When Removing Reductions Goes Wrong: Auditing Reduction Placement in Production ML-DSA Implementations

Sunwoo Lee, Hyuk Lim, and Seunghyun Yoon

Korea Institute of Energy Technology (KENTECH)
{sunwoolee,hlim,syoon}@kentech.ac.kr

Abstract. Implementing post-quantum signatures correctly in production cryptographic libraries remains challenging even after standardization. ML-DSA implementations rely on NTT-based polynomial arithmetic with lazy Montgomery reductions, and omitting a reduction may be either a valid optimization or a latent arithmetic defect. In practice, reduction calls are often removed for performance, memory, or embedded-deployment reasons, but the required correctness condition is inter-procedural: a site that appears redundant locally may be load-bearing for a later InvNTT stage. In this work, we present a certificate-backed audit methodology for reduction placement in production ML-DSA implementations. Starting from the conservative pq-crystals topology, our analysis propagates coefficient bounds across the full signing path and classifies reduction sites as redundant or necessary. The key technical ingredient is an exact-integer recovery result for sparse-challenge products, which tightens post-InvNTT bounds from $(-Q, Q)$ to $[-\tau\eta, \tau\eta]$ and separates safe omissions on sparse-product paths from load-bearing dense-product sites. Applying the methodology to eight ML-DSA libraries, we uncover a previously unreported defect in wolfSSL's memory-optimized `WOLFSSL_DILITHIUM_SMALL` path, where omitted post-matrix-multiplication reductions cause overflow, non-conformant arithmetic, and signing failure while surviving the implementation's existing KAT tests. The site classifications are backed by replayable SMT-LIB2 certificates, with the core integer-bound lemmas cross-checked in an axiom-free Coq development.

Keywords: Post-Quantum Cryptography · ML-DSA · Implementation Correctness · Modular Reduction · SMT-Based Verification

1 Introduction

The finalization of FIPS 204 [28] marks ML-DSA's transition from standardization to deployment. ML-DSA—formerly Dilithium [12]—is now implemented in production cryptographic libraries, including wolfSSL [34], AWS-LC [3], BoringSSL [16], Go stdlib [33], liboqs [30], PQClean [31], and SymCrypt [23]. These

implementations rely on NTT-based polynomial arithmetic with lazy Montgomery reduction [19, 22, 32]: intermediate values are not reduced after every operation but are normalized only at selected program points.

This distinction matters in production code. Implementations remove reductions for performance, memory, or embedded-deployment reasons, but at the source level a safe optimization and a defect can look identical: a reduction call has disappeared. The practical audit question is therefore simple: in a production ML-DSA implementation, which missing reductions are safe, and which are load-bearing?

Scope and Baseline. This paper studies implementation correctness, not cryptanalytic security: reduction placement is a deployment-critical correctness condition, and removing the wrong reductions can produce non-conformant signatures, signing failures, and undefined signed-overflow behavior in production code. We take the conservative reduction schedule of the pq-crystals reference implementation [12] as our baseline. The reference inserts `reduce32` at sites where coefficient growth is not easily discharged by a local bound; this acts as a position-by-position safety net rather than a formal minimality argument, and every such site is therefore a candidate for our analysis.

Production Libraries Deviate from the Baseline. Production implementations often depart from this conservative placement. In our audit, AWS-LC [3] omits reductions that our classification later confirms redundant, while the memory-optimized `WOLFSSL_DILITHIUM_SMALL` mode in wolfSSL [34] omits additional post-matrix-multiplication reductions that our analysis finds load-bearing. Each omission therefore makes an implicit correctness claim about the removed reduction. What is missing is a lightweight, implementation-independent criterion for checking such claims: KAT testing only exercises a small input distribution, per-function verifiers do not propagate bounds across function boundaries, and full-implementation proofs require per-library proof engineering that is difficult to repeat across many independently maintained codebases. Source-level deletion alone does not distinguish safe sparse-product omissions from unsafe dense-product omissions; we provide a criterion that does.

Consequence in wolfSSL SMALL. wolfSSL's SMALL path omits several sparse-challenge-product reductions that our analysis confirms to be redundant: the values reaching these sites are already well within the range where `reduce32` acts as the identity. However, the same path also omits the post-matrix-multiplication `reduce32` at three program points—key generation, signing, and verification—which are load-bearing. This single defect manifests as four correctness failures: signed 32-bit integer overflow in the inverse NTT, $\mathbb{Z}_q$-incorrect outputs, deterministic signing failure, and FIPS 204 non-compliance. It survives the implementation's existing KAT tests because it is inter-procedural: each function is correct in isolation, but the pipeline overflows when an unreduced dense matrix-vector accumulation flows into InvNTT.

Our Approach. Our analysis combines a scheme-level bound argument with replayable SMT certification. The key observation is that the NTT multiplication pipeline returns exact integers—not merely residues modulo Q—for sparse-challenge products. This tightens post-InvNTT bounds from $(-Q, Q)$ to $[-\tau\eta, \tau\eta]$, an improvement of over sixty-fold (Theorem 1, Section 3.1), and explains why several sparse-challenge-product reductions are redundant. We then propagate arithmetic bounds across the full ML-DSA signing path and ask, for each reduction in the pq-crystals baseline, whether removing it can drive any reachable coefficient outside the reduction's fixed-point region. Each site classification is backed by an SMT-LIB2 certificate: UNSAT certifies redundancy, while SAT provides a concrete witness for necessity. Unlike per-function verifiers such as CryptoLine, which verify that individual routines compute the correct output, our compositional queries track stage-to-stage preconditions across function boundaries—the inter-procedural reasoning needed to expose wolfSSL's overflow. Together with the centered-input semantics of `BitPack`, this classifies a majority of the baseline reductions as safe to remove and identifies the precise placements where removal is unsafe.

The wolfSSL failure is not exposed as a wrong answer on ordinary test vectors; it is a precondition failure that appears only when a valid upstream range is allowed to flow into a later lazy-reduction stage—the structural reason it survives existing KAT-based testing.

Contributions.

1. **Production reduction-placement defect.** Auditing eight production ML-DSA libraries, we uncover a previously unreported reduction-placement defect in wolfSSL's `WOLFSSL_DILITHIUM_SMALL` mode. The defect is one missing-reduction pattern that appears at three program points—key generation, signing, and verification—and manifests as signed 32-bit integer overflow, $\mathbb{Z}_q$-incorrect output, signing failure, and FIPS 204 non-compliance despite passing the implementation's existing KAT suite.
2. **Reduction-placement audit.** We give a certificate-backed criterion for separating safe reduction omissions from load-bearing ones. Applied to the pq-crystals reference topology, the criterion classifies four sites as necessary and six as redundant. Each classification is backed by a replayable SMT-LIB2 certificate: an UNSAT result for redundancy or a concrete SAT witness for necessity.
3. **Compact formal core.** The audit is enabled by Theorem 1: for sparse-challenge products, the Montgomery-form NTT pipeline returns exact integers rather than only residues modulo Q. This tightens post-InvNTT bounds from $(-Q, Q)$ to $[-\tau\eta, \tau\eta]$. The remaining compositional InvNTT bounds are checked by a reproducible Z3-based bound-propagation procedure provided in the supplementary material.

Positioning. We position this work as a practical, certificate-backed audit of reduction placement in production ML-DSA implementations, supported by a

compact scheme-specific formal core. The formal core identifies which reference-topology reductions are load-bearing under the integer bounds required by the signing pipeline, and the cross-library audit tests whether production implementations preserve this placement contract. The wolfSSL defect is therefore not presented as a standalone bug report, but as the production consequence of an omitted load-bearing reduction.

Comparison with Prior Work. Existing formal tools excel at verifying that a given implementation correctly computes the intended function. CryptoLine [15, 20, 8] can confirm that a given `reduce32` call computes the correct residue, but it is not designed to decide whether that call is semantically necessary across the full signing path; that requires tracking the sparse-challenge structure $\|c\|_1 = \tau$ and the resulting bounds through subsequent stages. Jasmin/EasyCrypt [2, 1] can prove the pq-crystals reference correct with all 10 reductions present, but such a proof does not by itself identify which reductions are redundant or audit omissions in a distinct production codebase such as wolfSSL. Our analysis is complementary: we classify reduction *sites*, not implementations. Section 8 provides a detailed comparison.

2 Background

2.1 NTT Arithmetic and Lazy Montgomery Reduction

Lattice-based cryptographic schemes operate over $R_Q = \mathbb{Z}_Q[X]/(X^n + 1)$. Polynomial multiplication is computed via the NTT: $a \cdot b = \mathrm{InvNTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$. The NTT uses butterfly operations in $d = \log_2 n$ layers.

Montgomery Reduction. ML-DSA uses Montgomery multiplication [25] for modular arithmetic. Given a $2W$-bit product a, Montgomery reduction computes a W-bit representative $t \equiv a \cdot R^{-1} \pmod{Q}$ without trial division:

$$m \leftarrow (a \bmod R) \cdot (-Q^{-1} \bmod R) \bmod R, \quad t \leftarrow \frac{a + m \cdot Q}{R}, \tag{1}$$

where $R = 2^W$ is the Montgomery radix. The division by R is exact: a right shift. Seiler [32] proves the strict inequality $|t| < Q$ for a single reduction; we extend this compositionally to the full InvNTT pipeline.

Cooley–Tukey (CT) Butterfly (Forward NTT).

$$t \leftarrow \texttt{montgomery_reduce}(\zeta \cdot a[j+\ell]), \tag{2}$$

$$a[j+\ell] \leftarrow a[j] - t, \quad a[j] \leftarrow a[j] + t. \tag{3}$$

Gentleman–Sande (GS) Butterfly (Inverse NTT).

$$t \leftarrow a[j], \tag{4}$$

$$a[j] \leftarrow t + a[j+\ell], \tag{5}$$

$$a[j+\ell] \leftarrow \texttt{montgomery_reduce}(\zeta \cdot (t - a[j+\ell])). \tag{6}$$

The key structural property: Montgomery reduction in (2) and (6) bounds its output to $(-Q, Q)$, but the additions and subtractions in (3) and (5) receive *no* intermediate reduction—the "lazy reduction" strategy employed by all reference implementations [12, 6].

