

The Lean Transparency Log: Distributing Kernel-Checked Correctness Evidence for Deployed Ed25519 Implementations

Olaf Horvath

Olaf.Horvath@zkdefi.org ORCID 0009-0004-8008-5805

July 2026 (revised)

Abstract

Interactive theorem provers can certify functional correctness of deployed cryptographic code, but the resulting assurance is expensive to consume: re-checking a realistic proof corpus requires a proof toolchain and hours of kernel time, which excludes almost every downstream user. We describe the Lean Transparency Log (LTL), an RFC 9162-style transparency log whose leaves are *replay attestations*: signed statements that the Lean 4 proofs of a specific Rust repository, at a specific git commit, re-check with exactly their documented axiom sets. Consumers verify one signature and a logarithmic inclusion proof in milliseconds; the kernel time is paid once, by the log operator.

This paper makes the trust model precise and proves the consumer-facing security claims. We define the attestation-transparency setting, give an explicit adversary model in which the operator may be malicious, and prove: completeness and soundness of the inclusion verifier (soundness via an explicit reduction extracting a SHA-256 collision), the analogous consistency statement, safety of the consumer’s head-pinning state machine (equivocation by the operator yields publicly verifiable evidence), and *verdict integrity*—consumers re-derive verification verdicts locally from observed axiom cones, so the operator is trusted only for *observations*, never for *verdicts*. A further design choice ties the log to its own subject matter: tree heads are signed by a binary built from the very Ed25519 implementation whose correctness certificates are leaves of the log. We report a small production deployment covering four verified production Ed25519 implementations, state exactly what the accumulated evidence does and does not establish, and outline the mechanization of this paper’s theorems in Lean as the natural next step.

1 Introduction

Formal verification of deployed cryptographic code has matured from research prototypes to substantial artifacts: verified-by-construction libraries such as HACL* [13] and Fiat-Crypto [15] ship in mainstream software, and post-hoc verification pipelines such as Aeneas [11] make it possible to state and prove theorems about existing production Rust code. The corpus underlying this paper is of the latter kind: four production Ed25519 implementations—upstream `curve25519-dalek/ed25519-dalek` and three deployed forks (Solana, RISC Zero, Betrusted)—each carry Lean 4 [12] certificates, proven against that fork’s own extracted model, covering field arithmetic over $\mathbb{F}_{2^{255}-19}$, the complete twisted Edwards laws [18, 19], scalar arithmetic mod ℓ , encoding/decoding with constructive point decompression, and a four-tier characterization of signature verification [16, 17] whose strongest tier states: the extracted verifier accepts iff the signature’s R component decompresses to a valid curve point equal to $[k](-A) + [s]B$. Section 2 states these theorems precisely.

The economics of *consuming* such evidence are poor: re-checking one fork’s certificates takes ≈ 30 minutes of Lean kernel time and a pinned toolchain. A wallet, a package manager, or an autonomous agent choosing a cryptographic backend cannot pay this per decision—and need not: a deterministic re-check yields a fact that can be attested once and distributed. This is the classic transparency-log trade—Certificate Transparency [1, 2] for certificate issuance, Sigstore’s Rekor [5] for signing events and supply-chain attestations [6], key transparency [7], checksum databases—applied to a payload with different trust semantics: evidence of machine-checked mathematical truth, together with its exact assumption set.

Contributions. The hash structure and proof algorithms are RFC 9162 verbatim, and we claim no novelty for any individual component. The contributions are:

1. **A precise trust model for attestation transparency over machine-checked proofs** (§4), in which the log operator is trusted for *observations* (“this is what the kernel printed”) but never for *verdicts* (“these proofs are acceptable”), because consumers re-derive every verdict locally from the observed axiom cones carried in each attestation.
2. **Security proofs for the consumer-facing claims** (§6): completeness and soundness of the RFC 9162 inclusion verifier as used here (soundness as an explicit extractor that turns any accepting proof for a non-member leaf into a SHA-256 collision), the analogous consistency statement, safety of the consumer’s pin-store state machine, and verdict integrity. The statements are elementary but, written out, they pin down exactly which assumption carries which claim.
3. **Boundary-exact axiom auditing** (§5): observed axiom cones are matched against per-theorem documented boundaries *exactly, in both directions*—an unexpected axiom and a missing boundary axiom are both flagged.
4. **A self-referential (not circular) signing design and a deployed instance** (§7, §8): tree heads are signed by a binary built from the pinned source of exactly the Ed25519 implementation attested in the log, with the operator’s own Merkle self-check of that leaf embedded in every signature; and a small, reproducible production deployment over the four-fork corpus, including a measurement of proof portability across real forks.

Non-claims. The LTL does not mechanize cryptographic security proofs—that bridge is being built by EasyCrypt and its relatives [10]. It does not establish correctness of any binary, of SHA-512, of wire-format parsers, of the signing path, or any side-channel property; §8 enumerates the assumption set in full. It bridges an adjacent, mostly empty gap: type-theory-certified artifacts have no distribution infrastructure, and cryptographic transparency infrastructure has never carried kernel-checked mathematics.¹

2 Background: the proof corpus

The corpus is a stack of theorems about extracted code, each stated through a denotation from machine representation to mathematics. Field elements are five 51-bit limbs denoting $\llbracket (a_0, \dots, a_4) \rrbracket = \sum_i a_i 2^{51i} \bmod p$ with $p = 2^{255} - 19$, and every operation carries a two-clause specification—the value is right *and* the representation invariant is preserved, e.g.

$$\forall a b. \text{ bnd } a \Rightarrow \text{ bnd } b \Rightarrow \exists c. \text{ mul } a b = \text{ ok } c \wedge \text{ bnd } c \wedge \llbracket c \rrbracket = \llbracket a \rrbracket \cdot \llbracket b \rrbracket.$$

¹The acronym LTL collides with linear temporal logic [20]; the collision is acknowledged.

