instead, it changes how the existing verification work is executed after the client has formed and partitioned a batch, mapping decompression, subgroup membership tests, hash-to-curve, scalar multiplication, Miller-loop accumulation, and final exponentiation to a staged SIMD pipeline.

BLS and pairing implementations. Production BLS libraries and specialized pairing implementations provide the two lower-level foundations on which AVXPoS builds. Libraries such as `blst` provide the verification interfaces used by Ethereum clients, while earlier SIMD and vector ECC work develops the field- and curve-arithmetic techniques needed for high-throughput implementations on modern CPUs [10, 21, 24, 41]. AVXPoS connects these two layers: it preserves the production BLS API and verification semantics while organizing the client-formed work that reaches that interface so that the underlying SIMD arithmetic remains effective.

8 Conclusion

We presented AVXPoS, a client-aware batching framework for increasing BLS verification headroom in the Prysm Ethereum Proof-of-Stake client. It combines Go-side range formation with width-adaptive AVX-512 execution and mergeable native verification state while preserving Prysm's batching, verification, and failure semantics. On a commodity two-core host, AVXPoS gains 1.44–$2.10\times$ at small batch sizes that bracket the observed post-aggregation gossip quantiles and reaches $3.18\times$ at the controlled VMS capacity point. On a production checkpoint-backfill path, it reduces complete verifier cost by $1.81\times$ at the protocol's 128-block request cap.

The broader lesson is that production cryptographic acceleration depends not only on kernel throughput, but also on how much independent work survives the client's own scheduling decisions before the native backend runs. Worker threads and vector lanes draw from the same pool of verification instances, so the native range width that survives client partitioning is a major determinant of realized speedup. AVXPoS shows that preserving useful per-range work and adapting execution width can turn SIMD kernel gains into client-visible verification headroom without changing consensus-facing semantics.

The Lighthouse mapping in Appendix B shows how the same client/backend boundary could be instantiated in another consensus client, but does not establish performance portability without an implementation and evaluation there. The main limitation of the evaluated Prysm path remains its inherited linear failure-isolation cost.

References

[1] The RISC-V vector extension, version 1.0. RISC-V International specification, 2021. https://docs.riscv.org/reference/isa/unpriv/v-st-ext.html.

[2] Advanced Micro Devices. Understanding AVX-512 and validating usage on AMD EPYC. AMD Blog, 2026. AMD product blog.

[3] J.P. Aumasson, Q.T.M. Nguyen, and A. Sanso. Security of BLS batch verification. Ethereum Research, 2021. Ethereum Research thread.

[4] The Go Authors. Command cgo. Go package documentation, 2026. https://pkg.go.dev/cmd/cgo.

[5] The Go Authors. Package runtime. Go package documentation, 2026. https://pkg.go.dev/runtime.

[6] P.S.L.M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks (SCN)*, LNCS 2576, pages 257–267. Springer-Verlag, 2002. https://doi.org/10.1007/3-540-36413-7_19.

[7] D.J. Bernstein, B.B. Brumley, M.-S. Chen, and N. Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *USENIX Security Symposium*, pages 845–862, 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein.

[8] D.J. Bernstein and T. Lange. Explicit-formulas database. Online database. https://hyperelliptic.org/EFD/.

[9] J.-L. Beuchat, J.E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In *Pairing-Based Cryptography (Pairing)*, LNCS 6487, pages 21–39. Springer-Verlag, 2010. https://doi.org/10.1007/978-3-642-17455-1_2.

[10] A.K. Bhattacharya, S. Karati, A. Das, D.R. Chowdhury, B. Bellur, and A. Iyer. Use of SIMD features to speed up eta pairing. In *E-Business and Telecommunications - International Joint Conference (ICETE)*, CCIS 455, pages 137–154. Springer-Verlag, 2012. https://doi.org/10.1007/978-3-662-44791-8_9.

[11] D. Boneh, J. Bradley, S. Gorbunov, R.S. Wahby, H. Wee, C.A. Wood, and Z. Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-06, Internet Engineering Task Force, 2025. https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/06/.

[12] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology (EUROCRYPT)*, LNCS 2656, pages 416–432. Springer-Verlag, 2003. https://doi.org/10.1007/3-540-39200-9_26.

[13] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology (ASIACRYPT)*, LNCS 2248, pages 514–532. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-45682-1_30.

[14] A.D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[15] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction. Electric Coin Co. Blog Post, 2017. https://electriccoin.co/blog/new-snark-curve.

[16] A. Budroni and F. Pintore. Efficient hash maps to G2 on BLS curves. *Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 33(3):261–281, 2022. https://doi.org/10.1007/s00200-020-00453-9.

[17] V. Buterin. Fast verification of multiple BLS signatures. Ethereum Research, 2019. Ethereum Research thread.

[18] H. Cheng, G. Fotiadis, J. Großschädl, and D. Page. Fast AVX-512 implementation of the optimal ate pairing on BLS12-381. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2025(4):848–872, 2025. https://doi.org/10.46586/tches.v2025.i4.848-872.

[19] H. Cheng, G. Fotiadis, J. Großschädl, and P.Y.A. Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022. https://doi.org/10.46586/tches.v2022.i2.41-68.

[20] H. Cheng, G. Fotiadis, J. Großschädl, P.Y.A. Ryan, and P.B. Rønne. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021. https://doi.org/10.46586/tches.v2021.i4.618-649.

[21] H. Cheng, J. Großschädl, J. Tian, P.B. Rønne, and P.Y.A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In *Selected Areas in Cryptography (SAC)*, LNCS 12804, pages 698–719. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-81652-0_27.

[22] Ethereum Foundation. Deneb mainnet preset. Ethereum Consensus Specs, 2026. https://github.com/ethereum/consensus-specs/blob/dev/presets/mainnet/deneb.yaml.

[23] Ethereum Foundation. Phase 0 – the beacon chain. Ethereum Consensus Specs, 2026. https://ethereum.github.io/consensus-specs/phase0/beacon-chain/.

[24] A. Faz-Hernández, J. López, and R. Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)*, 45(3):1–35, 2019. https://doi.org/10.1145/3309759.

