# WeaveTLS: High-Throughput Cross-Connection ML-DSA Authentication in Mutual TLS

*Ganqin Liu[1], Hao Cheng[1,2]*, and Jipeng Zhang[3]*

[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]*National University of Singapore, Singapore*

ganqin.liu@mail.sdu.edu.cn hao.cheng@sdu.edu.cn jp-zhang@outlook.com

## Abstract

Mutual TLS (mTLS) authenticates both peers and therefore incurs post-quantum signature costs on every connection. Concurrent handshakes expose independent ML-DSA operations, but executing them jointly is difficult: signing is rejection-divergent, verification uses heterogeneous keys, and synchronous TLS APIs expose authentication work one connection at a time.

We present WEAVETLS, a wire-transparent architecture that executes ML-DSA authentication across concurrent TLS connections. Its primitive interface combines rejection-aware slot refill with per-request expanded-key handles, supporting both unrelated client keys and shared issuer keys. A stackless OpenSSL continuation lets an nginx worker suspend authentication, expose work from other connections, and execute compatible operations through optimized single-request, four-request, or eight-request AVX-512 kernels without fibers or cross-thread handoff. WEAVETLS preserves the TLS authentication barrier, certificate validation, and wire protocol.

On an AMD Ryzen 9 9950X3D, WEAVETLS improves one-core nginx mTLS throughput by 2.81–4.31× over OpenSSL's default ML-DSA path and by 2.19–3.29× over a synchronous reference-C control in the same provider across ML-DSA-44/65/87. At the primitive boundary, expanded-key eight-request verification is 1.65–2.23× faster than matched cached AVX2, and rejection-aware refill makes ML-DSA-65 signing 1.90× faster than otherwise identical lockstep scheduling. Cohort publication also weakens client-visible rejection timing under load, reducing attempt-count/latency correlation to 0.063 at concurrency 16 and 0.008 at 64; singleton execution retains the signal.

## 1 Introduction

Post-quantum authentication is entering deployed TLS stacks. NIST standardized the Module-Lattice-Based Digital Signature Algorithm (ML-DSA) in FIPS 204 [30], RFC 9881 specifies its X.509 identifiers [28], the IETF TLS working group is standardizing ML-DSA signature schemes for TLS 1.3 [17], and OpenSSL 3.5 already exposes all three parameter sets to applications [34]. This progress resolves much of the protocol and library integration, but leaves the per-connection cost of post-quantum authentication. Existing measurements identify authentication as a major source of handshake cost [22,36,44].

The cost is especially relevant to mutual TLS (mTLS), which service meshes and zero-trust internal services use to authenticate every connection [33, 53]. Each full handshake makes the server sign its CertificateVerify and verify both the client's certificate and the client's CertificateVerify (Section 2.3). Although each connection invokes these operations separately, an event-driven server has many such handshakes live at once, exposing a natural source of cross-connection parallelism: independent ML-DSA operations that can share a SIMD engine without changing the TLS wire protocol or peer behavior.

Prior work shows that vectorized cryptography can improve end-to-end TLS performance. ENG25519 [48] integrates optimized X25519/Ed25519 into OpenSSL, but notes that cross-instance signing and verification depend on peer inputs and require asynchronous request collection across TLS executions. Prior work leaves this integration problem open for peer-dependent authentication within the TLS execution model.

Exploiting that concurrency takes more than conventional SIMD batching. Three constraints meet at the primitive–TLS boundary:

1. ML-DSA signing is *rejection-divergent*: requests require different numbers of attempts, so a fixed SIMD cohort either idles completed request slots or runs at the pace of its slowest member.
2. mTLS verification is *key-heterogeneous*: client CertificateVerify operations usually use unrelated public keys, while issuer verification exposes selective key reuse. A same-key batch API fragments client verification, whereas independent handling discards issuer-key reuse.
3. TLS creates a *closed-loop supply problem* behind

*Corresponding author.

OpenSSL's synchronous high-level EVP interface: if one handshake blocks while waiting for a batch, it withholds the additional connection-level work needed to fill that batch.

These constraints meet at the interface between the cryptographic primitive and the TLS event loop. That interface must support per-job retirement for divergent signing, per-request key identity for heterogeneous verification, and nonblocking collection of compatible work across live TLS connections.

WEAVETLS addresses that joint problem by separating TLS authentication state from the synchronous cryptographic call, combining a batch-capable ML-DSA primitive with a stackless TLS continuation. The primitive independently retires and refills signing jobs, while per-request expanded-key handles support both unrelated client keys and selective key sharing. Suspended authentication state moves into the SSL object, and libssl returns a nonblocking `SSL_ERROR_WANT_BATCH_AUTH` condition. The nginx worker can therefore continue advancing other handshakes, exposing the cross-connection work needed to form a batch without OpenSSL ASYNC fibers or cross-thread handoff.

On the next event-loop turn, the worker forms compatible cohorts and routes each finite cohort among optimized single-request, four-request, and eight-request AVX-512 execution according to measured total cost. We call these execution rungs B1, B4, and B8. Verification can complete several requests together, while rejection-aware signing can stream a larger queue through a bounded active working set. The execution width is therefore independent of the number of jobs admitted by the batching interface.

The stackless path defers server CertificateVerify signing and verification of the client's CertificateVerify; certificate-chain verification stays inline on the optimized single-request path. All three server-side authentication operations remain mandatory. The continuation preserves the TLS authentication barrier: batching changes when authentication work executes, not which checks authorize the connection. WEAVETLS leaves TLS messages, certificates, and peer behavior unchanged.

Across ML-DSA-44/65/87, WEAVETLS improves one-core nginx mTLS throughput by 2.81–4.31× over OpenSSL default and by 2.19–3.29× over a synchronous reference-C control in the same provider. At the primitive boundary, expanded-key B8 verification is 1.65–2.23× faster than a matched cached-AVX2 verifier, while rejection-aware refill makes B8 ML-DSA-65 signing 1.90× faster than otherwise identical lockstep scheduling with a 256-message work supply. Differential tests validate every execution route against a reference implementation, and all 186,270 measured handshakes complete successfully.

Rejection-aware signing remains variable-time, so we also evaluate the client-visible timing boundary. When one client controls all connections, cohort publication reduces the Pearson correlation between signing attempt count and handshake latency from 0.312 to 0.063 at concurrency 16 and from 0.133 to 0.008 at concurrency 64. This is an occupancy-dependent mitigation, not constant-time signing: singleton execution retains the rejection-timing signal.

This paper makes four contributions:

- **Primitive interface.** We design a cross-request ML-DSA interface with independent job retirement and per-request expanded-key handles, supporting heterogeneous keys and selective key aliasing. This boundary lets rejection-divergent signing and heterogeneous-key verification share one SIMD backend without imposing a same-key batch abstraction.
- **TLS architecture.** We build a wire-transparent, stackless OpenSSL/nginx path that stores suspended authentication state in the SSL object, letting the event loop advance other connections and expose cross-connection work. It avoids ASYNC fibers and cross-thread handoff while preserving the TLS authentication barrier and wire protocol.
- **Scheduling.** We develop an occupancy-aware scheduler that separates collection depth from SIMD execution width, combining rejection-aware slot refill with finite-cohort B1/B4/B8 selection by measured total drain cost rather than saturated per-operation throughput. Cohort publication separately controls when completed jobs become externally visible.
- **Implementation and evaluation.** We implement all three FIPS 204 parameter sets and integrate them with OpenSSL and nginx. On one server core, WEAVETLS improves mTLS throughput by 2.81–4.31× over OpenSSL default and by 2.19–3.29× over a synchronous reference-C control in the same provider.

## 2 Background

### 2.1 ML-DSA execution structure

ML-DSA, standardized in FIPS 204 as the successor to CRYSTALS-Dilithium [13, 30], operates in $R_q = \mathbb{Z}_q[X]/(X^{256}+1)$ for $q = 8{,}380{,}417$ and uses SHAKE throughout hashing, expansion, and sampling [29]. Of the three standardized post-quantum primitives (ML-DSA, ML-KEM [7, 31], and SLH-DSA [32]), only ML-DSA repeats its main signing path until it accepts a candidate, which gives it a distinct batching problem. Here $A \in R_q^{k\times\ell}$, $s_1 \in R_q^{\ell}$, and $s_2, t_0, t_1 \in R_q^{k}$; $\tau$ is the challenge's Hamming weight; and $d, \gamma_1, \gamma_2, \beta$, and $\omega$ are the FIPS 204 rounding and acceptance parameters. Table 5 in Appendix A lists the per-set dimensions that determine the polynomial, hashing, and rejection-loop workloads.

We follow the FIPS 204 notation $\mathsf{pk} = (\rho, t_1)$ and $\mathsf{sk} = (\rho, K, tr, s_1, s_2, t_0)$, where $As_1 + s_2 = t_1 2^d + t_0$, and write a signature as $\sigma = (\tilde{c}, z, h)$. For pure-mode signing, $M^\star$ denotes the FIPS 204 message representative and *rnd* the 32-byte signing randomizer. Algorithm 1 shows the structure relevant to our

**Algorithm 1** Core ML-DSA signing and verification dataflow.

**procedure** SIGN($\mathsf{sk}, M^\star, rnd$)
  Decode $\mathsf{sk}$; $\mu \leftarrow H(tr \| M^\star)$
  $\rho' \leftarrow H(K \| rnd \| \mu)$; $\widehat{A} \leftarrow \text{EXPANDA}(\rho)$; $\kappa \leftarrow 0$
  **loop**
    $y \leftarrow \text{EXPANDMASK}(\rho', \kappa)$; $\kappa \leftarrow \kappa + \ell$
    $w \leftarrow \text{NTT}^{-1}(\widehat{A} \odot \text{NTT}(y))$
    $(w_1, w_0) \leftarrow \text{DECOMPOSE}(w)$
    $\tilde{c} \leftarrow H(\mu \| \text{ENCODE}(w_1))$; $c \leftarrow \text{SAMPLEINBALL}(\tilde{c})$
    $z \leftarrow y + cs_1$; $r_0 \leftarrow w_0 - cs_2$; $v \leftarrow ct_0$
    **if** $\|z\|_\infty \geq \gamma_1 - \beta$ **or** $\|r_0\|_\infty \geq \gamma_2 - \beta$ **or** $\|v\|_\infty \geq \gamma_2$ **then**
      **continue**
    **end if**
    $h \leftarrow \text{MAKEHINT}(r_0 + v, w_1)$
    **if** $\text{wt}(h) > \omega$ **then**
      **continue**
    **end if**
    **return** $\text{SIGENCODE}(\tilde{c}, z, h)$
  **end loop**
**end procedure**

**procedure** VERIFY($\mathsf{pk}, M^\star, \sigma$)
  Decode $(\rho, t_1)$ and $(\tilde{c}, z, h)$
  **if** decoding fails **or** $\|z\|_\infty \geq \gamma_1 - \beta$ **then**
    **return** 0
  **end if**
  $\mu \leftarrow H(H(\mathsf{pk}) \| M^\star)$; $c \leftarrow \text{SAMPLEINBALL}(\tilde{c})$; $\widehat{A} \leftarrow \text{EXPANDA}(\rho)$
  $w' \leftarrow \text{NTT}^{-1}(\widehat{A} \odot \text{NTT}(z) - \text{NTT}(c) \odot \text{NTT}(t_1 2^d))$
  $w'_1 \leftarrow \text{USEHINT}(h, w')$
  **return** $[\tilde{c} = H(\mu \| \text{ENCODE}(w'_1))]$
**end procedure**

implementation: matrix expansion, polynomial transforms, candidate acceptance, and the fixed verification path.

Verification has no iterative acceptance loop. Its matrix–vector computation, hint reconstruction, and challenge comparison follow a fixed stage structure, so independent verifications admit lockstep cross-request execution. Which axis supplies parallelism still changes by stage, since matrix expansion exposes $k\ell$ independent SHAKE streams within one request (30 for ML-DSA-65), enough to occupy an eight-way Keccak implementation, whereas the polynomial stages expose the request axis.