Point operations are proven to implement the complete twisted Edwards addition law on $E : -x^2 + y^2 = 1 + d x^2 y^2$ over $\mathbb{F}_{2^{255}-19}$,

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 + x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right),$$

including the completeness fact that makes it branch-free ($a = -1$ is a square and d a non-square in $\mathbb{F}_{2^{255}-19}$, so the denominators never vanish [18]).

The signature apex as a lifting ladder. The signature-tier result is not one theorem but a ladder of four, each lifting the previous one to a stronger domain; the payload the log distributes is the *conjunction* of the four, and their separation is what makes the residual hypotheses legible. Write $\text{accept}(A, m, R, s)$ for “the extracted verifier returns ok”, let k be the scalar produced by the hash oracle $H(R, A, m)$ with *no properties assumed of H* , and let r_1 be the canonical byte encoding of the signature’s declared point R . Each tier is proven for the extracted code under the wire-format hypotheses $\mathcal{W}$ (the signature parses to an internal representation and the relevant compressed points re-encode; these outcomes are assumed, not proven—their byte-level specifications are the R5 frontier of §9).

T1 (byte apex). $\text{accept}(A, m, R, s) \Leftrightarrow \text{compress}([s]B - [k]A) = r_1$. Acceptance is byte-equality of the verifier’s recomputed encoding with the signature’s R bytes—a statement purely about the extracted control flow.

T2 (canonical half-lift). The recomputed bytes $\text{compress}([s]B - [k]A)$ are the canonical encoding of the group element $[k](-A) + [s]B$; that is, compress agrees on this input with the mathematical canonical-encoding function. T1 and T2 give $\text{accept} \Leftrightarrow \text{enc}([k](-A) + [s]B) = r_1$.

T3 (injectivity / point equation). Canonical encodings are injective on $E(\mathbb{F}_{2^{255}-19})$: if a valid curve point P has $\text{enc}(P) = r_1$ then $P = [k](-A) + [s]B$. Injectivity is exactly where non-squareness of d re-enters—it keeps $1 + dy^2 \neq 0$, so the curve equation determines x^2 from y and the encoding is one-to-one.

T4 (constructive full lift). $\text{accept}(A, m, R, s) \Leftrightarrow \text{decompress}(R) = [k](-A) + [s]B$, with the extracted decompress proven to realize the mathematical inverse of enc : exact byte parsing, the $(p+3)/8$ -power square root, and sign-bit root selection (the two roots x and $p-x$ differ in parity since p is odd, so the stored sign bit selects correctly).

The lift is monotone in strength—T1 is about bytes the code emits, T4 is about the group element a third party would recover from R —and each step names precisely one new mathematical fact (canonicity, injectivity, constructive inversion). A consumer that only trusts byte equality can stop at T1; a consumer reasoning about the underlying group element relies on T4. Both are in the corpus, separately certified, and the log carries all four so the consumer chooses the tier, not the operator.

Axiom cones. Each theorem’s *axiom cone*—the set of axioms its proof ultimately depends on, as reported by Lean’s `#print axioms`—is pinned exactly: the standard three axioms for the foundational certificates, plus an enumerated oracle boundary (SHA-512 and the wire-format types) at the four apex tiers. It is this exact set, not a pass/fail label, that each leaf carries and each consumer re-checks (§5.2). Appendix D restates the ladder with the Lean theorem names; Appendix C lists the per-fork allowed sets verbatim.

3 Related work

Certificate Transparency [1, 2] supplies the data structure and proof algorithms, used here unchanged; the underlying history-tree technique originates with Crosby and Wallach [3]. Dowling, Günther, Herath and Stebila [4] give formal security definitions and proofs for the CT primitives (logging schemes, inclusion, consistency); the analysis in §6 is in the same spirit, specialized to this system’s verifier and stated so that each claim can later be mechanized in Lean (§10). Rekord within Sigstore [5] is the closest deployed system: a transparency log over signing events and supply-chain attestations such as in-toto [6] link metadata; its payloads attest *process* (who signed, how an artifact was built), whereas LTL leaves attest kernel-checked mathematical statements together with their assumption sets, and the consumer re-derives verdicts rather than trusting labels. Key transparency [7] and checksum databases share the pattern with different payloads. Proof-carrying code [8] ships proofs to consumers who check them; the LTL serves consumers who cannot run any checker, replacing proof transport with attestation, inclusion, and signature—at the cost of trusting the operator’s kernel run, a cost the design minimizes (§4) but does not eliminate. Cheval, Moreira and Ryan formally verify transparency protocols themselves [9]; our direction is the complement (we log the verification), and §10 proposes meeting in the middle. Verified Merkle tree implementations exist, notably in EverCrypt [14]; §10 builds on that precedent rather than claiming it.

4 System and trust model

4.1 Roles and scheme syntax

The system has exactly two roles with deliberately asymmetric costs and capabilities. The *operator* (one per log) owns a Lean toolchain, replays proof corpora, holds the log’s signing key, and bears append-only obligations. *Consumers* (unbounded in number) hold the operator’s public key, receive small evidence files, and verify with roughly 25 lines of standard-library code (Appendix B). Nothing a consumer does requires a theorem prover.