[25] A. Faz-Hernandez, S. Scott, N. Sullivan, R.S. Wahby, and C.A. Wood. Hashing to Elliptic Curves. Internet Engineering Task Force (IETF) Request for Comments (RFC) 9380, 2023. https://www.rfc-editor.org/rfc/rfc9380.

[26] R. Granger and N.P. Smart. On computing products of pairings. Cryptology ePrint Archive, Paper 2006/172, 2006. https://eprint.iacr.org/2006/172.

[27] D. Hayashida, K. Hayasaka, and T. Teruya. Efficient final exponentiation via cyclotomic structure for pairings over families of elliptic curves. Cryptology ePrint Archive, Paper 2020/875, 2020. https://eprint.iacr.org/2020/875.

[28] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Intel developer documentation, 2026. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[29] M. Joye and C. Tymen. Compact encoding of non-adjacent forms with applications to elliptic curve cryptography. In *Public Key Cryptography (PKC)*, LNCS 1992, pages 353–364. Springer-Verlag, 2001. https://doi.org/10.1007/3-540-44586-2_26.

[30] Ray Kinsella, Chris MacNamara, and Georgii Tkachuk. Intel AVX-512: Packet processing with the Intel AVX-512 instruction set. Intel solution brief, revision 003, 2021.

[31] Chester Lam. Zen 5's AVX-512 frequency behavior. Chips and Cheese, 2025. https://chipsandcheese.com/p/zen-5s-avx-512-frequency-behavior.

[32] C.H. Lim and P.J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology (CRYPTO)*, pages 249–263. Springer-Verlag, 1997. https://doi.org/10.1007/BFb0052240.

[33] G. Liu, H. Cheng, G. Fotiadis, J. Zhang, and J. Großschädl. Efficient SIMD implementation of the BLS signature scheme using intel AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2026(3):566–591, 2026. https://doi.org/10.46586/tches.v2026.i3.566-591.

[34] P.L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. https://doi.org/10.2307/2007888.

[35] Offchain Labs. Prysm: An Ethereum consensus client. GitHub repository, 2026. https://github.com/OffchainLabs/prysm.

[36] T. Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020. https://eprint.iacr.org/2020/972.

[37] O. Reparaz, J. Balasch, and I. Verbauwhede. Dude, is my code constant time? In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1697–1702. IEEE, 2017. https://doi.org/10.23919/DATE.2017.7927267.

[38] M. Scott. A note on group membership tests for G1, G2 and GT on BLS pairing-friendly curves. Cryptology ePrint Archive, Paper 2021/1130, 2021. https://eprint.iacr.org/2021/1130.

[39] Sigma Prime. Lighthouse: An Ethereum consensus client. GitHub repository, 2026. v8.2.1, https://github.com/sigp/lighthouse/tree/v8.2.1.

[40] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017. https://doi.org/10.1109/MM.2017.35.

[41] Supranational. blst: BLS12-381 signature library. GitHub repository, 2026. https://github.com/supranational/blst.

[42] F. Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory (TIT)*, 56(1):455–461, 2010. https://doi.org/10.1109/TIT.2009.2034881.

[43] R.S. Wahby and D. Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(4):154–179, 2019. https://doi.org/10.13154/tches.v2019.i4.154-179.

[44] B. Zhang, H. Cheng, J. Großschädl, and P.Y.A. Ryan. High-throughput EdDSA verification on intel processors with advanced vector extensions. Cryptology ePrint Archive, Paper 2025/1658, 2025. https://eprint.iacr.org/2025/1658.

[45] J. Zhang, J. Huang, L. Zhao, D. Chen, and Ç.K. Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In *USENIX Security Symposium*, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-jipeng.

[46] J. Zheng, H. Zhu, Y. Dong, Z. Song, Z. Zhang, Y. Yang, and Y. Zhao. Faster post-quantum TLS 1.3 based on ML-KEM: implementation and assessment. In *European Symposium on Research in Computer Security (ESORICS)*, LNCS 14983, pages 123–143. Springer-Verlag, 2024. https://doi.org/10.1007/978-3-031-70890-9_7.

A Additional implementation details

A.1 Base formulas

The low-level formulas used by the backend are recorded in this appendix. The vectorization strategies for these formulas follow prior work [18, 33]. Points are represented in Jacobian coordinates unless noted otherwise.

For a point $P = (X_P, Y_P, Z_P)$ in $E_1(\mathbb{F}_p)$ or in $E_2(\mathbb{F}_{p^2})$, the Explicit-Formulas Database (EFD) formula `dbl-2009-l` [8] computes $R = [2]P = (X_R, Y_R, Z_R)$ as follows.

$$X_R = 9X_P^4 - 8X_P Y_P^2,$$
$$Y_R = 3X_P^2(12X_P Y_P^2 - 9X_P^4) - 8Y_P^4, \quad Z_R = 2Y_P Z_P.$$

For points $P = (X_P, Y_P, Z_P)$ and $Q = (X_Q, Y_Q, Z_Q)$, the EFD formula `add-2007-bl` computes $R = P + Q$ as:

$$X_R = 4S^2 - 4TU^2,$$
$$Y_R = 2S(4X_P Z_Q^2 U^2 - X_R) - 8Y_P Z_Q^3 U^3,$$
$$Z_R = 2Z_P Z_Q U$$

where the intermediate values S, U and T are defined as:

$$S = Y_Q Z_P^3 - Y_P Z_Q^3, \quad U = X_Q Z_P^2 - X_P Z_Q^2,$$
$$T = X_Q Z_P^2 + X_P Z_Q^2.$$

The unified DADD operation follows the constant-time `blst` convention: $P \oplus Q$ evaluates either $P + Q$ or $[2]P$ with the same instruction schedule. It computes

$$X_R = S^2 - U^2 T, \quad Y_R = S^2(U^2 U_1 - X_R) - U^3 S_1,$$
$$Z_R = UZZ,$$

where the parameters are selected as

$$S = Y_Q Z_P^3 - Y_P Z_Q^3, \quad U = X_Q Z_P^2 - X_P Z_Q^2,$$
$$T = X_Q Z_P^2 + X_P Z_Q^2, \quad U_1 = X_P Z_Q^2,$$
$$S_1 = Y_P Z_Q^3, \quad ZZ = Z_P Z_Q.$$

for addition, and

$$S = 3X_P^2, \quad U = 2Y_P, \quad T = 2X_P,$$
$$U_1 = X_P, \quad S_1 = Y_P, \quad ZZ = Z_P.$$

for doubling.