Signing instead repeats candidate generation until all norm and hint bounds hold. ML-DSA's Fiat–Shamir-with-aborts structure [26, 27] therefore gives independent messages different attempt counts [2, 12], so cross-request signing must accommodate requests that finish after different numbers of iterations. FIPS 204 separately permits caching expanded keys [30, Sec. 3.6.3]. The cacheable verification state consists of public deterministic functions of pk: the NTT-domain public matrix, the transformed public vector, and the public-key hash. Because an mTLS workload may present unrelated client keys alongside repeated issuer keys, cross-request execution must preserve both key identity and selective reuse.

## 2.2 AVX-512 and the choice of lane axis

AVX-512 exposes sixteen 32-bit lanes per 512-bit ZMM register, an element width that matches ML-DSA arithmetic: since $q < 2^{23}$, a coefficient and its reduced intermediates fit in one lane. Mask registers provide per-lane predication, and AVX-512VBMI/VBMI2 byte permutes support ML-DSA's packed encoding boundaries. What remains is which values occupy those lanes.

An *intra-instance* layout places coefficients from one polynomial in the same register, exposing parallelism within a single operation but forcing intra-register lane rearrangement in the NTT stages that pair coefficients inside it. At the *cross-instance* endpoint, each lane holds the same coefficient index from a different request, so every lane executes the same butterfly with one broadcast twiddle and native masks track request validity independently. That layout pays for it with transposition at the packed-byte boundary, a live working set that scales with the request count, and a need for enough independent operations to fill the request axis. Neither endpoint dominates. Intermediate layouts fill a register with fewer than sixteen requests by giving each request several adjacent coefficients, trading a smaller active request set for local rearrangement in the final transform stages.

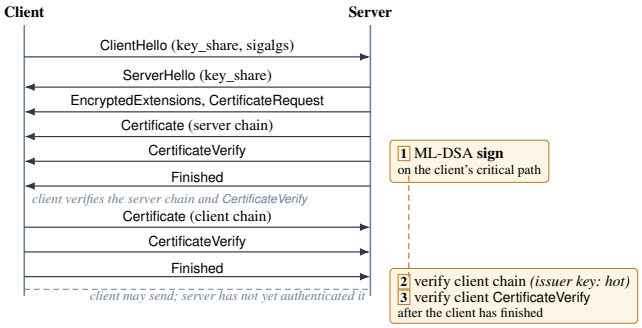



Figure 1: TLS 1.3 handshake with client authentication. Each full handshake contributes one server-side ML-DSA signature 1 on the client's critical path and two mandatory verifications 2 3 after the client's final flight. The server must complete both checks before it authenticates the client, even though the client may finish its side of the handshake first.

## 2.3 The TLS 1.3 mutual-authentication handshake

TLS 1.3 authenticates in a single round trip [38]. Figure 1 shows the full handshake with client authentication enabled and identifies the three server-side ML-DSA operations relevant to this work.

The server signs its own CertificateVerify (1) before sending Finished, placing that operation on the client's critical path. After the client's final flight, the server performs two mandatory verifications: one on the issuing certificate authority's (CA's) signature over the client certificate (2) and the other on the client's CertificateVerify (3). Each full mTLS handshake therefore contributes a fixed 2:1 verify-to-sign mix.

Two properties shape the rest of the paper. The verification classes differ in key locality: certificate-chain verification

repeatedly uses the issuer's public key, whereas the CertificateVerify check uses the client's own, so the workload combines stable key reuse with heterogeneous end-entity keys. Each handshake also invokes EVP separately and owns its own transcript, verification result, and authentication decision. Cross-connection execution therefore needs a mechanism that suspends and resumes authentication without changing TLS semantics. OpenSSL's ASYNC framework provides one, suspending provider calls through resumable fibers and application-visible wait descriptors [35]. This design suits accelerators with a separate scheduling domain, but for CPU-resident batching those transitions run on the core already driving the event loop.

# 3 Cross-Request ML-DSA Engine

WEAVETLS's ML-DSA engine separates request identity from SIMD layout: it receives independent messages, key state, and result locations, then routes each stage along the request or coefficient axis matching its available parallelism. That boundary lets signing retire requests independently and verification retain one key identity per request.

## 3.1 Primitive boundary and parallelism routing

The engine exposes one primitive interface inside and outside TLS. Signing accepts an expanded secret key, arrays of independent messages, and one hedging input per message; verification accepts packed signatures and messages together with one expanded-public-key handle per request; both return results independently for each request. The TLS provider is one producer of these arrays, but the primitive neither parses TLS objects nor depends on OpenSSL job state.

Throughout the engine, *request identity* stays independent of physical SIMD placement. Each request owns its message, key state, output, and, for signing, rejection-loop state. We use *lane* for one physical vector element and *slot* for one independent request position in an active working set, so a slot says where a request currently executes, not what it is. That lets the engine change vector organization between stages without changing primitive semantics. An implementation with $B$ simultaneously active slots is a $B$-slot *execution rung*, and we provide B1, B4, and B8. Here $B$ counts independent requests, not vector lanes: B1 still exploits intra-request AVX-512 parallelism, whereas B4 and B8 use our folded cross-request layout. Execution width is likewise distinct from the number of requests a caller admits; Section 4 composes finite TLS cohorts from these rungs.

The engine changes the SIMD axis at selected stage boundaries according to the parallelism each operation exposes. Polynomial arithmetic uses 32-bit lanes, so at B4 and B8 a 512-bit ZMM register holds $B$ request slots and $F = 16/B$ adjacent coefficients from each, giving a folded request–coefficient layout. SHAKE instead operates on eight independent 64-bit Keccak states, which the request axis fills directly only when it supplies enough streams; below that, stage-specific intra-request parallelism supplies the rest. Explicit boundaries let each operation use its natural SIMD axis without imposing a global layout. Sections 3.2–3.4 instantiate this routing for folded polynomial execution, expanded-key matrix products, and rejection-divergent signing.

## 3.2 Coefficient-major execution

Section 2.2 leaves two endpoints: a pure cross-request layout needs sixteen simultaneous requests, while an intra-request layout must rearrange lanes wherever butterfly partners share a register. The engine executes between them. Take one polynomial position across $B$ independent requests, writing $p_0, \ldots, p_{B-1}$ for those polynomials, and let $F = 16/B$. Vector row $r$ is

$$P[r] = \langle p_0[Fr{:}Fr{+}F{-}1], \ldots, p_{B-1}[Fr{:}Fr{+}F{-}1] \rangle.$$

Each request owns one contiguous $F$-lane group of a ZMM register: $F = 2$ adjacent coefficients per request at B8, and $F = 4$ at B4, where a request's fold is exactly one 128-bit quarter.

A butterfly whose coefficient distance is at least $F$ pairs coefficients at the same lane position of two rows, so it stays a cross-vector butterfly with a broadcast twiddle and no movement between request slots; only the last $\log_2 F$ layers pair coefficients inside one request's own fold. Elementwise stages (pointwise multiplication, reduction, rounding, norm checks) are insensitive to $F$. B4 and B8 are separate native-width datapaths; neither runs by padding a narrower cohort into a wider kernel. Each working set carries a $B$-bit validity mask, one bit per request. Kernels check coefficient lanes and emit a 16-bit failure predicate. The engine reduces this predicate by disjunction over the $F$ lanes owned by each request and clears only that request's mask bit. A request therefore fails exactly when one of its own coefficients fails.

Encoded objects cross into the folded form once: public keys and signatures arrive instance-major, so the decoder widens coefficients within one object and then transposes into folded rows that arithmetic keeps until final encoding. Hint decoding shares this boundary, clamping every index before the access it controls and recording format violations, so a malformed request stays on the common schedule.

**Folded transforms.** Folding pays only if the butterflies that dominate the transform stay in registers between layers. The eight NTT layers run as two register-resident passes of four on a width-independent schedule that uses the same distances and twiddle indices at every $B$, with $F$ deciding only how they map onto rows. The two rungs therefore differ only in where a distance falls inside a row. B4 transposes four consecutive folded rows inside each 128-bit quarter, which contains

one request's fold. Its last two layers then become full-width cross-vector butterflies. B8 resolves its single fold-local layer with local shuffles and a masked add/subtract. Neither transformation moves data between request slots. Twiddles carry a precomputed Montgomery companion $\zeta q^{-1}$, reducing the correction operand to one low-word multiply. Tiling regroups only independent butterflies, so the schedule preserves the reference reduction points and coefficient bounds. Appendix C gives the layer distances, the transpose network, and the shuffle schedules.

**Eight-way Keccak and stream grouping.** Every SHAKE stream runs on an eight-way AVX-512 Keccak-f[1600] permutation over eight independent 64-bit states, reused from the published vectorized lineage [1, 23, 41]. What the folded layout constrains is not that permutation but the grouping around it: a group consumes eight streams and leaves its output transposed across those states, whereas polynomial arithmetic consumes $B$ requests as folded 32-bit rows. Each hashing stage therefore fixes which algorithmic axis supplies its eight streams and where the two representations convert.

ExpandMask makes that choice concrete. Its input is fixed at a 64-byte $\rho'$ prefix and a little-endian 16-bit nonce, so the engine writes the padded block straight into the transposed eight-stream state instead of assembling it in memory. A secondary axis then enters only when the request axis cannot saturate the engine. At B8 it can: eight requests give eight streams, and one group yields one $y$ polynomial for the working set. Four requests would leave every group half used, so B4 also recruits the polynomial axis ExpandMask already carries. ML-DSA-65's five $y$ polynomials across four requests give 20 streams, filling two groups and a half-used third instead of five half-used groups. Section 3.3 applies the same rule to matrix expansion.

**Arithmetic streaming boundary.** Matrix products stream without materializing intermediate matrices. One fused kernel carries a multiply–accumulate result through reduction and into the inverse transform, which folds its final scaling and correction into the store. An out-of-place forward-transform entry removes a pre-transform copy of $y$. Multipliers shared across request slots reuse the same precomputed Montgomery companion. Section 3.3 describes how cold matrix expansion and expanded-key state supply this boundary.

## 3.3 Streaming matrix products and expanded keys

Signing and verification both evaluate an NTT-domain product

$$\widehat{w}_i = \sum_{j=0}^{\ell-1} \widehat{A}_{i,j} \odot \widehat{x}_j, \qquad 0 \le i < k,$$

where $x$ is $y$ during signing and $z$ during verification. What differs is how $\widehat{A}$ reaches this boundary. Signing always consumes a resident matrix prepared once from the signing key. Verification supplies it in one of two ways: a *hot* request reuses an expanded-key entry, while a *cold* request generates $\widehat{A}$ from its own seed. On the cold path, the $B$ requests carry unrelated seeds. For each $(i, j)$, a group of eight SHAKE128 states absorbs $\rho \| j \| i$. The request axis fills those groups at B8; below B8 the secondary axis is the $\ell$ matrix-XOF streams of one row, so $\widehat{A}_{i,0}, \ldots, \widehat{A}_{i,\ell-1}$ are grouped across the $B$ requests. Because rejection sampling accepts only a sparse subset of candidates, the parser compacts the accepted ones while preserving stream order.

The accepted coefficients initially form $B$ instance-major polynomials. The cold path therefore runs SHAKE, reject-parse, transpose, and Montgomery multiply–accumulate consecutively. It transposes each $(i, j)$ polynomial into coefficient-major form and immediately accumulates it into row $i$, keeping at most one row of $\widehat{A}$ live. Installing an expanded-key entry materializes the whole matrix, but the same pass writes each accepted polynomial directly to its destination while the row remains resident. No temporary copy of $A$ exists. Cross-request execution accelerates the serial hashes and polynomial stages, though matrix expansion already exposes $k\ell$ independent XOF streams inside one request, limiting the extra speedup available during cold expansion.