GS Addition Branch Accumulation. In the GS butterfly (5), the addition branch $a[j] \leftarrow t + a[j+\ell]$ is not Montgomery-reduced, so its magnitude can grow across layers; the subtraction branch (6) is reset to $(-Q, Q)$ by Montgomery reduction. In the worst case, a coefficient taking the addition branch in all d layers grows by a factor 2^d. Thus, for ML-DSA with $d = 8$ and input bound $A \leq Q$, the coarse lazy-accumulation bound is $2^8 Q = 2{,}145{,}386{,}752$, below $2^{31} = 2{,}147{,}483{,}648$ by only 0.1%. This calculation is not the certified centered output bound used in Theorem 1; rather, it illustrates why missing pre-InvNTT reductions are dangerous. The conforming pipeline is much tighter: our compositional bound-propagation procedure certifies $B_{\max} \leq 4{,}201{,}525 < Q$ after accounting for the exact butterfly topology and twiddle factors.

2.2 Reduction Sites and ML-DSA Constraints

ML-DSA implementations use several reduction operators with different semantic roles. Some reductions normalize arithmetic values into a smaller centered range, while others establish representation preconditions for downstream routines such as decomposition or encoding. For our audit, the relevant question is whether a reduction changes any value that can actually reach its program point.

We use the fixed-point regions in Table 1 as the local redundancy criterion: if all values reaching a program point lie in the corresponding region, the reduction is a no-op there; if any reachable value can leave the region, the reduction is load-bearing.

Table 1: Reduction operators in ML-DSA and their fixed-point regions.

Operator	Fix(r)	Width
`reduce32`	$[-2^{22}, 2^{22})$	$2^{23} \approx 8.4$M
`caddq`	$\{a : a \geq 0\}$	(asymmetric)
`freeze`	$\{a : 0 \leq a < Q\}$	$Q = 8{,}380{,}417$

A reduction at program point p is *fixed-point redundant* if every reachable value already lies within the corresponding fixed-point region; in that case, the

operator is a no-op at p. If reachable values can leave this region, the reduction is not locally redundant, and we determine necessity by checking the downstream arithmetic or representation precondition it is meant to establish. Thus, for representation-changing operators such as `caddq` and `freeze`, necessity depends not only on arithmetic magnitude but also on how later routines interpret the representation. Under the reference arithmetic-shift model, the fixed-point region of `reduce32` is the half-open interval $[-2^{22}, 2^{22})$; we use 2^{22} as the displayed threshold for headroom ratios, and all reported redundant bounds lie far from either endpoint.

ML-DSA Constraints. ML-DSA (FIPS 204 [28]) samples secret vectors s_1, s_2 with $\|s_i\|_\infty \leq \eta$ ($\eta \in \{2, 4\}$), a mask vector y whose coefficients lie in $[-\gamma_1+1, \gamma_1]$ ($\gamma_1 \in \{2^{17}, 2^{19}\}$) as produced by `ExpandMask` (FIPS 204, Algorithm 34), and a challenge c via `SampleInBall` with exactly τ non-zero entries, each in $\{-1, +1\}$ ($\tau \in \{39, 49, 60\}$).

The critical domain-specific property is the sparse challenge's ℓ_1-norm: $\|c\|_1 = \tau$, so any coefficient of the negacyclic convolution $c \cdot s_1$ satisfies $|(c \cdot s_1)_k| \leq \tau \cdot \eta$. A generic interval analysis treating c as dense with $\|c\|_\infty \leq 1$ yields the looser bound $n \cdot \eta$—up to 6.6× worse.

2.3 Verification Scope and SMT Certificates

Scope of Existing Tools. Existing PQC verification tools primarily verify that a fixed implementation computes a fixed specification. CryptoLine [15, 20, 8] requires per-routine annotations, while Jasmin/EasyCrypt [2, 1] requires reimplementation in a verification-oriented DSL. These approaches are complementary to our goal: we classify reduction *sites* at the scheme level and then audit production libraries against that classification.

The wolfSSL defect is inter-procedural. Each Montgomery reduction and butterfly is locally correct, but omitting the reduction after matrix multiplication allows the accumulated bound to flow into InvNTT, where GS addition branches amplify it past `INT32_MAX`. Detecting this requires propagating bounds across stages, not only checking each routine in isolation.

SMT Certificates. Satisfiability Modulo Theories (SMT) solvers [26] extend SAT solving with background theories such as bit-vectors and integer arithmetic. An SMT query asks whether there exists an assignment satisfying a set of constraints. The solver returns SAT, together with a concrete witness, or UNSAT, certifying that no such assignment exists.

We use SMT in two ways. **QF_BV** models the fixed-width arithmetic under the implementation contract used by our replay model and tested targets: low-word extraction for the `int32_t` conversion, two's-complement signed interpretation, and arithmetic right shift for signed shifts. **QF_LIA** models scheme-level integer constraints such as sampling ranges and sparse-challenge bounds. In our setting, an UNSAT query certifies that a reduction is redundant for all valid ML-DSA inputs, while a SAT query provides a concrete witness showing

that the reduction is necessary. This distinction is important for our audit: KAT testing checks finitely many vectors, whereas an SMT certificate quantifies over all inputs satisfying the ML-DSA constraints.

Why SMT? SMT gives us more than a finite test-vector check: our queries encode *scheme constraints*—sampling bounds, sparsity, and parameter ranges—directly as assertions. The resulting classification is implementation-independent and can be used as a common audit baseline across conforming libraries. This differs from implementation verifiers such as CryptoLine [15], which operate on specific routines and codebases rather than on reduction sites as reusable audit objects. Section 5.2 demonstrates the KAT-vs-SMT contrast on the wolfSSL defect.

3 Analysis Methodology

Our methodology combines three ingredients: a tight bound on InvNTT output for sparse-challenge products, a simple criterion for classifying each reduction site, and SMT-backed certification of every claim.

3.1 Tight Bounds via Exact Integer Recovery

The standard analysis of the InvNTT gives an output bound of $(-Q, Q)$—correct but opaque, as it makes every post-InvNTT reduction appear necessary. We show that for sparse-challenge products, the actual output equals the mathematical integer product, enabling far tighter bounds. The argument proceeds through three lemmas, a proposition, and a theorem.

Lemma 1 (Modular correctness of the Montgomery NTT pipeline). *For all input polynomials $a, b \in R_Q$, the Montgomery-form NTT multiplication pipeline Φ satisfies*

$$\Phi(\hat{a}, \hat{b})_k \equiv (a \cdot b)_k \pmod{Q} \qquad \text{for every coefficient } k,$$

where $\hat{a} = \mathrm{NTT}(a)$, $\hat{b} = \mathrm{NTT}(b)$, and $\cdot$ denotes negacyclic convolution in $R_Q = \mathbb{Z}_Q[X]/(X^n + 1)$.

Proof. The forward NTT maps negacyclic convolution in R_Q to componentwise multiplication, and the inverse transform applies the corresponding inverse twiddle schedule and final n^{-1} scaling. Each Montgomery reduction preserves the intended residue while changing only the representative; the per-multiplication factor R^{-1} is exactly cancelled by the final $n^{-1}R^2$ scaling, which converts the Montgomery representative back to the coefficient domain. Hence $\Phi(\hat{a}, \hat{b})_k \equiv (a \cdot b)_k \pmod{Q}$ for every k. □

Lemma 2 (Compositional InvNTT output bound). *Consider the reference-style ML-DSA inverse NTT routine ($Q = 8{,}380{,}417$, $R = 2^{32}$, $d = 8$*

GS butterfly layers, $n = 256$), including its final Montgomery scaling. Suppose its NTT-domain input is a vector $\hat{x}$ of signed centered integer representatives satisfying the input contract

$$|\hat{x}_i| < Q \qquad \text{for every } i \in \{0, \ldots, n-1\}.$$

These are the signed integer representatives of the NTT/Montgomery-domain values held immediately before `invntt_tomont`*; the bound is on these representatives, not on coefficient-domain values produced after the final scaling. Then:*
(i) every Montgomery reduction performed by the routine satisfies $|t| < Q$;
(ii) every final output coefficient x_i satisfies $|x_i| \le B_{\max}$, with $B_{\max} \le 4{,}201{,}525 < Q$.

Proof. (i) For a single Montgomery reduction, $|t| < Q$ is known [32]. We model each reduction with exact 64-bit $\times$ 32-bit semantics via QF_BV and, for each of the 8 GS layers and the final scaling, assert $|t| \ge Q$. Z3 returns UNSAT for all 9 queries (Table 2). These 9 queries certify the coarse invariant that every Montgomery call returns a value in $(-Q, Q)$ at each layer and at the final scaling step. (ii) The finer per-butterfly bounds used to compute $B_{\max}$ are certified compositionally by the Z3-based bound-propagation procedure summarized below and detailed in the supplementary material (Table 3). The procedure tracks one per-coefficient integer bound B_i through each of the 8 GS butterfly layers and the final Montgomery scaling, for each power-of-two dimension $n \in \{2, 4, \ldots, 256\}$; at $n = 256$ this yields 3,584 per-butterfly QF_BV queries and gives $\max_i B_i \le 4{,}201{,}525$. □

Bound propagation summary. We make the procedure underlying Lemma 2(ii) explicit so that the result is self-contained. The propagation tracks, for each coefficient position i, a single integer bound $|\hat{x}_i| \le B_i$. Initially $B_i \leftarrow Q - 1$, matching the input contract. For each Gentleman–Sande butterfly with inputs u, v and twiddle factor ζ, the reference inverse NTT computes

$$u' \leftarrow u + v, \qquad v' \leftarrow \mathrm{Mont}\big(\zeta \cdot (u - v)\big).$$

The unreduced addition branch is propagated by integer interval arithmetic; the Montgomery branch is certified by a per-call bit-vector query. For input-difference bound $D = B_u + B_v$ and twiddle ζ, let $M_\zeta(D)$ denote the certified output bound produced by the replayable QF_BV query:

$$\forall d,\ |d| \le D \ \Rightarrow\ |\mathrm{Mont}(\zeta\, d)| \le M_\zeta(D).$$

Each query also certifies $M_\zeta(D) < Q$. The propagated updates are

$$B_{u'} \leftarrow B_u + B_v, \qquad B_{v'} \leftarrow M_\zeta(B_u + B_v).$$

After the 8 GS layers, the final Montgomery scaling by the constant factor F used by ML-DSA's InvNTT is applied coefficient-wise, giving $B_i^{\mathrm{out}} \leftarrow M_F(B_i)$ and $B_{\max} = \max_i B_i^{\mathrm{out}}$. For $n = 256$ the replayed certificates give $B_{\max} = 4{,}201{,}525 < Q$.