For Miller loop line doubling, let $T = (X_T, Y_T, Z_T) \in E_2(\mathbb{F}_{p^2})$ and $R = [2]T$. The tangent-line coefficients are

$$\ell_0 = 6X_T^3 - 4Y_T^2, \quad \ell_1 = 3X_T^2 Z_T^2, \quad \ell_2 = 2Y_T Z_T^3.$$

For mixed line addition with affine $Q = (X_Q, Y_Q, 1)$, the coefficients of the line through T and Q are:

$$\ell_0 = 4X_Q(Y_Q Z_T^3 - Y_T) - 2Y_Q Z_R,$$
$$\ell_1 = 2(Y_Q Z_T^3 - Y_T), \quad \ell_2 = Z_R.$$

After precomputing the affine $\mathbb{G}_1$ input as $(-2x_P, 2y_P)$, line evaluation produces the sparse $\mathbb{F}_{p^{12}}$ element

$$\ell(P) = (\ell_0 - 2x_P \ell_1 v) + (2y_P \ell_2 v)w,$$

which is consumed by the sparse $\mathbb{F}_{p^{12}}$ multiplication in the Miller function update.

For the tower-field operations, we use the standard BLS12-381 tower $\mathbb{F}_p \to \mathbb{F}_{p^2} \to \mathbb{F}_{p^6} \to \mathbb{F}_{p^{12}}$ with $u^2 = -1$, $v^3 = u + 1$, and $w^2 = v$. Thus an element of $\mathbb{F}_{p^{12}}$ is written as $\alpha = a + a'w$, where $a, a' \in \mathbb{F}_{p^6}$ and $a = a_0 + a_1 v + a_2 v^2$. For $\alpha = a + a'w$ and $\beta = b + b'w$, the Karatsuba $\mathbb{F}_{p^{12}}$ multiplication used by the backend is

$$\alpha\beta = (ab + a'b'v) + ((a + a')(b + b') - ab - a'b')w.$$

The corresponding Karatsuba squaring is

$$\alpha^2 = ((a + a')(a + a'v) - aa' - aa'v) + 2aa'w.$$

In the Miller loop, line values have the sparse shape

$$\beta = (b_0 + b_1 v) + b_4 vw.$$

For an ordinary $\mathbb{F}_{p^6}$ element $a_0 + a_1 v + a_2 v^2$, multiplication by $b_0 + b_1 v$ gives

$$r_0 = a_2 b_1 (u + 1) + a_0 b_0,$$
$$r_1 = a_0 b_1 + a_1 b_0,$$
$$r_2 = a_2 b_0 + a_1 b_1,$$

and multiplication by $b_4 v$ gives

$$r_3 = a_2 b_4 (u + 1), \qquad r_4 = a_0 b_4, \qquad r_5 = a_1 b_4.$$

Together these coefficients form $(r_0 + r_1 v + r_2 v^2) + (r_3 + r_4 v + r_5 v^2)w$. Following [18], the width-2 tower-field kernels exploit the representation $\alpha = a + a'w$: the square and sparse-multiplication kernels distribute the two $\mathbb{F}_{p^6}$ components across the batch layout so four $\mathbb{F}_{p^6}$ multiplications can run before reduction.

A.2 Width-adaptive formula sources

Table 7 lists the width-specific formulas used by the low-level backend and identifies their formula basis. Here, "EFD" denotes the Explicit-Formulas Database [8]. The vectorization column follows the hierarchical notation of [33]: an $(x \times y \times z)$ implementation denotes parallelism at the extension-field, base-field, and lane levels, respectively. For $\mathbb{F}_p$-only operations, the notation omits the extension-field dimension. Entries such as "4×1 / 2×2" denote a layout remapping used by the corresponding formula. The concrete $\mathbb{F}_p/\mathbb{F}_p^2$ arithmetic, vector packing/unpacking, and layout-transfer routines follow the AVX-512IFMA arithmetic framework of Cheng et al. [18], with minor local modifications.

Table 7: Formula sources and width-adaptive uses in the low-level backend.

Operation	n	Vectorization	Formula basis
E_1 point DADD	2	4×1 / 2×2	[33], Algorithm 4
	4	2×1	[33], Algorithm 12
	8	1×1	[41]
E_2 point DADD	2	$2\times2\times1$	[33], Algorithm 12
	4	$1\times2\times1$	[41]
	8	$1\times1\times1$	[41]
E_1 point addition	2	4×1 / 2×2	[33], Algorithm 3
	4	2×1	[33], Algorithm 11
	8	1×1	EFD, add-2007-bl
E_2 point addition	2	$2\times2\times1$	[33], Algorithm 11
	4	$1\times2\times1$	EFD, add-2007-bl
	8	$1\times1\times1$	EFD, add-2007-bl
E_1 point doubling	2	4×1 / 2×2	[33], Algorithm 2
	4	2×1	[33], Algorithm 10
	8	1×1	EFD, dbl-2009-l
E_2 point doubling	2	$2\times2\times1$	[33], Algorithm 10
	4	$1\times2\times1$	EFD, dbl-2009-l
	8	$1\times1\times1$	EFD, dbl-2009-l
$\mathbb{F}_p$ sqrt	2	fallback scalar	–
	4	4×2	–
	8	8×1	–
Isogeny map E_2	2	$4\times1\times1$	[33], Algorithm 6
	4	$1\times2\times1$	[43], Section 4
	8	$1\times1\times1$	[43], Section 4
$\mathbb{F}_{p^{12}}$ square	2	$4\times1\times1$	[18], Algorithm 6
	4	$1\times2\times1$	karatsuba
	8	$1\times1\times1$	karatsuba
$\mathbb{F}_{p^{12}}$ sparse mul	2	$4\times1\times1$	[18], Algorithm 4
	4	$1\times2\times1$	schoolbook
	8	$1\times1\times1$	schoolbook
Line doubling	2	$2\times2\times1$	[18], Algorithm 7
	4	$1\times2\times1$	EFD, dbl-2009-l
	8	$1\times1\times1$	EFD, dbl-2009-l
Line addition	2	$2\times2\times1$	[18], Algorithm 9
	4	$1\times2\times1$	EFD, add-2007-bl
	8	$1\times1\times1$	EFD, add-2007-bl

B Lighthouse integration analysis

The portability of AVXPoS depends on whether its client-side organization can cross language and runtime boundaries other than Prysm's Go/C interface. We analyze Lighthouse v8.2.1 [39] as a Rust client architecture and identify where its existing verification pipeline could expose the same native range interface. This analysis characterizes the required integration boundary and client-specific engineering; quantitative Lighthouse evaluation remains future work.