Expanded-key entries are keyed by public-key identity. Each stores the parsed NTT-domain matrix $\widehat{A}$, $\mathrm{NTT}(t_1 2^d)$, and $H(\mathsf{pk})$ under least-recently-used (LRU) eviction, and an invocation receives one entry pointer per request, naming unrelated keys or aliasing one issuer key. Hot and cold verification requests never share one invocation, so a miss never demotes an otherwise hot working set to expansion, and a cold invocation expands directly into the entries it has been given.

Three checks preserve key-state integrity. A candidate hit found through a short seed tag is confirmed against the complete encoded public key, which alone establishes identity. The cache pins every referenced entry until the invocation completes. An entry becomes valid only after complete installation, so no request observes a partially constructed key.

Because each request carries its own handle, the interface needs no same-key batch API: the generic kernel gathers and transposes coefficients from $B$ independent handles, while a pointer-equality test selects a broadcast kernel whenever every handle aliases one entry, as in issuer verification. That test reads only handle identity, never signature contents or secret state. An ML-DSA-65 entry consumes 36.1 KiB, so cache capacity is a deployment parameter and not a property of the interface, and B1, B4, and B8 share one handle representation, leaving what is cached unchanged as rungs compose.

## 3.4 Rejection-aware signing

The signer holds a queue of pending jobs and a working set of $B$ request slots. Each iteration computes one candidate per occupied slot and combines the norm and hint checks into

a success mask. A failed request keeps its slot, its message identity, and its rejection state, and advances its own nonce; a successful request encodes its signature and retires, after which its slot takes the next queued job. When fewer jobs remain than the working set holds, the unused slots leave the occupancy mask and their output goes to scratch storage: the vector unit still evaluates them, but they are never retired and nothing they produce reaches a caller buffer.

Refill needs a queue deeper than one working set, since a call carrying at most $B$ jobs drains before any slot can be replaced. The primitive therefore accepts arbitrary depth, and the deployed provider admits a *superbatch* of up to 64 compatible jobs: depth is what the queue admits, width is what one iteration executes. A retirement callback keeps internal slot reuse separate from external completion. The primitive *retires* a request as soon as its slot succeeds, handing over the fully encoded signature and dropping every reference to its storage; whether the caller *publishes* that completion immediately or holds it until a later boundary is a scheduling decision made in Section 4.

Each attempt executes one pass of Algorithm 1's rejection loop for every occupied slot and advances those slots' nonces (Algorithm 2, Appendix C). Refill preserves every FIPS 204 acceptance condition and changes only which request occupies a slot once an attempt succeeds, so a slot falls idle only while the admitted queue is draining.

The scheduling effect is clearest in units of *slot iterations*: one lane of the $B$-wide working set executing one pass of the rejection loop, counted independently of cycle cost. One iteration of the working set therefore costs $B$ slot iterations whether or not every slot is occupied, because the vector unit evaluates the whole set. Let $T \in \{1,2,\dots\}$ be the attempts one signature requires, geometric with per-attempt success probability $p$, so $\mathbb{E}[T] = 1/p$. Under an unbounded refill queue, every slot always holds a request that still needs work, costing $\mathbb{E}[T]$ slot iterations per completed request. A lockstep group of $B$ requests instead holds all $B$ slots until its slowest member succeeds, costing

$$\mathbb{E}[\max_i T_i] = \sum_{r\geq 0} \left[1 - (1-(1-p)^r)^B\right]$$

per completed request, so the ratio of the two isolates the schedule.

A finite queue never reaches that limit. Once the admitted queue empties, a retiring slot cannot be refilled, so the working set runs at falling occupancy while each iteration still costs the full $B$ slot iterations, raising slot work per completed request above $\mathbb{E}[T]$. The model accounts for that tail. Measured cycles additionally carry packing, retirement, memory traffic, and kernel cost, so slot work and cycles measure complementary layers; Section 6 reports both.

**Hedged randomness.** Randomness is job-local: each request carries its own 32-byte *rnd*, from which the signer derives that request's $\rho' = H(K \,\|\, rnd \,\|\, \mu)$ as FIPS 204 specifies. Since

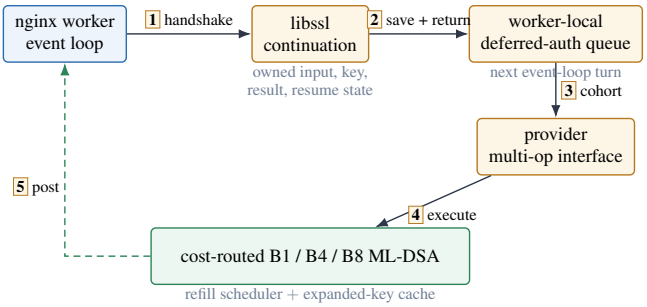



Figure 2: Stackless authentication data path. 1–2: an eligible operation's inputs and resume state move into the `SSL` object and libssl returns `SSL_ERROR_WANT_BATCH_AUTH`. 3–4: the worker's deferred-authentication queue is flushed as cohorts through the provider's multi-operation entry point. 5: once the whole flush has completed, its connections resume through nginx's ordinary posted events. No suspended stack or provider frame crosses these boundaries, and the path adds no fiber, wait descriptor, mutex, helper thread, or cross-thread handoff.

$\rho'$ and the nonce belong to the job rather than the slot, a job refilling a slot brings its own randomness and begins its own rejection state, so slot reuse never shares a masking sequence between requests. Deterministic signing is the same path through a wrapper supplying the standardized all-zero *rnd*; both modes present identical input lengths to every hash and share one dataflow.

## 4 System Design

WEAVETLS makes TLS authentication a stackless continuation owned by the connection. Instead of preserving a provider call frame, libssl stores the authentication inputs and resume state in the `SSL` object and returns control to the event-loop worker. The worker can then advance other connections, expose compatible work, and execute a finite cohort through the provider's multi-operation entry point. Returning from the current TLS call breaks the closed-loop supply problem, since one handshake no longer prevents the worker from discovering the other requests needed to populate a cross-request execution rung. Figure 2 shows this data path.

Ownership follows component boundaries. The `SSL` object holds the suspended connection state, the nginx worker owns collection and scheduling, and the provider performs only finite multi-operation execution. No suspended C stack or provider frame crosses these boundaries. A *cohort* is a compatible set of requests passed to one multi-operation call and served by the B1/B4/B8 rungs of Section 3.1; one worker flush may form several cohorts.

## 4.1 Provider interception and the multi-operation boundary

The provider registers the three standard ML-DSA names for key management and signatures. OpenSSL property selection directs those fetches to WEAVETLS while applications and peers retain the standard algorithm identifiers. Key management imports packed octet strings, and `OSSL_FUNC_PROVIDER_GET_CAPABILITIES` advertises the corresponding TLS signature schemes. Because provider dispatch is single-operation, the multi-operation boundary sits outside it: the module exports two additional symbols, one per operation, which the patched libssl resolves from the already-loaded module with `dlopen` and `dlsym`, leaving the published `OSSL_DISPATCH` table unchanged.

Suspension makes that boundary necessary, since the call frame carrying an operation's inputs unwinds completely before the worker executes anything. libssl therefore takes ownership of the `EVP_MD_CTX`, copies the transcript-derived message, allocates or copies the signature buffer, and calls `EVP_PKEY_todata` on the key bound to that operation, keeping a clear-on-free copy of the private octet string for signing or the public one for verification. The library attempts the export before suspension. Export failure does not bypass authentication: libssl still defers the request and later executes it through the retained EVP context.

A flush partitions its pending requests into cohorts and calls the private entry point once per cohort with the arrays the primitive boundary of Section 3.1 expects: keys, messages, signature buffers, and per-request result slots. Requests join one cohort when they agree on operation, parameter set, and message length, and, for signing, on the key bytes themselves; equal length is required because the eight-way Keccak stage absorbs one common input length across its eight streams. Verification cohorts compare no key material and may therefore carry unrelated public keys, matching the heterogeneous-key interface of Section 3.3, whereas signing cohorts share one key, as the refill queue of Section 3.4 requires and TLS server-key reuse already provides.

Whatever the private path does not execute falls back to ordinary EVP. The entry point accepts or declines a cohort as a whole, and a request whose key export failed is never offered to it; the flush completes each remaining request synchronously on the context it still owns. Because that context carries the originally selected provider and key, fallback preserves the selected algorithm and operation inputs. Every request executes on exactly one path.

## 4.2 Stackless suspension and the authentication barrier

Server CertificateVerify suspends inside message construction. The library assembles the transcript-derived input and transfers the signature context and a copy of that input to the `SSL` object. It then returns `SSL_ERROR_WANT_BATCH_AUTH` before writing any byte of the message body. The enclosing `WPACKET` is local to the write state machine's frame and cannot outlive that return. The state machine therefore discards it and rewinds to the pre-work state of the same write step. On resume, it rebuilds the message around the signature produced by the worker.

Incoming client CertificateVerify suspends one step later, in the post-processing state after message parsing. The `SSL` object owns the received signature bytes, transcript-derived input, exported public key, signature context, and verdict slot until the worker fills it. This state precedes every transition that advances the handshake past CertificateVerify. While the verdict is pending, the server cannot read Finished and therefore cannot authenticate the peer, deliver application data, or issue a session ticket. A failed verdict raises the same decrypt-error alert and bad-signature reason as the synchronous check. Batching thus changes when the check runs, not whether it must succeed before the peer is authenticated.

The deployed path suspends server CertificateVerify signing and the client CertificateVerify check, the two operations whose placement exposes cross-connection work, and keeps certificate-chain validation inline on cached B1. All three remain mandatory, and both client-authentication verifications must succeed before the barrier is crossed.

The nginx worker treats the new error as a nonblocking condition and links the connection onto a worker-local deferred-authentication queue, distinct from nginx's posted-event queues. One flush event is posted to the next event-loop turn and never re-posted, so every connection suspended before that turn begins joins the same flush. The collection window lasts one loop turn, and the worker does not wait for a target width. `SSL_batch_auth_flush` then partitions the queued requests into cohorts, runs each through the private entry point, completes the remainder through EVP, and returns with every request finished, after which the worker posts those connections to nginx's ordinary posted-event queue. Since connection state stays in its own `SSL` object and collection and execution both run on the owning worker, this path transfers nothing between threads and needs no `ASYNC_JOB`, alternate stack, wait descriptor, flusher thread, or cross-thread wakeup.

## 4.3 Finite-cohort cost routing

A cohort admits several decompositions: eight verifications can run as one B8 call, as two B4 calls, or as eight serial B1 calls. The dispatcher takes whichever minimizes the measured cost of draining that finite cohort, rather than the widest linked rung or the one with the best saturated throughput. Signing adds the drain of Section 3.4: once the admitted queue empties, retired slots cannot be refilled and the remaining iterations run at falling occupancy, so a wider working set can lose to a narrower one on a short cohort.

Saturated throughput does not predict finite-cohort cost.

Deep-queue per-operation figures omit the drain tail and a rung's fixed per-call costs keep even rejection-free verification away from its limit; the slot-work model predicts scheduling work rather than cycles, so it sets no thresholds either. We therefore calibrate whole-cohort cycle cost directly, timing each candidate decomposition of finite cohorts in the binary that links every production rung. Calibration is offline: the runtime dispatcher applies the resulting thresholds as a fixed comparison and benchmarks nothing on the critical path. The thresholds depend on the processor, linked binary, and bounded admission policy. The routing mechanism is portable, but each deployment must recalibrate its crossover values; Section 6 reports those used here.

The deployed verification engine takes at most eight requests per call, so a longer cohort is split into successive calls. Within one call, a singleton routes to B1, two to four requests to B4, five to B4 followed by B1, and six or more to B8; two ML-DSA-87 requests are one calibrated parameter-specific exception and run as two serial B1 calls. The signing engine takes up to 64 jobs per call, which the refill scheduler feeds through a narrower working set, and selects between serial B1 and B4 at a per-parameter-set job count. B8 signing kernels are linked and evaluated, but under the current 64-job bound their total-cost advantage over B4 stays inside the measurement noise, so the calibrated selector never chooses them; B8 remains the widest verification rung. Sparse load consequently reaches this selector as short cohorts, which it routes to B1.

