Verification Theatre: False Assurance in Formally Verified Cryptographic Libraries

Nadim Kobeissi

Symbolic Software, Paris, France
nadim@symbolic.software

Abstract. Every formally verified system embeds a *verification boundary*: the interface between code with machine-checked proofs and code that is trusted without them. We study what happens when this boundary is not communicated clearly. Through a case study of CE Labs's libcrux and hpke-rs cryptographic libraries, we present thirteen vulnerabilities that escaped formal verification. Nine reside in unverified code, including a cross-backend endianness bug that caused real decryption failures in Signal's post-quantum ratchet, a missing mandatory X25519 validation, nonce reuse via integer overflow, and two FIPS 204 specification violations in the ML-DSA verifier. Four reside in formally verified specification and proof code: in ML-KEM, a wrong decompression constant, a missing inverse NTT, and a false serialization proof; in ML-DSA, a wrong multiplication specification that renders axiomatized AVX2 proofs unsound. From these findings, we develop a taxonomy of five verification boundary failure types, a lightweight auditing methodology for detecting them, and a comparative analysis with AWS's verified libcrypto. The same failure types arise in both projects, but their management—through systematic documentation, proof execution in CI, and clear scope communication—varies significantly. We call the gap between verification claims and verification reality *"verification theatre"*, and propose concrete practices for closing it.

Keywords: high-assurance cryptography · formal verification · verification boundaries · cryptographic engineering · post-quantum cryptography

1 Introduction

The formal verification of cryptographic implementations has produced some of the most impressive results in applied computer science. Projects such as HACL⋆ [23], Jasmin [1], Vale [9], and Fiat-Crypto [19] have demonstrated that it is possible to produce machine-checked proofs that cryptographic code is correct with respect to its specification, free of memory safety violations, and resistant to timing side channels. These are genuine achievements that meaningfully advance the security of deployed systems.

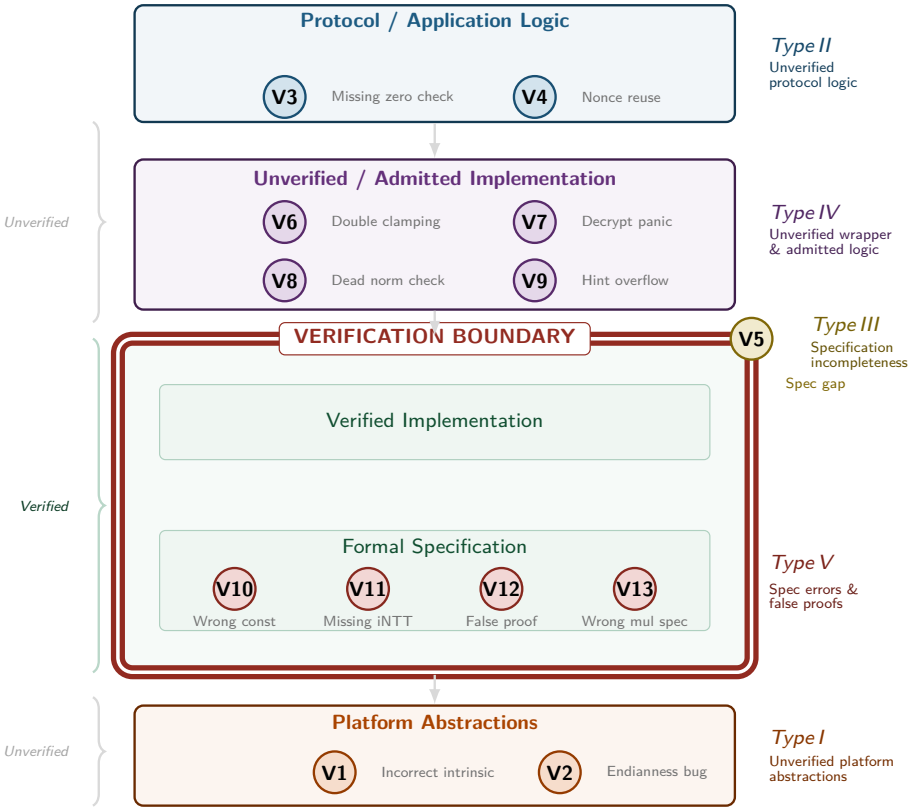

Fig. 1. Taxonomy of verification boundary failures. The double-lined box marks the **verification boundary**: code inside has machine-checked proofs; code outside is trusted without them. Type I–IV failures arise at or beyond the boundary. Type V failures are the most insidious—they corrupt the formal foundation *within* the boundary, making proofs vacuous. Each circle marks a concrete vulnerability at its failure location.

However, every formally verified system embeds a *verification boundary*: the interface between code for which machine-checked proofs exist and code that is trusted without such proofs. Compilers, operating systems, hardware, platform intrinsics, and "glue code" all lie outside the verified perimeter. The existence of this boundary is well understood—CompCert [32] explicitly documents which compiler passes are verified, seL4 [26] carefully delineates assumptions about hardware behavior—but it is not always communicated clearly.

When a cryptographic library is marketed as "formally verified" without qualification, users reasonably interpret this as a comprehensive guarantee. CE Labs, the company behind the libcrux [13] and hpke-rs [11] cryptographic libraries, describes itself as providing "services and software for high assurance software in order to establish trust into your critical systems." The libcrux repository is

subtitled "the formally verified crypto library," and its documentation claims that "every implementation in libcrux is formally verified." These libraries are deployed widely. Libcrux provides the ML-KEM implementation used internally at Google [21]. The hpke-rs library is a dependency of Signal's `signal-crypto` crate [47] and of OpenMLS [36].

We present thirteen vulnerabilities across libcrux, hpke-rs, and the libcrux-psq protocol library. Nine reside in unverified code. Two were independently discovered: a platform-dependent cryptographic output failure reported by Valsorda [50], and a cross-backend endianness inconsistency in the ML-KEM incremental API reported by GuuJiang [22] that caused real decryption failures in Signal's post-quantum ratchet [46]. The remaining seven, identified by us, include a missing mandatory X25519 validation, nonce reuse via integer overflow, ECDSA signature malleability, an Ed25519 key generation defect, a denial-of-service via an unhandled AES-GCM decryption error, and two FIPS 204 [34] specification violations in the ML-DSA verifier—a norm check rendered dead code by a wrong constant and a missing final bound check in hint deserialization. Four reside in formally verified specification and proof code: in ML-KEM, a wrong decompression constant, a missing inverse NTT, and a false serialization proof; in ML-DSA, a wrong multiplication specification in the AVX2 intrinsics axioms that makes verified proofs unsound. The specification-level bugs survive because the build system admits proofs by default, a proof tactic silently accepts all claims in lax mode, axioms are never proved, and full verification is never triggered in any automated pipeline.

From these thirteen findings, we develop three contributions:

1. A **taxonomy of five verification boundary failure types** that captures the structural reasons why bugs escape formal verification in practice (Section 3).
2. A **lightweight auditing methodology** for detecting verification boundary failures, validated by its application to the libcrux codebase (Section 4).
3. A **comparative analysis** of verification boundary practices across two verified cryptographic libraries—CE Labs's libcrux and AWS's verified libcrypto [2]—showing that the failure types are general but that their management varies significantly (Section 8).

We define *verification theatre* as the practice of deploying formal methods in a manner that:

1. establishes machine-checked proofs for a subset of a system's code and properties, where some proofs may themselves be unchecked, vacuous, or built on false axioms,
2. communicates these proofs to users and stakeholders using language that implies comprehensive coverage, and
3. fails to adequately disclose the verification boundary, the unverified components, the unchecked proofs, or the properties not covered.

2 Background

2.1 The Formal Verification Pipeline

Modern cryptographic verification typically proceeds through a multi-stage pipeline. We describe the methodology used by CE Labs for libcrux, as documented in their `Architecture.md` [13] and associated tooling descriptions.

Stage 1: Specification. A high-level specification of the cryptographic algorithm is written in a specification language. In the case of libcrux, specifications are written in hacspec, a subset of Rust designed for formal reasoning, or directly in F* [48].

Stage 2: Implementation. An optimized implementation is written in Rust (for code verified via hax [6]) or in Low* (for code in the HACL* tradition [23,39]). The implementation may include multiple backends targeting different instruction set architectures (portable, AVX2, NEON/ARM64).

Stage 3: Extraction and Proof. The hax toolchain extracts the Rust implementation to F*, where automated and semi-automated provers verify properties including:

- **Functional correctness**: the implementation matches the specification.
- **Runtime safety**: the code is free of panics, array bounds violations, and integer overflow.
- **Secret independence**: the execution trace does not depend on secret inputs (constant-time behavior at the source level).

For HACL*-derived code, the Low* implementation is extracted to C via KaRaMeL [39] and wrapped in a safe Rust API.

Stage 4: Compilation and Deployment. The verified source code is compiled by an unverified compiler (typically LLVM via `rustc`) targeting the deployment platform. Platform-specific intrinsics may be substituted at compile time based on the target architecture's feature set.

Each stage introduces a trust boundary. The formal proofs established in Stage 3 hold only for the extracted model; they do not automatically transfer through compilation, platform-specific code selection, or interaction with unverified library components.

2.2 Verification Status of libcrux

CE Labs's own documentation reveals that the verification status of libcrux is significantly more nuanced than marketing materials suggest. The ML-KEM verification status document [16] enumerates three levels of verification for each module: *lax checking* (the F* type checker accepts the extracted code), *runtime safety* (freedom from panics and precondition violations), and *correctness* (functional equivalence to the specification).