Stage	Operation	Bound update
Input	reduced NTT-domain coefficient	$B_i \leftarrow Q-1$
GS addition	$u' = u + v$	$B_{u'} \leftarrow B_u + B_v$
GS Montgomery	$v' = \mathrm{Mont}(\zeta(u-v))$	$B_{v'} \leftarrow M_\zeta(B_u + B_v)$
Final scaling	$x_i = \mathrm{Mont}(F \cdot x_i)$	$B_i^{\mathrm{out}} \leftarrow M_F(B_i)$
Output	all coefficients	$B_{\max} = \max_i B_i^{\mathrm{out}}$

The proof uses two levels of reasoning. The control-flow topology and the addition-branch recurrence are handled as integer interval propagation; each Montgomery output bound $M_\zeta(D)$ is discharged in QF_BV under the same two's-complement/arithmetic-shift implementation contract used by our replay model. The integer abstraction is used only to compose certified per-butterfly bounds across the GS topology; the bit-level Montgomery step itself is not approximated.

Two consequences follow. First, $B_{\max}$ applies to any reference-style InvNTT call whose input satisfies the contract $|\hat{x}_i| < Q$, regardless of how $\hat{x}$ was produced. In particular, when $\hat{x}$ is produced by coefficientwise Montgomery multiplication—i.e., each entry is the Montgomery-reduced product of two NTT-domain coefficients—every input coefficient to the InvNTT lies in $(-Q, Q)$ and the contract holds; this is the case used by Theorem 1. Second, $B_{\max}$ is *not* a bound on unreduced dense matrix-vector accumulations such as the $\ell(Q-1)$-magnitude inputs that arise in Section 5.2 when the post-`matrix_mul reduce32` is omitted; the hypothesis of Lemma 2 no longer holds there, which is exactly why that omission is load-bearing.

Table 2: Strict Montgomery bound certificates for ML-DSA InvNTT. All 9 queries return UNSAT, certifying $|t| < Q$ (part (i) of Lemma 2).

Layer	B_{in}	Result
GS layers 0–7	1.40×10^{14}–1.80×10^{16}	all UNSAT
Final scaling	9.01×10^{13}	UNSAT

Proposition 1 (Residue-to-integer recovery threshold). *Let $v_k \in \mathbb{Z}$ and $c_k \in \mathbb{Z}$ satisfy $v_k \equiv c_k \pmod{Q}$, $|v_k| \le B_{\max} < Q$, and $|c_k| \le T_Q$ where $T_Q = Q - 1 - B_{\max}$. Then $v_k = c_k$ as integers.*

Proof. Write $v_k = c_k + mQ$ for some integer m. If $m \neq 0$, then $|m| \ge 1$ and $|c_k| = |v_k - mQ| \ge |m|Q - |v_k| \ge Q - B_{\max}$. The hypothesis $|c_k| \le T_Q = Q - 1 - B_{\max}$ contradicts this, forcing $m = 0$. □

Once the true product is known to lie inside the recovery window $[-T_Q, T_Q]$, the modular output has no room to represent any other integer congruent to it modulo Q.

Table 3: Compositional SMT verification of pipeline output bounds for part (ii) of Lemma 2. Each Montgomery reduction is verified via a QF_BV query. At n=256, $\max_k B_{\text{out}}(k) \le 4{,}201{,}525$.

n	Layers	SMT queries	$\max_k B_{\text{out}}(k)$	Status
2	1	7	4,196,100	UNSAT
4	2	20	4,200,785	UNSAT
8	3	52	4,200,857	UNSAT
16	4	128	4,200,949	UNSAT
32	5	304	4,201,062	UNSAT
64	6	704	4,201,196	UNSAT
128	7	1,600	4,201,350	UNSAT
256	8	3,584	4,201,525	UNSAT

Lemma 3 (Sparse-challenge convolution bound). *Let $c \in R_Q$ have at most τ nonzero coefficients, each in $\{-1, +1\}$, and let $u \in R_Q$ satisfy $\|u\|_\infty \le \beta$. Then $\|c \cdot u\|_\infty \le \tau\,\beta$.*

Proof. Each coefficient $(c \cdot u)_k$ is a signed sum of at most $\|c\|_0 \le \tau$ terms of the form $\pm c_i\, u_{k-i}$ in R_Q (with negacyclic sign on the wrap). Triangle inequality: $|(c \cdot u)_k| \le \sum_{i:c_i \neq 0} |u_{k-i}| \le \tau\beta$. □

Theorem 1 (Exact integer recovery for sparse products). *For ML-DSA ($Q = 8{,}380{,}417$, $R = 2^{32}$, $n = 256$), let $c \in R_Q$ be a challenge polynomial with at most τ nonzero coefficients in $\{-1, +1\}$, and let $u \in R_Q$ satisfy $\|u\|_\infty \le \beta$. If $\tau\,\beta \le T_Q = Q - 1 - B_{\max}$, then the Montgomery-form NTT multiplication pipeline returns the exact integer product:*

$$\Phi(\hat{c}, \hat{u})_k = (c \cdot u)_k \qquad \text{for every coefficient } k.$$

With $B_{\max} \le 4{,}201{,}525$ from Lemma 2(ii), $T_Q \ge 4{,}178{,}891$.

Proof. Set $v_k = \Phi(\hat{c}, \hat{u})_k$ and $c_k = (c \cdot u)_k$. By Lemma 1, $v_k \equiv c_k \pmod{Q}$. The coefficient-wise Montgomery multiplications $\hat{c} \circ \hat{u}$ before the inverse NTT produce signed representatives in $(-Q, Q)$, so the input contract of Lemma 2 holds for the InvNTT input; part (ii) then gives $|v_k| \le B_{\max} < Q$. By Lemma 3 and the hypothesis $\tau\beta \le T_Q$, we have $|c_k| \le \tau\beta \le T_Q$. Proposition 1 then gives $v_k = c_k$. □

Numerically, the recovery window is enormous: $T_Q/(\tau\eta) \ge 4{,}178{,}891/196 > 21{,}000$, so the sparse-challenge product occupies less than 0.005% of the integer-recovery window. Theorem 1 instantiates twice for ML-DSA: with $u = s_i$, $\beta = \eta$ giving $\tau\beta \le 196$ (Steps 11, 14, 12, 15a); and with $u = t_0$, $\beta = 2^{D-1} = 4{,}096$ giving $\tau\beta \le 245{,}760$ (Step 18). Both are well below $T_Q \ge 4{,}178{,}891$. This is the point at which the classification changes: under the generic post-InvNTT bound $(-Q, Q)$ these sites appear to require reduction; under exact integer recovery

they inherit the much smaller $[-\tau\beta, \tau\beta]$ bound and become fixed-point no-ops. Together with FIPS 204 `BitPack`'s centered-input semantics that makes Step 21 `freeze` unnecessary, this classifies 6 of the 10 reference-topology reduction sites as safe to remove.

Coq Cross-Check of the Arithmetic Core. The arithmetic proof obligations used in the paper-level argument (Proposition 1, the sparse-convolution bound of Lemma 3, the concrete $\tau\eta$ and T_Q instantiations, and the headroom calculations) are also checked in an axiom-free Coq development (`01-reduction-site-proofs/coq/correctness.v` in the supplementary material). The bit-vector Montgomery bound and the compositional value of $B_{\max}$ are certified separately by the replayable SMT-LIB2 queries described above.

3.2 Site Classification Criterion

We now turn the bounds from Section 3.1 into a per-site audit criterion. For each candidate reduction site p, the input to the criterion is a reachable bound B_p on the value arriving at p and the fixed-point threshold T_r of the reduction operator r used at that site. We summarize their separation by the *headroom ratio*

$$\rho_p \;=\; \frac{B_p}{T_r}.$$

For arithmetic reductions such as `reduce32`, the interpretation is direct. If $\rho_p < 1$, then every reachable value already lies inside the fixed-point region of r, so the reduction is a semantic no-op at p and is classified as *redundant*. If $\rho_p \geq 1$, the fixed-point criterion alone cannot justify removal: the reachable bound crosses the identity-region threshold, so we require a necessity witness or a downstream-precondition argument before classifying the reduction as load-bearing.

This distinction is important because not all reductions serve the same role. Some sites are arithmetic normalizations, where the fixed-point region is the relevant correctness condition. Others are representation-changing operators that establish preconditions for downstream routines. For example, `caddq` is necessary before `HighBits` because it restores a non-negative representative, even if overflow is not the issue. Conversely, `freeze` before signature encoding is unnecessary in our topology because FIPS 204 `BitPack` accepts centered z directly. Thus, our criterion has two layers: a local fixed-point test for arithmetic no-ops, followed by a downstream-precondition check for representation-sensitive sites.

The output of the criterion is a site classification used by the cross-library audit:

$$p \in S^* \quad \text{if the reduction at } p \text{ is load-bearing,}$$

and otherwise the site is marked safe to omit under the ML-DSA constraints. In Section 4, this criterion classifies four sites in the audited reference-style C topology as load-bearing and six as redundant; Section 3.3 explains how each classification is backed by an SMT certificate.

3.3 SMT Certification Procedure

Every redundancy and necessity claim is certified by an SMT query of the form:

$$\exists\, \boldsymbol{v} :\ \phi_{\text{scheme}}(\boldsymbol{v})\ \wedge\ f(\boldsymbol{v}) \notin \text{Fix}(r), \tag{7}$$

where ϕ_{scheme} encodes scheme constraints—sampling bounds and sparsity—while f computes the value reaching the reduction site and $\text{Fix}(r)$ is the exact fixed-point region of the operator r used at that site (Table 1). For `reduce32`, this region is the half-open interval $[-2^{22}, 2^{22})$ under the reference arithmetic-shift model. An UNSAT result certifies redundancy: no scheme-valid input can drive $f(\boldsymbol{v})$ outside $\text{Fix}(r)$, so the reduction is a no-op for all valid inputs. A SAT result certifies necessity: the solver returns a concrete witness—specific values of y, c, s_1 etc.—that demonstrate the reduction is needed.

Modeling Montgomery Reduction in QF_BV. To certify the strict bound $|\texttt{montgomery_reduce}(x)| < Q$, we model the Montgomery reduction with exact bit-vector semantics. The pq-crystals reference implementation computes:

Listing 1.1: Montgomery reduction in pq-crystals (reduce.c).

```
int32_t montgomery_reduce(int64_t a) {
  int32_t t;
  t = (int32_t)a * QINV; // low 32 bits
  t = (a - (int64_t)t * Q) >> 32; // exact division by R
  return t;
}
```

Implementation contract for QF_BV replay. The C expression used in the reference-style Montgomery reduction contains operations whose behavior is not fully specified by ISO C across all targets: converting a 64-bit value to `int32_t` when the value is outside the signed 32-bit range is implementation-defined, and right-shifting a negative signed integer is implementation-defined. Our QF_BV certificates therefore do not claim portability over arbitrary C implementations. Instead, they certify the implementation contract used by the tested two's-complement targets and by our replay model: the cast to `int32_t` extracts the low 32 bits and interprets them as a signed two's-complement value, and the final signed right shift is arithmetic.