**Cohort publication.** Retirement stays internal: the deployed path passes no retirement callback, and a suspended connection resumes only after `SSL_batch_auth_flush` returns. Because one flush may form several cohorts, the set that becomes externally visible together is the flush, so publication is never finer than one multi-operation call. It is independent of width selection, and it changes when a completion is visible rather than the rejection-dependent work behind it; Section 5 analyzes the observable that remains.

# 5 Security Analysis

## 5.1 Threat model and security invariants

Our adversary controls remote peers that open and close connections, choose handshake timing, present arbitrary certificates and signatures, observe their own latency and failures, and coordinate connections to influence cohort composition. The trusted computing base is the server process, the provider, and secret-key files; ML-DSA cryptanalysis, local privilege escalation, and a compromised local co-tenant are out of scope.

Each request still computes the same FIPS 204 operation under the same acceptance predicate, carrying its own rejection state, and cross-request execution alters neither the ML-DSA algorithm nor the wire format, so analyses of its abort structure continue to apply [2]. Batching introduces shared scheduling: several requests occupy one execution and publication boundary, and a handshake suspends and resumes around work belonging to others. Four invariants constrain this sharing. *Barrier preservation*: batching and suspension change when authentication work executes, never whether a connection may cross the TLS authentication barrier before every mandatory check has succeeded. *Request isolation*: malformed input, rejection state, validity masks, outputs, and verdicts belonging to one request cannot affect another. *Key-state integrity*: every verification request uses expanded state derived from its own public key, except where handles intentionally alias one shared issuer entry. *Completion semantics*: batching changes execution order and publication time, not the operation performed or its accept/reject result. Section 5.4 adds a bounded-admission property and the residual leakage that none of these invariants removes.

## 5.2 Authentication and request isolation

The authentication barrier is positional. A suspended client CertificateVerify is parked in the post-processing state of that message, which precedes every transition that advances the handshake; while the verdict is pending the server cannot read Finished and therefore cannot authenticate the peer, deliver application data, or issue a session ticket. A failed verdict raises the same alert and error as the synchronous check, and server CertificateVerify suspends before any byte of the message body is written, so suspension never exposes partially authenticated wire state. Deferring a check moves it in time without moving it across the barrier.

Within one execution, three mechanisms enforce request isolation (Section 3.2). Every decoding error clears only the originating request's validity bit. The decoder clamps each untrusted index before use and records the failure. The engine reduces each lane-failure mask only over lanes owned by the same request, so malformed input cannot clear another request's bit. Later arithmetic operates on sanitized request state, and the final verdict includes the validity mask. Section 6.2 validates this empirically with a negative-test matrix that corrupts every signature field across several request positions and rejects only the corrupted requests.

Expanded-key entries carry the invariant across invocations (Section 3.3). Aliasing therefore records that two requests presented the same key, whose expanded form is a public function of it and exposes no signing secret.

## 5.3 Signing leakage and publication

ML-DSA signing is rejection-dependent and therefore variable-time. Refill changes how efficiently slots are used, not the secret-dependent acceptance predicate: our signer computes rejected candidates and drives its slot masks along

data-dependent paths. Attempt counts appear nowhere in the wire format, but completion time is remotely observable.

Hedged signing addresses a different problem. Every signature receives fresh, independent 32-byte randomness through the randomized FIPS 204 interface, so repeated messages get independent masking sequences, addressing the reuse exploited by differential fault attacks on deterministic lattice signing [9, 37]. Hedging does not hide rejection timing. Within our remote threat model, the relevant observable is client-visible completion latency. Local cache, physical-trace, and fault attacks are out of scope, and WEAVETLS offers no resistance to them. In both hedged and deterministic modes, our signer retains the rejection-dependent intermediate state and sampling behavior that prior attacks target [8, 25, 49, 52].

That observable is sensitive to publication policy. Publishing a request as soon as the primitive retires it couples its completion time directly to its own attempt count. *Cohort publication* instead withholds completion until the enclosing multi-operation call returns, and in the deployed worker until the whole flush returns, so a request's visible latency is mixed with that of the other requests published alongside it. Where occupancy is high enough that a flush publishes several requests, this attenuates the measured correlation between a connection's attempt count and its own observable latency. The signing work itself still depends on rejection count, and a singleton cohort publishes one request and retains the full signal, so the effect scales with occupancy and does not provide constant-time signing. Section 6.5 quantifies the residual signal.

### 5.4 Batch shaping and resource interference

Cohort membership follows public request shape and, for signing, the server-selected key, so a remote attacker can deliberately fragment compatible work, with three consequences. Occupancy degrades: incompatible shapes split one flush into short cohorts and remove SIMD gain. Requests interfere: a request waits for the others in the finite publication group it is flushed with, so cross-request latency coupling is real but bounded by that group. Ordinary capacity remains consumable: an attacker can exhaust TLS handshake and signature capacity exactly as against an unbatched server, which nginx connection, timeout, and rate limits govern.

The design bounds batching-specific delay because no request waits for a target width. Collection ends at the next event-loop turn, not at a target occupancy. No batching timer exists on the path, admitted cohorts are finite, and the superbatch size caps signing admission. The dispatcher sends a short cohort to calibrated B1 or B4 execution without holding it. Fragmentation therefore reduces throughput without creating an unbounded wait. An invalid verification input consumes at most one slot after bounded decoding, and the state machine selects the server credential before signing, so a peer can request only protocol-defined CertificateVerify operations. Flush behavior does leak coarse aggregate load: a lone connection takes the B1 route while compatible connections at a busy server share B4/B8 service, so probes can distinguish idle from loaded operation classes. That signal depends on public request shape and aggregate occupancy rather than on signing-key material or tenant plaintext, and WEAVETLS offers no workload confidentiality among mutually distrustful clients sharing one worker.

### 5.5 State ownership and cleanup

Suspension removes a call frame instead of relocating it: neither a provider frame nor an ASYNC alternate stack survives the return. That also removes the stack-sizing dependency ASYNC imposes when a post-quantum provider call overruns the default fiber allocation, since cryptography runs on the ordinary stack of the worker owning the connection. The `SSL` object owns every value that must outlive the frame: the signature context, transcript-derived input, exported key copy, output buffer, and verdict slot. The library clear-frees these values on completion or connection destruction (Section 4.1). Each path also marks the requests it completes, and the flush's fallback loop visits only those still pending, so a deferred request runs on the private multi-operation path or on the retained EVP context rather than on both. Non-benchmark builds run under sanitizers (Appendix A).

## 6 Evaluation

We answer five questions. (1) *Correctness*: do the SIMD engine, refill scheduler, and stackless continuation preserve per-request cryptographic and TLS semantics? (2) *Primitive performance*: how much do cross-request execution, expanded-key reuse, and the B1/B4/B8 kernels improve ML-DSA cost? (3) *Scheduling*: how much does refill recover from signing divergence, and why does finite-cohort routing rank widths differently from saturated throughput? (4) *End-to-end impact*: do those gains survive OpenSSL and nginx well enough to improve one-core mTLS throughput and latency? (5) *Timing and portability*: how does cohort publication affect client-visible rejection timing, and do the performance mechanisms transfer to a second AVX-512 processor?

### 6.1 Methodology

The server is an AMD Ryzen 9 9950X3D. Microbenchmarks and the nginx worker run pinned to physical core 8 with boost disabled, the governor set to `performance`, a measured 4.30 GHz load frequency, and the core's simultaneous-multithreading sibling idle; capacity experiments drive the server from a co-located loopback load generator on disjoint cores. A process result is the median of 31 samples and each table reports the median of three fresh process starts. Across-process spread is 0.7–2.6% for verification and below

1.7% for signing, so we treat differences within 3% as noise throughout.

Primitive comparators are pq-crystals reference C, its official AVX2 implementation, Xu et al.'s published AVX-512 Dilithium implementation [46], our optimized B1 rung, and a matched cached-AVX2 verifier holding the same public-key-derived objects as our backend. The system experiments use OpenSSL 3.5.7 and nginx 1.29.8 with an all-ML-DSA certificate chain, one pinned worker, a 64-entry expanded-key cache, and no helper thread, and each concurrent client completes a TLS 1.3 full mTLS handshake and closes. Three configurations share certificates, nginx, and CPU placement. *OpenSSL default* represents the deployed implementation boundary. *Provider C* keeps the WEAVETLS provider and key management but signs and verifies synchronously with reference C, making it a system-level control instead of an optimized primitive baseline. WEAVETLS uses the stackless continuation and production B1/B4/B8 dispatcher; its single-request rung uses Xu-derived FIPS 204 AVX-512 with cached expanded public keys. Position-balanced runs report client concurrency $c = 64$, our saturated comparison point, and have zero client failures. Appendix A gives the measurement contract, the comparator boundaries, and the run bookkeeping.

## 6.2 Correctness

For all three parameter sets, we differentially test eight-way SHAKE, every SIMD arithmetic kernel, packing, verification, and both schedulers against FIPS 202, pq-crystals intermediate values, or complete reference outputs. The folded schedules reproduce reference signatures byte for byte under deterministic and hedged signing, the attempt count per completed signature matches the reference rejection behavior, and every superbatch job retires once after packing a reference-valid signature. We test isolation with a negative matrix that corrupts the signature, message, public key, hint ordering, hint weight, trailing hint bytes, norm bound, and challenge across several request positions and across two requests at once, and the verifier rejects only the corrupted requests. The production dispatcher exercises every verification cohort size $n = 1, \ldots, 8$ and each signing crossover in both modes; OpenSSL state-machine tests cover suspension, server-CertificateVerify reconstruction, the client-verdict barrier, and the EVP fallback. Appendix B gives the test matrix and boundary cases.

## 6.3 Primitive performance

**Primitive speedups.** Against established single-instance paths, cold B8 verification is 1.61× faster than the reproduced AVX-512 comparator and 2.90× faster than official AVX2, and caching removes matrix expansion and parsing for a further 3.01×; under a 256-message work supply the B8 refill signer is 16.2× faster than reference C, 2.23× faster than cached AVX2, and 1.41× faster than the optimized Xu-derived B1 rung (Table 1). Because those signing rows use different cached-state and work-supply boundaries, the matched comparisons below carry the causal claims.

Table 1: ML-DSA-65 primitive costs (cycles per completed operation). Verification starts from packed public keys. Signing starts from cached or expanded secret-key state, with key precomputation timed separately, and its rows differ in supply: the AVX2 row is a cached-key residual, the B1 row is isolated single-request execution, and the WEAVETLS row is fed by a 256-message queue. Table 2 covers the expanded-key verification boundary.

| Implementation | Verify | Sign |
|---|---|---|
| pq-crystals reference C | 371,420 | 1,554,041 |
| pq-crystals AVX2 | 120,654 | 214,477[†] |
| Xu et al. AVX-512 | 67,015 | n/a |
| Xu-derived FIPS 204 B1 | 111,154 | 135,926 |
| WEAVETLS B8 cold/refill | **41,655** | **96,154** |

[†]Cached-key residual; setup measured separately.

**Expanded-key verification.** Table 2 provides a matched comparison: both paths start from the same expanded public-key-derived state and run in one binary on one input set. Against the cached-AVX2 verifier, the heterogeneous-key B8 path is 1.65–2.23× faster, our headline expanded-key result, with a complementary 1.31–1.83× against the isolated optimized B1 verifier. The final three columns instead hold the key constant, comparing eight separately stored copies with eight pointers to one entry: broadcasting adds 1.25–1.37×. That figure is the benefit of selective same-key reuse, such as one issuer certificate, not a general verification gain.

**Refill versus lockstep.** On ML-DSA-65 the B8 refill scheduler requires 96,154 cycles per completed signature, versus 183,032 cycles for the otherwise identical lockstep scheduler, a 1.90× improvement. The comparison holds instruction selection, SIMD layout, cached signing-key state, and work supply constant, isolating the scheduling effect from the arithmetic kernels. Hedged signing presents the same input lengths to every hash as deterministic signing and measures within 0.2% of it.