Table 1. ML-KEM verification status as documented by CE Labs [16]. "✓" indicates completed proofs; "–" indicates "needs proofs." The NEON backend has no completed runtime safety or correctness proofs for any module.

Backend	Module	Lax	Safety	Correct.
Gen.	const_time_ops	✓	✓	✓
	hash_functions	✓	✓	✓
	ind_cpa	✓	✓	✓
	ind_cca	✓	✓	✓
	polynomial	✓	✓	✓
	ntt/invert_ntt	✓	✓	–
	matrix/serial./sampling	✓	–	–
Port.	arith/ntt/serial./compr	✓	✓	✓
	sampling	✓	✓	–
AVX2	arith/ntt/serialize	✓	✓	✓
	compr/sampling	✓	✓	–
NEON	arith	✓	–	–
	ntt	✓	–	–
	compress	✓	–	–
	serialize	✓	–	–
	sampling	✓	–	–

Table 1 reproduces the verification status for the ML-KEM implementation at the time of our audit. The table reveals that the NEON (ARM64) backend—the very backend implicated in the November 2025 platform-dependent output failure—has "needs proofs" for *all* modules across both runtime safety and correctness. The portable and AVX2 backends have substantially better coverage, but even the generic modules that apply across all backends have incomplete correctness proofs for NTT, matrix operations, serialization, and sampling.

3 A Taxonomy of Verification Boundary Failures

From our analysis of the libcrux ecosystem, we identify five categories of verification boundary failure that arise in formally verified cryptographic libraries. Each category is exemplified by concrete vulnerabilities described in detail in Sections 5 and 6; we present the taxonomy first to provide a framework for reading the individual findings.

Type I: Unverified Platform Abstractions. The verified code depends on platform-specific primitives (intrinsics, assembly routines, hardware instructions) that are treated as abstract operations in the verification model. The formal proofs assume these operations implement their specified semantics, but the actual implementations may contain bugs. *Examples*: V1 (Section 5.1), where an ARM64 SHA-3 intrinsic fallback was marked `#[hax_lib::opaque]`, causing the verifier

to accept it as an axiom whose implementation was incorrect on certain platforms; V2 (Section 5.2), where the portable ML-KEM backend used inconsistent endianness with SIMD backends in its `from_bytes/to_bytes` serialization, causing cross-backend state incompatibility [22].

Type II: Unverified Protocol Logic. The verified code implements cryptographic primitives correctly, but the protocol-level logic that *uses* these primitives contains defects. *Examples*: V3 (Section 5.3), a missing X25519 zero check required by RFC 9180 [4]; V4 (Section 5.4), a nonce reuse vulnerability caused by a `u32` sequence counter that silently wraps in release builds.

Type III: Specification Incompleteness. The code is verified correct with respect to its formal specification, but the specification does not capture all security-relevant properties required by the deployment context. *Example*: V5 (Section 5.5), where the ECDSA implementation is provably correct per its F⋆ specification but lacks low-S normalization required by practical deployments.

Type IV: Unverified Wrapper Logic. The verified core is correct, but the unverified API layer or algorithm-level code that composes verified building blocks introduces defects. *Examples*: V6 (Section 5.6), where an Ed25519 key generation wrapper applies clamping before hashing; V7 (Section 5.7), where an AES-GCM decryption wrapper calls `.unwrap()` instead of propagating errors; V8 (Section 5.8), where the ML-DSA verifier uses a doubled norm bound that renders a FIPS 204 security check dead code; V9 (Section 5.9), where ML-DSA hint deserialization checks the previous iteration's count instead of the current one, skipping a mandatory bound validation.

Type V: Specification Errors and False Proofs. The formal specification itself is wrong, or the proofs contain false claims that are silently accepted by the build system. *Examples*: V10 (Section 6.1), a decompression specification that disagrees with FIPS 203 [35]; V11 (Section 6.2), a missing inverse NTT in the encryption spec; V12 (Section 6.3), a false serialization proof claiming 12-bit values fit in 1 bit; V13 (Section 6.4), an AVX2 multiplication specification in ML-DSA that computes x^2 instead of $x \cdot y$.

Type V failures are the most insidious because they undermine the foundation on which all other verification rests. Even complete proofs only establish correctness with respect to the wrong definition (V10, V11) or are themselves false and silently accepted (V12).

4 Auditing Methodology

We describe a lightweight methodology for auditing verification boundaries in formally verified cryptographic libraries. Our approach requires no specialized tooling beyond a text editor, a terminal, and familiarity with the verification framework employed. We applied this methodology to libcrux and its associated libraries in February 2026 and confirmed its effectiveness through the thirteen vulnerabilities described in Sections 5 and 6.

4.1 Step 1: Map the Verification Boundary

Identify all code paths that are excluded from formal verification. In practice, this requires examining:

- **Extraction annotations.** Search for framework-specific markers that exclude code from extraction (e.g., `#[hax_lib::opaque]` in hax, `crucible_llvm_unsafe_assume_spec` in SAW). Each marker represents an axiom: the verifier accepts the function's specification without proof.
- **Build system configuration.** Examine Makefiles, CI workflows, and build scripts for module admission flags (`ADMIT_MODULES`, `--admit_smt_queries true`) and variables that control whether proofs are checked (`VERIFY_SLOW_MODULES`).
- **API wrappers and glue code.** Identify all functions that call verified primitives but are not themselves extracted or verified. These are the Type IV attack surface.
- **Platform-specific code paths.** Check whether all target architectures have completed proofs. Conditional compilation (`#[cfg(...)]`) often selects between verified and unverified implementations.

4.2 Step 2: Audit Specifications Against Standards

For each verified primitive, compare the formal specification against the relevant standard (FIPS, RFC, or reference implementation). Focus on:

- **Constants.** Verify that rounding constants, modular reduction parameters, and domain separation tags match the standard. Copy-paste errors between similar functions are common (cf. V10).
- **Domain transformations.** Check that NTT/inverse NTT, Montgomery multiplication, and other domain changes are applied at the correct points. Omitting a transformation may produce a type-correct but mathematically wrong expression (cf. V11).
- **Bound claims.** Verify that lemmas asserting output bounds claim the correct bit width. A lemma claiming a k-bit output is bounded by 1 instead of k is trivially false but may survive in an admitted module (cf. V12).

4.3 Step 3: Verify That Proofs Are Checked

Confirm that the build system actually dispatches proof obligations to the solver:

- **Search for admission patterns.** Grep for `admit`, `tadmit`, `sorry`, `Admitted`, `unsafe_assume_spec`, or equivalent constructs in the proof code and build configuration.
- **Check CI pipelines.** Determine whether full verification runs on every commit, only on scheduled builds, or never. If proofs only run on schedule, check whether the schedule has ever triggered successfully.
- **Test tactic behavior in lax mode.** Examine custom tactics for lax-mode escape hatches (e.g., `if lax_on () then iterAll tadmit`). A tactic that admits all goals in lax mode renders any proof that uses it vacuous when the module is admitted.

4.4 Step 4: Cross-Reference Documentation

Compare the verification status claimed in documentation against the build system's actual behavior. Discrepancies between a status document claiming "correctness: yes" and a Makefile listing the same module in `ADMIT_MODULES` indicate that documentation is manually maintained and has drifted from reality.

5 Boundary Vulnerabilities in Unverified and Admitted Code

We describe nine vulnerabilities at or beyond the verification boundary, including two in modules that are claimed to be verified but whose proofs are admitted without checking.

5.1 V1: Platform-Dependent Cryptographic Output Failure

In November 2025, Filippo Valsorda independently reported [50] that libcrux-ml-dsa v0.0.3 produced different public keys and signatures on different platforms given identical deterministic inputs. The bug resided in the `_vxarq_u64` intrinsic wrapper (Listing 1.1), which implements the XAR operation used in SHA-3's Keccak-f permutation. The fallback passed incorrect arguments to the shift operations, corrupting SHA-3 digests on ARM64 platforms without hardware SHA-3 support. This is a **Type I** failure: the intrinsic was marked `#[hax_lib::opaque]`, and the entire NEON backend had no completed proofs for runtime safety or correctness (Table 1).

5.2 V2: Cross-Backend State Incompatibility in Incremental API

On December 27, 2025, GuuJiang independently reported [22] that the ML-KEM incremental API (`encapsulate1`/`encapsulate2`) produced incompatible state when the first call executed on one backend (e.g., portable) and the second on a SIMD backend (AVX2 or NEON), or vice versa.

The root cause was an endianness inconsistency in the portable backend's `from_bytes`/`to_bytes` serialization routines for the intermediate encapsulation state. The portable implementation serialized field elements in a different byte order than the AVX2 and NEON backends. When the incremental API serialized state after `encapsulate1` on one backend and deserialized it on another for `encapsulate2`, the reconstructed polynomial coefficients were incorrect, causing decapsulation to fail.

This bug had direct downstream impact: it caused real decryption failures in Signal's post-quantum ratchet protocol [46], affecting users on macOS, Windows, iOS, and Android—platforms where the libsignal library might select different backends across process restarts or device migrations. Signal versions v0.76.3 through v0.86.9 were affected; the fix was included in libsignal v0.86.12 (January 15, 2026).

CE Labs merged the fix on January 6, 2026 [42], without issuing a security advisory. This is a **Type I** failure: the incremental API code resides in the "unextracted" category (3,198 LOC that are never extracted to F*), and the cross-backend serialization consistency property is not captured by any formal specification. Like V1, it demonstrates that platform abstraction mismatches can produce silently wrong cryptographic output.