We phrase the system as an *attestation-transparency scheme*, in the style of the logging schemes of Dowling et al. [4], so that the security goals below can name its algorithms precisely.

Definition 1 (Attestation-transparency scheme). *A scheme Π is a tuple of algorithms over a hash function H and a signature scheme Sig :*

- $\text{KeyGen} \rightarrow (sk, pk)$: the operator’s head-signing keypair.
- $\text{Append}(sk, \mathbf{D}, a) \rightarrow (\mathbf{D}', \sigma)$: appends attestation-leaf a to the ordered leaf list $\mathbf{D}$, returning the new list and a signed tree head $\sigma = \text{Sig.Sign}(sk, (|\mathbf{D}'|, \text{MTH}(\mathbf{D}'), t))$.
- $\text{ProveIncl}(\mathbf{D}, m) \rightarrow P$ and $\text{VerifyIncl}(pk, d, m, \sigma, P) \rightarrow \{0, 1\}$: the membership proof and its verifier (§5.3, Appendix B).
- $\text{ProveCons}(\mathbf{D}, n_0) \rightarrow C$ and $\text{VerifyCons}(pk, \sigma_0, \sigma_1, C) \rightarrow \{0, 1\}$: the append-only (consistency) proof between two signed heads and its verifier (§5.3).
- $\text{Verdict}(\text{allowed}, a) \rightarrow \{\text{clean}, \neg\text{clean}, \perp\}^{|a|}$: the consumer’s per-certificate verdict function (§5.2), parameterized by the consumer’s own allowed-axiom table *allowed* and taking no operator label as input.

MTH , $\text{ProveIncl}/\text{VerifyIncl}$ and $\text{ProveCons}/\text{VerifyCons}$ are the RFC 9162 algorithms, defined in §5.3; Append and Verdict are specific to this system.

4.2 Adversary model

We consider a probabilistic polynomial-time adversary $\mathcal{A}$ that controls the network (may reorder, replay, drop, or forge messages to consumers) and may *be* the operator. A malicious operator may sign arbitrary tree heads, construct arbitrary leaves, present different views to different consumers, and label attestations arbitrarily. The single capability we do *not* model cryptographically is falsification of kernel observations: an operator who reports an axiom cone that the Lean kernel never printed is lying about a physical event on its own machine, and no log structure can exclude this; §4.4 isolates this residual trust precisely. Standard assumptions: SHA-256 is collision resistant; Ed25519 (as instantiated by the signing binary) is EUF-CMA secure; the consumer obtained the operator’s true public key (trust on first use; §9).

4.3 Security goals

G1 (Membership). If a consumer accepts a receipt for attestation a against a signed head, then a is a leaf of the tree committed by that head—any other outcome exhibits a SHA-256 collision or an Ed25519 forgery. (Theorem 2, Proposition 1.)

G2 (Append-only with fork evidence). A consumer’s accepted view of the log only ever grows by extension; if the operator presents conflicting views to two consumers, the union of their pin stores contains publicly verifiable evidence of equivocation (two validly signed conflicting heads). (Theorem 3, Proposition 1.)

G3 (Verdict integrity). The verdict a consumer derives for a certificate depends only on the observed axiom cone in the leaf and the consumer’s *own* copy of the allowed axiom sets; the operator’s pass/fail labels have no influence. (Proposition 2.)

4.4 The residual trust, isolated

Goals G1–G3 reduce the operator’s trusted role to a single sentence: “*the reported `#print axioms` output is what the kernel printed for this commit.*” Everything else—membership, history, verdicts—is either cryptographically enforced or locally re-derived. An operator that labels a dirty cone “clean” gains nothing (G3); an attestation that omits observed cones is treated as unverifiable; an operator that rewrites history is caught with transferable evidence (G2). An operator that fabricates observations can only be caught by independent replay, which any party with a Lean toolchain can perform from the pinned commit—the design makes such an audit cheap to *target* (the claim is exact: repository, commit, toolchain, expected cones) even though it is expensive to *run*.

4.5 Verdicts are the consumer’s, not the operator’s

The design choice behind G3 is what most distinguishes this system from prior attestation transparency, so we state it as a principle rather than a mechanism. In systems like Rekor [5] a consumer learns *that* something was attested and trusts the issuer’s assessment of it; the payload’s meaning is the issuer’s to declare. Here the payload is a set of *observations*—the literal `#print axioms` output per theorem—and the assessment (*clean* or *not*) is computed by Verdict (Definition 1) from those observations against the consumer’s own table *allowed*. Concretely:

- The allowed set `allowed(c)` is not shipped by the operator at verification time; it is part of the consumer’s tooling, small enough to audit by hand (Appendix C: 7–11 axiom names per fork), and *re-derivable from first principles*—Lean’s foundational three, plus, for the

apex tiers, named placeholders for exactly those primitives the theorem deliberately leaves opaque (the hash, the wire format).

- That an independently written **allowed** meets the deployed observations *exactly* is engineered, not coincidental: the corpus is minimized so that every axiom in a cone earns its place, and any reasonable reconstruction of “what a correct proof of this statement must assume” lands on the same finite set. When the consumer’s requirement meets the supply exactly, verification is a set equality.
- When it does not—a consumer who additionally requires SHA-512 itself proven, say—the gap is exact and itemized (the boundary axioms of Appendix C), and the consumer’s options are honest: accept a *named* residual, decline, or discharge the missing boundary and let the resulting certificate enter the log. The log is additive in the same way requirements are; a stricter table is a roadmap, not a rejection.

The operator, in this picture, is not a judge whose verdict one trusts but a witness whose *observations* one re-adjudicates. G3 (§4.3, Proposition 2) is the formal statement that this re-adjudication ignores the operator’s opinion entirely.