**Execution-rung characterization.** On a 256-message queue, B8 lowers signing cost by 6.0–10.7% relative to B4 and is the fastest deep-queue rung for every parameter set; the independent ML-DSA-65 width run gives 96,078 cycles, within 0.1% of the scheduler comparison above. Table 8 reports the full width matrix, whose columns characterize rungs under different supply regimes, not deployed routing.

**Ablations.** Folding pays only in the final transform layers: elementwise cost remains unchanged between B4 and B8, while B4's local transpose reduces its forward transform by 12.8% and its inverse by 10.0%. Recruiting a second algorithmic

Table 2: Expanded-public-key verification (cycles/operation), measured in one binary on one input set. "Copied" uses eight equal but separately stored entries; "Shared" aliases one entry.

| | Heterogeneous-key comparison | | | Shared-key comparison | | |
|---|---|---|---|---|---|---|
| Set | Cached AVX2 | B8 heterogeneous | Gain | Copied | Shared | Gain |
| ML-DSA-44 | 22,684 | 10,155 | 2.23× | 10,226 | 8,172 | 1.25× |
| ML-DSA-65 | 27,074 | 13,847 | 1.96× | 13,848 | 10,821 | 1.28× |
| ML-DSA-87 | 34,170 | 20,694 | 1.65× | 20,667 | 15,050 | 1.37× |

axis pays wherever the request axis cannot fill a Keccak group: at B4, grouping ExpandMask by request and $y$ polynomial reduces signing by 12.1%, and grouping matrix expansion by request and row reduces ML-DSA-65 cold verification by 25.0%. Appendix C reports the remaining kernel-level ablations.

**Finite-cohort scheduling.** Because a short cohort drains through partially occupied iterations, a wider kernel starts behind B1 and may need tens of jobs to recover that cost. We time every decomposition of finite cohorts drawn from a 256-message pool in the binary that links all six B4/B8 kernels. The first cohort sizes routed from B1 to B4 are 11, 24, and 40 jobs for ML-DSA-44/65/87. Verification remains on B4 below its B8 crossover. Near the 64-job cap, B8's largest median improvement over B4 is about 3%, inside our noise threshold, so B8 has no stable total-cost advantage under the bounded policy and the deployed signing selector never chooses it. Deep-queue throughput therefore does not predict finite-cohort cost. These thresholds characterize this binary and processor, not the primitive in general.

**Refill work model.** Measured in slot work instead of cycles over 12,288 signatures per parameter set, and including its finite-queue drain term, the model of Section 3.4 predicts measurement within 2.1% everywhere (Table 7), so cycles and slot work describe complementary layers whose ratios are not interchangeable.

## 6.4 End-to-end throughput and latency

Table 3 compares the three configurations across all parameter sets. The stackless path is 2.81–4.31× faster than OpenSSL default and 2.19–3.29× faster than the same-provider reference-C control, with 99th-percentile latency (p99) of 20.35–24.34 ms against 59.47–106.85 ms for OpenSSL and 45.70–79.55 ms for Provider C. Rates are stable across cyclic position, with the largest within-configuration max/min spread below 1%, and all 186,270 measured handshakes succeed.

A concurrency sweep confirms that $c = 64$ is a saturation measurement rather than a concurrency-specific optimum. With one sequential client no cross-request cohort forms, so the final path uses B1 and reaches 644.5 ML-DSA-65 handshakes/s against 512.7 for Provider C and 454.1 for OpenSSL;

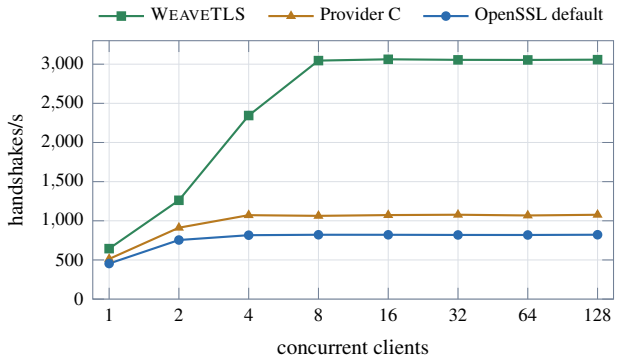



Figure 3: ML-DSA-65 handshake throughput versus concurrent clients on one server core (medians of three position-balanced process starts; logarithmic $x$ axis). All variants share certificates, nginx, and CPU placement; Provider C and WEAVETLS also share provider integration.

by eight clients it reaches 3,045.8, within 0.3% of the $c = 64$ point, while Provider C and OpenSSL saturate by four. Figure 3 plots the full curve.

Width telemetry explains why the primitive gains yield end-to-end improvements: they come from workload-aware composition, not from batching every authentication operation. At low concurrency the selector uses optimized B1; once the event loop exposes overlapping verification work, B4 and B8 absorb that cross-connection parallelism, and at $c = 64$ verification is predominantly B4. Event-loop phasing meanwhile forms mostly two-request signing cohorts, which the selector sends through Xu-derived B1 instead of paying a rejection-divergent drain, so batching every operation would be slower here.

Latency falls for the same structural reason that bounds the deferral: since collection lasts only until the next event-loop turn, a suspended request waits only for the finite flush that publishes it. Under saturation the reduction in authentication service time dominates that bounded deferral, so p99 latency falls rather than rises.

## 6.5 Client-visible rejection timing

We compare eager and cohort publication (Section 5) at concurrency 1, 16, and 64, joining each client's complete

Table 3: One-core TLS comparison at $c = 64$ (three-process medians). "Provider C" is synchronous reference C in the same provider, distinct from the optimized B1 route; "Final" denotes the WEAVETLS stackless configuration.

| | Throughput (handshakes/s) | | | Speedup | | p99 latency (ms) | | |
|---|---|---|---|---|---|---|---|---|
| Set | OpenSSL | Provider C | Final | vs OpenSSL | vs Provider C | OpenSSL | Provider C | Final |
| ML-DSA-44 | 1,186 | 1,520 | **3,336** | 2.81× | 2.19× | 59.47 | 45.70 | **20.35** |
| ML-DSA-65 | 826 | 1,078 | **3,051** | 3.70× | 2.83× | 87.05 | 65.88 | **22.85** |
| ML-DSA-87 | 663 | 870 | **2,858** | 4.31× | 3.29× | 106.85 | 79.55 | **24.34** |

Table 4: Client-visible ML-DSA-65 rejection timing, pooled over three runs. $c$ is client concurrency; $\bar{n}$ is mean signing-cohort size; subscripts E/C denote eager/cohort publication; $r$ is attempt/latency Pearson correlation and BA denotes balanced accuracy (50% is chance).

| $c$ | $\bar{n}$ | $r_E$ | $r_C$ | $\mathrm{BA}_E$ | $\mathrm{BA}_C$ |
|---|---|---|---|---|---|
| 1 | 1.0 | 0.565 | 0.572 | 69.1% | 67.0% |
| 16 | 15.3 | 0.312 | 0.063 | 61.5% | 50.4% |
| 64 | 37.5 | 0.133 | 0.008 | 54.6% | 49.1% |

TCP+TLS handshake latency to the server's signing-attempt count through a wire-visible signature tag; all 140,322 client records join. Besides Pearson correlation we train a one-threshold latency classifier on a tag-disjoint split, which tests whether latency alone supports a simple one-versus-multiple-attempt distinguisher; it does not test resistance to arbitrary inference (Appendix A).

At concurrency 16 and 64, cohort publication lowers correlation to 0.063 and 0.008 and balanced accuracy to 50.4% and 49.1%; at concurrency one the signing cohort contains one job and the two policies expose comparable signals. The benefit therefore depends on occupancy: cohort publication attenuates the client-visible signal for a loaded server, while signing remains variable-time and singleton rejection timing still requires a constant-work or masked signer. Verification raises no analogous secret-dependent timing question because its inputs are public. We do not characterize occupancies between the measured $c = 1$ and $c = 16$ points.

## 6.6 Portability

Repeating the ML-DSA-65 primitive experiment on an Intel Core i3-1005G1 (Ice Lake) preserves the cross-request benefit: after the same differential tests, folded B8 is 1.98× faster than the optimized Xu-derived B1 route for cold verification and B4 refill signs 1.70× faster than the otherwise identical lockstep scheduler. Since wide-vector execution can select a lower sustained Turbo operating point, we measure effective frequency separately: B8 settles 5.9% below AVX2 with Turbo enabled yet still sustains 1.97× its verification throughput (2.08× with Turbo disabled, both near 1.19 GHz). The mechanism transfers while the calibrated crossovers do not, so a new deployment recalibrates (Appendix D).

# 7 Related Work

**Post-quantum authentication in TLS.** Benchmarks of post-quantum TLS place most of the added handshake cost in authentication; the cost grows with chain depth and network loss [22, 36, 44]. Measurements of HTTPS and service-mesh sidecars show how this per-connection CPU cost scales [33, 53]. Several lines of work change the protocol to avoid the cost. KEMTLS replaces the handshake signature with a key-encapsulation mechanism (KEM) [10, 39, 40], and TLS batch-signing amortizes signatures over a Merkle tree [4]. Both approaches alter the protocol or require peer support, whereas WEAVETLS retains the standardized ML-DSA authentication messages. Thread-pool PQ-TLS designs also stay wire-transparent and parallelize independent operations across cores [14], a different and non-exclusive execution domain from the cross-request SIMD parallelism WEAVETLS exploits inside one worker.

**Vectorized ML-DSA and batching.** Zheng et al. and Xu et al. reduce shuffle, load, sampling, and packing overhead for Dilithium under AVX2 and AVX-512 [46, 51], building on an established transform lineage [1, 41]. Xu et al. provide our strongest primitive-level verification comparator. Huang et al. use 16-bit NTTs for small products on ARM [18]. These implementations supply the optimized primitive boundaries in our comparisons. Türe and Cenk batch Dilithium algebraically by turning matrix–vector into matrix–matrix products [45], assuming a shared key, whereas mTLS combines heterogeneous end-entity keys with selective issuer-key reuse. GPU implementations reach high aggregate throughput at large batch depth, with transfer cost and a separate scheduling domain [42, 43], and classical batch verification folds many equations into one randomized check [3, 6] but returns a single verdict where TLS needs one per connection. Among prior cross-instance SIMD implementations, AVX-512 batch EdDSA verification [47] and cross-instance AVX-512 SIKE [11] are closest in structure; both lack ML-DSA's iterative signing rejection loop, so their schedules need no per-request retirement and refill.

**Cryptography inside TLS stacks.** ENG25519 and

OpenSSLNTRU jointly design the primitive and its OpenSSL integration [5, 48], and Faster Post-Quantum TLS carries AVX-512 ML-KEM into a real handshake [50]; together they establish that primitive gains can survive the full protocol stack. ENG25519 is also the closest precedent for our question. It batches X25519 key generation eight ways, an operation needing nothing from the peer, but reports that derivation, signing, and verification all consume peer messages; collecting eight of them would mean refactoring the TLS application, against its goal of leaving application source untouched. It names batched TLS-layer integration as future work. WEAVETLS pays that cost for post-quantum authentication, where every handshake operation is peer-dependent, modifying libssl and nginx but nothing on the wire.

Asynchronous offload differs in where the work runs. SSLShader converts synchronous per-connection cryptography into batched asynchronous work behind an ordinary TLS server [20], but its primitives have no rejection loop and its batches cross into a device with its own scheduling domain and preferred batch size. Intel QuickAssist's `async_mode_nginx` drives nginx through OpenSSL's ASYNC pause/resume path [35] to a hardware coprocessor [19] at production rates, relying on dedicated silicon to amortize each fiber and descriptor transition. WEAVETLS instead keeps execution on the CPU worker that owns the event loop, organizing per-worker state as scalable network stacks do [21].