5.3 V3: Missing X25519 All-Zero Validation

RFC 9180, Section 7.1.4 [4], mandates that HPKE implementations "MUST check whether the Diffie-Hellman shared secret is the all-zero value and abort if so." The hpke-rs library's Diffie-Hellman implementation returns the raw X25519 output without any zero check (Listing 1.2). An attacker who supplies a low-order public key can force the shared secret to zero, making the entire HPKE key schedule deterministic. This is a **Type II** failure: the X25519 primitive is correct, but the protocol wrapper omits a mandatory validation [30].

5.4 V4: Nonce Reuse via Sequence Number Overflow

The hpke-rs library stores the HPKE sequence number as a `u32`, with a maximum value of $2^{32} - 1$. RFC 9180 [4] requires checking against $2^{96} - 1$. The overflow guard compares a `u32` (cast to `u128`) against $2^{96} - 1$, a comparison that can never be true—the check is dead code. In release builds, the counter silently wraps to zero, reusing nonces and enabling plaintext recovery and universal forgeries [8]. This is a **Type II** failure [29].

5.5 V5: ECDSA Specification Gap (Signature Malleability)

The libcrux ECDSA P-256 implementation validates that r and s are nonzero and within the curve order but does not check $s \leq n/2$. Low-S normalization, required by major deployed systems including Bitcoin Core [51] and Ethereum, ensures signature uniqueness. The implementation is provably correct per its F* specification, which faithfully models ECDSA as defined in FIPS 186-5 [33]. However, the specification does not capture the application-level uniqueness constraint that deployed systems require. This is a **Type III** failure: the formal specification is correct with respect to the standard but incomplete with respect to deployment requirements [28].

5.6 V6: Ed25519 Double Clamping

RFC 8032 [25] specifies Ed25519 key generation as: hash the seed with SHA-512, then clamp the result. The libcrux `key_gen` wrapper applies clamping to the raw seed *before* passing it to the verified `secret_to_public` function, which hashes and clamps again. This reduces effective seed entropy by 5 bits. This is a **Type IV** failure: the verified core is correct, but the unverified wrapper constrains its input unnecessarily [31].

Table 2. ML-DSA verifier norm check bounds. The buggy bound $2\gamma_1 - \beta$ exceeds the maximum possible $\|\mathbf{z}\|_\infty$ for every parameter set, making the check vacuously true.

Param. Set	γ_1	β	Max $\|\mathbf{z}\|_\infty$	Buggy bound
ML-DSA-44	2^{17}	78	131 072	262 066
ML-DSA-65	2^{19}	196	524 288	1 048 380
ML-DSA-87	2^{19}	120	524 288	1 048 456

5.7 V7: Denial of Service via AES-GCM Decryption Panic

The libcrux-psq crate [12] implements a post-quantum pre-shared-key protocol. In the `decrypt_out` method, the AES-GCM 128 decryption path calls `.unwrap()` on the decryption result instead of propagating the error (Listing 1.3). A single malformed ciphertext crashes the process. Beyond the denial-of-service impact, this defect renders the PSQ implementation not IND-CCA secure: the IND-CCA security game requires a decryption oracle that returns $\perp$ on invalid ciphertexts, but a panicking implementation terminates execution instead, making the decryption oracle unavailable after the first invalid query. This is a **Type IV** failure: the verified primitive correctly returns an error on authentication failure, but the unverified wrapper discards it [27].

5.8 V8: ML-DSA Verifier Norm Check Is Dead Code

The ML-DSA verifier checks whether the signer response vector $\mathbf{z}$ has an infinity norm below a bound that prevents secret key leakage. FIPS 204 [34], Algorithm 8 (ML-DSA.Verify_internal), step 13 specifies:

$$\text{return } \|\mathbf{z}\|_\infty < \gamma_1 - \beta \text{ and } \tilde{c} = \tilde{c}'$$

The signer (Algorithm 7, step 23) uses the same bound $\gamma_1 - \beta$. The libcrux verifier uses a doubled constant (Listing 1.4): the expression `2 << GAMMA1_EXPONENT` computes $2 \cdot 2^{\gamma_{1\text{exp}}} = 2\gamma_1$, not γ_1. The signer correctly uses `1 << GAMMA1_EXPONENT`. The signer response $\mathbf{z}$ is deserialized from a γ_1-encoded representation where each coefficient lies in $[-\gamma_1, \gamma_1]$, so $\|\mathbf{z}\|_\infty \leq \gamma_1$. Since $\gamma_1 < 2\gamma_1 - \beta$ for all parameter sets (Table 2), the check can never trigger—it is dead code.

Concrete impact. A conforming signer never produces $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ (the signing algorithm retries), but a malicious signer can produce signatures with $\gamma_1 - \beta \leq \|\mathbf{z}\|_\infty \leq \gamma_1$ that pass libcrux's verifier but are correctly rejected by any FIPS 204-conforming implementation. This creates interoperability hazards in post-quantum certificate chains, firmware signing, and consensus protocols where libcrux nodes accept signatures that conforming nodes reject.

Provenance. Introduced in commit `326c837a33` [41] on January 7, 2025, during a refactoring pass. The signer's correct bound was already present; the verifier's

was written incorrectly. This is a **Type IV** failure: the ML-DSA signing and verification logic resides in `ADMIT_MODULES` while the verified building blocks (SIMD arithmetic, NTT, encoding primitives) are in `VERIFIED_MODULES`.

5.9 V9: ML-DSA Hint Deserialization Missing Final Bound Check

ML-DSA signatures contain a hint vector $\mathbf{h}$ with a maximum of ω total true hints. FIPS 204 [34], Algorithm 21 (HintBitUnpack), step 4 specifies that for every row i: if $y[\omega+i] > \omega$ then return $\perp$. The libcrux implementation checks the wrong variable (Listing 1.5): the overflow condition tests `previous_true_hints_seen > max_ones_in_hint` instead of `current_true_hints_seen > max_ones_in_hint`.

For the last row ($i = k - 1$), the current count is read, hint indices are processed and applied via `set_hint`, and `previous_true_hints_seen` is updated—but the overflow check will not fire until the next iteration, which never comes. The subsequent padding check is also skipped when `previous_true_hints_seen` $\geq$ `max_ones_in_hint`, as the loop range becomes empty.

Concrete impact. A crafted signature can set the last row's cumulative count above ω. The inner loop then reads trailer bytes (the cumulative count bytes for each row) as hint indices, setting arbitrary positions in the last polynomial's hint vector. This produces a hint vector that a conforming implementation would reject, creating cross-implementation verification disagreement and violating strong unforgeability (multiple distinct serializations decode to the same logical signature).

Provenance. The loop structure was established in commit `83ffae727d` on January 2, 2025, modified in commit `404115bc08` on January 13, 2025, and further restructured in commit `d979b001f9` [40] on November 17, 2025 as a workaround for a bug in Eurydice that prevented early returns. The Eurydice workaround's flag-and-break pattern obscured the missing bound check. This is a **Type IV** failure in an admitted module.

6 Bugs in the Verified Code

We now present four bugs in formally verified specification and proof code—three in ML-KEM and one in ML-DSA—demonstrating Type V failures. These were discovered by applying the auditing methodology of Section 4 to the libcrux codebase.

6.1 V10: Wrong Specification for ML-KEM Decompression

FIPS 203 definition. The decompression function in FIPS 203 [35] is defined as $\text{Decompress}_d(y) = \lceil q/2^d \cdot y \rfloor$, implemented as $\lfloor (q \cdot y + 2^{d-1})/2^d \rfloor$. The rounding constant 2^{d-1} is the standard round-half-up correction for integer division by 2^d.

Table 3. Counterexamples showing disagreement between the F⋆ specification (using 1664) and FIPS 203 (using 2^{d-1}). The specification produces wrong values for all $d \in \{1, 4, 5, 10, 11\}$.

d	x	**Spec (1664)**	**Correct (2^{d-1})**	**FIPS 203**
1	0	832	0	0
1	1	2496	1665	1665
4	0	104	0	0
4	1	312	208	208
10	0	1	0	0
10	1	4	3	3

Specification defect. In `Spec.MLKEM.Math.fst` [14], the `decompress_d` function uses the constant 1664 instead of 2^{d-1} (Listing 1.6). The constant 1664 equals $\lfloor q/2 \rfloor$ where $q = 3329$. This value is correct in the *companion* function `compress_d`, where integer division is by q and the rounding constant should be $\lfloor q/2 \rfloor = 1664$. The bug is a copy-paste error: the same constant was used in both functions despite the different divisors.

Counterexamples. Table 3 shows that the specification disagrees with FIPS 203 on every input for $d \in \{1, 4, 5, 10, 11\}$. In particular, decompress_d$(d, 0) \neq 0$ for all $d < 12$—decompressing zero does not yield zero. The bug affects all five parameter values of d used across the three ML-KEM parameter sets.

Implementation is correct. The Rust implementation in the portable compress module (`compress.rs`) computes the correct formula $(x \cdot q + 2^{d-1})/2^d$ using bit shifts. The implementation matches FIPS 203; the specification does not.

Why undetected. This is a specification-level bug: the mathematical reference that correctness proofs are verified against computes a different function than FIPS 203. Even if all proofs were completed and fully verified, they would prove the implementation correct against the wrong mathematical definition.