B.1 Existing batching boundary

Lighthouse's `crypto/bls` crate already forms a vector of SignatureSet values and sends it to `verify_signature_sets` in `crypto/bls/src/impls/blst.rs`. This is the closest analogue of Prysm's VerifyMultipleSignatures boundary. Before the final multi-pairing call, Lighthouse has already selected a nonzero random coefficient for every set, validated and aggregated the public keys belonging to that set, obtained its aggregate signature, and retained the 32-byte signing root. Consequently, an AVXPoS backend would be inserted at this final verifier boundary rather than in gossip queues or per-message validation.

The proposed mapping is direct:

Lighthouse value	Native range input
aggregate signature for one set	one compressed 96-byte signature
aggregate public key for the set	one compressed 48-byte public key
signing root	one contiguous 32-byte message
random coefficient	one nonzero 64-bit scalar

Thus Lighthouse would not need to wait for a larger batch: it could pass the batch it already formed across the Rust/C boundary, preserving the client's existing batching latency.

B.2 Required adapter

A concrete integration would add an optional Rust crate exposing a narrow C ABI and change only the backend selection inside `verify_signature_sets`. The adapter would first normalize the four columns above into contiguous arrays. It would then partition the set vector into ranges, let each worker absorb its range into an opaque native pairing context, merge the committed contexts, and request one final verification. SIMD packing, width-$8/4/2/1$ stage selection, and temporary curve representations would remain inside C; Rust would retain ownership of the input arrays and worker scheduling.

Four details require client-specific engineering beyond a mechanical reuse of the Prysm bridge. First, a SignatureSet may contain several public keys, so Lighthouse must preserve its existing per-set aggregation and rogue-key checks before constructing one native row. Second, Rust ownership requires the FFI calls to be synchronous and forbids the native context from retaining pointers into movable Rust buffers. Third, backend selection should perform a runtime check for AVX-512F, DQ, BW, VL, and IFMA before entering C, because Lighthouse binaries are deployed on heterogeneous x86 systems. Finally, native rejection must return through Lighthouse's existing invalid batch path; diagnostic isolation and any portable-backend retry remain client policy.

B.3 Scope

This mapping identifies a concrete placement for the AVXPoS state interface that leaves Lighthouse's consensus batching policy unchanged. Its performance depends on Lighthouse's production batch shapes and the cost of Rust-side normalization. A quantitative evaluation would require production traces, removal of duplicate conversion between native and portable paths, valid and invalid batch tests, runtime feature-gate tests, and end-to-end measurements on representative hardware.

C Prysm semantic compatibility

C.1 Randomized-batch coefficients

Prysm forms the VerifyMultipleSignatures request before sampling one fresh nonzero 64-bit coefficient r_i per item. If an adversary knew the coefficients while choosing two controlled signatures, it could replace valid signatures with $\sigma_1' = \sigma_1 + \Delta$ and $\sigma_2' = \sigma_2 - (r_1 r_2^{-1})\Delta$ for any $\Delta \in \mathbb{G}_2$. The perturbations cancel in $\sum_i r_i \sigma_i'$, so the randomized equation can accept even though neither modified item verifies alone. Soundness thus requires unpredictability until the submitted inputs are fixed, not permanent confidentiality after verification [3, 17]. AVXPoS neither regenerates nor reuses the coefficients and retains mask-based scalar processing as a conservative implementation discipline.

C.2 Fault isolation

AVXPoS preserves Prysm's error-propagation boundary by treating each native worker range as an all-or-nothing verification unit. The bridge returns the same Boolean outcome as the original backend and does not locate faulty consensus objects inside C. If decoding, subgroup checking, range processing, commit, or final verification fails, worker-local native state is released and failure is returned to the Go caller.

This boundary matches the structure of Prysm's verification code. A SignatureBatch is verified by calling VerifyMultipleSignatures, and its verbose mode first tries the batch and then rechecks entries one by one only after a rejection. The gossip batch verifier builds a temporary aggregate set, applies duplicate removal and equal-message aggregation to that temporary set, and reports the same batch error back to the waiting messages; each caller then uses its existing per-message verification policy. Deposit and pending-deposit verification paths likewise construct arrays of signatures, messages, and public keys and call VerifyMultipleSignatures directly [23, 35, 41]. These implementation facts allow AVXPoS to keep failure diagnosis above the native backend, with no need for a C-side bad-lane search.

The same boundary limits the denial-of-service implications of failed batches. Invalid inputs can still force Prysm onto slower isolation paths, but AVXPoS does not add backend retries or mutation of caller-owned Go batches. The gossip verifier also contains a regression test asserting that the batch verifier does not mutate caller-provided SignatureBatch sets after duplicate removal and aggregation. AVXPoS therefore changes neither the number of diagnostic checks nor Prysm's queueing, duplicate-removal, validation, and peer/message handling policies. Both the initial native rejection and the complete inherited isolation path are measured in Appendix D.5: AVXPoS lowers the absolute failure cost, and the client's $O(n)$ diagnostic policy carries over unchanged.