Prior work addresses fast ML-DSA kernels, same-key or large-batch execution, and asynchronous TLS offload separately. WEAVETLS combines heterogeneous per-request key state, rejection-divergent signing, and event-loop-resident finite cohorts. Prior work also documents wide-vector frequency effects and motivates vector-aware scheduling [15,16]. Section 6.6 measures these effects on Ice Lake.

## 8 Discussion

WEAVETLS targets concurrent full mTLS handshakes, where each connection performs one signature and two verifications. Dense bursts feed the cross-request kernels, whereas persistently low concurrency leaves the selector on optimized B1 execution and delivers only the singleton-path benefit. Session resumption bypasses most certificate authentication, so a deployment with a high resumption rate exposes proportionally less ML-DSA work to WEAVETLS; we characterize full handshakes rather than resumed sessions, and the realized saving scales with their share of a workload.

WEAVETLS does not require cross-worker batching because queues, expanded-key caches, and execution are worker-local. Each nginx worker forms and executes its own cohorts without cross-thread handoff, and we evaluate one worker to isolate per-core capacity; we do not claim a multicore scaling factor. Cached-key execution also depends on workload locality: issuer verification exposes repeated public keys, whereas heterogeneous client CertificateVerify keys may see lower hit rates under high key churn; a cold request stays correct and expands its matrix, so churn costs the cached-key advantage without affecting correctness. Wire transparency does not imply deployment transparency: peers need no modification, but deployment requires coordinated changes to libssl and nginx and a private provider multi-operation entry point.

The stackless continuation, finite-cohort cost routing, and the separation of queue depth from execution width are systems mechanisms not tied to one parameter set. Folded arithmetic kernels and the expanded-key representation are instead ML-DSA-specific, and rejection-aware refill applies to iterative operations whose jobs can retire independently, so WEAVETLS is a jointly designed cryptographic interface and TLS scheduler for ML-DSA, not a generic batching framework. Deep-queue ranking and deployed routing answer different questions. Our B8 refill result characterizes a 256-message saturated supply, whereas TLS exposes much shorter signing cohorts that the selector routes to B1 or B4 by finite total cost, so B8 being the fastest deep-queue rung remains consistent with the deployed selector never choosing it.

Cohort publication attenuates the measured remote rejection-timing signal under load, but it does not make ML-DSA signing constant-time. A deployment that requires rejection-independent timing still needs a constant-work or otherwise masked signer.

## 9 Conclusion

WEAVETLS turns independent ML-DSA authentication operations from concurrent TLS connections into finite CPU-resident execution cohorts without changing the TLS wire protocol. Three mechanisms make that possible. Request identity and expanded-key state remain attached to each request, independent of its SIMD position. Rejection-aware retirement and refill let divergent signing jobs leave and enter the working set independently. A stackless event-loop continuation routes each finite flush among B1, B4, and B8 by measured cost.

Across ML-DSA-44/65/87, B8 expanded-key verification is 1.65–2.23× faster than a matched cached-AVX2 verifier, and rejection-aware refill makes B8 ML-DSA-65 signing 1.90× faster than an otherwise identical lockstep scheduler. On one server core the nginx path is 2.81–4.31× faster than OpenSSL default and 2.19–3.29× faster than a synchronous reference-C control in the same provider. Cohort publication substantially attenuates the measured client-visible rejection signal under load, while singleton signing remains variable-time.

Cross-connection SIMD requires more than a wider primitive call. The TLS software boundary requires that request identity remain independent of SIMD placement, divergent jobs retire independently, and the event loop expose finite

work without waiting for a target batch width. Together, the cryptographic interface and TLS scheduler convert CPU-resident SIMD parallelism into end-to-end mTLS capacity while preserving the standardized wire protocol.

## References


[1] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *Applied Cryptography and Network Security (ACNS)*, pages 853–871. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-09234-3_42.

[2] M. Barbosa, G. Barthe, C. Doczkal, J. Don, S. Fehr, B. Grégoire, Y. Huang, A. Hülsing, Y. Lee, and X. Wu. Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. In *Advances in Cryptology (CRYPTO)*, pages 358–389. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-38554-4_12.

[3] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Advances in Cryptology (EUROCRYPT)*, pages 236–250. Springer-Verlag, 1998. https://doi.org/10.1007/BFb0054130.

[4] D. Benjamin. Batch signing for TLS. Expired Internet-Draft draft-ietf-tls-batch-signing-00, Internet Engineering Task Force, 2020. https://datatracker.ietf.org/doc/html/draft-ietf-tls-batch-signing-00.

[5] 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. USENIX Association, 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein.

[6] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. https://doi.org/10.1007/s13389-012-0027-1.

[7] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018. https://doi.org/10.1109/EuroSP.2018.00032.

[8] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and Reload: a cache attack on the BLISS lattice-based signature scheme. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 323–345. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-53140-2_16.

[9] L. G. Bruinderink and P. Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(3):21–43, 2018. https://doi.org/10.13154/tches.v2018.i3.21-43.

[10] S. Celi, A. Faz-Hernández, N. Sullivan, G. Tamvada, L. Valenta, T. Wiggers, B. Westerbaan, and C. A. Wood. Implementing and measuring KEMTLS. In *Progress in Cryptology (LATINCRYPT)*, pages 88–107. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-88238-9_5.

[11] 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.

[12] J. Devevey, P. Fallahpour, A. Passelègue, and D. Stehlé. A detailed analysis of Fiat-Shamir with aborts. In *Advances in Cryptology (CRYPTO)*, pages 327–357. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-38554-4_11.

[13] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Dilithium: a lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(1):238–268, 2018. https://doi.org/10.13154/tches.v2018.i1.238-268.

[14] S.-W. Eum, M.-H. Song, and H.-J. Seo. Performance analysis of a thread pool-based parallel execution model for hybrid post-quantum TLS 1.3 handshakes. Cryptology ePrint Archive, Paper 2026/440, 2026. https://eprint.iacr.org/2026/440.

[15] M. Gottschlag, P. Brantsch, and F. Bellosa. Automatic core specialization for AVX-512 applications. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 25–35. ACM, 2020. https://doi.org/10.1145/3383669.3398282.

[16] M. Gottschlag, P. Machauer, Y. Khalil, and F. Bellosa. Fair scheduling for AVX2 and AVX-512 workloads. In *USENIX Annual Technical Conference (ATC)*, pages 745–758. USENIX Association, 2021. https://www.usenix.org/conference/atc21/presentation/gottschlag.

[17] T. Hollebeek, S. Schmieg, and B. Westerbaan. Use of ML-DSA in TLS 1.3. Internet-Draft draft-ietf-tls-mldsa-03, Internet Engineering Task Force, May 2026. https://datatracker.ietf.org/doc/draft-ietf-tls-mldsa/03/.

[18] J. Huang, A. Adomnicăi, J. Zhang, W. Dai, Y. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen. Revisiting Keccak and Dilithium implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):1–24, 2024. https://doi.org/10.46586/tches.v2024.i2.1-24.

[19] Intel Corporation. Intel QuickAssist technology (QAT) async mode NGINX. Software distribution, 2023. https://github.com/intel/asynch_mode_nginx.

[20] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2011. https://www.usenix.org/conference/nsdi11/sslshader-cheap-ssl-acceleration-commodity-processors.

[21] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502. USENIX Association, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[22] P. Kampanakis, P. Panburana, E. Daw, and D. V. Geest. The viability of post-quantum X.509 certificates. Cryptology ePrint Archive, Paper 2018/063, 2018. https://eprint.iacr.org/2018/063.

[23] Keccak Team. eXtended Keccak code package (XKCP). Software distribution, 2024. https://github.com/XKCP/XKCP.

[24] R. Kinsella, C. MacNamara, and G. Tkachuk. Intel AVX-512: Packet processing with the Intel AVX-512 instruction set. Intel solution brief, revision 003, 2021.

[25] Z. Liu, A. Wang, C. Wei, Y. Ding, J. Zhang, A. Liu, and L. Zhu. Release the power of rejected signatures: an efficient side-channel attack on the ML-DSA cryptosystem. Cryptology ePrint Archive, Paper 2025/582, 2025. https://eprint.iacr.org/2025/582.

[26] V. Lyubashevsky. Fiat-Shamir with aborts: applications to lattice and factoring-based signatures. In *Advances in Cryptology (ASIACRYPT)*, pages 598–616. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-10366-7_35.

[27] V. Lyubashevsky. Lattice signatures without trapdoors. In *Advances in Cryptology (EUROCRYPT)*, pages 738–755. Springer-Verlag, 2012. https://doi.org/10.1007/978-3-642-29011-4_43.

[28] J. Massimo, P. Kampanakis, S. Turner, and B. Westerbaan. Internet X.509 public key infrastructure—algorithm identifiers for the module-lattice-based digital signature algorithm (ML-DSA). Technical Report RFC 9881, RFC Editor, Oct. 2025. https://www.rfc-editor.org/rfc/rfc9881.html.

[29] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical Report FIPS 202, U.S. Department of Commerce, 2015. https://doi.org/10.6028/NIST.FIPS.202.

[30] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Technical Report FIPS 204, U.S. Department of Commerce, Aug. 2024. https://doi.org/10.6028/NIST.FIPS.204.

[31] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report FIPS 203, U.S. Department of Commerce, Aug. 2024. https://doi.org/10.6028/NIST.FIPS.203.

[32] National Institute of Standards and Technology. Stateless hash-based digital signature standard. Technical Report FIPS 205, U.S. Department of Commerce, Aug. 2024. https://doi.org/10.6028/NIST.FIPS.205.

[33] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. M. Munafò, K. Papagiannaki, and P. Steenkiste. The cost of the "S" in HTTPS. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 133–140. ACM, 2014. https://doi.org/10.1145/2674005.2674991.

[34] OpenSSL Project. OpenSSL 3.5 final release. OpenSSL Library announcement, Apr. 2025. https://openssl-library.org/post/2025-04-08-openssl-35-final-release/.

[35] OpenSSL Project. *ASYNC_start_job: Asynchronous Job Management Functions*, 2026. https://docs.openssl.org/3.5/man3/ASYNC_start_job/.

[36] C. Paquin, D. Stebila, and G. Tamvada. Benchmarking post-quantum cryptography in TLS. In *Post-Quantum Cryptography (PQCrypto)*, pages 72–91. Springer-Verlag, 2020. https://doi.org/10.1007/978-3-030-44223-1_5.

[37] P. Ravi, M. P. Jhanwar, J. Howe, A. Chattopadhyay, and S. Bhasin. Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of NIST candidates. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 427–440. ACM, 2019. https://doi.org/10.1145/3321705.3329821.

[38] E. Rescorla. The transport layer security (TLS) protocol version 1.3. Technical Report RFC 8446, RFC Editor, Aug. 2018. https://www.rfc-editor.org/rfc/rfc8446.html.

[39] P. Schwabe, D. Stebila, and T. Wiggers. Post-quantum TLS without handshake signatures. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1461–1480. ACM, 2020. https://doi.org/10.1145/3372297.3423350.

[40] P. Schwabe, D. Stebila, and T. Wiggers. More efficient post-quantum KEMTLS with pre-distributed public keys. In *European Symposium on Research in Computer Security (ESORICS)*, pages 3–22. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-88418-5_1.

[41] G. Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.

[42] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao. High-throughput GPU implementation of Dilithium post-quantum digital signature. *IEEE Transactions on Parallel and Distributed Systems*, 35(11):1964–1976, 2024. https://doi.org/10.1109/TPDS.2024.3453289.

[43] S. Shen, H. Yang, W. Li, and Y. Zhao. cuML-DSA: optimized signing procedure and server-oriented GPU design for ML-DSA. *IEEE Transactions on Dependable and Secure Computing*, 22(3):2295–2307, 2025. https://doi.org/10.1109/TDSC.2024.3494835.

[44] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis. Post-quantum authentication in TLS 1.3: a performance study. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020. https://www.ndss-symposium.org/ndss-paper/post-quantum-authentication-in-tls-1-3-a-performance-study/.