The bug survives because the IND-CPA module—whose correctness proof would need to relate the implementation to this specification—is listed in `ADMIT_MODULES` in the build configuration (Listing 1.7). When a module is admitted, all SMT queries are accepted without checking, so any inconsistency between the specification and implementation is silently ignored. Meanwhile, CE Labs's `verification_status.md` [16] claims that `ind_cpa` has completed proofs for lax checking, runtime safety, and correctness.

Provenance. Commit `c1935b9cf8` [14], dated August 12, 2024, with message "spec." The bug has been present since inception.

6.2 V11: Missing Inverse NTT in Encryption Spec

FIPS 203 requirement. Algorithm 14 of FIPS 203 [35] (K-PKE.Encrypt) specifies:
- Step 19: $\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{A}^T \circ \hat{y}) + \mathbf{e}_1$
- Step 21: $v \leftarrow \text{NTT}^{-1}(\hat{t}^T \circ \hat{y}) + e_2 + \mu$

Both require applying the inverse NTT to the NTT-domain product before adding coefficient-domain error terms.

Specification defect. In `Spec.MLKEM.fst` [14], the `ind_cpa_encrypt_unpacked` function correctly applies `vector_inv_ntt` when computing $\mathbf{u}$ but omits `poly_inv_ntt` when computing v (Listing 1.8). The function `vector_dot_product_ntt` returns an NTT-domain polynomial, while `error_2` and `mu` are coefficient-domain polynomials. The NTT representation of a polynomial is a distinct sequence of 256 field elements bearing no algebraic relationship to the coefficient representation. Adding them directly produces a result that is neither the correct NTT-domain nor coefficient-domain value—a mathematically incoherent operation.

Notably, the same specification correctly applies `poly_inv_ntt` in the *decryption* function `ind_cpa_decrypt_unpacked`, confirming that the omission in the encryption path is an oversight, not a deliberate design choice.

Implementation is correct. The Rust implementation in `matrix.rs` correctly applies `invert_ntt_montgomery` before adding the error and message terms.

The F* postcondition for this function, however, references the wrong specification—it asserts equivalence to the buggy spec without the inverse NTT.

Why undetected. The function `compute_ring_element_v` is marked `verification_status(lax)`—its postcondition is only lax-checked, not proven. Furthermore, the IND-CPA module is in `ADMIT_MODULES` (Listing 1.7), so even the lax-checked postcondition is never verified against the specification.

Provenance. Same commit as V10: `c1935b9cf8` [14], August 12, 2024.

6.3 V12: False Proof in ML-KEM Serialization

The false lemma. In `libcrux-ml-kem/src/vector/portable/serialize.rs` [15], the lemma `deserialize_12_bit_vec_lemma_bounded` claims that 12-bit deserialized values are bounded by 1 (Listing 1.9). The lemma asserts that a function producing 12-bit values only ever produces values that fit in 1 bit. This is a false mathematical statement: the 12-bit version was copy-pasted from the 1-bit version with the bound left unchanged.

Three layers of silent failure. The false lemma survives due to three interacting mechanisms:

1. **Slow modules are admitted by default.** The portable serialize module is listed in `SLOW_MODULES` in the build configuration. The base Makefile sets `VERIFY_SLOW_MODULES ?= no`, causing slow modules to be added to `ADMIT_MODULES` and verified with `--admit_smt_queries true`, which accepts all SMT queries without checking.
2. **The proof tactic has a lax-mode escape hatch.** The tactic `prove_bit_vector_equality'` checks whether it is running in lax mode; if so, it calls `tadmit` on all goals unconditionally (Listing 1.10).
3. **Full verification is never triggered.** The CI workflow runs lax checking on every PR (`./hax.py prove --admit`) but only runs full verification on scheduled or manual builds. Even on full builds, `VERIFY_SLOW_MODULES=yes` is never set in any CI configuration, Makefile target, or script. The false proof has never been checked by any automated system.

Documentation contradiction. Three sources within the same codebase give three different answers about the verification status of this code:

- `verification_status.md` [16]: claims the portable serialize module is fully verified (lax checking: yes, runtime safety: yes, correctness: yes).
- The file header comment: `//! Verification status: Lax`.
- The actual build configuration: admitted by default; proofs never checked.

Provenance. Commit `5f6e3c2083` [15], dated March 25, 2025, with message "fix(ml-kem): deserialize_10/12: remove `admit`." The `deserialize_10` version was added correctly with bound 10. The `deserialize_12` version was copy-pasted from `deserialize_1` with the bound left as 1. A commit titled "remove admit" introduced a false proof.

6.4 V13: Wrong Multiplication Specification in ML-DSA AVX2 Proofs

The wrong definition. In `Spec.Intrinsics.fsti`, the function `i16_mul_32extended` specifies the semantics of a 16-to-32-bit widening multiplication used in AVX2 intrinsic proofs (Listing 1.11). The function takes two parameters `x` and `y` but the body computes `(cast x) *! (cast x)`—squaring `x` instead of multiplying `x` by `y`. The parameter `y` is completely ignored.

Impact chain. The function is marked `@@ "opaque_to_smt"`, hiding the buggy definition from Z3. All reasoning about its behavior goes through approximately 40 `val` declarations (axioms) in the same file, for which *no corresponding F⋆ implementation file exists*—every lemma is an unproven axiom. These axioms claim correct $x \cdot y$ behavior that contradicts the actual x^2 definition. For example, `mm256_madd_epi16_lemma` specifies AVX2's `VPMADDWD` as $a_{2k} \cdot b_{2k} + a_{2k+1} \cdot b_{2k+1}$; the actual definition computes $a_{2k}^2 + a_{2k+1}^2$, ignoring b entirely.

Table 4. Threat model and exploitability for each vulnerability. "Latent" indicates specification-level bugs whose deployed impact is contingent on future changes that align the implementation with the (wrong) specification.

ID	Type	Adversary Model	Exploitability	Impact
V1	I	None (platform-dependent)	Deterministic on affected ARM64	Incorrect ML-DSA output
V2	I	None (platform-dependent)	Cross-backend state exchange	Decryption failure (Signal PQ ratchet)
V3	II	Active: chosen public key	Single key exchange	Deterministic shared secret
V4	II	Any sender; long session	After 2^{32} encryptions	Nonce reuse; plaintext recovery
V5	III	Passive: signature observer	Any ECDSA signature	Signature malleability
V6	IV	None (systemic)	All Ed25519 key generations	5-bit entropy reduction
V7	IV	Active: chosen ciphertext	Single malformed ciphertext	Process crash (DoS)
V8	IV	Active: crafted signature	Any ML-DSA verification	Verifier accepts non-conforming sigs
V9	IV	Active: crafted signature	Any ML-DSA verification	Hint overflow; strong unforg. violation
V10	V	Latent	Spec adopted as impl. reference	Wrong decompression values
V11	V	Latent	Spec adopted as impl. reference	Missing inverse NTT
V12	V	Latent	Spec adopted as impl. reference	False bound proof accepted
V13	V	Latent	Spec adopted as impl. reference	Unsound AVX2 encoding proofs

The ML-DSA extraction Makefile lists modules including `Simd.Avx2.Encoding.Error.fst` and `Simd.Avx2.Encoding.Commitment.fst` in `VERIFIED_MODULES`, establishing the intent to verify them against these axioms. However, the extraction directory is currently empty—no `.fst` files have been extracted from Rust code. The wrong axioms are a foundation for future proofs that do not yet exist, a pre-verification infrastructure bug.

Structural comparison with V10–V12. V10 and V11 are specification errors in admitted modules. V12 is a false lemma silently accepted by a lax-mode tactic. V13 introduces a new failure mode: the definition is wrong, the axioms are false, and any future modules verified against these axioms would inherit the unsoundness—even with all proofs fully dispatched to Z3. Every ML-DSA proof for the AVX2 backend—the code path that executes on every modern x86-64 server and desktop—is built on approximately forty axioms that are provably false, and no amount of future verification effort can remedy this without first discovering the wrong definition. Any deployment that relied on libcrux's "formally verified" designation to justify using ML-DSA—government procurement, financial infrastructure, firmware signing—did so on a verification foundation that is mathematically unsound.

Provenance. The copy-paste error (duplicating `cast x` instead of writing `cast y`) follows the same pattern as V10 (duplicating 1664 instead of `pow2 (d-1)`) and V12 (duplicating bound 1 instead of 12).

Threat model summary. Table 4 summarizes the adversary model and exploitability conditions for all thirteen vulnerabilities. V1–V7 and V8–V9 are exploitable in current deployments under the stated adversary models; V10–V13 are latent risks whose impact is contingent on future implementation changes that align code with the wrong specification or on future proofs built on false axioms.

Table 5. Lines of Rust source code in libcrux-ml-kem and libcrux-intrinsics, classified by verification status. "Verified" means the corresponding F* module has its SMT queries dispatched to Z3; "Admitted" means the module is in `ADMIT_MODULES` or `SLOW_MODULES` and passes `--admit_smt_queries true`; "Opaque" means the code is marked `#[hax_lib::opaque]` and treated as an axiom; "Unextracted" means the code is never extracted to F*.

Component	Verified	Admitted	Opaque	Unextracted	Total
ML-KEM generic modules	8,986	1,464	—	3,198	13,648
Portable backend	1,431	748	—	—	2,179
AVX2 backend	1,732	—	—	—	1,732
NEON backend	—	792	—	—	792
libcrux-intrinsics	—	—	2,457	—	2,457
Total	**12,149**	**3,004**	**2,457**	**3,198**	**20,808**
Fraction of total	58.4%	14.4%	11.8%	15.4%	100%

7 Quantitative Boundary Analysis

The vulnerabilities in Sections 5 and 6 show that bugs cluster at the verification boundary. To understand how much code is exposed, we quantify the boundary across the libcrux ecosystem. We measure three dimensions: the fraction of code that is actually verified, the fraction of proof obligations that are dispatched to the solver, and the downstream exposure of unverified code paths.