Under this contract, we encode the input a as a 64-bit bit-vector, extract the low word with `(_ extract 31 0)`, interpret it as signed by `(_ sign_extend 32)`, and encode the final shift using `bvashr`. Each query asserts the layer-specific input bound $|a| \le B_{\text{in}}$ and asks whether $|t| \ge Q$ is satisfiable; UNSAT certifies $|\texttt{montgomery_reduce}(a)| < Q$ under this implementation contract. The query quantifies over all 64-bit assignments satisfying the bound; no sampling or test-vector enumeration is used. The complete replayable SMT-LIB2 query template is provided in the supplementary material.

Redundancy Certification (QF_LIA). The following query certifies that `reduce32` at Step 12 ($z = y + cs_1$) is redundant for ML-DSA-44. It encodes the scheme constraints—$y \in [-\gamma_1 + 1, \gamma_1]$ (the asymmetric range produced by `ExpandMask`) and the convolution bound $|cs_1| \leq \tau\eta$—and asks whether the result can fall outside the `reduce32` fixed-point interval $[-2^{22}, 2^{22})$:

```
(set-logic QF_LIA)
(declare-fun y () Int)
(declare-fun cs1 () Int)      ; a single convolution coefficient
(assert (and (<= (- 131071) y) (<= y 131072)))  ; y in [-gamma1+1, gamma1]
(assert (and (<= (- 78) cs1)   (<= cs1 78)))    ; |cs1| <= tau*eta
(assert (or (>= (+ y cs1) 4194304)
            (<  (+ y cs1) (- 4194304))))  ; outside [-2^22, 2^22)?
(check-sat)  ; UNSAT => z always in [-2^22, 2^22) => reduce32 redundant
```

The solver returns UNSAT: no combination of valid y and cs_1 can place z outside the half-open fixed-point interval $[-2^{22}, 2^{22})$. This single query replaces enumerating tens of millions of boundary combinations—and unlike exhaustive testing, it provides a mathematical proof.

Necessity Certification with Concrete Witness. For Step 15 ($w - cs_2$), the query asks whether any scheme-valid input places $w - cs_2$ outside the half-open `reduce32` fixed-point interval $[-2^{22}, 2^{22})$:

```
(set-logic QF_LIA)
(declare-fun w () Int)
(declare-fun cs2 () Int)
(assert (and (<= 0 w) (< w 8380417)))        ; w in [0, Q) after caddq
(assert (and (<= (- 78) cs2) (<= cs2 78)))   ; |cs2| <= tau*eta
(assert (or (>= (- w cs2) 4194304) (<  (- w cs2) (- 4194304))))
(check-sat)  ; SAT => necessary!
(get-model)  ; witness: w = 8380416, cs2 = -78 => w-cs2 = 8380494
```

The solver returns SAT with witness $w = 8{,}380{,}416$, $cs_2 = -78$, giving $w - cs_2 = 8{,}380{,}494 > 2^{22}$. This is a concrete input that any correct reference-style implementation must handle; without `reduce32`, the downstream `Decompose` receives an out-of-range value.

Scalability. Two optimizations preserve soundness. First, negacyclic convolution is index-shift invariant, so a single representative coefficient suffices per query. Second, we solve under relaxed cardinality ($\leq \tau$ non-zeros); redundancy under relaxation implies redundancy under the exact constraint.

Verification Time. On a standard single-threaded workstation, the 68 SMT-LIB2 certificates finish in under 2 minutes (each QF_LIA query < 5 s, each QF_BV query < 0.3 s); the compositional bound-propagation procedure also runs in minutes on the same setup. All proofs are deterministic

and independently replayable. The supplementary material provides the 68 SMT-LIB2 certificates (`01-reduction-site-proofs/`) and the compositional bound-propagation generator, per-query result summary, and verification script (`02-compositional-bound-queries/`).

4 Tracing Bounds Through the ML-DSA Signing Path

We trace coefficient bounds through every step of `ML-DSA.Sign_internal` (FIPS 204, Algorithm 7 [28]), the internal signing routine reached from the external `ML-DSA.Sign` wrapper (FIPS 204, Algorithm 2), and classify each reduction in the reference signing-path topology. Figure 1 maps the 21 steps to the 10 candidate reduction sites and previews the audit result of Section 5: 4 sites are necessary (collected as S^*), and 6 are redundant.

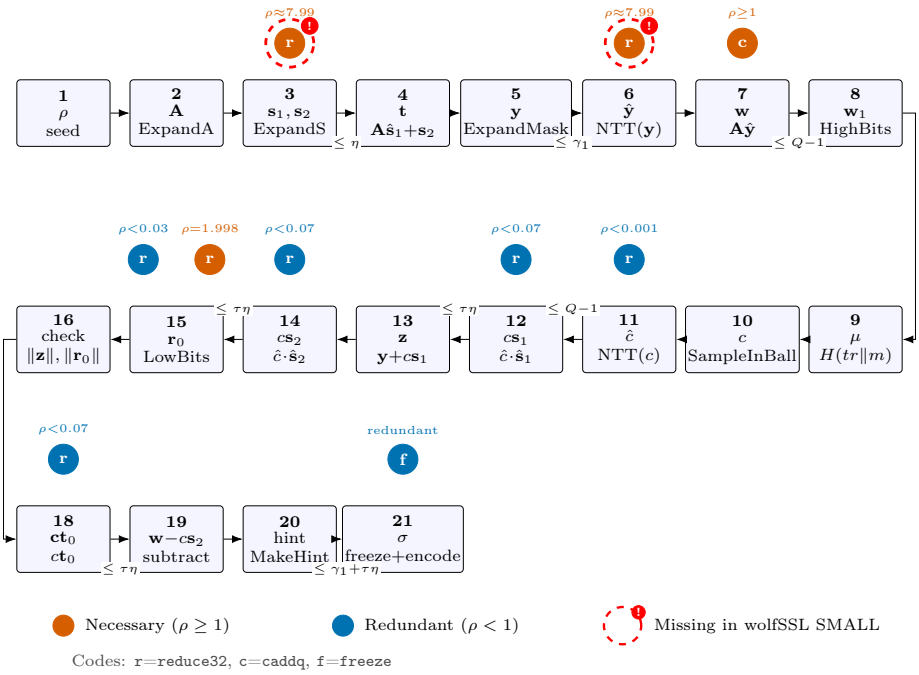

Fig. 1: ML-DSA-44 signing path with the 10 candidate reduction sites. Box numbers follow the reference-topology reduction-site labels used throughout Section 4 and Table 4. Filled nodes = necessary; open nodes = redundant. Dashed circles mark the necessary reductions missing in wolfSSL's `WOLFSSL_DILITHIUM_SMALL` build path. Per-site classification details and the verification analog of r_6 are discussed in Section 5.2.

Table 4: Core reduction-site classification used in the audit.

Site class	Sites	Classification
Dense pre-InvNTT products	r_3, r_6	Necessary
Canonicalization	c_7	Necessary
Pre-`Decompose` subtraction	r_{15}	Necessary in reference topology
Sparse-challenge products	$r_{11}, r_{12}, r_{14}, r_{15a}, r_{18}$	Redundant
Centered signature encoding	f_{21}	Redundant

4.1 Load-Bearing Sites: Dense Products and Canonicalization

Steps 3 and 6: Dense Products, Reduction before *InvNTT.* These steps compute $\mathrm{InvNTT}(\hat{A}\hat{s}_1) + s_2$ and $\mathrm{InvNTT}(\hat{A}\hat{y})$, involving the uniform matrix A whose coefficients span $[0, Q)$. The reduction sites r_3 and r_6 are placed by pq-crystals *between* the NTT-domain matrix-vector multiplication and the InvNTT call:

```
/* sum of l Montgomery prods */
matrix_pointwise_montgomery(t, A, s1hat);

/* <-- r3 (or r6 for sign) */
polyveck_reduce(t);
polyveck_invntt_tomont(t);
```

The bound that matters for these sites is therefore *the input to the InvNTT*—the unreduced accumulation of ℓ Montgomery-reduced products, each bounded by $Q - 1$:

$$|t_k| \;\le\; \ell\,(Q-1) \qquad \text{for } \ell \in \{4, 5, 7\} \text{ across ML-DSA-44/65/87.}$$

Without this reduction the InvNTT receives a magnitude up to $\ell(Q-1)$; the GS-butterfly addition branch can then double the bound across layers. For ML-DSA-87, this reaches $2^d \cdot 7(Q-1)$ after d=8 layers, which exceeds the signed 32-bit range (Section 5.2; Coq theorem `dsa87_overflow_witness` in `01-reduction-site-proofs/coq/correctness.v`). The headroom ratio against the displayed 2^{22} boundary of the half-open `reduce32` fixed-point interval is

$$\rho \;=\; \frac{\ell(Q-1)}{2^{22}} \;\in\; \{7.99,\, 9.99,\, 13.99\} \geq 1 \quad\Longrightarrow\quad \textbf{necessary.}$$

SMT certification: Z3 returns SAT with witness ℓ Montgomery-reduced terms each saturating at $\pm(Q-1)$ and aligning in sign, giving $|t_k| = \ell(Q-1) > 2^{22}$.

Step 7: ***`caddq`*** *before HighBits.* `HighBits` (FIPS 204, Algorithm 37) returns the high-order component r_1 produced by `Decompose` (Algorithm 36). FIPS 204 `Decompose` performs $r^+ \leftarrow r \bmod {}^+ q$ internally, so the FIPS algorithm itself does not require a separate `caddq`. The audited reference-style C topology, however,

uses an optimized `HighBits`/`Decompose` code path that assumes its input is already a canonical non-negative representative; therefore `caddq` restores w to $[0, Q)$ before this call. The InvNTT can produce negative representatives such as $w_k = -1$, so omitting `caddq` violates this representation precondition (the optimized code path then yields $\texttt{HighBits}(-1) = -1 \neq Q/\alpha - 1$). The `caddq` operator maps $a \mapsto a + Q \cdot \mathbf{1}[a < 0]$, restoring non-negativity. No overflow-guarding reduction can substitute for this canonicalization step. SMT witness: $w_k = -1$. **Necessary** under the reference C topology (canonicalizing).