[45] N. D. Türe and M. Cenk. Efficient batch algorithms for the post-quantum CRYSTALS Dilithium signature scheme and CRYSTALS Kyber encryption scheme. Cryptology ePrint Archive, Paper 2024/1397, 2024. https://eprint.iacr.org/2024/1397.

[46] R. Xu, D. He, M. Luo, C. Peng, and X. Zeng. Optimizing Dilithium implementation with AVX2/AVX-512. *ACM Transactions on Embedded Computing Systems*, 23(6):1–30, 2024. https://doi.org/10.1145/3687309.

[47] B. Zhang, H. Cheng, J. Großschädl, and P. Y. A. Ryan. High-throughput EdDSA verification on Intel processors with advanced vector extensions. In *Selected Areas in Cryptography (SAC)*. Springer-Verlag, 2025. Cryptology ePrint Archive, Paper 2025/1658. https://eprint.iacr.org/2025/1658.

[48] 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*, pages 6381–6398. USENIX Association, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-jipeng.

[49] Y. Zhao, W. Cheng, Z. Qiao, Y. Liu, and Y. Zhou. Rejection matters: efficient non-profiling side-channel attack on ML-DSA via exploiting public templates. Cryptology ePrint Archive, Paper 2026/056, 2026. https://eprint.iacr.org/2026/056.

[50] 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)*, pages 123–143. Springer-Verlag, 2024. https://doi.org/10.1007/978-3-031-70890-9_7.

[51] J. Zheng, H. Zhu, Z. Song, Z. Wang, and Y. Zhao. Optimized vectorization implementation of CRYSTALS-Dilithium. *IEEE Transactions on Dependable and Secure Computing*, 22(5):5730–5743, 2025. https://doi.org/10.1109/TDSC.2025.3575915.

[52] Y. Zhou, W. Wang, Y. Sun, and Y. Yu. Rejected signatures' challenges pose new challenges: key recovery of CRYSTALS-Dilithium via side-channel attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2025. Cryptology ePrint Archive, Paper 2025/214. https://eprint.iacr.org/2025/214.

[53] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan. Dissecting overheads of service mesh sidecars. In *ACM Symposium on Cloud Computing (SoCC)*, pages 142–157. ACM, 2023. https://doi.org/10.1145/3620678.3624652.

## A Detailed Measurement Methodology

**Primitive boundary.** For the B8 cold-verification comparison, one timed invocation receives eight distinct packed

Table 5: FIPS 204 parameter sets and encoded object sizes in bytes.

| Set | $k$ | $\ell$ | $\tau$ | $\lvert\mathsf{pk}\rvert$ | $\lvert\mathsf{sk}\rvert$ | $\lvert\sigma\rvert$ |
|---|---|---|---|---|---|---|
| ML-DSA-44 | 4 | 4 | 39 | 1,312 | 2,560 | 2,420 |
| ML-DSA-65 | 6 | 5 | 49 | 1,952 | 4,032 | 3,309 |
| ML-DSA-87 | 8 | 7 | 60 | 2,592 | 4,896 | 4,627 |

$(\mathsf{pk}, \sigma, M)$ tuples, reconstructs a fresh context, and returns an eight-bit verdict. Cost per verification is the elapsed batch cycles divided by eight. The single-instance baseline loops over the same eight-element pool inside one timed region and divides by eight. Signing starts from cached or expanded secret-key state and includes message hashing, sampling, rejection, packing, and request retirement. Key precomputation has its own reported timing outside this operation boundary.

The expanded-public-key experiment applies the same exclusion to both paths. Setup for both the pq-crystals AVX2 baseline and the B8 backend materializes $A$, $\mathrm{NTT}(t_1 2^d)$, and $H(\mathsf{pk})$ before the timed region. A single benchmark binary then times AVX2, eight distinct entry objects, eight copied objects that hold the same key, and eight aliases of one object over the same pool of distinct valid messages and signatures. We report three fresh processes; the copied-versus-aliased comparison isolates the broadcast dispatch from key or code-layout changes.

The refill work experiment uses 24 independent key/message trials of 512 signatures for each parameter set (12,288 signatures per row). It records total candidate attempts and occupied B8 iterations for the same inputs under refill and lockstep scheduling. The geometric model estimates $p$ from the aggregate attempts and includes the measured finite-queue drain. We compare its error with request-slot work and report the cycle ratio as a separate implementation measurement.

The peer-visible timing experiment instead measures complete TCP+TLS latency at the client. Server instrumentation records the true attempt count and an eight-byte tag taken from the CertificateVerify signature; the client's TLS message callback extracts the same public bytes for an offline join. The instrumentation adds no field to the wire. The client controls all connections at concurrency 1, 16, or 64 over loopback, avoiding wide-area network variation at this external boundary. Eager/cohort order alternates over three fresh four-second server processes. Pearson correlation uses the pooled joined records. The binary classifier labels one attempt versus multiple attempts, chooses a latency threshold on the 80% of tags whose integer value is nonzero modulo five, and reports balanced accuracy on the disjoint remainder.

Each sample contains at least 32 consecutive calls within one outer TSC interval. We compute each process median from at least 31 samples; at least three process starts check layout and address-space layout randomization (ASLR) sensitivity. We report medians and process spread. Verdicts and signature bytes feed a volatile sink, and request-slot assignment rotates between iterations. Instrumented stage profiles use separate builds whose stage sum must remain within 5% of their instrumented end-to-end cost. The signer profile covers 6,144 independent signatures per parameter set and identifies the remaining optimization targets and their upper bounds.

The implemented-width experiment rebuilds each ML-DSA parameter set at B4 and B8, runs the complete differential test after each of the six builds, and then records cold/hot verification and 256-message refill signing in three fresh processes. We measure B1 in isolated binaries. Each signing process checks 64 deterministic and hedged signatures byte for byte; each verification process accepts 16 valid inputs and rejects corruptions in the challenge, $z$, hint tail, and message before timing. The cold B1 measurement includes unpacking, natural-order four-way matrix expansion within one request, and one verification, whereas the hot measurement starts from the expanded public-key objects.

The Xu B1 signer uses that work's AVX-512 forward and inverse transforms and pointwise multiplication, our FIPS 204 transcript wrapper and cached key schedule, and our eight-way sampler, which is 23% faster than the four-way one it replaces. Its transforms consume and produce the reference coefficient order, so they need no repacking step. The upstream per-row hint routine fails our FIPS 204 byte-validation test because it reuses the final row's $ct_0$ product and hint buffer. Our B1 route retains Xu's arithmetic and uses the reference fused hint loop. Each process checks 64 signatures, half deterministic and half hedged, byte for byte against the reference.

We time both cached single-instance signers in binaries that do not link the cross-instance engine, keeping the single-request boundary independent of the production dispatch binary. The isolated measurements hold a 0.4–0.9% spread across parameter sets. The common verification harness supplies the expanded-key AVX2 measurement and gives both paths the same boundary.

A separate production-crossover benchmark links all six B4/B8 kernels at once, draws repeated finite cohorts of 1–64 jobs from the same 256-message pool, and times the B1/B4/B8 candidates over nine samples in each of six fresh processes. This boundary includes the deployed binary layout and the rejection drain paid by each finite cohort and supplies the dispatch thresholds. The deep-queue experiment separately ranks saturated primitive throughput.

**Environment controls.** The benchmark driver records CPU model and microcode, kernel, compiler and flags, source revisions, resolved `-march`, governor, boost state, core frequency, core/SMT placement, and relevant environment variables. The driver rejects runs with boost enabled or a non-performance governor. We calibrated TSC cycles against hardware core cycles on both CCDs; their difference was below 2% under

the measured configuration.

**System runs.** The server worker and client processes receive explicit disjoint CPU sets. The nginx configuration uses a reuseport backlog of 8,192 and error logging at `error`; informational close messages otherwise produce thousands of log writes per second. Every data point records duration, concurrency, successful and failed handshakes, p50/p90/p99/max latency, operation count, average admitted group size, hot/cold cache counts, and per-core CPU utilization. Here p50, p90, and p99 denote the 50th, 90th, and 99th latency percentiles. A data point is invalid if a client fails, the server log reports a verification-isolation failure, TCP SYN retransmission creates a network/client limit for a server-saturation result, or any client CPU overlaps the worker's CPU set.

The TLS comparison uses OpenSSL default and Provider C as controls. OpenSSL default represents the deployed library boundary. Provider C keeps WEAVETLS key management, algorithm registration, certificate chain, nginx path, client, and CPU placement, but executes the provider's reference-C sign and verify routines synchronously. The WEAVETLS configuration adds the SSL-owned continuation and direct B1/B4/B8 entry point. For each of ML-DSA-44/65/87, three fresh four-second processes use cyclic orders {default,C,final}, {C,final,default}, and {final,default,C} at $c = 64$, placing every configuration in each ordinal position once. We disable logging in timing runs; all 186,270 handshakes complete successfully. The runs use deterministic signing. A separate instrumented run records group size and chosen width; the correctness and primitive tests exercise hedged signing.

## B Additional Correctness Cases

The eight-way SHAKE implementation matches scalar FIPS 202 on short inputs, multi-block inputs, and six-block streaming output. For ML-DSA-44, ML-DSA-65, and ML-DSA-87, differential tests compare every SIMD kernel with pq-crystals reference intermediate values. Lockstep and refill signing produce byte-identical signatures to the deterministic reference for 32 and 64 messages, respectively; hedged signing does the same for 64 messages supplied with identical per-message randomness. In both modes, the retirement callback fires once per job after packing a complete signature, and its totals agree with the signer's statistics.

At the system level, the test suite forces every verification $n = 1, \ldots, 8$ boundary and each finite-queue signing crossover (10/11, 23/24, and 39/40 for ML-DSA-44/65/87). It executes deterministic and hedged signing and reference-verifies every output. OpenSSL's SSL API and TLS 1.3 message tests cover server-message reconstruction and the pending client-verdict barrier. Full TLS runs for all three parameter sets exercise signing and client CertificateVerify through the continuation while the certificate-chain signature follows the inline cached B1 path.

B8 decodes ExpandMask output directly from Keccak words into folded rows while B4 reaches the same rows through the generic byte path; every width retains that byte path as a differential control. The verification negative matrix includes bit flips in the packed signature, message, and public key; non-increasing hint indices; hint weight above $\omega$; nonzero unused hint bytes; a coefficient exactly at the rejected norm bound; and a changed challenge hash. We place each corruption in the first, last, and two middle request positions, and in a two-request case. The kernel, negative-case, and signing tests cover all six B4/B8 builds; operation-specific B1 tests cover all three parameter sets. Seed scans exercise low-acceptance parser paths in matrix and challenge expansion. A non-benchmark build runs sanitizers; benchmark binaries retain the production optimization flags and no instrumentation.

## C Primitive Design Alternatives

**Where the gains come from.** Starting from the first correct cross-instance implementation, vectorized packing and decoding cut the decode stage by 66%. Fusing NTT layers and matrix rows reduced arithmetic and memory traffic. The largest late improvement was the Keccak rewrite: 883 versus 1,132 cycles per eight-way permutation, reducing cold verification from about 47,300 to about 41,500 cycles and signing from 106,800 to about 100,700 cycles. The shared-matrix companion and inverse-NTT boundary pass then reduce the isolated row stage by another 4.3–5.7%, and the fixed-input ExpandMask path reduces the complete signer by 4.1–5.0% across parameter sets. Expanded-key caching produces a larger verification gain because it removes the stage whose internal 30-way XOF parallelism cross-instance SIMD cannot accelerate.

Two late signing experiments refine the implementation without changing the scheduler or provider boundary. Table 6 isolates all signing rows from pointwise MAC through decomposition and the complete ExpandMask stage. Every variant is bit-for-bit identical to its generic control.