Code coverage. We partitioned the libcrux-ml-kem crate and its immediate dependency libcrux-intrinsics into four categories by examining extraction annotations, build configuration, and module admission lists. For each Rust source file, we determined whether a corresponding F* module exists in the extraction output, and if so, whether that module is dispatched to the SMT solver or admitted. Table 5 reports the results.

Only 58.4% of the Rust code that constitutes an ML-KEM deployment has its proofs actually checked by the SMT solver. The remaining 41.6% is either admitted with `--admit_smt_queries true` (14.4%), axiomatized via `#[hax_lib::opaque]` (11.8%), or never extracted to F* at all (15.4%). The NEON backend—which serves all ARM64 deployments including iOS and recent Android devices—is entirely admitted: all 792 lines have their proofs accepted without checking. The `libcrux-intrinsics` crate, which provides the platform abstraction layer implicated in V1, is entirely opaque to the verifier: its 2,457 lines of SIMD intrinsic wrappers are treated as axioms. Within the generic modules, the 1,464 admitted lines include `ind_cpa.rs` (1,185 LOC)—the IND-CPA encryption core whose correctness proof would need to relate the implementation to the specifications containing V10 and V11—and `sampling.rs` (279 LOC). The 3,198 unextracted lines include the incremental API—the very code implicated in V2's cross-backend endianness failure—multiplexing logic, and the top-level `mlkem.rs` public interface.

Table 6. Proof obligation status for extracted F* modules in libcrux-ml-kem under the default build configuration. "Checked" modules have SMT queries dispatched to Z3; "Admitted" modules pass `--admit_smt_queries true`.

Category	Modules	Fraction
Fully checked (SMT dispatched)	127	92.0%
Admitted via `ADMIT_MODULES`	9	6.5%
Admitted via `SLOW_MODULES`	2	1.4%
Total extracted modules	**138**	100%

Table 7. Downstream exposure of each vulnerability class. Consumers are identified from public dependency declarations and published integration documentation.

Vuln.	Affected Component	Downstream Consumers
V1	NEON SHA-3 (libcrux-ml-kem)	Google (internal ML-KEM) [21]
V2	Incremental API (libcrux-ml-kem)	Signal (libsignal PQ ratchet) [46]
V3	X25519 DH (hpke-rs)	Signal (`signal-crypto`) [47], OpenMLS [36]
V4	HPKE seq. number (hpke-rs)	Signal, OpenMLS
V5	ECDSA P-256 (libcrux)	Any libcrux ECDSA consumer
V6	Ed25519 keygen (libcrux)	Any libcrux Ed25519 consumer
V7	AES-GCM decrypt (libcrux-psq)	PSQ protocol consumers
V8–V9	ML-DSA verify / hint deser.	Google (internal ML-DSA) [21]; any libcrux ML-DSA consumer
V10–V12	ML-KEM spec / proofs	All ML-KEM consumers (spec risk)
V13	ML-DSA AVX2 spec / proofs	Google (internal ML-DSA) [21]; any libcrux ML-DSA consumer

Proof obligation coverage. We examined the F* extraction Makefile to determine how many of the 138 extracted F* files (69 `.fst` implementations and 69 `.fsti` interfaces) have their SMT queries actually checked under the default build configuration (`VERIFY_SLOW_MODULES=no`). Table 6 reports the results.

At first glance, 92% of modules being checked appears reassuring. However, the 11 admitted modules include the most security-critical components: `Ind_cpa.fst` (the IND-CPA encryption core), `Sampling.fst` (rejection sampling), all seven NEON backend modules (the entire ARM64 code path), and `Vector.Portable.Serialize.fst` (the module containing the false proof V12). The modules whose correctness postconditions reference the wrong specifications (V10, V11) are also in the admitted category. As detailed in Section 6.3, no CI configuration ever sets `VERIFY_SLOW_MODULES=yes`; the slow modules have never been checked by any automated system.

Downstream exposure. The unverified and admitted code paths are not confined to libcrux's internal use. Table 7 maps each vulnerability to the downstream consumers that transitively depend on the affected code.

V2, V3, and V4 are the most exposed: V2 caused real decryption failures in Signal's post-quantum ratchet, while hpke-rs (V3, V4) is a direct dependency of Signal's `signal-crypto` crate and of OpenMLS, both of which are deployed at scale. V8 and V9 affect all consumers of libcrux's ML-DSA implementation, including Google's internal usage [21]: both are FIPS 204 specification violations in

the core verification path that cause libcrux to accept signatures that conforming implementations reject. V10–V12 do not currently affect deployed code because the Rust implementations are correct, but they represent a latent risk: any future refactoring that aligns the implementation with the (wrong) specification—a natural action when a developer trusts the formally verified spec—would silently introduce the bugs into production. V13 extends this pattern to ML-DSA: the AVX2 intrinsics specification contains a wrong multiplication definition, and any future proofs built on its axioms would inherit the unsoundness.

8 Comparative Analysis

To establish that our taxonomy describes a general phenomenon rather than an idiosyncrasy of one project, we examine verification boundary practices across AWS's cryptographic verification efforts, which use SAW [20], Cryptol, HOL Light, and CBMC rather than hax and F*.

8.1 Overview

AWS's verification effort spans multiple projects. The aws-lc-verification repository [2] uses SAW [20] for x86-64 and C code, NSym for AArch64 assembly, Cryptol for specifications, and Coq for elliptic curve properties, covering "*portions of* AWS libcrypto." More recently, AWS has expanded to s2n-bignum [24] (assembly-level arithmetic verified in HOL Light) and mlkem-native [37] (a FIPS 203 implementation using CBMC for C memory/type safety and HOL Light for assembly correctness). Both s2n-bignum and mlkem-native publish structured *soundness documents* [18,5] that enumerate verified properties, explicit assumptions, residual risks, and known gaps—a practice we return to in Section 8.3.

Table 8 summarizes the verified algorithms in aws-lc-verification. The scope is explicitly limited: only SHA-2 (384/512), HMAC-SHA384, AES-KW(P), and AES-GCM (256-bit) are verified in CI, each for specific platform configurations. The repository also contains proof scripts for ECDSA, ECDH, and HKDF, but these are currently disabled in CI and removed from the README—a clear acknowledgment that their verification status has regressed.

8.2 Same Failure Types, Different Practices

All five verification boundary failure types from our taxonomy are present in the AWS verification effort. The critical difference lies in how they are documented and managed.

Type I: Unverified Platform Abstractions. AWS-LC's SAW proofs verify x86-64 code produced by Clang 10. The README explicitly documents that "the verification results also apply to any compiler that produces semantically equivalent code" but notes the caveat `NoInline`: over 15 functions are patched with

Table 8. Verified algorithms in AWS-LC verification [2]. Each algorithm is verified only on specified platforms with explicit caveats.

Algo.	Platform	Tool	Caveats
SHA-2 (384/512)	SandyBr.+	SAW	NoEngine, MemCorrect
SHA-2 (384/512)	neo.	SAW, NSym	NoEngine, NoInline, MemCorrect, ArmSpecGap, ToolGap, LaxPointer
HMAC-SHA384	SandyBr.+	SAW	NoEngine, MemCorrect, InitZero, NoInline, CRYPTO_once_Correct
AES-KW(P) 256	SandyBr.+	SAW	InputLength, MemCorrect, NoInline
AES-GCM 256	SandyBr.–Sky.	SAW	MemCorrect, NoInline, GcmSpecGap, GcmMultipleOf16, GcmADNotVerified, GcmIV9Tag16, GcmWellFoundedInduction

`noinline` to prevent compiler optimizations that would break SAW's verification. The ARM assembly verification uses NSym, a separate tool with a separate specification (`ArmSpecGap`).

By contrast, libcrux's NEON backend is listed in `ADMIT_MODULES` with no documentation of why, and ARM64 intrinsics are marked opaque in the extraction without any accompanying caveat list.

Type II: Protocol Logic. AWS-LC does not verify protocol-level usage of its primitives—the verification scope explicitly covers API operations, not protocol compliance. This is the same boundary as libcrux's unverified hpke-rs protocol layer, but AWS-LC does not market protocol compliance as part of its verification story.

Type III: Specification Incompleteness. The AWS caveats table includes `GcmMultipleOf16` (whole blocks only), `GcmADNotVerified` (additional data not verified), and `GcmIV9Tag16` (only 12-byte IVs and 16-byte tags). These are explicit acknowledgments of specification incompleteness. The same boundary type exists in libcrux's ECDSA specification, which omits low-S normalization, but libcrux does not document which properties the specification does or does not capture.

Type IV: Wrapper Logic. Both projects have unverified wrapper code. AWS-LC explicitly lists functions whose verification assumes correct behavior of unverified components: `OPENSSL_malloc`, `OPENSSL_free`, `CRYPTO_refcount_inc`, `ERR_put_error`, and others are documented as "not verified, and assumed to behave correctly." Libcrux's wrapper layer (e.g., `key_gen` in `src/signature.rs`) has no comparable documentation of trust assumptions.