Step 15: `reduce32` before Decompose. After Step 7, $w \in [0, Q)$. The sparse-challenge product $cs_2 = \text{InvNTT}(\hat{c} \cdot \hat{s}_2)$ has $|cs_2| \leq \tau\eta = 78$ by Theorem 1. The subtraction:

$$|w_k - (cs_2)_k| \leq (Q-1) + \tau\eta = 8{,}380{,}494,$$

which lies outside the half-open `reduce32` fixed-point interval $[-2^{22}, 2^{22})$. $\rho = 8{,}380{,}494/4{,}194{,}304 = 1.998$: **necessary**. Z3 returns SAT with witness $w_k = Q - 1 = 8{,}380{,}416$, $(cs_2)_k = -78$, giving $w_k - (cs_2)_k = 8{,}380{,}494$. Without `reduce32`, this value reaches `Decompose` without first being normalized into the half-open interval $[-2^{22}, 2^{22})$ expected by the reference-topology reduction schedule.

4.2 Redundant Sites: Sparse-Challenge Products and Centered Encoding

Five of the six redundant sites lie on *sparse-challenge product paths*. Theorem 1 converts the generic post-InvNTT bound $(-Q, Q)$ into tight integer bounds determined by $\|c\|_1 = \tau$, making each site a fixed-point no-op for `reduce32`. Table 5 summarizes the resulting bounds; all four sites yield $\rho < 0.07$, far inside the half-open `reduce32` fixed-point interval $[-2^{22}, 2^{22})$. Without Theorem 1, the conservative bound at Step 12 would give $\rho = 2.03$, making the site appear load-bearing; the tight bound converts this to $\rho = 0.031$, a 65× improvement. SMT certification returns UNSAT for each redundancy claim and SAT (with the bound saturated by aligning τ challenge entries in sign) for each tightness check.

Table 5: Sparse-challenge sites classified as redundant by Theorem 1.

Site	Value computed	Worst-case bound	ρ
r_{11}, r_{14}	$cs_i = \text{InvNTT}(\hat{c} \cdot \hat{s}_i)$	$\tau\eta \leq 196$	< 0.0001
r_{12}	$z = y + cs_1$	$\gamma_1 + \tau\eta = 131{,}150$ (ML-DSA-44)	0.031
r_{15a}	$w_0 - cs_2$ after `Decompose`	$\gamma_2 + \tau\eta = 262{,}084$ (ML-DSA-65)	0.063
r_{18}	$ct_0 = \text{InvNTT}(\hat{c} \cdot \hat{t}_0)$	$\tau \cdot 2^{D-1} = 245{,}760$ (ML-DSA-87)	0.059

Table 6: Redundancy of `reduce32` on $z = y + cs_1$ across ML-DSA parameter sets. UNSAT (*Redundancy*) certifies that z never leaves the half-open `reduce32` fixed-point interval $[-2^{22}, 2^{22})$; SAT (*Bound tight*) returns a concrete witness saturating the bound $B = \gamma_1 + \tau\eta$.

Param set	γ_1	τ	η	B	ρ	Redundancy	Bound tight
ML-DSA-44	2^{17}	39	2	131,150	3.1%	UNSAT	SAT
ML-DSA-65	2^{19}	49	4	524,484	12.5%	UNSAT	SAT
ML-DSA-87	2^{19}	60	2	524,408	12.5%	UNSAT	SAT

Step 21: `freeze` before `sigEncode`. A naive analysis would conclude that z must be reduced to $[0, Q)$ before encoding, but FIPS 204 `BitPack` (Algorithm 17) accepts polynomial coefficients in $[-a, b]$, and `sigEncode` (Algorithm 26, line 3) invokes it as $\texttt{BitPack}(z[i], \gamma_1 - 1, \gamma_1)$. The packing maps $z[j]$ to $\gamma_1 - z[j] \in [0, 2\gamma_1 - 1]$ for any $z[j] \in [-\gamma_1 + 1, \gamma_1]$ – including negative z. No `freeze` is required. **Redundant** (and indeed pq-crystals' `polyz_pack` omits it).

4.3 Summary: Which Reductions Can Be Removed?

The 4 reductions that *must* remain in the audited reference-style C topology are:

$$S^* = \left\{\texttt{reduce32}_{\text{Step 3}},\ \texttt{reduce32}_{\text{Step 6}},\ \texttt{caddq}_{\text{Step 7}},\ \texttt{reduce32}_{\text{Step 15}}\right\}. \quad (8)$$

The remaining 6 are safe to remove: five lie on sparse-challenge product paths where Theorem 1 guarantees exact integer recovery, so intermediate values never exceed the representable range; the sixth is the `freeze` at Step 21, which is unnecessary because FIPS 204 `BitPack` accepts centered z. Removing any element of S^* from the audited reference-style C topology causes overflow, $\mathbb{Z}_q$-incorrect output, or representation-precondition violation downstream. Figure 2 visualizes this separation: the log-scale y-axis makes the four-order-of-magnitude gap between necessary and redundant sites visible—Step 15 lies near $Q \approx 8.4$M, while the dense pre-InvNTT sites at Steps 3 and 6 are higher still ($\ell(Q-1)$, tens of millions), all above the displayed 2^{22} boundary; redundant sites span 78 to 2.6×10^5, all well below it. No site falls in the ambiguous band, indicating that the ρ criterion is not borderline for any ML-DSA reduction. Table 6 confirms that the $z = y + cs_1$ redundancy holds across all three ML-DSA parameter sets ($\rho \leq 12.5\%$ in every case).
The complete 21-step signing-path bound trace—per-step variables, operators, $\|v\|_\infty$ bounds, classifications, and SMT tightness checks—is provided in the supplementary material.

5 Cross-Library Audit

We audit eight production ML-DSA implementations spanning four reduction strategies: explicit per-site reductions, always-reduced arithmetic, implicit Mont-

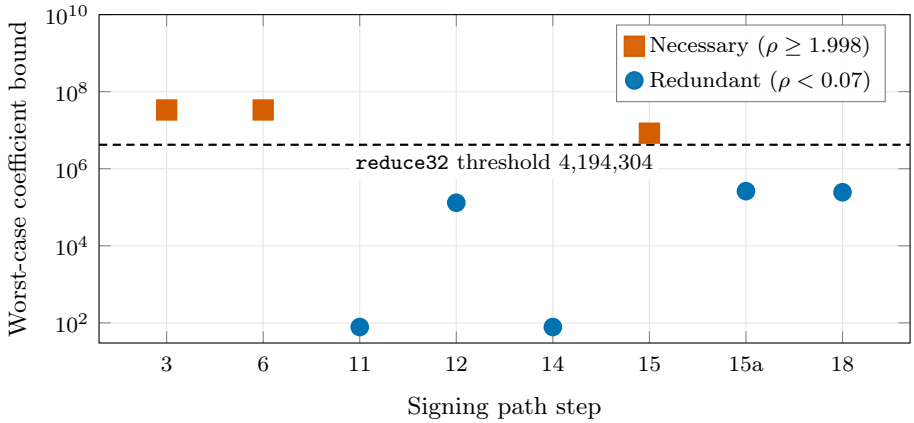

Fig. 2: Worst-case coefficient bounds at each `reduce32` site in ML-DSA-44 signing, using the reference-topology step labels from Figure 1. The dashed line marks the displayed 2^{22} boundary of the half-open `reduce32` fixed-point interval; sites above are necessary (Steps 3, 6, 15), sites below are redundant (Steps 11, 12, 14, 15a, 18). The separation is clean: every necessary site has $\rho \geq 1.998$ or $\rho \geq 7.99$, while every redundant site has $\rho < 0.07$. The canonicalizing `caddq` at Step 7 is necessary by precondition; `freeze` at Step 21 is unnecessary because `BitPack` accepts centered z.

gomery normalization, and hybrid native backends. The audit reveals three distinct reduction-placement behaviors: *conservative* implementations that retain all relevant reductions, *safely optimized* implementations that remove only redundant sites, and an implementation with an *unsafe omission* of load-bearing reductions. Without the classification S^*, AWS-LC's safe removals and wolfSSL's unsafe removals are indistinguishable at the code-review level: both simply appear as removed reductions.

5.1 Audit Setup and Results

For each library, we extracted the ML-DSA signing implementation, mapped reduction calls to the 21-step signing trace, checked the four load-bearing sites in S^* for presence, and recorded whether the six redundant sites were retained or removed. A missing necessary reduction is classified as an *unsafe omission*; a removed redundant site is treated as a safe optimization. The full per-step trace is in the supplementary material; core sites are summarized in Table 4.

We audited 8 libraries spanning four reduction strategies. The explicit per-site group comprises pq-crystals [12], PQClean [31], AWS-LC [3], and wolfSSL [34]. The always-reduced group comprises BoringSSL [16] and SymCrypt [23]. Go stdlib [33] uses implicit Montgomery normalization, and liboqs [30] uses a hybrid native backend via mldsa-native [24]. Table 7 reports

the pinned commit used for each library. For wolfSSL, we audited both the default path and `-DWOLFSSL_DILITHIUM_SMALL`; only the latter omits the load-bearing post-matrix-multiplication reductions.

Table 7: Safety-critical reduction audit at pinned revisions. ✓: present; ×: missing necessary reduction; S: subsumed by a stronger invariant; T: topology-absent. wolfSSL was audited in both the default and `WOLFSSL_DILITHIUM_SMALL` modes; the unsafe omission appears only in the latter.

Library	Commit	r_3	r_6	c_7	r_{15}
pq-crystals	`6e00625c5b29`	✓	✓	✓	✓
AWS-LC	`830d7041abe1`	✓	S	✓	✓
wolfSSL SMALL	`ca5b484e23be`	×	×	✓	T
PQClean	`3730b32aa50b`	✓	✓	✓	✓
liboqs	`c58e93e97f46`	✓	S	✓	✓
BoringSSL	`305bcfce00b1`	S	S	S	S
SymCrypt	`9bf32df23d33`	S	S	S	S
Go stdlib	`9777ceceec8f`	S	S	S	S

All audited libraries except wolfSSL's SMALL mode maintain the necessary reductions. wolfSSL SMALL is the sole outlier, missing the post-matrix-multiplication `reduce32` at Steps 3 and 6. The r_{15} site does not appear in wolfSSL's signing topology, which subtracts cs_2 after `Decompose`.