C.3 Rogue-key handling

AVXPoS does not change which aggregate-signature API is considered safe for a given consensus path. In Prysm's BLS package, AggregateVerify is documented as rogue-key sensitive and deprecated except for specification tests [12, 35]. Same-message aggregate checks use FastAggregateVerify with validator public keys admitted through Ethereum's deposit proof-of-possession process, and attestation paths validate the indexed-attestation structure before checking the aggregate signature [11, 23]. Mixed-message batches use randomized VerifyMultipleSignatures rather than deterministic aggregation [3, 17]; Prysm samples one random scalar per item and rejects empty or length-mismatched inputs before entering the native range scheduler. AVXPoS enters only after Prysm has made the API choice, so rogue-key assumptions and randomization semantics are unchanged.

The optional raw-public-key aggregation entry point does not alter this trust boundary. It accelerates decompression, subgroup checking, and point aggregation for compressed keys, but it neither admits a validator nor treats subgroup membership as a substitute for proof of possession. Prysm's normal verification paths use affine validator keys already resident in the cache after the existing admission process, so raw-key aggregation is not part of the end-to-end evaluation. No validator-admission, proof-of-possession, or key cache logic is changed by AVXPoS.

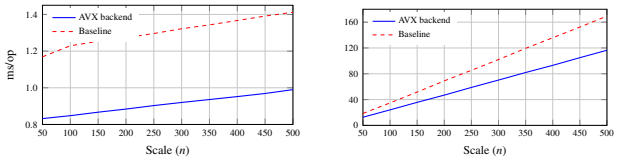
D Additional evaluation details

This appendix collects the measurements that support the evaluation in Section 6: worker-sensitivity sweeps, mainnet workload traces, the invalid-batch failure benchmark, the batch-aware worker cap, the secondary Zen 5 platform and the effect of the client thread budget, and the low-level and mid-level microbenchmarks, parameter choices, and tail schedules behind the client-level results.

D.1 Single-worker and worker sensitivity

The evaluated verification paths are measured through Prysm's Go API with one AVX worker in Figure 6, so that batching effects appear without scheduling; the `blst` baseline curve keeps the client's full thread budget, so the comparison is conservative. The interval includes bridge conversion, native state allocation, C-side accumulation, and final verification. FastAggregateVerify gains moderately because most batching occurs in public-key aggregation before one final pairing check, whereas VerifyMultipleSignatures scales with input size because more independent messages expose more pre-pairing and pairing-accumulation work.

Worker sensitivity on the same-message path is plotted in Figure 7. Its best latency is reached at four workers, the number of hardware threads, but that is only 1.06× better than



(a) FastAggregateVerify. (b) VerifyMultipleSignatures.

Figure 6: Single-worker latency of AVXPoS and Prysm's `blst` backend.



Figure 7: Worker sensitivity of FastAggregateVerify at $n = 512$.

a single worker because the final pairing check is serial, and requesting more workers than threads does not help. In the same sweep, VerifyMultipleSignatures at $n = 256$ is measured 2.25× faster with four workers than with one, since each worker owns an independent pairing context. Four workers are therefore used for both paths.

Batch-aware worker cap. The worker-cap measurements are reported in Table 5 of Section 6. They connect to the steady-state trace of Table 8 as follows: $p95 = 3$ is one of the sizes the cap helps; at $p50 = 1$ a single worker is already used; and at $p99 = 7$ the rule returns the default four workers. The artifact exposes the cap as an opt-in mode (`PRYSM_AVX_BATCH_WORKERS=auto`); promoting it to a default would first require a per-platform sweep of batch size against worker count.

D.2 Large-batch VMS sweep

Table 3 is continued above $n = 32$ on the reference platform in Figure 8. The gap is above 2.5× across the whole plotted range and both curves are close to linear in n, so the capacity point at $n = 1000$ is representative of the slope rather than a special case.

D.3 All-subnets mainnet trace

This experiment and the recovery experiment that follows require a node synchronized to the live mainnet head and were collected on the Zen 5 secondary platform (Appendix D.7). Both characterize the workload the client forms, which is a property of the protocol and the client's batching policy rather than of the host. We instrumented Prysm's existing batching policy and collected two sequential one-hour mainnet runs

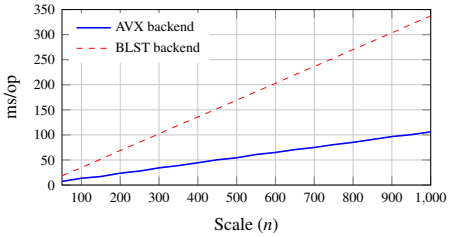


Figure 8: Multi-threaded end-to-end latency of VerifyMultipleSignatures with AVXPoS and Prysm's `blst` backend on the reference platform.

with `--subscribe-all-subnets`, one for `blst` and one for AVXPoS. This opt-in policy subscribes the node to all attestation and sync-committee subnets, placing the trace at the upper end of a single node's verification load. The privacy-preserving tracer records sizes, API names, latencies, flush reasons, and outcomes, but not messages, signatures, public keys, or peer identifiers. Both runs completed with no malformed records, dropped trace events, or invalid batches. The two windows were collected on 2026-07-30 with the node at a live synchronized head under the Fulu fork (fork version `0x06000000`, with no later fork scheduled at the time), and together span mainnet slots 14,883,061–14,883,675 (epochs 465,095–465,114).

Table 8: One-hour all-subnets mainnet traces. Size columns report $p50/p95/p99/$ max before deduplication and after equal-message aggregation; slot time is $p50/p95$ BLS API time in ms.

Backend	Batches	Batch size		Slot time
		Before dedup.	After aggr.	(ms)
`blst`	280,400	12/85/162/1000	1/3/7/83	905.1/1176.7
AVXPoS	307,598	12/82/164/1000	1/2/6/93	722.4/921.8

As shown in Table 8, subscribing to all subnets raises the pre-aggregation input to $p99 = 162$–164 and occasionally reaches Prysm's 1000-item flush limit. The 5 ms timer still triggers 280,396 of 280,400 `blst` batches and 307,581 of 307,598 AVXPoS batches. Equal-message aggregation then collapses the mixed-message input to $p50/p95/p99 = 1/3/7$ for `blst` and $1/2/6$ for AVXPoS; the resulting VerifyMultipleSignatures (VMS) calls have the same size distributions. The trace also observes 480 and 472 FastAggregateVerify calls, respectively, with a median of 510 public keys.