Type V: Specification Errors. AWS-LC documents the `GcmSpecGap` caveat: "AES-GCM is verified using an implementation-specific Cryptol spec describing an optimized form of AES-GCM," and the `SAWCore_Coq` caveat notes that "certain facts are admitted in Coq code." The s2n-bignum soundness document [18]

Table 9. Verification boundary practices compared.

Dimension	AWS-LC Verification	libcrux
Scope claims	"portions of AWS libcrypto"	"the formally verified crypto library"
Caveat docs	26 named caveats; structured soundness documents [18,5]	Manual `verification_status.md`; no caveats
Proofs in CI	Every push/PR (5 parallel jobs)	Lax on PR; full on schedule only; slow modules never verified
Admitted proofs	`do_prove` defaults true; explicit per-function `unsafe_assume_spec`	`ADMIT_MODULES` + auto-admitted `SLOW_MODULES`; lax-mode escape hatch

goes further, organizing residual risks into four explicit gaps: specification-to-requirements (our Type III), model-to-system (Type I), proof infrastructure (Type V), and integration (Type IV). These are the same structural risks as libcrux's wrong decompression constant, false serialization proof, and wrong multiplication specification, but they are systematically documented rather than hidden.

8.3 Key Structural Differences

Table 9 contrasts the verification boundary practices of the two projects across four dimensions.

The most significant difference is in proof execution. AWS-LC verification runs five parallel CI jobs on every push and pull request, providing immediate feedback on proof regressions. Libcrux runs only lax checking on pull requests; full verification is conditioned on scheduled or manual dispatch, and even then does not set `VERIFY_SLOW_MODULES=yes`, meaning slow modules—including the one containing the false proof V12—are admitted even in "full" verification.

The second difference is in boundary documentation. AWS-LC's README says "portions of" in the first sentence; libcrux's tagline says "the formally verified crypto library." The s2n-bignum and mlkem-native soundness documents [18,5] go further still: they tabulate residual risks, explicit assumptions (64-bit mode, little-endian byte order, uninterrupted execution), and known gaps in proof infrastructure, and openly identify planned improvements such as machine-checked links between C headers and formal specifications. Libcrux has no comparable documentation. AWS-LC's narrower scope makes exhaustive documentation more tractable, but the structural practices are not inherently scope-limited.

9 Discussion

Verification theatre (Section 1) is not the same as formal verification performed in good faith with clear communication about its scope. The distinction lies in

the gap between what is claimed and what is delivered. Our comparative analysis (Section 8) demonstrates that this gap is not inherent to formal verification but is a choice: AWS-LC and libcrux both have verification boundaries, but they communicate them very differently.

Our findings with V10–V13 reveal that the problem extends beyond the verification boundary. The formally verified code itself can contain wrong specifications that disagree with the standards they model (V10, V11), false proofs that are silently accepted (V12), and wrong definitions hidden behind false axioms (V13), while documentation claims full verification. V8 and V9 demonstrate that the pattern extends to ML-DSA: the core verification and signature encoding logic resides in admitted modules, and straightforward FIPS 204 violations in this code go undetected. When specifications are wrong, proofs are never checked, and documentation says "yes" when the answer is "no," the gap between claim and reality encompasses the verified core itself.

9.1 Admitted Proofs as Systemic Risk

The `ADMIT_MODULES` pattern revealed by V10–V13 represents a systemic risk that is distinct from—and arguably more dangerous than—the verification boundary failures of V1–V7. The boundary failures (V1–V7, V8–V9) arise from code that was never claimed to be verified, or from admitted modules whose proofs are never checked. The admitted-proof failures (V10–V13) arise from code that *is* claimed to be verified but where the proofs are silently not checked or built on false axioms.

The three-layer mechanism described in Section 6.3—default admission of slow modules, lax-mode tactic escape hatches, and manually maintained status documents—creates a failure mode where specifications can contain mathematical errors (V10, V11), proofs can make false claims (V12), axiomatized lemmas can contradict the definitions they abstract (V13), and documentation can claim full verification—all simultaneously.

9.2 Recommendations

We propose the following practices, informed by both our negative findings in libcrux and the positive practices observed in AWS's verified libcrypto:

R1: Machine-readable verification manifests. Every release should include a manifest of which code paths are verified, against which properties, and for which target architectures. The manifest should be generated from the build system, not manually maintained. AWS's soundness documents for s2n-bignum and mlkem-native [18,5]—which enumerate verified properties, assumptions, residual risks, and open gaps—provide a model for the content; deriving such manifests automatically from the proof infrastructure would close the documentation-drift gap exposed by V10–V12.

R2: Proofs in CI on every commit. All proof obligations should be discharged in the continuous integration pipeline on every push and pull request, following the model used by AWS-LC verification. Proofs that are too slow for per-commit CI should be flagged as incomplete in the verification manifest rather than silently admitted. The `ADMIT_MODULES` pattern—where modules are admitted by default and full verification is never triggered—should be considered a verification defect. Axiomatized intrinsic specifications (`val` declarations without implementations) should be proved against a reference model or tested exhaustively (cf. V13).

R3: No lax-mode escape hatches in tactics. Proof tactics should fail loudly when invoked in lax mode, not silently admit all goals. The pattern `if lax_on () then iterAll tadmit` converts a false proof into a "verified" proof whenever the module is admitted. Tactics should either run their full prover or raise an error indicating that the proof was not checked.

R4: Clear scope communication. Claims of formal verification should be accompanied by explicit qualifications: which properties, which code paths, which platforms. The phrase "formally verified" without qualification should be reserved for systems where the verification boundary has been minimized to the hardware level, as in CompCert [32] or seL4 [26].

R5: Defense in depth. Formal verification should be presented as one component of a defense-in-depth strategy. Our findings demonstrate that traditional practices—code review, specification compliance auditing, cross-platform testing, and fuzzing—catch categories of bugs that formal verification, as currently practiced, does not.

9.3 Limitations

Scope and generalizability. Our taxonomy is derived from two projects (CE Labs's libcrux and AWS's verification efforts including aws-lc-verification, s2n-bignum, and mlkem-native) using hax/F* and SAW/Cryptol/HOL Light/CBMC respectively. Our evidence base does not cover Jasmin, Lean 4, or Dafny; projects using these tools may exhibit additional failure types or be immune to certain types due to architectural differences. The concrete auditing patterns (e.g., `ADMIT_MODULES`, `#[hax_lib::opaque]`) are stack-specific, though the high-level steps—map the boundary, audit specs against standards, verify proofs are checked, cross-reference documentation—are tool-agnostic. The vulnerabilities reflect the state of the codebase at the time of our audit (February 2026); our contribution is the structural patterns they reveal, not the individual bugs.

Specification bugs vs. deployed impact. V10–V13 reside in the F* specification and proof code; the Rust implementations are correct and no deployed system currently produces wrong output. The risk is indirect: a developer who trusts the formally verified specification and refactors the implementation to match it would silently introduce the specification's errors into production code.

Table 10. Disclosure timeline for the vulnerabilities reported in this paper.

Date	Event
Nov 2025	Valsorda publicly reports V1 [50]; CE Labs fixes the bug. No security advisory issued by CE Labs; advisory filed by Birr-Pixton [7].
Dec 27, 2025	GuuJiang publicly reports V2 [22]; cross-backend endianness bug in ML-KEM incremental API causing Signal PQ ratchet failures [46].
Jan 6, 2026	CE Labs merges fix for V2 [42]. No security advisory issued.
Feb 5, 2026	We submit PRs with tested fixes for V3–V6 [30,29,28,31].
Feb 5–6	CE Labs blocks our GitHub account, closes all four PRs, citing a "documented Security Policy" that was not present in the repository at the time of disclosure.
Feb 9	CE Labs merges our fixes without attribution to the original authors.
Feb 12	CE Labs publishes response [17] acknowledging V1 and V6, omitting V3–V5 and V7, and stating "no bugs have been found in the verified code."
Feb 2026	We discover V7, V10–V13; V7 disclosed via public commit [27] as no reporting channel remained available.
Feb 17	We disclose V8–V9: two FIPS 204 specification violations in ML-DSA (verifier norm check dead code, missing hint bound check).
Mar 4	CE Labs merges fixes for V8 [43] and V9 [44], crediting the discoverer.

Completeness. Our audit focused on ML-KEM, ML-DSA, HPKE, ECDSA, Ed25519, and AES-GCM within the libcrux ecosystem; we do not claim the thirteen vulnerabilities are exhaustive. Other primitives in libcrux (e.g., SHA-2, Chacha20-Poly1305) were not audited. The verification boundary failures we identify may exist in other components.

9.4 Disclosure Timeline

Table 10 summarizes the disclosure process.

We followed full disclosure practices for all vulnerabilities reported in this paper. Our initial disclosures followed the coordinated model: on February 5, 2026, we submitted pull requests with tested fixes for V3–V6 directly to CE Labs's public repositories [30,29,28,31], each including a description, root cause analysis, and working patch. Within 24 hours, CE Labs blocked our GitHub account and closed all four PRs without technical review. On February 9, CE Labs merged our fixes without attribution to the original authors. On February 12, CE Labs published a response [17] that acknowledged V1 and V6, omitted V3–V5 and V7, and stated "no bugs have been found in the verified code"—a claim directly contradicted by V8–V13. Coordinated disclosure requires a willing counterparty; when that channel was severed and the vendor publicly mischaracterized the find-

ings, full disclosure became necessary to correct the record and allow downstream consumers to assess their exposure [45]. V10–V13 reside in publicly available F⋆ specification and proof code; disclosing them does not provide adversaries with an attack capability, as the deployed Rust implementations are correct.