5.2 wolfSSL SMALL: Missing Load-Bearing Reductions

The memory-optimized `WOLFSSL_DILITHIUM_SMALL` mode omits the post-matrix-multiplication `reduce32` at three program points—key generation, signing, and verification—corresponding to Steps 3 and 6 in our reference numbering plus the verification analog of Step 6. The canonicalizing `caddq` at Step 7 is present; the r_{15} site does not appear in wolfSSL's topology, which subtracts cs_2 from the post-`Decompose` value w_0 rather than from w directly. This build configuration is relevant to memory-constrained embedded deployments, where reducing temporary storage and memory traffic is often a design goal.

Steps 3 and 6—Missing `reduce32` after `matrix_mul` Causes Signed 32-Bit Overflow. The root cause is in `dilithium_matrix_mul_c()`, which computes the NTT-domain matrix-vector product $w = A \cdot \hat{z}$. The pq-crystals reference calls `polyveck_reduce()` after this multiplication to normalize each coefficient into the small centered range required before InvNTT; wolfSSL's SMALL mode omits this call.

The concrete code (wolfSSL `dilithium.c`, SMALL mode) is:

Listing 1.2: wolfSSL SMALL mode: matrix-vector multiply without reduce (simplified).

```
/* SMALL mode: per-element Montgomery reduction in sword32 */
for (e = 0; e < 256; e++)
    r[e] = mont_red((sword64)a[0*256+e] * y[0*256+e]);
for (j = 1; j < l; j++)
    for (e = 0; e < 256; e++)
        r[e] += mont_red((sword64)a[j*256+e] * y[j*256+e]);
/* DEFAULT mode calls dilithium_reduce32() here.
   SMALL mode omits this reduction. */
```

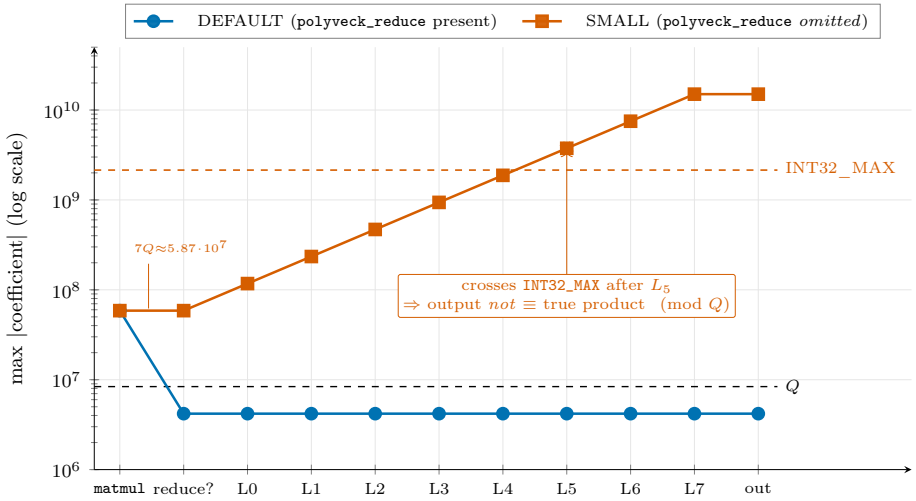

Fig. 3: DEFAULT vs. SMALL InvNTT pipelines for ML-DSA-87. DEFAULT (blue, solid) normalizes coefficients into the small centered range after the post-`matrix_mul` reduction; SMALL (orange, square markers) omits the call, so the $7Q$ accumulation crosses `INT32_MAX` after L_5, triggering signed-overflow behavior. Per-layer crossing details are in Section 5.2.

Each `mont_red()` returns a value in $(-Q, Q)$. After accumulating l columns ($l = 7$ for ML-DSA-87), each `r[e]` lies in $(-7Q, 7Q) \approx (-58.7\text{M}, +58.7\text{M})$. Compare with the pq-crystals reference:

Listing 1.3: pq-crystals reference: `polyveck_reduce()` after matrix multiply.

```
polyvec_matrix_pointwise_montgomery(&t1, mat, &s1hat);
polyveck_reduce(&t1); /* <-- normalizes before InvNTT */
polyveck_invntt_tomont(&t1);
```

Without reduction, the subsequent InvNTT's GS butterfly doubles the magnitude at each addition layer. After 7 doublings (the output of layer L_6, at GS-stride $\ell = 128$), position 0 accumulates the sum of 128 unreduced coeffi-

Table 8: Concrete overflow instance for ML-DSA-87 SMALL mode; full input data in the supplementary material.

Component	Value	Note
ρ (public seed)	`f5491b44...d25d37`	32-byte hex
Overflow row	6 (of 8)	Row of matrix A
az_sum (NTT-domain)	1,907,109,921	88.8% of `INT32_MAX`
ct1_sum (msg-dep.)	−241,616,264	Subtracted in verification
Total at InvNTT pos 0	2,148,726,185	100.06% of `INT32_MAX`
Excess over `INT32_MAX`	+1,242,538	Exceeds signed 32-bit range; wraps on tested targets
`HighBits` (DEFAULT)	15	Correct
`HighBits` (SMALL)	14 ≠ 15	verification diverges

cients:

$$128 \times l(Q-1) = 128 \times 58{,}662{,}912 = 7.51 \times 10^9 > \texttt{INT32_MAX} = 2.15 \times 10^9.$$

(The first crossing actually occurs one layer earlier, at the output of L_5, where the all-addition path accumulates to approximately 3.76×10^9; we use the larger L_6 value as a clearer "well past INT32_MAX" witness for the figure.) At the C-language level, this signed 32-bit integer overflow invalidates the execution semantics. On the tested two's-complement targets, and in our bit-vector replay model, the overflow manifests as wraparound. The result is $\mathbb{Z}_q$-incorrect: on algebraically zero input $f[i] = lQ \equiv 0 \pmod{Q}$, the observed wraparound execution yields $-83{,}956 \not\equiv 0 \pmod{Q}$. Figure 3 contrasts the DEFAULT and SMALL pipelines on a log scale, showing how the single missing `polyveck_reduce()` call cascades to overflow.

Concrete Overflow Instance. The Coq development proves the bound *structure* that permits the overflow (Lemma 'dsa87_overflow_witness': $2^d \cdot \ell Q >$ `INT32_MAX`); the specific witness values reported below were generated by the SMT-guided search bundled in the supplementary material, and the resulting signed-overflow behavior is confirmed at runtime by UBSan when running the bundled C reproducer. The public seed ρ yields a matrix A whose row 6 has an unusually high NTT-domain coefficient sum. Combined with a crafted sparse vector z with 5 non-zero positions per column, chosen by exhaustive search over $\binom{256}{5}$ candidates, the NTT-domain accumulation $\hat{A}[6] \cdot \hat{z}$ sums to 1,907,109,921 at position 0. After subtracting the message-dependent ct_1 contribution, the total reaches 2,148,726,185—exceeding `INT32_MAX` by 1,242,538, a margin of 0.06% (Table 8).
The `HighBits` divergence (15 vs. 14) propagates through the commitment hash, producing $\tilde{c} \neq c$ and deterministic verification failure.

Verification Analog. Verification computes $w' = \text{InvNTT}(\hat{A} \cdot \hat{z} - \hat{c} \cdot \hat{t}_1 \cdot 2^d)$ in FIPS 204 `ML-DSA.Verify_internal` (Algorithm 8, line 9) via the same matrix-multiplication routine. wolfSSL's SMALL mode omits the post-`matrix_mul`

`reduce32` here as well, so the signed 32-bit integer overflow can manifest in verification independently of signing – this is the third program point of the same defect, and is the basis of the cross-verifier inconsistency analyzed in Section 5.3.

Root Cause Analysis. Table 9 shows the complete picture. wolfSSL's SMALL path removes 3 redundant reductions at Steps 11, 14, and 15a, all with $\rho < 0.06$, and also omits the unnecessary `freeze` at Step 21. However, the same path removes the post-`matrix_mul` `reduce32` at Steps 3 and 6, as well as the verification analog of Step 6, all with $\rho \geq 7.99$ at the pre-InvNTT site. The r_{15} site does not arise in wolfSSL's topology because `Decompose` is applied before the cs_2 subtraction. The headroom ratio ρ cleanly separates safe from load-bearing removals: every redundant site has $\rho < 0.07$, every necessary overflow-guarding site has $\rho \geq 1.998$ at Step 15, and the pre-InvNTT sites at Steps 3, 6, and $6'$ have $\rho \geq 7.99$.

Table 9: wolfSSL SMALL mode reduction decisions vs. our classification. $\mathbf{No}_N$: absent necessary reduction (bug). No_R: absent redundant reduction (correct removal). $\mathrm{N/A}_{\mathrm{topo}}$: site absent because of a different signing topology.

Step	Operator	ρ	Status	Classification
3	`reduce32`	7.99–13.99	$\mathbf{No}_N$	Necessary (pre-InvNTT)
6	`reduce32`	7.99–13.99	$\mathbf{No}_N$	Necessary (pre-InvNTT)
$6'$ (verify)	`reduce32`	7.99–13.99	$\mathbf{No}_N$	Necessary (pre-InvNTT)
7	`caddq`	—	Yes	Necessary
11	`reduce32`	< 0.001	No_R	Redundant
14	`reduce32`	< 0.001	No_R	Redundant
15	`reduce32`	1.998	$\mathrm{N/A}_{\mathrm{topo}}$	Necessary in reference
15a	`reduce32`	0.063	No_R	Redundant
18	`reduce32`	< 0.06	No_R	Redundant
21	`freeze`	—	No_R	Redundant

KAT-vs-SMT Demonstration. The scope limits of existing tools (Section 2.3) play out concretely on this defect. Extracting the matrix-multiply routines from both implementations (full source in the supplementary material) and running them on an SMT-found input yields a NIST output of $-23{,}077$ versus a wolfSSL output of $-8{,}403{,}494$—a difference of exactly $-Q = -8{,}380{,}417$, the $\mathbb{Z}_q$-incorrectness caused by the missing `reduce32`. The contrast with KAT-style regression testing is structural rather than empirical. By construction, any scheme-constrained input keeps the pre-InvNTT per-coefficient magnitude below $l(Q-1) \approx 5.8 \times 10^7$ for ML-DSA-87, well within `INT32_MAX`; the all-addition GS path that lifts this to $7.51 \times 10^9 >$ `INT32_MAX` requires the worst-case sign-aligned configuration that our SMT query produces as a witness but that a

finite test-vector corpus is unlikely to sample. KAT-style testing therefore cannot, in general, decide whether such a witness exists; the SMT query establishes existence over all scheme-constrained inputs.