Because the backends ran in consecutive network windows with different event counts, the per-slot times characterize workload scale; the controlled backend comparisons are those of Section 6. Even under this stress subscription, equal-message aggregation makes large mixed-message VMS calls uncommon, so the $n = 512$ and $n = 1000$ VMS sweeps remain capacity measurements; checkpoint backfill in Sec-

tion 6 supplies the production distinct-message workload up to the mainnet request cap of 128.

Two later six-hour all-subnets runs also exposed large VMS calls on Prysm's unchanged pending-attestation path: 685 of 1,362 `blst` calls and 724 of 1,310 AVXPoS calls contained at least 128 inputs; maxima reached 21,422 and 22,313.

D.4 Default-subnet transient recovery

We separately test whether a short head gap can expose large VMS calls under Prysm's default subnet subscriptions. In each of five runs, the node first reaches the live mainnet head. An opt-in fault injector then withholds the first target block arriving after the trigger for exactly four slots (48 seconds), at both gossip and request–response import paths, while normal attestation processing continues. Once the gate opens, we record for 30 seconds the actual VerifyMultipleSignatures calls issued by Prysm's existing pending-attestation path. The injector changes neither the client's batching and validation logic nor its BLS backend policy.

Table 9: Default-subnet recovery after a controlled four-slot block delay. Inputs are the total items submitted by pending-path VMS calls; "last call" is measured from block release.

	VMS calls			Last call
Run	Total	≥ 128	Inputs	(s)
1	8	6	5,233	22.69
2	6	5	4,344	13.83
3	3	2	1,052	12.50
4	7	5	4,469	11.46
5	4	4	3,489	26.78
All	28	22	18,587	26.78

Across the five episodes, the median, 95th-percentile, and maximum VMS call sizes are 871, 873, and 874; 22 of 28 calls contain at least 128 inputs, and all calls complete with a valid result. The last observed pending-path VMS call occurs 11.46–26.78 seconds after block release across runs. The controlled gap thus shows that large VMS inputs arise during transient recovery even under default subnet subscriptions, complementing the steady-state stress picture of the preceding all-subnets trace. Backend performance at these larger sizes is measured by the controlled VMS sweep of Section 6.

D.5 Invalid-batch failure cost

We characterize Prysm's inherited failure path with a local controlled benchmark on synthetic inputs. Each input contains a distinct message. In the invalid variant, the middle entry is a correctly encoded, subgroup-valid signature produced by the expected key over a different message; it therefore reaches cryptographic batch rejection rather than a cheaper decoding error. We use $n = 32$ and $n = 128$ from the backfill experiment and $n = 874$, the maximum observed in the default-subnet recovery trace. On the reference host with turbo enabled and `GOMAXPROCS=4`, each case runs for three seconds and the median of five repetitions is reported in Table 10.

Table 10: Valid and one-invalid SignatureBatch latency (median ms). "Reject" is the initial VMS call; "full invalid" additionally includes Prysm's verbose isolation scan, and the speedup is that of the full invalid path.

	Valid		Reject		Full invalid		
Inputs	AVXPoS	`blst`	AVXPoS	`blst`	AVXPoS	`blst`	Speedup
32	4.107	12.433	4.386	12.424	33.350	51.273	1.54×
128	13.855	44.379	14.963	44.372	130.874	204.376	1.56×
874	90.085	294.648	97.612	294.699	890.378	1379.643	1.55×

The initial rejection costs approximately the same as a successful VMS call and retains its acceleration: the `blst`/AVXPoS ratio is 2.83×, 2.97×, and 3.02× at the three sizes. Once that call rejects, VerifyVerbosely checks every entry rather than stopping after the bad one. Subtracting the rejection time gives isolation costs of 29.0/115.9/792.8 ms for AVXPoS and 38.8/160.0/1084.9 ms for `blst`. That stage is dominated by unbatched Verify, which gains only 1.39× (Table 3), so it reduces the complete-path gain to approximately 1.55×.

AVXPoS thus lowers absolute failure cost at every measured size while keeping the number of fallback checks unchanged. The $O(n)$ isolation scan itself is client policy inherited from Prysm, and handling repeated invalid batches remains the role of the client's admission, queueing, and peer-scoring machinery. Malformed encodings return a decoding error before this diagnostic scan is reached; the artifact covers that cheaper behavior with a separate correctness test.

D.6 Frequency behavior on the reference platform

Intel documents that sustained AVX-512 can select a lower turbo-frequency level, so a wider instruction mix can win issue slots and lose clocks [30]. The main results run with turbo enabled because that is how a deployed node runs. Here we measure the frequency each backend actually reaches and repeat the comparison with turbo disabled, where both backends are pinned to the 1.20 GHz base frequency.

The speedups hold under both power policies, and the reported configuration is the conservative one, as recorded in Table 11: with turbo disabled every path improves further, from 3.18× to 3.52× at the capacity point and from 2.90× to 3.19× at $n = 32$. The direction is what Intel's documented behavior predicts. A 10-second `perf stat` run of VerifyMultipleSignatures at $n = 1000$ with turbo enabled measures a process-wide effective frequency (cycles per task-clock) of 3.193 GHz for AVXPoS and 3.385 GHz for `blst`: the vectorized instruction mix runs at a 5.7% lower clock, which

Table 11: Reference-platform latency with turbo enabled (the reported configuration) and disabled (controlled). Medians of five three-second repetitions in ms.

		Turbo enabled			Turbo disabled		
Operation	n	AVXPoS	`blst`	Sp.	AVXPoS	`blst`	Sp.
Verify	1	0.803	1.117	1.39×	2.169	3.096	1.43×
FAV	512	0.938	1.418	1.51×	2.520	4.114	1.63×
VMS	1	1.078	1.649	1.53×	2.932	4.652	1.59×
	8	1.844	3.868	2.10×	4.973	11.016	2.22×
	32	4.285	12.443	2.90×	11.191	35.655	3.19×
	1000	106.159	337.308	3.18×	274.820	968.115	3.52×

narrows but does not erase its advantage. Because the two backends execute different instruction mixes, this reading describes the deployed configuration as a whole. Reporting the turbo-enabled numbers in Section 6 matches how the client is actually run and understates the backend gap by roughly a tenth.