5 The log construction

5.1 Leaves: replay attestations

A leaf is the canonical JSON serialization of an attestation recording: the subject repository URL and git commit (which cryptographically pins the entire source tree); the toolchain versions; the resource-control regime under which the replay ran; and, per certificate, its name, replay status, and the *observed axiom cone*—the exact output of Lean’s `#print axioms` for that theorem. For the corpus of §2 each attestation carries 16 certificates. Appendix A gives the leaf schema.

5.2 Boundary-exact auditing

Every certificate c has a documented allowed axiom set $\text{allowed}(c)$. Foundational certificates must carry exactly Lean’s three standard axioms (`propext`, `Classical.choice`, `Quot.sound`); the four signature-tier certificates additionally carry a per-fork, explicitly enumerated boundary (an opaque SHA-512 oracle and opaque wire-format types—e.g., eleven axioms in total for the upstream fork). Writing $\text{obs}(c)$ for the observed cone recorded in the leaf, define

$$\text{clean}(c) : \iff \text{obs}(c) = \text{allowed}(c) \quad (\text{equality of finite sets}).$$

Deviation in *either* direction—an unexpected axiom, or a missing boundary axiom—falsifies **clean**. The second direction matters: a missing boundary axiom means the theorem proved is not the theorem documented (e.g., a hash oracle discharged by a placeholder rather than kept opaque). Each source repository enforces the same discipline in its own check scripts; the log mirrors those sets, and consumers carry their own copies.

5.3 Tree, heads, receipts

Let H be SHA-256. Define, for a byte string d and 256-bit values x, y :

$$h_{\text{leaf}}(d) = H(0x00 \parallel d), \quad h_{\text{node}}(x, y) = H(0x01 \parallel x \parallel y).$$

For a leaf list $D = [d_0, \dots, d_{n-1}]$ the RFC 9162 tree head is

$$\begin{aligned} \text{MTH}([]) &= H(\varepsilon), & \text{MTH}([d]) &= h_{\text{leaf}}(d), \\ \text{MTH}(D) &= h_{\text{node}}(\text{MTH}(D[0:k]), \text{MTH}(D[k:n])), \end{aligned}$$

where k is the largest power of two strictly less than n . The *inclusion path* for index m is

$$\text{Path}(m, [d]) = [], \quad \text{Path}(m, D) = \begin{cases} \text{Path}(m, D[0:k]) \parallel [\text{MTH}(D[k:n])] & m < k, \\ \text{Path}(m - k, D[k:n]) \parallel [\text{MTH}(D[0:k])] & m \geq k, \end{cases}$$

and the consumer's root-reconstruction function $\text{Root}(v, m, n, P)$ is the evident dual (Appendix B): fold the path back up, choosing left/right by comparing m with k at each level.

Consistency. A consistency proof C lets a consumer check that a size- n_1 tree *extends* a size- n_0 tree it already pinned, $0 < n_0 \leq n_1$. We give the verifier as a function ConsRec that reconstructs *both* committed roots from C ; it is the recursive counterpart of RFC 9162 §2.1.4, and we use this form (rather than the RFC's iterative one) because the proofs of §6 induct on it. On a proof C interpreted as a list of nodes, with a flag b recording whether the size- n_0 subtree's root is carried implicitly (the pinned root) or explicitly in C :

$$\text{ConsRec}(n_0, n, C, b, r) = \begin{cases} (r, r) & n_0 = n, \ b, \ C = [], \\ (s, s) & n_0 = n, \ \neg b, \ C = [s], \\ (x, \text{h}_{\text{node}}(y, s)) & n_0 \leq k, \ C = C' \parallel [s], \\ (\text{h}_{\text{node}}(s, x'), \text{h}_{\text{node}}(s, y')) & n_0 > k, \ C = C' \parallel [s], \end{cases}$$

where k is the largest power of two below n , $(x, y) = \text{ConsRec}(n_0, k, C', b, r)$ in the third case, and $(x', y') = \text{ConsRec}(n_0 - k, n - k, C', \perp, r)$ in the fourth (any shape mismatch rejects). The consumer accepts C between signed heads (n_0, r_0) and (n_1, r_1) iff $n_0 = 0$, or $\text{ConsRec}(n_0, n_1, C, \top, r_0) = (r_0, r_1)$. We verified that this recursive form agrees with the deployed iterative RFC 9162 verifier by *exhaustive* differential testing over every pinned/current size pair $1 \leq n_0 \leq n_1 \leq 256$, each with the honest proof and four adversarial mutations (wrong old root, wrong new root, truncated and padded proofs): 164,224 verifier invocations, full agreement. The inclusion verifier of Appendix B was checked the same way (164,479 invocations over all $m < n \leq 256$).

The operator signs tree heads $(n, \text{MTH}(D), t)$ with Ed25519; a *receipt* for a leaf is its index, its sibling path, and a signed head.

5.4 The consumer pin store

Each consumer maintains a local pin $(n_{\text{pin}}, r_{\text{pin}})$, updated by the following state machine on receiving a validly signed head (n', r') :

- $n' = n_{\text{pin}}$: accept iff $r' = r_{\text{pin}}$; a mismatch is reported as *equivocation*, the pair of signed heads is retained as evidence, and the state is poisoned (unrecoverable).
- $n' > n_{\text{pin}}$: accept iff a consistency proof from $(n_{\text{pin}}, r_{\text{pin}})$ to (n', r') verifies; then update the pin.
- $n' < n_{\text{pin}}$: reject (rollback).