5.3 Correctness Consequences

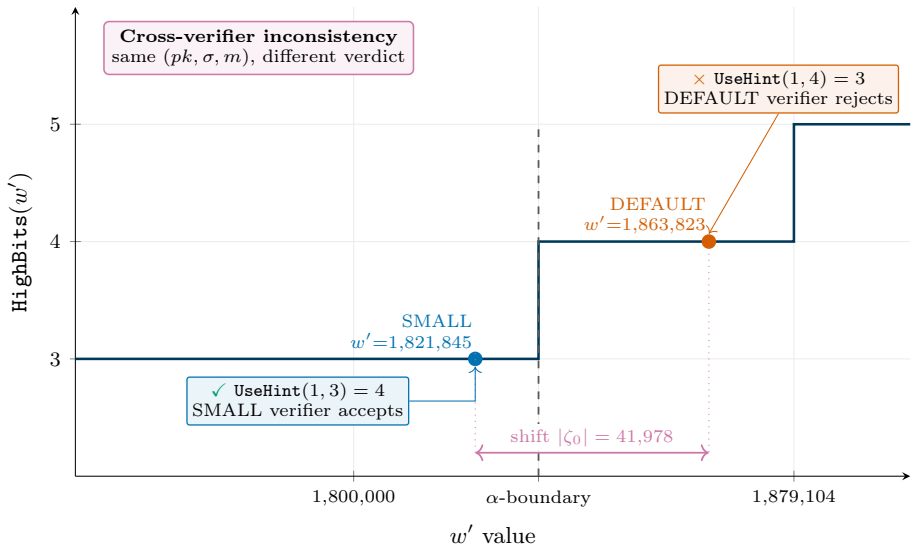

Fig. 4: Cross-verifier disagreement at a `HighBits` boundary. The observed wraparound shifts $w'[6][0]$ by $|\zeta_0| = 41{,}978$, crossing a step boundary so SMALL and DEFAULT land in different bins; a single hint flip then reconciles one build but breaks the other.

All four failure modes affect only the `WOLFSSL_DILITHIUM_SMALL` code path and are classified as CWE-190 and CWE-682; the default build is unaffected. Although the affected code path is not wolfSSL's default build, it targets precisely the memory-constrained deployment class—embedded devices, smart cards, automotive ECUs, and IoT systems—where PQC implementations face the strongest pressure to remove memory traffic and reductions. The primary impact is deterministic signing failure—a denial of service. Additionally, the signed 32-bit integer overflow causes a *cross-verifier inconsistency*: the same public key, signature, and message together produce different verification outcomes depending on the build mode.

Mechanism (Reproduced, Not Formally Proved). We reproduce this mechanism with a minimal C harness rather than treating it as a formal theorem; the harness fixes the relevant wolfSSL build path and input state, then observes the

HighBits transition. ML-DSA verification (FIPS 204 `ML-DSA.Verify_internal`, Algorithm 8, line 9) computes $w' = \text{InvNTT}(\hat{A} \cdot \hat{z} - \hat{c} \cdot \hat{t}_1 \cdot 2^d)$ and derives $w'_1 = \texttt{UseHint}(h, w')$. The observed wraparound shifts $w'[6][0]$ by exactly $-|\texttt{ZETAS_MONT}[0]| = -41{,}978$, pushing the value across an α-boundary:

$$\texttt{HighBits}(w'[6][0] = 1{,}863{,}823) = 4 \quad \text{(DEFAULT, correct)},$$
$$\texttt{HighBits}(w'[6][0] = 1{,}821{,}845) = 3 \quad \text{(SMALL, shifted by } -41{,}978).$$

By flipping a single hint bit ($h[6][0] : 0 \to 1$), the SMALL-mode verifier reconstructs the correct $w_1 = 4$ via $\texttt{UseHint}(1, 3) = 4$, while the DEFAULT-mode verifier computes $\texttt{UseHint}(1, 4) = 3 \neq 4$ and rejects. The SMALL-mode verifier accepts a signature that the DEFAULT-mode verifier rejects—an interoperability failure. Figure 4 traces the mechanism geometrically.

FIPS 204 Conformance Violation. The signed 32-bit integer overflow produces InvNTT outputs *not congruent modulo* Q (Table 10):

Table 10: $\mathbb{Z}_q$-incorrectness: InvNTT output on algebraically zero input $f[i] = lQ \equiv 0$.

Parameter set	Correct output	Observed output $\equiv 0 \pmod{Q}$?
ML-DSA-44 ($l = 4$)	0	$-83{,}956$ NO; two boundary crossings
ML-DSA-65 ($l = 5$)	0	$-83{,}956$ NO; two boundary crossings
ML-DSA-87 ($l = 7$)	0	$-125{,}934$ NO; three boundary crossings

The error is $k \times |\texttt{ZETAS_MONT}[0]|$ where k is the number of 2^{32}-boundary crossings. The implementation clause of FIPS 204 permits alternative procedures only when they produce the correct output for every input. The SMALL-mode InvNTT violates this requirement for all three parameter sets.

Impact Assessment. The cross-verifier inconsistency means that a protocol deploying SMALL-mode verifiers and DEFAULT-mode verifiers can disagree on signature validity for specific inputs. We do not claim an EUF-CMA break. The demonstrated impact is correctness: FIPS 204 non-compliance, undefined signed 32-bit overflow behavior, deterministic signing failure, and interoperability failure between build modes. In particular, the cross-verifier inconsistency is an interoperability failure, not a forgery attack under the standard EUF-CMA model.

Minimal Correctness Reproducers. The supplementary material includes minimal correctness reproducers for each of the four `WOLFSSL_DILITHIUM_SMALL` failure modes discussed above (`03-wolfssl-small-reproducers/`): signed 32-bit integer overflow, non-congruent InvNTT output modulo Q, deterministic signing failure, and FIPS 204 non-compliance manifest as cross-verifier disagreement.

Table 11: Evidence summary for the major claims in this paper. All 68 SMT-LIB2 certificates decide under Z3's default tactics.

Claim	Evidence	Tier
Reduction-site classification (S^*) across ML-DSA-44/65/87	39 SMT-LIB2 certificates: ten reduction sites across three parameter sets give 30 base certificates, plus 9 tightness witnesses	SMT-certified
Strict Montgomery per-layer bound (Lemma 2(i))	15 QF_BV certificates	SMT-certified
Cross-scheme analysis (ML-KEM, FN-DSA)	14 SMT-LIB2 certificates	SMT-certified
Compositional pipeline bound (Lemma 2(ii))	6,399 Z3 Python API queries	SMT-certified
Integer-bound and centering arithmetic core	Coq development, 44 lemmas and 1 theorem, axiom-free	Coq-checked
wolfSSL SMALL failure mechanisms	Minimal C harnesses and UBSan traces	Reproduced

These are correctness reproducers, not exploit demonstrations; the cross-library audit table that contextualizes them is in `04-cross-library-audit/`.

6 Validation

6.1 Evidence Summary and Reproducibility

We do not claim that the entire wolfSSL implementation is formally verified. Instead, we mechanize the integer-bound arguments underlying the reduction-site classification, certify bit-vector subclaims with replayable SMT queries, and reproduce implementation-specific failures with C harnesses. The evidence is intentionally tiered: **SMT-certified** (a Z3/SMT-LIB2 query yields the expected SAT/UNSAT verdict and is deterministically replayable); **Coq-checked** (the proposition is proved in our axiom-free Coq development and accepted by Coq's kernel); **Reproduced** (a minimal C harness, compiled with the documented flags and UBSan where applicable, exhibits the predicted behaviour at runtime). The first two tiers machine-certify the arithmetic core; the implementation-level manifestations are reproduced experimentally rather than treated as formal theorems. Table 11 maps each major claim to its evidence and tier; per-scheme headroom maxima are $\rho_{\max} = 7.99/9.99/13.99$ for ML-DSA-44/65/87 (each: 10 sites, 6 redundant, 4 necessary).

7 Discussion

Other NIST Lattice Schemes. The same fixed-point analysis explains why the ML-DSA redundancy does not transfer mechanically to the other NIST lattice

standards. ML-KEM [27] (FIPS 203, $Q = 3{,}329$, 16-bit Barrett) has fixed-point width $Q/2 = 1{,}664$, only a factor of 2,500 smaller than `reduce32`'s 2^{22}, so all 6 candidate reductions exceed the threshold ($\rho > 1$). FN-DSA [14] ($q = 12{,}289$, $W = 16$) has only two bits of arithmetic slack, and its Gaussian signing path lacks the sparse-challenge exactness condition; every NTT site has $\rho_{\mathrm{NTT}} \in [3.38, 4.12]$, mandating 3–4 intermediate reductions per transform. Thus ML-DSA occupies a distinct regime: enough slack for sparse-challenge products to make some reductions redundant, but not enough to remove reductions on dense matrix products. Full cross-scheme SMT certificates are in the artifact.

Nature of the Analysis. Our methodology is a human-driven formal analysis, not a push-button automated verifier. The bound derivations require domain expertise: identifying the sparse-challenge ℓ_1-norm structure, mapping FIPS 204 algorithm steps to reduction sites, and formulating SMT queries with correct scheme constraints. The SMT solver automates the *verification*—determining whether a bound is tight and whether a counterexample exists—but not the *formulation*. The 68 SMT-LIB2 certificates and the bound-propagation procedure are the reproducible artifacts; the methodology that produced them is not push-button.

Scope and Limitations. Our analysis targets FIPS 204-conforming production C implementations using the standard ML-DSA modulus $Q = 8{,}380{,}417$ and Montgomery arithmetic—the configuration used by all eight surveyed libraries. Alternative NTT strategies using smaller primes such as 3329, 257, and 769 are employed by some optimized assembly implementations but are outside our scope; our audit addresses the standard arithmetic as specified in FIPS 204. Assembly-optimized implementations such as AVX2 and Neon use the same reduction schedules as their C counterparts and are covered by the same bound analysis; however, vector-instruction-level verification such as CryptoLine [15] is complementary. Note that even when a reduction is certified redundant, vector implementations may retain it because removing a single scalar operation within a SIMD lane offers no performance benefit [4]. Our goal is correctness classification, not optimization. Alternative modular reduction strategies such as Barrett [32] and Plantard require different fixed-point analyses and are left for future work.