10 Related Work

Verified cryptographic libraries. HACL⋆ [23,53] and its successor EverCrypt [38] proved memory safety, functional correctness, and secret independence for a comprehensive set of cryptographic primitives in F⋆. EverCrypt is notable for its precise accounting of trust assumptions, explicitly identifying the compiler, Vale embedding semantics, and hardware as trusted components. Libcrux's legacy primitives (ECDSA, Ed25519) derive from HACL⋆, making our Type III and Type IV findings (V5, V6) directly relevant to the HACL⋆ lineage. Jasmin [1] verifies at the assembly level, eliminating the compiler from the trusted computing base; its remaining unverified passes are explicitly documented [3]. FiatCrypto [19] generates verified field arithmetic from Coq specifications, trusting only the Coq kernel and an unverified printer to C. Each of these projects documents its verification boundary clearly—a practice our findings show is not universal.

Verified compilers and the trust boundary. CompCert [32] provides machine-checked proofs for its compilation passes but explicitly identifies its parser, assembler, and linker as unverified. Yang et al.'s Csmith campaign [52] found zero wrong-code bugs in CompCert's verified middle-end but did find bugs in the unverified front-end, directly paralleling our finding that libcrux's verified core is more robust than its wrappers. Our work extends this observation: the verified core itself can contain specification errors (V10, V11), false proofs (V12), and false axioms (V13) that survive when the build system does not check them. Thompson's "Reflections on Trusting Trust" [49] remains relevant to any project whose proofs are dispatched through unverified SMT solvers.

Boundary documentation in practice. The seL4 microkernel [26] provides the gold standard for documenting verification boundaries, maintaining a dedicated page listing every trust assumption. Boston et al. [10] describe the verification of AES-256-GCM and SHA-384 in AWS-LC using SAW [20]; AWS has since expanded to HOL Light–based verification in s2n-bignum [24,18] and CBMC-based verification in mlkem-native [37,5], both with structured soundness claims. We provide the first comparative analysis of these boundary practices against another verified library in Section 8.

Positioning our contribution. Our term "verification theatre" deliberately echoes Schneier's "security theater" [45]—deploying formal verification with unqualified claims creates an appearance of correctness while leaving significant code paths unverified or verified against wrong specifications. Prior work has focused on

developing verification techniques and documenting trust assumptions; our contribution is orthogonal: the first taxonomy of verification boundary failure types validated by concrete vulnerabilities, the first auditing methodology for detecting them, and the first comparative analysis of boundary practices across verified cryptographic libraries.

11 Conclusion

We have analyzed thirteen vulnerabilities in CE Labs's libcrux, hpke-rs, and libcrux-psq cryptographic libraries—two discovered independently (by Valsorda [50] and GuuJiang [22]) and eleven identified by us—each of which escaped the formal verification methodology employed. The first nine exist at or beyond the verification boundary, including two FIPS 204 specification violations in the ML-DSA verifier that cause libcrux to accept non-conforming signatures. The final four exist in formally verified specification and proof code: in ML-KEM, a decompression specification that disagrees with FIPS 203, a missing inverse NTT in the encryption specification, and a false serialization proof; in ML-DSA, a wrong multiplication specification that renders axiomatized AVX2 proofs unsound.

We developed a taxonomy of five verification boundary failure types and a lightweight auditing methodology for detecting them, validated across both ML-KEM and ML-DSA implementations. A comparative analysis with AWS's verified libcrypto confirmed that the same failure types are endemic to formally verified cryptographic libraries, but that their management—through explicit caveat documentation, proof execution in CI, and clear scope communication—varies significantly between projects.

These findings do not diminish the value of formal verification as a technique. Machine-checked proofs of functional correctness, memory safety, and secret independence are genuine contributions to software security. What our findings challenge is the practice of presenting partial verification as comprehensive assurance—and, more pointedly, the practice of claiming verification when the build system defaults to not checking proofs.

We urge the formal verification community to adopt precise, qualified language when describing verification achievements; to treat verification boundaries as first-class security-relevant documentation; to ensure that build systems actually verify the proofs they claim to check; and to recognize that formal methods complement, rather than replace, the engineering practices that remain essential to producing secure cryptographic software.

Acknowledgements. We thank Filippo Valsorda and GuuJiang for independently reporting V1 and V2, respectively. We thank Toby Fox for providing musical inspiration that was integral to the production of this research work.

Generative AI Usage

The writing of this paper was assisted by AI tools: Claude was used to sketch an initial skeleton for the paper and to help integrate research results into a coherent narrative. All AI-generated prose was manually validated and verified for correctness. All citations were added by humans and not by AI. None of this paper's research findings were AI-assisted.

References

1. 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: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823. ACM (2017)
2. Amazon Web Services: AWS LibCrypto Formal Verification. `https://github.com/awslabs/aws-lc-verification` (2024)
3. Arranz-Olmos, S., Barthe, G., Blatter, L., Grégoire, B., Laporte, V., Torrini, P.: The Jasmin compiler preserves cryptographic security. arXiv preprint arXiv:2511.11292 (2025)
4. Barnes, R., Bhargavan, K., Lipp, B., Wood, C.: Hybrid Public Key Encryption. RFC 9180 (2022)
5. Becker, H., Chapman, R., Harrison, J.: mlkem-native soundness. `https://github.com/pq-code-package/mlkem-native/blob/main/SOUNDNESS.md` (2026)
6. Bhargavan, K., Buyse, M., Franceschino, L., Hansen, L.L., Kiefer, F., Schneider-Bensch, J., Spitters, B.: hax: Verifying security-critical Rust software using multiple provers. In: Verified Software: Theories, Tools and Experiments (VSTTE 2024). pp. 96–119. Springer (2024)
7. Birr-Pixton, J.: RUSTSEC-2025-0133: Incorrect calculation on aarch64. `https://rustsec.org/advisories/RUSTSEC-2025-0133.html` (2025)
8. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In: 10th USENIX Workshop on Offensive Technologies (WOOT '16) (2016)
9. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium. pp. 917–934 (2017)
10. Boston, B., Breese, S., Dodds, J., Dodds, M., Huffman, B., Petcher, A., Stefanescu, A.: Verified cryptographic code for everybody. In: Computer Aided Verification (CAV 2021). pp. 645–668. Springer (2021)
11. CE Labs: hpke-rs: Pure Rust implementation of HPKE. `https://github.com/celabshq/hpke-rs` (2024)
12. CE Labs: libcrux-psq: Post-quantum pre-shared key protocol. `https://github.com/celabshq/libcrux/tree/main/libcrux-psq` (2024)
13. CE Labs: libcrux: The formally verified crypto library for Rust. `https://github.com/celabshq/libcrux` (2024)
14. CE Labs: ML-KEM specification commit `c1935b9cf8`. GitHub Commit, `https://github.com/celabshq/libcrux/commit/c1935b9cf8` (2024), august 12, 2024. Commit message: "spec"

15. CE Labs: ML-KEM serialization proof commit 5f6e3c2083. GitHub Commit, https://github.com/celabshq/libcrux/commit/5f6e3c2083 (2025), march 25, 2025. Commit message: "fix(ml-kem): deserialize_10/12: remove admit"
16. CE Labs: ML-KEM verification status. https://github.com/celabshq/libcrux/blob/main/libcrux-ml-kem/proofs/verification_status.md (2025), accessed February 2026
17. CE Labs: The strengths and limits of formal verification. https://cryspen.com/post/strengths-and-limitations/ (2026), published February 12, 2026
18. Chapman, R., Harrison, J.: s2n-bignum soundness. https://github.com/awslabs/s2n-bignum/blob/main/doc/s2n_bignum_soundness.md (2026)
19. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic–with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy. pp. 1202–1219. IEEE (2019)
20. Galois, Inc.: SAW: The software analysis workbench. https://saw.galois.com (2024)
21. Google Bug Hunters: Formally verified post-quantum algorithms. https://bughunters.google.com/blog/6038863069184000/formally-verified-post-quantum-algorithms (2024)
22. GuuJiang: Inconsistent ML-KEM encapsulation state between portable and SIMD backends. GitHub Issue, https://github.com/celabshq/libcrux/issues/1275 (2025)
23. HACL Community: HACL*: A formally verified cryptographic library. https://hacl-star.github.io (2024)
24. Harrison, J.: s2n-bignum: A collection of bignum arithmetic routines with machine-checked formal proofs in HOL Light. https://github.com/awslabs/s2n-bignum (2021)
25. Josefsson, S., Liusvaara, I.: Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 (2017)
26. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. ACM (2009)
27. Kobeissi, N.: DoS via crafted ciphertext on libcrux-psq's AESGCM ciphersuite. GitHub Commit, https://github.com/nadimkobeissi/libcrux/commit/96d3d8a751aa536a619476a8b45c7c12b51a74ec (2026)
28. Kobeissi, N.: ECDSA signature malleability via lack of low-S normalization. GitHub Pull Request, https://github.com/celabshq/libcrux/pull/1315 (2026)
29. Kobeissi, N.: Nonce reuse vulnerability in hpke-rs. GitHub Pull Request, https://github.com/celabshq/hpke-rs/pull/118 (2026)
30. Kobeissi, N.: Reject X25519 all-zero DH outputs per RFC 9180 section 7.1.4. GitHub Pull Request, https://github.com/celabshq/hpke-rs/pull/117 (2026)
31. Kobeissi, N.: Unnecessary "double clamping" in Ed25519 reduces seed entropy by 5 bits. GitHub Pull Request, https://github.com/celabshq/libcrux/pull/1316 (2026)
32. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
33. National Institute of Standards and Technology: Digital Signature Standard (DSS). FIPS PUB 186-5 (2023)
34. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. FIPS 204 (2024)