A freshness policy bounds head age. The full log is also published as a git repository: one file per leaf, plus the signed head history since publication began (heads signed before the mirror existed were not retained). Any cloner can therefore recompute every prefix root from the public leaves and check every published head against its prefix root and signature without consistency proofs—a low-infrastructure witness mechanism [2]; a standalone ≈ 150 -line standard-library verifier ships in the mirror.

6 Security analysis

This section proves the claims G1–G3 of §4.3. The statements are not deep—inclusion and consistency security for RFC 6962/9162 trees is folklore, and was treated formally by Dowling et al. [4]—but writing them out for *this* system serves two purposes: it pins down exactly which assumption carries which consumer-facing claim, and it produces statements in a form ready for mechanization in Lean (§10), where they will re-enter the log as leaves.

We write $\text{Root}(v, m, n, P)$ for the consumer’s root reconstruction: it is defined by $\text{Root}(v, m, 1, []) = v$ and, for $n > 1$ with k the largest power of two below n and $P = P' \parallel [s]$,

$$\text{Root}(v, m, n, P) = \begin{cases} h_{\text{node}}(\text{Root}(v, m, k, P'), s) & m < k, \\ h_{\text{node}}(s, \text{Root}(v, m - k, n - k, P')) & m \geq k, \end{cases}$$

rejecting on any length mismatch. The consumer accepts a receipt (d, m, P) against a head (n, r) iff $m < n$ and $\text{Root}(h_{\text{leaf}}(d), m, n, P) = r$.

Lemma 1 (Domain separation). *No leaf preimage equals a node preimage as a byte string: for all d, x, y , $0x00 \parallel d \neq 0x01 \parallel x \parallel y$.*

Proof. The first byte differs. □

Lemma 1 forecloses the classic cross-type confusion in which an adversary presents an interior node’s 64-byte child concatenation as a “leaf” (or vice versa) to move a value between levels of the tree [3, 4]; in the proofs below it guarantees that when two preimages of the same hash value are compared, a leaf/node type mismatch already constitutes inequality of strings, hence a collision.

Theorem 1 (Inclusion completeness). *For every non-empty leaf list D with $|D| = n$ and every $m < n$,*

$$\text{Root}(h_{\text{leaf}}(D[m]), m, n, \text{Path}(m, D)) = \text{MTH}(D).$$

Proof. Structural induction on n . For $n = 1$: $\text{Path}(0, [d]) = []$ and $\text{Root}(h_{\text{leaf}}(d), 0, 1, []) = h_{\text{leaf}}(d) = \text{MTH}([d])$. For $n > 1$ with split point k , suppose $m < k$ (the case $m \geq k$ is symmetric). Then $\text{Path}(m, D) = \text{Path}(m, D[0:k]) \parallel [\text{MTH}(D[k:n])]$, and by the induction hypothesis

$$\text{Root}(h_{\text{leaf}}(D[m]), m, k, \text{Path}(m, D[0:k])) = \text{MTH}(D[0:k]),$$

so the outer step yields $h_{\text{node}}(\text{MTH}(D[0:k]), \text{MTH}(D[k:n])) = \text{MTH}(D)$. □

Both soundness theorems below rest on a single collision-extraction fact, which we isolate first. Call a reconstruction a *hash-fold over T* if it computes its output by emitting $h_{\text{node}}(\cdot, \cdot)$ at every internal node of the honest tree T and the leaf hash at each leaf. The inclusion reconstruction $\text{Root}(h_{\text{leaf}}(\cdot), \cdot, n, \cdot)$ is a hash-fold over the size- n tree; so is the size- n_1 component of the consistency verifier **ConsRec** (§5.3). The values folded in may be adversarial; only the *shape* matches T .

Lemma 2 (Root binding). *Let F be a hash-fold over the honest Merkle tree T of a leaf list D , run on otherwise adversarial inputs (a leaf value and sibling nodes it folds in), and suppose F ’s output equals $\text{MTH}(D)$. Then either (i) at some internal node F ’s h_{node} argument pair differs from T ’s while the two h_{node} values agree—an explicit SHA-256 collision—or (ii) F ’s computation coincides with T node-for-node: every value F emits and every sibling node F consumes equals the corresponding node of T , and F ’s leaf inputs are the true leaves of D .*

Proof. Induction on the height of T . At the root both values equal $\text{MTH}(D)$ by hypothesis, and each is h_{node} of an argument pair—for F that pair is (its recursive child value, the sibling it consumed there); for T it is (MTH of the left block, MTH of the right block). Equal h_{node} values of *distinct* argument pairs are a collision (case i); otherwise the two pairs coincide *as pairs*, so simultaneously F ’s recursive child value equals T ’s child *and the sibling F consumed equals T ’s other child*, and the hypothesis applies to the child subtree. At a leaf, F folds $h_{\text{leaf}}(d')$ where T has $h_{\text{leaf}}(D[j])$; equality with $d' \neq D[j]$ is again a collision, and Lemma 1 rules out a leaf value coinciding with an internal-node value by type, so the two cases are exhaustive and exclusive. In case (ii) the argument-pair coincidence at every level is exactly the claim that both emitted values and consumed siblings match T . $\square$

Theorem 2 (Inclusion soundness: position binding). *There is an explicit algorithm $\mathcal{E}$ (running in time $O(n)$ hash evaluations) such that: whenever an adversary outputs a leaf list D with $|D| = n$, an index $m < n$, a leaf $d \neq D[m]$, and a path P with $\text{Root}(h_{\text{leaf}}(d), m, n, P) = \text{MTH}(D)$, $\mathcal{E}(D, m, d, P)$ outputs a SHA-256 collision.*