8 Related Work

Implementation Bugs in Deployed PQC. Our work contributes to a growing body of research demonstrating that post-quantum implementations—even those derived from well-studied reference code—harbor non-trivial bugs. Guo et al. [18] exploited decryption-failure vulnerabilities in LAC, a NIST Round 2 candidate, to mount a practical CCA key-recovery attack. KyberSlash [5] discovered secret-dependent timing vulnerabilities across multiple ML-KEM libraries—including

the pq-crystals reference, Go stdlib, and CIRCL—demonstrating that even mature, widely-deployed code paths contain exploitable flaws. Our findings are orthogonal: KyberSlash finds *timing* side channels; we find *algebraic correctness* bugs. Both share the structural observation that production PQC code diverges from the reference in subtle ways that survive standard testing.

Formal Verification of Cryptographic Implementations. CryptoLine [15] verifies functional correctness and range properties of NTT assembly routines at the instruction level. Hwang et al. [20] applied CryptoLine to verify NTT multiplications for Kyber, SABER, and NTRU across AVX2 and Cortex-M4 platforms, demonstrating that individual NTT kernels compute the intended operations correctly. More recently, Chiu et al. [8] extended CryptoLine's range-checking capabilities to handle Barrett and Plantard arithmetic in addition to Montgomery. These are impressive verification achievements for per-function correctness. However, CryptoLine's per-function scope means it cannot determine whether a reduction is *redundant*: verifying that `reduce32` at Step 12 correctly computes $a - \lfloor aQ^{-1} \rceil \cdot Q$ does not reveal that the input already lies in the `reduce32` fixed-point interval $[-2^{22}, 2^{22})$, making the call a no-op. This requires tracking the sparse-challenge structure ($\|c\|_1 = \tau$) across the full signing path—beyond any single function's scope.

Jasmin/EasyCrypt [2, 1] provides machine-checked correctness, IND-CCA security, and constant-time proofs for complete cryptographic implementations, with a recent CRYPTO 2024 result [1] for ML-KEM. Fiat-Crypto [13] synthesizes verified field arithmetic; SILVER [21] verifies statistical independence properties of masked hardware. These tools verify that a *given* implementation is correct; our analysis determines the *minimum* set of reductions needed for correctness—a complementary question. A Jasmin proof of the pq-crystals signing routine succeeds with all reductions present but cannot reveal which are unnecessary, nor detect the post-matrix-multiplication `reduce32` missing at 3 program points in wolfSSL.

NTT Arithmetic and Lazy Reduction. Seiler [32] proves the strict single-reduction bound $|t| < Q$ in their Lemma 1, where t denotes the Montgomery-reduction output. We extend this compositionally: the challenge is not the single-reduction bound, which is tight and analytically simple, but the *pipeline* bound through 8 GS layers with unreduced addition branches. The erroneous analytical bound $A + dQ$, which ignores addition-branch doubling, illustrates why compositional reasoning benefits from SMT certification.

A substantial body of work optimizes NTT implementations on various platforms [4, 9, 17, 7]. These works modify *how* NTT arithmetic is performed but inherit reduction *sites* from the reference without formal justification. Our audit methodology provides that justification: implementers can verify that any proposed reduction schedule satisfies $\rho < 1$ at every remaining site.

Abstract Interpretation for Numerical Programs. Navas et al. [29] apply abstract interpretation [10] to verify that one reduction can be removed from Longa and

Naehrig's NTT variant [22]—a single-site result for a non-standard NTT. Generic abstract-interpretation tools such as Astrée [11] verify absence of runtime errors via interval and polyhedral domains. The fundamental challenge for these tools on NTT pipelines is *interval widening*: the Montgomery reduction's true output range $(-Q, Q)$ is a domain-specific invariant that standard widening operators cannot recover from the bit-level computation, which involves intentional `int32` truncation. Once this tight bound is lost, the accumulation loop's output range widens to the full `int32` range, and subsequent overflow alarms become indistinguishable from false positives on the intentional truncation. Additionally, abstract interpretation cannot express the ℓ_1-norm constraint ($\|c\|_1 = \tau$) that enables tight sparse-challenge bounds: standard interval analysis yields $n \cdot \eta = 512$ for the product $c \cdot s_1$ versus our tight bound $\tau\eta = 78$.

9 Conclusion

Developers are removing modular reductions from production ML-DSA code to optimize performance and memory, but not every omitted reduction is equally safe. We showed that reduction placement is an inter-procedural correctness condition: a call that appears unnecessary at one program point may be load-bearing for a later NTT stage. Using SMT-backed bound propagation, we classified the audited reference-style C topology into 4 sites that must remain and 6 sites that are safe to remove.

This classification explains both safe and load-bearing reduction omissions in production code. Auditing 8 ML-DSA libraries uncovered a reduction-placement defect in wolfSSL's memory-optimized `WOLFSSL_DILITHIUM_SMALL` mode, where post-matrix-multiplication reductions are omitted in key generation, signing, and verification. The omission causes signed 32-bit integer overflow, $\mathbb{Z}_q$-incorrect InvNTT output, deterministic signing failure, and FIPS 204 non-compliance, while surviving the implementation's existing KAT tests. We responsibly disclosed the issue to the affected maintainer, and a fix for the affected code path was available before public release of this manuscript. The classification and impact analysis are supported independently by range analysis, replayable SMT certificates, and reproduced correctness harnesses.

The main technical reason the classification is sharp is exact integer recovery for sparse-challenge products. The Montgomery-form NTT pipeline does not merely return residues modulo Q in these cases: our certificates show that $T_Q \geq 4{,}178{,}891 \gg \tau\eta \leq 196$, tightening the post-InvNTT output range from $(-Q, Q)$ to $[-\tau\eta, \tau\eta]$. This separates safe omissions on sparse-challenge paths from load-bearing dense-product sites.

More broadly, our results suggest that PQC implementation optimization should treat reduction removal as a proof obligation, not as a local code cleanup. KATs are useful regression tests, but they cannot justify a reduction schedule over all valid inputs. A small set of replayable bound certificates can provide that missing criterion.

References

1. Almeida, J.B., Arranz Olmos, S., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Léchenet, J.C., Low, C., Oliveira, T., Pacheco, H., Quaresma, M., Schwabe, P., Strub, P.Y.: Formally Verifying Kyber: Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt. In: Advances in Cryptology – CRYPTO 2024. LNCS, vol. 14921, pp. 384–421. Springer (2024)
2. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-Assurance and High-Speed Cryptography. In: ACM CCS. pp. 1807–1823 (2017). https://doi.org/10.1145/3133956.3134078
3. Amazon Web Services: AWS-LC: AWS Libcrypto. https://github.com/aws/aws-lc (2024)
4. Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 221–244 (2022)
5. Bernstein, D.J., Bhargavan, K., Bhasin, S., Chattopadhyay, A., Chia, T.K., Kannwischer, M.J., Kiefer, F., Paiva, T.B., Ravi, P., Tamvada, G.: KyberSlash: Exploiting Secret-Dependent Division Timings in Kyber Implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(2), 209–234 (2025)
6. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367 (2018)
7. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Progress in Cryptology – AFRICACRYPT 2019. Lecture Notes in Computer Science, vol. 11627, pp. 209–228. Springer (2019)
8. Chiu, C.M., Liu, J., Tsai, M.H., Shi, X., Wang, B.Y., Yang, B.Y.: Algebraic Linear Analysis for Number Theoretic Transform in Lattice-Based Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(3), 668–692 (2025)
9. Chung, C.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.J., Yang, B.Y.: NTT Multiplication for NTT-Unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. TCHES **2021**(2), 159–188 (2021)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252 (1977)
11. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée Analyzer. In: ESOP. LNCS, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3
12. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. TCHES **2018**(1), 238–268 (2018)
13. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple High-Level Code for Cryptographic Arithmetic – With Proofs, Without Compromises. In: IEEE Symposium on Security and Privacy (S&P). pp. 1202–1219 (2019)

14. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. NIST PQC Round 3 Submission, https://falcon-sign.info/ (2020)
15. Fu, Y.F., Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Signed Cryptographic Program Verification with Typed CryptoLine. In: ACM CCS. pp. 1591–1606 (2019). https://doi.org/10.1145/3319535.3354199
16. Google: BoringSSL. https://github.com/google/boringssl (2024)
17. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, D.: Compact dilithium implementations on Cortex-M3 and Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) **2021**(1), 1–24 (2021)
18. Guo, Q., Johansson, T., Yang, J.: A Novel CCA Attack Using Decryption Errors Against LAC. In: Advances in Cryptology – ASIACRYPT 2019. LNCS, vol. 11921, pp. 82–111. Springer (2019)
19. Harvey, D.: Faster Arithmetic for Number-Theoretic Transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002
20. Hwang, V., Liu, J., Seiler, G., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(4), 718–750 (2022)
21. Knichel, D., Sasdrich, P., Moradi, A.: SILVER – Statistical Independence and Leakage Verification. In: Advances in Cryptology – ASIACRYPT 2020. LNCS, vol. 12491, pp. 787–816. Springer (2020)
22. Longa, P., Naehrig, M.: Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security (CANS 2016). Lecture Notes in Computer Science, vol. 10052, pp. 124–139. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_8
23. Microsoft: SymCrypt: Core Cryptographic Function Library. https://github.com/microsoft/SymCrypt (2024)
24. mldsa-native Contributors: mldsa-native: Secure, Fast, and Portable C90 Implementation of ML-DSA. https://github.com/pq-code-package/mldsa-native (2024)
25. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985)
26. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. pp. 337–340 (2008)
27. National Institute of Standards and Technology: FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM). https://csrc.nist.gov/pubs/fips/203/final (2024)
28. National Institute of Standards and Technology: FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA). https://csrc.nist.gov/pubs/fips/204/final (2024)
29. Navas, J.A., Dutertre, B., Mason, I.A.: Verification of an Optimized NTT Algorithm. In: Verified Software. Theories, Tools, and Experiments (VSTTE 2020). LNCS, vol. 12549, pp. 144–160. Springer (2020)
30. Open Quantum Safe Project: liboqs: Open Quantum Safe Library. https://github.com/open-quantum-safe/liboqs (2024)
31. PQClean Contributors: PQClean: Clean Implementations of Post-Quantum Cryptography. https://github.com/PQClean/PQClean (2024)

32. Seiler, G.: Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography. Cryptology ePrint Archive, Paper 2018/039 (2018), https://eprint.iacr.org/2018/039
33. The Go Authors: The Go Programming Language. https://github.com/golang/go (2024)
34. wolfSSL Inc.: wolfSSL Embedded SSL/TLS Library. https://github.com/wolfSSL/wolfssl (2024)