Table 6: Late signing optimization reductions (medians).

| Set | MAC–decompose | ExpandMask stage | Complete signer |
|---|---|---|---|
| ML-DSA-44 | 4.3% | 21.2% | 4.13% |
| ML-DSA-65 | 4.7% | 21.0% | 4.13% |
| ML-DSA-87 | 5.7% | 21.1% | 5.02% |

**Folded transform schedules.** The eight NTT layers factor into two register-resident passes of four. Pass one loads the strided tile $\{j_0 + Br : 0 \le r < 16\}$ and executes coefficient distances 128, 64, 32, and 16; pass two loads the consecutive tile $\{16j_0 + r\}$ and executes 8, 4, 2, and 1, each tile holding its sixteen rows in registers throughout. At $F = 4$ the last two layers pair coefficients inside one four-lane fold, so running

**Algorithm 2** Rejection-aware refill of a $B$-request signing engine.

**Require:** compatible job queue $Q[0 \dots n-1]$; width $B$; *active* a $B$-bit slot-occupancy mask

$next \leftarrow 0$; $active \leftarrow 0$
**for** $\lambda \leftarrow 0$ **to** $B-1$ **do**
  **if** $next < n$ **then**
    $job_\lambda \leftarrow Q[next]$; LOAD($\lambda, job_\lambda$)
    $active_\lambda \leftarrow 1$; $next \leftarrow next + 1$
  **else**
    BINDDUMMY($\lambda$)
  **end if**
**end for**
**while** $active \neq 0$ **do**
  $(ok, \Sigma) \leftarrow$ ATTEMPT($active$)
  **for all** $\lambda$ such that $active_\lambda \wedge ok_\lambda$ **do**
    $\sigma_\lambda \leftarrow$ SIGENCODE($\Sigma_\lambda$)
    RETIRE($job_\lambda, \sigma_\lambda$)
    **if** $next < n$ **then**
      $job_\lambda \leftarrow Q[next]$; LOAD($\lambda, job_\lambda$)
      $next \leftarrow next + 1$
    **else**
      $active_\lambda \leftarrow 0$; BINDDUMMY($\lambda$)
    **end if**
  **end for**
**end while**

them in place would leave each multiply doing half a vector of useful work. B4 instead applies a $4 \times 4$ dword transpose inside each 128-bit quarter. The paired coefficients then occupy corresponding lanes of different vectors, and their twiddles are no longer uniform. The four coefficient positions of a fold need four values, packed into one 128-bit pattern and broadcast across the request quarters. Because the network is an involution, the same sequence restores the layout. B8's single fold-local layer needs no transpose. Two `vpshufd` patterns duplicate the even and odd coefficients of each pair across its qword. One Montgomery multiply applies the row's twiddle, which remains uniform across lanes, to the duplicated odd copy. A masked add/subtract writes the sum into even lanes and the difference into odd lanes. In the eight-way Keccak permutation, a two-round ping-pong representation holds the state in named vector variables, so $\pi$ becomes a renaming rather than data movement. At B8, ExpandMask keeps only the squeezed 64-bit words the parameter set's $\gamma_1$ width decodes: 72 at ML-DSA-44 and 80 at the other two sets.

**Cold matrix-expansion details.** Each $(i, j)$ group of eight SHAKE128 states absorbs $\rho \| j \| i$ under a fixed squeeze budget, and a continuing state supplies further blocks for any stream that falls short of 256 accepted coefficients. Because rejection sampling accepts only a sparse subset of candidates, the parser compacts the accepted ones with `vpcompressd`, preserving stream order.

**Shared-matrix/inverse-NTT boundary.** The precomputed Montgomery companion accounts for 2.0–4.3% and boundary fusion adds 1.5–2.4%. Two counterbalanced ML-DSA-65 whole-signer A/B blocks improve by 1.5–1.6%. The out-of-place forward entry removes one 8 KiB copy per $y$ polynomial, 40 KiB per ML-DSA-65 B8 iteration. The companion table costs 16/30/56 KiB for ML-DSA-44/65/87, but streaming the row product removes a materialized row vector whose size scales with the execution width, so the net B8 working set falls by 16/18/8 KiB. Carrying a register-sized MAC tile into the first inverse layer spills; a spill-free pair schedule adds at most 0.6% and slows the raw stage. The retained streaming schedule gives the best measured combination of register pressure, stage cost, and complete-signer cost.

**Fixed-input ExpandMask boundary.** Constructing the padded 66-byte input in the transposed Keccak state saves 6.8–7.6%; avoiding byte buffers and the coefficient transpose supplies the rest. Both 18- and 20-bit decoders are bit-for-bit identical to the control. Counterbalanced uninstrumented builds reduce the complete refill signer by 4.13%, 4.13%, and 5.02% for ML-DSA-44/65/87. In the ML-DSA-65 stage profile, scheduler work is identical across variants, and the instrumented end-to-end cost falls by 4.8%.

At full-stack scale, a same-period interleaving gives a median 2,848.2 handshakes/s for the direct structure-of-arrays (SoA) provider and 2,829.7 for the byte-path provider, a 0.65% difference within the 3% system noise threshold. The primitive improvement is measurable in isolation, but batching and TLS scheduling dominate the complete-stack rate.

For ML-DSA-65, the 50 eight-way permutations required by the algorithm account for 44,150 of the remaining 49,404 ExpandMask cycles. The non-permutation remainder is only 1.78% of total iteration cost. Thus, even an ideal implementation of the remaining work can improve the whole iteration by at most 1.78% and is unlikely to change the system result.

Table 7: Finite-queue refill model. Drain is refill-tail overhead; Model and Measured report lockstep/refill request-slot-work ratios. Cycle speedups appear separately in Table 1.

| Set | $\mathbb{E}[T]$ | Drain | Model | Measured |
|---|---|---|---|---|
| ML-DSA-44 | 4.42 | 2.1% | 2.46× | 2.44× |
| ML-DSA-65 | 4.83 | 2.2% | 2.47× | 2.53× |
| ML-DSA-87 | 3.83 | 1.9% | 2.43× | 2.38× |

**Implemented widths across parameter sets.** Table 8 gives the three-parameter-set, three-rung matrix. Moving the folded layout beyond ML-DSA-65 required two mode-specific boundaries: ML-DSA-44's $(q-1)/88$ decomposition alphabet packs four six-bit $w_1$ coefficients into three bytes, and its B8 ExpandMask decoder consumes 18-bit $z$ coefficients; the other parameter sets use 20-bit coefficients. Both paths decode directly into the folded coefficient-major layout. Every B4/B8 cell matches reference intermediate values, passes the request-slot isolation matrix, and produces byte-exact deterministic and hedged signing before timing; the B1 verifier passes the valid/corrupted-input matrix, and the isolated B1 signer passes the byte-level tests.

The cold-verification and signing columns are the width measurements cited in Section 6. The hot-verification column comes from the per-width matrix harness; Table 2 uses a

Table 8: All-parameter-set B1/B4/B8 matrix (cycles/completed operation). The B1 entries are the operation-specific Xu-derived FIPS 204 single-request paths used by the production dispatcher. The cold B1 comparator uses four-way XOF expansion within its one request; Xu's published round-3 verifier expands eight ways and is correspondingly faster (Table 1).

| Set | $B$ | Cold verify | Refill sign | Hot verify |
|---|---|---|---|---|
| ML-DSA-44 | 1 | 66,733 | 104,948 | 18,604 |
| | 4 | 27,403 | 71,162 | 10,378 |
| | 8 | 25,155 | **66,885** | **9,742** |
| ML-DSA-65 | 1 | 111,154 | 135,926 | 21,073 |
| | 4 | 48,158 | 107,598 | **13,408** |
| | 8 | 41,655 | **96,078** | 14,065 |
| ML-DSA-87 | 1 | 174,675 | 138,455 | 27,019 |
| | 4 | 79,108 | 118,477 | **18,918** |
| | 8 | 70,369 | **107,802** | 21,247 |

separate common binary and input pool for its matched cross-implementation ratios.

The B4/B8 sign columns use a 256-message queue. B8 lowers this deep-queue cost by 6.0–10.7% relative to B4. B1-to-B8 deep-supply ratios are 1.57×, 1.41×, and 1.28×. The margin narrows with the parameter set because larger sets expose more intra-request hashing and arithmetic parallelism to B1, reducing the fraction that only cross-request execution can accelerate. The hot-verification minimum can occur at B4 because folding overhead and working-set locality matter once caching removes matrix expansion.

A second benchmark measures the ordering on finite TLS-sized cohorts, timing B1, B4, and B8 over cohort depths between 1 and 64 in the binary that links all six production B4/B8 kernels. Section 6 reports the routing thresholds it yields; Table 8 instead characterizes deep-queue per-operation throughput.

# D Secondary Intel Results

We repeat the ML-DSA-65 primitive experiment on an Intel Core i3-1005G1 (Ice Lake, two cores and four threads). The host runs Ubuntu 22.04 with a 6.8 kernel, and GCC 11.4 compiles the backend with `-O3 -march=native`. We pin the benchmark to CPU 0, run no benchmark task on its SMT sibling, and use AC power. The B8 verification and B4 signing builds pass the FIPS 202, folded-layout, reference-kernel, positive/negative verification, and deterministic/hedged signing tests before timing.

## D.1 Primitive performance

Table 9 reports medians of five fresh process starts with Turbo disabled and the governor set to `performance`. We measure the B1 signer and verifier in their isolated binary, without resident batch state. Expanded-public-key setup and signing-key setup lie outside all cached boundaries. The AVX2 and Xu-derived B1 columns are therefore optimized singleton implementations at the same primitive interfaces as the B8/B4 entries.

Table 9: ML-DSA-65 on Ice Lake (cycles per completed operation). B1 columns show isolated single-instance paths; WeaveTLS uses the folded width shown in parentheses.

| Boundary | AVX2 B1 | Xu B1 | WeaveTLS | vs. Xu B1 |
|---|---|---|---|---|
| Cold verification (B8) | 144,711 | 136,651 | **68,906** | 1.98× |
| Expanded, heterogeneous (B8) | 33,581 | 30,383 | **28,933** | 1.05× |
| Expanded, copied entry (B8) | 33,581 | 30,383 | **28,995** | 1.05× |
| Expanded, shared entry (B8) | 33,581 | 30,383 | **22,342** | 1.36× |
| Refill signing, cached key (B4) | 268,279 | 205,990 | **187,888** | 1.10× |

The folded geometry preserves the cross-request benefit on Ice Lake. B8 nearly halves cold-verification cost relative to the optimized Xu-derived B1 route; at the tighter expanded heterogeneous boundary its advantage is 5%. The copied-entry control holds the key constant without pointer aliasing; aliasing one entry lowers the B8 cost from 28,995 to 22,342 cycles. On signing, rejection-aware B4 refill lowers the cost from 318,729 cycles under the otherwise identical lockstep scheduler to 187,888 cycles, a 1.70× scheduler gain.

## D.2 Turbo-frequency sensitivity

Wide-vector execution can select a lower sustained Intel Turbo operating point [16, 24]. A dedicated ten-second cold-verification loop runs one implementation at a time, and effective frequency comes from `perf stat` cycles divided by task-clock. Input generation and validation precede the interval. Table 10 gives medians of five fresh process starts, with path order alternated between starts.

Table 10: Sustained verification and effective frequency on Ice Lake.

| Turbo | Path | GHz | verify/s | Gain |
|---|---|---|---|---|
| Off | WeaveTLS B8 | 1.193 | 17,114 | 2.08× |
| | AVX2 | 1.194 | 8,244 | n/a |
| On | WeaveTLS B8 | 3.189 | 46,835 | 1.97× |
| | AVX2 | 3.389 | 23,726 | n/a |

With Turbo disabled, both paths run at approximately 1.19 GHz and B8 completes 2.08× as many verifications. With Turbo enabled, B8 settles at 3.189 GHz versus 3.389 GHz for AVX2, a 5.9% lower effective frequency; its

sustained throughput remains 1.97× higher. These whole-verifier operating points include each implementation's complete instruction mix.

The result also quantifies a placement effect for the deployed same-worker path: wide-vector frequency affects the worker's service rate. Operators should calibrate width thresholds on each target processor instead of copying those from the AMD host.