Proof. $F = \text{Root}(h_{\text{leaf}}(d), m, n, \cdot)$ applied to P is a hash-fold over the honest tree T_D (it has the size- n tree’s shape; its inputs—the leaf d and the siblings in P —are adversarial), and by hypothesis its output is $\text{MTH}(D)$. Apply Lemma 2. Case (ii) is impossible: it would require F ’s emitted leaf at position m , namely $h_{\text{leaf}}(d)$, to equal T_D ’s leaf $h_{\text{leaf}}(D[m])$, which for $d \neq D[m]$ is itself a collision. So case (i) fires; $\mathcal{E}$ recomputes T_D ($O(n)$ hashes), replays the fold to locate the differing h_{node} (or leaf) pair, and outputs it. $\square$

Remark 1. Both soundness statements are unconditional in the same sense: they do not assert forgery is infeasible, they *construct* a SHA-256 collision from any successful forgery, so append-only and position security are *precisely* “SHA-256 is collision resistant”—no more, no less. The two theorems share Lemma 2, the only place hashing is reasoned about; this factoring is deliberate, as Lemma 2 is exactly what the Lean mechanization of §10 will carry, with collision resistance entering only as a documented boundary axiom, audited by the log like the SHA-512 oracle in the Ed25519 tiers.

Theorem 3 (Consistency soundness). *There is an explicit algorithm $\mathcal{E}'$, running in $O(n_1)$ hash evaluations, such that: whenever an adversary outputs leaf lists D_0, D_1 with $|D_0| = n_0 \leq n_1 = |D_1|$ and $D_0 \neq D_1[0:n_0]$, together with a proof C that the consumer’s verifier of §5.3 accepts, i.e. $\text{ConsRec}(n_0, n_1, C, \top, \text{MTH}(D_0)) = (\text{MTH}(D_0), \text{MTH}(D_1))$, $\mathcal{E}'(D_0, D_1, C)$ outputs a SHA-256 collision.*

Proof. ConsRec returns a pair; acceptance equates its second component with $\text{MTH}(D_1)$ and its first with $\text{MTH}(D_0)$. Reading the four cases, the second component emits h_{node} at every split of the size- n_1 tree—it is a hash-fold over the honest tree T_1 —while the first component reuses a sub-list of the very same proof nodes, namely those covering the index range $[0, n_0]$, and folds *only* those. We use the two components differently, so the delicate first component never enters the lemma.

Step 1 (the transcript nodes are genuine). Apply Lemma 2 to the second component against T_1 . Either it hits case (i)—output that collision—or (case ii) every value it emitted, in particular every proof node it consumed, equals the corresponding node of T_1 . Assume the latter; the proof nodes are now known to be genuine nodes of the honest tree T_1 .

Step 2 (the prefix roots collide). The nodes covering $[0, n_0]$ are the canonical RFC 9162 decomposition of that range into maximal perfect subtrees of T_1 ; by Step 1 they are genuine, so folding them—which is exactly what the first component does—yields the root of $D_1[0:n_0]$, i.e. the first component equals $\text{MTH}(D_1[0:n_0])$. But acceptance also equates the first component with $\text{MTH}(D_0)$. Hence $\text{MTH}(D_0) = \text{MTH}(D_1[0:n_0])$ while $D_0 \neq D_1[0:n_0]$.

Step 3 (descend). Since $|D_0| = |D_1[0:n_0]| = n_0$, the two honest trees have identical shape, so $T_{D_1[0:n_0]}$ is a hash-fold over T_{D_0} (same shape; its leaf inputs are the leaves of $D_1[0:n_0]$). Its output is $\text{MTH}(D_1[0:n_0]) = \text{MTH}(D_0)$ by Step 2, so Lemma 2 applies with $D = D_0$. Case (ii) would force the leaf inputs to equal D_0 , i.e. $D_1[0:n_0] = D_0$, contradicting the premise; so case (i) fires—an explicit collision. $\mathcal{E}'$ outputs whichever collision was found; recomputing T_0 and T_1 , it runs in $O(n_1)$ hash evaluations. $\square$

Proposition 1 (Pin-store safety). *Assume Ed25519 EUF-CMA security for the head-signing key and consider the state machine of §5.4. Then, except with the probability of a signature forgery or a SHA-256 collision:*

1. (Monotonicity) *If a consumer’s pin evolves through states $(n_1, r_1), \dots, (n_t, r_t)$, then $n_1 \leq \dots \leq n_t$, and for any leaf lists D_i the operator can exhibit with $\text{MTH}(D_i) = r_i$, $|D_i| = n_i$, each D_i is a prefix of D_{i+1} .*
2. (Fork evidence) *If two consumers with the same pinned key ever hold accepted heads (n, r) and (n, r') with $r \neq r'$, the pair of signed heads is transferable, publicly verifiable evidence that the key holder signed two conflicting views.*

Proof. (1) The machine accepts a larger size only with a verified consistency proof, so by Theorem 3 any exhibited leaf lists are prefix-ordered unless a collision is found; rollback is rejected syntactically. (2) Both heads carry valid signatures under the pinned key; under EUF-CMA, both were produced by the key holder, and $r \neq r'$ at equal size is precisely a split view. The evidence is transferable because verification requires only the public key. $\square$