D.7 Zen 5 secondary platform and the thread budget

We repeat the Prysm-boundary measurements of Table 3 on an AMD Ryzen 9 9950X3D (Zen 5) with 16 physical cores and 32 hardware threads, running Ubuntu 24.04.4, GCC 13.3, and Go 1.25.8, with frequency boost disabled and the `performance` governor. This host executes AVX-512 on a full-width 512-bit datapath and applies no large fixed AVX-512 frequency offset [2,31]; we observed no throughput instability from frequency transitions. Its absolute latency is far lower than the reference platform's, so it represents a high-throughput server-class deployment rather than a commodity one. The build retains the scalar width-one E_1 doubling formula, which is faster here (Appendix D.10), and the checkpoint-backfill, mainnet-trace, and recovery experiments of Section 6 were all collected on this host.

Table 12: Prysm main-path latency on the Zen 5 secondary platform, median of five three-second repetitions. The small VerifyMultipleSignatures sizes appear in the 32-thread columns of Table 4.

		Latency (ms)		
Operation	Inputs	AVXPoS	Prysm/`blst`	Speedup
Verify	1	0.495	0.578	1.17×
FastAggregateVerify	512	0.540	0.804	1.49×
VerifyMultipleSignatures	1000	7.923	23.823	3.01×

Why the relative gain is smaller here. The smaller speedups are a consequence of the client's thread budget, not of the microarchitecture. Prysm's worker count for a VerifyMultipleSignatures call is $\min(\texttt{GOMAXPROCS}, n)$, after which the bridge rounds each worker range up to a supported native width (1, 2, 4, 8, 12, 16, 24, then multiples of eight). With 32 threads available and $n \leq 32$, every range holds exactly one item, the native path therefore runs at width one, and the backend has no cross-signature work left to vectorize; the residual 1.37–1.46× comes from contiguous range preparation and the reduced allocation path rather than from SIMD.

The thread-budget experiment is reported in Table 4 of Section 6.4. We vary only `GOMAXPROCS`, which is the input to worker selection and constrains both backends identically; the hardware, the binaries, and the benchmark are unchanged, so the two backends are always compared under the same thread budget.

Two consequences follow. First, the platform gap in Section 6 is a range-occupancy effect: restricting this 32-thread host to four threads makes the same $n = 32$ call four width-8 ranges and raises the speedup from 1.39× to 3.39×, which is what the four-thread reference platform also reports. Second, AVXPoS substitutes SIMD width for thread count. At $n = 32$ the `blst` backend needs all 32 threads to reach 1.685 ms, while AVXPoS reaches 1.476 ms with four; across every measured size AVXPoS on four threads is at least as fast as `blst` on 32 (1.00–1.46×). A node can therefore either verify the same work with a smaller share of the machine, leaving threads for state transition, fork choice, and propagation, or run the client on a smaller host. Both readings support reporting a commodity four-thread machine as the reference platform.

D.8 Low-level formula kernel benchmarks

The cycle counts of the low-level curve, line, tower-field, and square-root kernels used by the batched backend are reported in Table 13. The width-8 implementations provide the largest improvements, with most curve and line operations achieving roughly 3× speedup over the scalar baseline. The narrow widths show why the selectors of Section 4.1 are platform-sensitive rather than fixed: at width 1 the margin over the scalar routine collapses to 1.19–1.23× for E_1 doubling and addition, and on the Zen 5 secondary platform the same width-one E_1 doubling is *slower* than the scalar routine (0.67×), which is why that host selects the scalar formula while the reference host selects the vector one (Appendix D.10). The two $\mathbb{F}_{p^{12}}$ tower operators list no width-2 entry: one $\mathbb{F}_{p^{12}}$ element already fills all eight lanes at width 1, so packing two instances adds inter-layout permutation without adding parallelism, and the two-instance kernel we measured beat neither platform's width-1 schedule. Those requests therefore dispatch to two width-1 invocations.

D.9 Mid-level verification kernel benchmarks

As described in Section 4.2, scalar multiplication uses a batched wNAF method whose precomputation window affects the overall cost. The mid-level kernels used in verification are summarized in Table 14, including point decompression, subgroup checks, scalar multiplication, hash-to-$\mathbb{G}_2$, and

Table 13: Benchmarks of low-level operations in the scalar baseline and AVXPoS.

Operation	n	Cycles per instance: SIMD	Cycles per instance: Scalar (`blst`)	Speedup
E_1 point addition	8	743.2	2431.0	3.271×
	4	815.5	2423.5	2.972×
	2	1007.0	2413.0	2.396×
	1	1325.0	2419.0	1.826×
E_1 point DADD	8	814.1	2632.0	3.233×
	4	887.0	2621.5	2.955×
	2	1080.0	2614.0	2.420×
	1	1439.0	2615.0	1.817×
E_1 point doubling	8	333.4	1155.9	3.467×
	4	433.2	1157.2	2.671×
	2	577.0	1143.5	1.982×
	1	932.0	1144.0	1.227×
E_1 mixed DADD	8	600.8	1998.9	3.327×
E_2 point addition	8	1966.1	6620.0	3.367×
	4	2206.0	6446.0	2.922×
	2	2354.5	6458.0	2.743×
	1	2896.0	6439.0	2.223×
E_2 point DADD	8	2199.5	7177.6	3.263×
	4	2468.8	7141.2	2.893×
	2	2623.0	7155.0	2.728×
	1	3108.0	7132.0	2.295×
E_2 point doubling	8	855.0	2643.5	3.092×
	4	993.2	2639.8	2.658×
	2	1245.0	2646.5	2.126×
	1	1756.0	2630.0	1.498×
E_2 mixed DADD	8	1618.9	5265.2	3.252×
line addition	8	1892.4	5241.0	2.769×
	4	2032.5	5238.5	2.577×
	2	2319.0	5223.5	2.252×
	1	3450.0	5221.0	1.513×
line doubling	8	1560.4	4092.8	2.623×
	4	1731.8	4081.2	2.357×
	2	1949.0	4080.0	2.093×
	1	2408.0	4062.0	1.687×
isogeny map E_2	8	4021.5	13247.1	3.294×
	4	4434.0	13269.8	2.993×
	2	5233.0	13253.5	2.533×
	1	5826.0	13239.0	2.272×
$\mathbb{F}_{p^{12}}$ sparse mul	8	2234.0	4979.0	2.229×
	4	2334.8	5024.2	2.152×
	1	2618.0	4983.0	1.903×
$\mathbb{F}_{p^{12}}$ square	8	1961.8	5429.8	2.768×
	4	2461.5	5399.2	2.193×
	1	2694.0	5377.0	1.996×
$\mathbb{F}_p$ sqrt	8	18244.8	58128.0	3.186×
	4	20802.8	57976.0	2.787×

Miller loops. The width-8 variants are consistently the fastest, while the smaller widths give the costs needed by the adaptive scheduler.