35. National Institute of Standards and Technology: Module-Lattice-Based Key-Encapsulation Mechanism Standard. FIPS 203 (2024)
36. OpenMLS Contributors: OpenMLS: A Rust implementation of the Messaging Layer Security protocol. https://github.com/openmls/openmls (2024)
37. pq-code-package: mlkem-native: A portable C implementation of ML-KEM. https://github.com/pq-code-package/mlkem-native (2025)
38. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., et al.: EverCrypt: A fast, verified, cross-platform cryptographic provider. In: 2020 IEEE Symposium on Security and Privacy. pp. 983–1002. IEEE (2020)
39. Protzenko, J., Zinzindohoé, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Bé guelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F*. Proceedings of the ACM on Programming Languages **1**(ICFP), 17:1–17:29 (2017)
40. Schneider-Bensch, J.: ML-DSA hint deserialization commit d979b001f9. GitHub Commit, https://github.com/celabshq/libcrux/commit/d979b001f9 (2025), november 17, 2025. Commit message: "We would like to use early returns below, but doing so triggers a bug in Eurydice"
41. Schneider-Bensch, J.: ML-DSA verifier norm bound commit 326c837a33. GitHub Commit, https://github.com/celabshq/libcrux/commit/326c837a33 (2025), january 7, 2025. Commit message: "Further macro monomorphization of parameter sets"
42. Schneider-Bensch, J.: Fix endianness in portable ML-KEM incremental API. GitHub Pull Request, https://github.com/celabshq/libcrux/pull/1276 (2026)
43. Schneider-Bensch, J.: [ML-DSA] fix incorrect check of signer response norm during verification. GitHub Pull Request, https://github.com/celabshq/libcrux/pull/1347 (2026), merged March 4, 2026
44. Schneider-Bensch, J.: [ML-DSA] fix incorrect hint decoding. GitHub Pull Request, https://github.com/celabshq/libcrux/pull/1348 (2026), merged March 4, 2026
45. Schneier, B.: Beyond Fear: Thinking Sensibly About Security in an Uncertain World. Copernicus Books (2003)
46. Signal Community: ML-KEM decapsulation failures in post-quantum ratchet. GitHub Issue, https://github.com/niclas-AXelorth/libsignal/issues/641 (2025)
47. Signal Messenger, LLC: libsignal: Platform-agnostic apis for Signal clients and servers. https://github.com/signalapp/libsignal (2024)
48. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in F*. ACM SIGPLAN Notices **51**(1), 256–270 (2016)
49. Thompson, K.: Reflections on trusting trust. Communications of the ACM **27**(8), 761–763 (1984)
50. Valsorda, F.: Incorrect ML-DSA output on Alpine Linux and Ampere Altra. GitHub Issue, https://github.com/celabshq/libcrux/issues/1220 (2025)
51. Wuille, P.: Dealing with Malleability. Bitcoin Improvement Proposal 62 (2014)
52. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. ACM (2011)

53. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1789–1806. ACM (2017)

A Code Listings

```
pub fn _vxarq_u64<const LEFT: i32, const RIGHT: i32>(
    a: uint64x2_t, b: uint64x2_t
) -> uint64x2_t {
    #[cfg(/* sha3 target feature available */)]
    unsafe { vxarq_u64::<RIGHT>(a, b) }
    #[cfg(/* sha3 target feature NOT available */)]
    {
        let a_xor_b = _veorq_u64(a, b);
        _veorq_u64(
            _vshlq_n_u64::<LEFT>(a_xor_b),
            _vshrq_n_u64::<RIGHT>(a_xor_b),
        )
    }
}
```

Listing 1.1. V1: The `_vxarq_u64` intrinsic wrapper. The fallback path (lines 7–11) was used on platforms without SHA-3 hardware support.

```
fn dh(alg: KemAlgorithm, pk: &[u8], sk: &[u8])
    -> Result<Vec<u8>, Error>
{
    match alg {
        KemAlgorithm::DhKem25519 => {
            let sk = X25519StaticSecret::from(sk_array);
            Ok(sk
                .diffie_hellman(&X25519PublicKey::from(pk_array
                    ))
                .as_bytes()
                .to_vec()) // No zero check
        }
        // ...
    }
}
```

Listing 1.2. V3: X25519 DH computation in hpke-rs. No validation is performed on the output.

```
AEADKey::AesGcm128(key) => {
    libcrux_aesgcm::AesGcm128::decrypt(
        plaintext, key, &self.nonce,
        aad, ciphertext, tag,
    )
    .unwrap();
    // .map_err(|_| AEADError::CryptoError)?;
}
```

Listing 1.3. V7: The AES-GCM decryption path in libcrux-psq (`src/aead.rs`). The correct error handling is commented out.

```
// Line 402-405 (verifier, BUG):
if vector_infinity_norm_exceeds::<SIMDUnit>(
    &deserialized_signer_response,
    (2 << GAMMA1_EXPONENT) - BETA,
//   ^^^^^^^^^^^^^^^^^^^^^^^^^ = 2*gamma_1 - beta
) {
    return Err(VerificationError::
        SignerResponseExceedsBoundError);
}
```

Listing 1.4. V8: The ML-DSA verifier norm check in `ml_dsa_generic.rs`. The bound `2 << GAMMA1_EXPONENT` computes $2\gamma_1$, not γ_1.

```
for i in 0..rows_in_a {
    let current_true_hints_seen =
        hint_serialized[max_ones_in_hint + i] as usize;

    if (current_true_hints_seen
            < previous_true_hints_seen)
        || (previous_true_hints_seen
            > max_ones_in_hint)
    //      ^^^^^^^^^^^^^^^^^^^^^^^^^^
    //      Checks PREVIOUS, not current.
    //      FIPS 204: y[w + i] > w (current > max)
    {
        malformed_hint = true;
        break;
    }

    for j in previous_true_hints_seen
             ..current_true_hints_seen {
        set_hint(out_hint, i,
            hint_serialized[j] as usize);
    }
    // ...
    previous_true_hints_seen =
        current_true_hints_seen;
}
```

Listing 1.5. V9: Hint deserialization in `signature.rs`. The overflow check uses the *previous* iteration's count instead of the current one.

```
let decompress_d (d: dT {d ≠ 12})
    (x: field_element_d d): field_element
```

```
  = let r = (x * v v_FIELD_MODULUS + 1664) / pow2 d in
    r
```

Listing 1.6. V10: The `decompress_d` specification in `Spec.MLKEM.Math.fst`. The rounding constant 1664 is incorrect; it should be `pow2 (d - 1)`.

```
ADMIT_MODULES = Libcrux_ml_kem.Ind_cpa.fst \
    Libcrux_ml_kem.Sampling.fst \
    Libcrux_ml_kem.Vector.Neon.Arithmetic.fst \
    Libcrux_ml_kem.Vector.Neon.Compress.fst \
    Libcrux_ml_kem.Vector.Neon.fsti \
    Libcrux_ml_kem.Vector.Neon.fst \
    Libcrux_ml_kem.Vector.Neon.Ntt.fst \
    Libcrux_ml_kem.Vector.Neon.Serialize.fst \
    Libcrux_ml_kem.Vector.Neon.Vector_type.fst
```

Listing 1.7. V10: Excerpt from the ML-KEM extraction Makefile showing admitted modules.

```
(* u: correct -- inverse NTT applied *)
let u = vector_add
  (vector_inv_ntt (matrix_vector_mul_ntt ...))
  error_1

(* v: BUG -- no inverse NTT *)
let v = poly_add
  (poly_add
    (vector_dot_product_ntt t_as_ntt r_as_ntt)
    error_2)
  mu
```

Listing 1.8. V11: Encryption spec in `Spec.MLKEM.fst`. The computation of **u** correctly applies `vector_inv_ntt`; the computation of v omits `poly_inv_ntt`, adding an NTT-domain polynomial to coefficient-domain terms.

```
let deserialize_12_bit_vec_lemma_bounded
    (v: t_Array u8 (sz 24))
  : squash (
    let result = ${deserialize_12} v in
    (forall (i: nat {i < 16}).
      Rust_primitives.bounded
        (Seq.index result.f_elements i)
        1))           (* <-- should be 12 *)
  = _ by (Tactics.GetBit.prove_bit_vector_equality'())
```

Listing 1.9. V12: The false lemma in `serialize.rs`. The bound should be 12, not 1. The tactic `prove_bit_vector_equality'` silently accepts this false claim because the module is admitted.

```
let prove_bit_vector_equality' (): Tac unit =
  if lax_on ()
  then iterAll tadmit
  else prove_bit_vector_equality'' ()
```

Listing 1.10. V12: The lax-mode escape hatch in `Tactics.GetBit.fst`. When `lax_on()` is true, all proof goals are admitted unconditionally.

```
[@@ "opaque_to_smt"]
let i16_mul_32extended (x y: i16): i32 =
  (cast x <: i32) *! (cast x <: i32)
  (*                  ^^^ should be: cast y *)
```

Listing 1.11. V13: The `i16_mul_32extended` specification in `Spec.Intrinsics.fsti`. The second parameter `y` is ignored; the body computes x^2 instead of $x \cdot y$.