Proposition 2 (Verdict integrity). *Fix a consumer with local allowed-set table $\text{allowed}(\cdot)$. For every attestation leaf a and certificate c in it, the consumer’s verdict is the predicate $\text{clean}(c) \Leftrightarrow \text{obs}_a(c) = \text{allowed}(c)$, a function of the leaf’s observed cones and the consumer’s table only. In particular, the operator’s embedded pass/fail labels are not an input, and for fixed leaf content the verdict is independent of anything the operator asserts.*

Proof. By construction of the consumer tooling: the verdict function takes $(\text{obs}_a, \text{allowed})$ and ignores the label fields in every branch; a certificate lacking an observed cone is mapped to **unverifiable**, not to a verdict. The operator’s proven/failed **status** label is consulted separately by the acceptance policy, but only *negatively*—a certificate the operator does not itself mark proven can never count—so no operator assertion can upgrade any verdict. $\square$

What is *not* proven. Propositions 1 and 2 together with Theorems 1–3 discharge G1–G3. They do not—and cannot—exclude an operator who fabricates observations (§4.4), and they say nothing about the mathematical content of the attested corpus, whose guarantees rest on the Lean kernel and the assumption set enumerated in §8. The division of labor is deliberate: the cryptographic layer makes the operator’s claims *exact, immutable, and attributable*; the deductive layer is what makes them *true*.

7 The self-referential signing loop

Tree heads are Ed25519 signatures, and this creates an opportunity for coherence: the log contains correctness certificates for an Ed25519 implementation. The LTL’s heads are therefore signed by a binary built from the pinned source tree of exactly the implementation attested in the log (serial backend pinned, matching the verified extraction), and—before signing—the operator runs the same Merkle inclusion verification a consumer runs, on the newest leaf attesting the signing implementation, against the tree about to be signed. The verdict is embedded in the signature block:

```

signing_backend: verified-dalek-serial
signing_library_source_commit: aa0f6ab...
signing_library_leaf_index: 8
signing_library_certificates_proven: 16/16
self_inclusion: verified

```

(These provenance fields ride alongside the signature as operator-provided context; they are not part of the signed payload, and a consumer relies on none of them—the acyclic chain below rests only on the signature and the leaf’s inclusion.) The signature vouches for the tree; the tree vouches for the code that produced the signature; and the two vouchings are different proof modalities (cryptographic and deductive), so the loop is self-referential without being circular. Concretely, a consumer’s verification order is a directed acyclic chain, no step trusting its own output: pin the operator key (assumed, once) → check the head signature (EUF-CMA) → verify the signing library’s leaf is included in that head (hashes only, no signature) → optionally rebuild that library from its pinned commit and re-check its certificates (Lean kernel). The self-reference is only that the code producing signatures also *appears as a subject* in the log; no check consumes the result it is establishing. The self-check always references the *newest* leaf attesting the signing library: after the re-attestation of §8, the referenced index advanced from 4 to 8 automatically, the loop re-anchoring itself to the fresh attestation without operator intervention.

The honest extent of this claim. The Lean certificates cover the *verification* path of the library (the theorems’ subject is the extraction image of that path); the *signing* path is not covered by any certificate and is declared trusted base. The design merely ensures the trusted signing code is the attested artifact rather than an unrelated third implementation, and that consumers can check as much. Signature verification on consumer machines can optionally run through the same certified-source binary, with the backend that actually ran recorded in every result and a fail-closed policy flag available. First-append bootstrapping is handled honestly: heads signed before the signing library’s attestation enters the log record `self_inclusion: library_not_in_log`.

8 Deployment and evidence

The LTL is deployed² with twelve leaves, produced by three full replay runs (one attestation per fork per run; 58–64 Lean files and ≈1,800s per fork, under hard memory caps and core pinning). Each successful run reports 16/16 certificates proven with boundary-exact cones, pinned to exact commits. The three runs correspond to three states of the world, and their coexistence in one append-only ledger is the point of the system:

- **Leaves 0–3 (failed run).** The first run’s audit step failed on two defects in the operator tooling (a path issue and a parser that mishandled Lean’s line-wrapped axiom lists for the eleven-axiom cones). The operator signed attestations *recording the failure* rather than suppressing the run. Both defects were fail-closed: valid proofs were rejected, invalid ones never accepted.
- **Leaves 4–7 (clean run).** After the fix, all four forks attested 16/16 boundary-exact at that day’s commits.

²Service: <https://l1l.zkdefi.org> (read-only HTTP API and documentation). Mirror: <https://github.com/saymrwulf/lean-transparency-log>. Operator and consumer tooling: <https://github.com/saymrwulf/proof-aware-crypto-tooling-agent>. The underlying proof corpora are in the `saymrwulf/*-ed25519-verified` repositories; every claim in this paper is re-checkable from these artifacts.

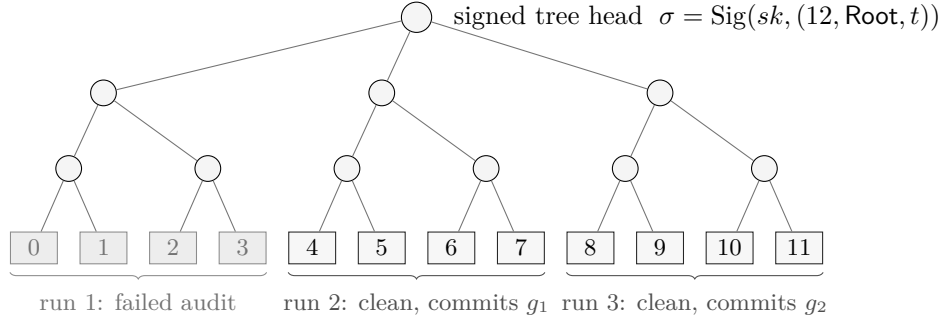


Figure 1: The deployed twelve-leaf log. Grey leaves 0–3 record the first run’s audit failure (retained, not erased); leaves 4–7 and 8–11 are two clean runs, at commit generations g_1 and g_2 across a subject-history rewrite. The internal layer is drawn schematically; the true shape is the RFC 9162 tree of §5.3. Every value in the figure is recomputable from the public leaves.