The one-instance vector paths reuse formula layouts from prior work [18, 33]. On our evaluation platforms, the 4×2 layout is less favorable than in the original reports, so the inherited one-instance paths show lower relative speedups than those papers report.

Table 14: Benchmarks of mid-level verification kernels in the scalar baseline and AVXPoS.

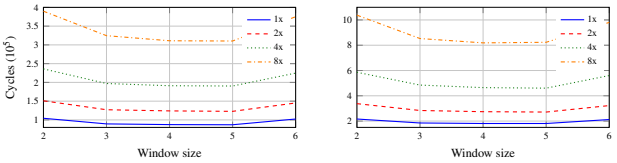
Operation	n	Cycles per instance: SIMD	Cycles per instance: Scalar (`blst`)	Speedup
uncompress $\mathbb{G}_1$	8	18834.1	58953.6	3.130×
	4	21517.8	58783.0	2.732×
uncompress $\mathbb{G}_2$	8	39059.9	119321.0	3.055×
	4	45077.2	119440.0	2.650×
subgroup check $\mathbb{G}_1$	8	50192.4	172677.8	3.440×
	4	88170.8	172478.0	1.956×
	2	108424.5	172944.0	1.595×
	1	124980.0	172387.0	1.379×
subgroup check $\mathbb{G}_2$	8	66109.9	205233.4	3.104×
	4	106629.5	205282.2	1.925×
	2	123937.5	205201.5	1.656×
	1	129471.0	207214.0	1.600×
E_1 scalar mul.	8	38802.6	132707.6	3.420×
	4	47626.2	131757.8	2.766×
	2	61368.5	132080.0	2.152×
	1	87018.0	132281.0	1.520×
E_2 scalar mul.	8	102132.1	320082.1	3.134×
	4	114907.2	321079.8	2.794×
	2	135698.5	322122.5	2.374×
	1	179908.0	321356.0	1.786×
hash-to-$\mathbb{G}_2$	8	234282.6	723457.1	3.088×
	4	258282.5	733298.8	2.839×
	2	327768.5	725886.5	2.215×
	1	546275.0	726497.0	1.330×
Miller loop	8	329341.0	999039.0	3.033×
	4	377010.8	998994.0	2.650×
	2	491028.0	999772.0	2.036×
	1	511083.0	1005027.0	1.966×

D.10 Scalar-multiplication window selection

Scalar multiplication uses a batched wNAF method. We tune the precomputation window separately for each vector width on both E_1 and E_2, and use the selected settings in the mid-level kernel benchmarks in Table 14. As shown in Figure 9, all paths use window size 5 except the width-8 E_2 path, where window size 4 is 0.6% faster; the width-one E_2 path is a tie between 4 and 5 to within 0.01% and takes the common setting. Repeating the sweep on the Zen 5 secondary platform selects the same windows, so this parameter transfers between the two hosts.

The width-one E_1 path does not. Its doubling steps can use either the vector formula or a conversion back to the scalar routine, and the choice inverts across platforms: on

the reference host the vector formula reduces width-one E_1 scalar multiplication from 108,565 to 87,311 cycles (1.24×), whereas on Zen 5 the scalar formula wins (Table 13). We therefore compile the reference build with `P1_DBLVEC=1` and the Zen 5 build without it. Figure 9 is measured on the reference platform in its shipping configuration.



(a) E_1 scalar multiplication. (b) E_2 scalar multiplication.

Figure 9: Cycle counts of vectorized scalar multiplication under different window sizes.

D.11 Mid-level tail paths

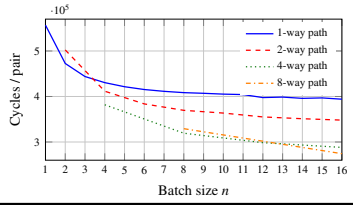
For SIMD-unfriendly sizes, the lower-cost choice is recorded in Table 15 between padded execution and narrower decomposition. A single width leaves unused lanes inactive; a sum denotes independent calls whose widths cover the logical batch.

Table 15: Selected schedules for SIMD-unfriendly batch sizes.

	Execution width for logical batch size n			
Operation	3	5	6	7
uncompress $\mathbb{G}_1$	4	8	8	8
uncompress $\mathbb{G}_2$	4	8	8	8
subgroup check $\mathbb{G}_1$	$2+1$	8	8	8
subgroup check $\mathbb{G}_2$	$2+1$	8	8	8
E_1 scalar mul.	4	$4+1$	8	8
E_2 scalar mul.	$2+1$	$4+1$	$4+2$	8
hash-to-$\mathbb{G}_2$	4	$4+1$	$4+2$	8

D.12 Miller-loop path selection

We benchmark independent SIMD Miller-loop paths and multi-pairing accumulation paths separately, and use the lowest-cost schedule for each logical batch size. The per-pair costs and selected schedules are reported in Figure 10. For example, 4×2 denotes two width-4 invocations, and $4\times2+1$ denotes two width-4 invocations followed by one width-1 invocation.



n	Path	n	Path	n	Path	n	Path
2	1×2	6	2×3	10	$4\times2+1\times2$	14	8×2
3	1×3	7	4×2	11	4×3	15	8×2
4	4	8	4×2	12	4×3	16	8×2
5	$4+1$	9	$4\times2+1$	13	$4\times3+1$		

Figure 10: Miller-loop cost and selected schedule by logical batch size.