fork	Lean files	apex-boundary axioms	SHA-512 in the boundary
upstream dalek	64	11	3-call streaming (new/update/finalize)
Solana (anza)	58	7	one <code>ed_sigs.sha512.hash3</code>
RISC Zero	63	8	one <code>verifying.sha512.hash3</code>
Betrusted	63	8	one <code>verifying.sha512.hash3</code>

Table 1: The four subject implementations. Each replay re-checks 16 certificates in $\approx 1,800$ s under memory caps and core pinning. The apex-boundary count is the size of the fork’s enumerated oracle boundary (Appendix C); it differs by fork because the SHA-512 surface and the byte-accessor shape differ. Proof-script divergence across forks is quantified in the portability paragraph below; the pure-mathematics files are byte-identical across all four.

- **Leaves 8–11 (clean run, new commits).** A subsequent documentation-only rewrite of the subject repositories’ histories changed their commit hashes. Because a leaf pins an exact commit (§5.1), the operator re-ran the full corpus and appended fresh attestations at the new commits rather than editing leaves 4–7. That the proof *files* survived the rewrite unchanged is corroborated from the log itself: leaves 4–7 and 8–11 carry identical certificate lists and identical observed axiom cones, re-checked by the kernel at both commit generations. (The pre-rewrite trees themselves are no longer distributed, so a direct tree diff is not among the public artifacts.)

This last event is a live exercise of the append-only discipline (G2): a change that a naive operator would have hidden by overwriting is instead absorbed by *addition*, leaving a permanent, publicly verifiable record that the subject histories changed and that the mathematics survived the change. The ledger—four failure leaves and eight success leaves across two commit generations—is a feature of the trust model, not clutter to be pruned (Figure 1).

What a verified receipt establishes. Under the assumptions enumerated below, a consumer who verifies a receipt knows: *the operator whose key I pinned attests that the Lean certificates of repository X at commit Y re-check, with per-certificate observed axiom cones as included—and this statement is part of the log presented to every other consumer.* Combined with local verdict re-derivation (Proposition 2), this yields source-level assurance for the pinned commit. It deliberately does *not* establish: correctness of any binary (consumers build from the pinned source; compilers are trusted base), correctness of SHA-512 (an opaque oracle in the theorems), correctness of the wire-format parsers (their outcomes are hypotheses

of the signature tiers), signing-side correctness, or side-channel properties.

The assumption set, in full. The Lean kernel and its three axioms plus mathlib; faithfulness of the Charon/Aeneas extraction [11]; each fork’s documented oracle boundary; operator key custody and trust-on-first-use key distribution (mitigated by publishing the key in two independent locations); collision resistance of SHA-256 for the log (Theorems 2, 3); unforgeability of Ed25519 for the heads (Proposition 1); and the consumer’s own ≈ 25 -line verifier (Appendix B).

An observational by-product: proof portability. Because the four corpora prove the same theorems against four independent extractions, the diff between proof files measures how portable proofs are across real forks. Pure-mathematics files (e.g., a carry-telescope lemma file) are byte-identical across all four; extraction-facing proof scripts diverge sharply where the forks’ code or the extractor’s naming differs (e.g., 215 changed lines for the byte-parser proofs on the two forks whose extraction produces a closure-based loader; 121 lines for the signature-glue proofs on the same-crate fork; 27 lines between the two structurally closest forks, tracking one fork’s `black_box` optimization barrier—its axiom-list entries and the operation reordering it induces). Per-target verification, in other words, is doing measurable work exactly where the targets actually differ.

9 Limitations

The deployment is small (one operator, twelve leaves, four subject repositories) and the operator is a single party; split-view defense currently rests on consumer-side pinning (Proposition 1) plus the public git mirror rather than an independent witness network. Key distribution is trust-on-first-use. The signing path of the dogfood binary is unverified (declared, not proven). The residual trust of §4.4—honesty of the operator’s kernel observations—is mitigated only by targeted independent replay. The corpus itself stops at source-level assurance: reproducible builds and side-channel evidence remain open, and ML-DSA slots in the head format are deliberately recorded as unavailable rather than backed by an unverified implementation.

10 Next step: verifying the accumulator itself

The natural continuation applies the corpus’s own discipline to the log’s cryptographic half. Theorems 1–3 and Proposition 1 were stated so that their mechanization is a translation task, not a research task: (i) inclusion completeness (Theorem 1) is assumption-free; (ii) inclusion soundness becomes the explicit extractor of Theorem 2, with SHA-256 collision resistance a documented boundary axiom audited exactly like the SHA-512 oracle in the Ed25519 tiers; (iii) likewise consistency (Theorem 3); (iv) domain separation (Lemma 1) is a one-line lemma; and (v) total correctness of the consumer’s pin-store state machine (Proposition 1). Verified Merkle implementations in F^* [14] and machine-checked transparency-protocol analyses [9] show these proofs are well within reach; the LTL-specific closure is where the certificates go: *into the log they defend, checked by the certified checker they specify*, alongside a consumer policy flag requiring the certified verifier. At that point both proving traditions in the composition run on certified code, and the remaining trusted base is two hash assumptions, a compiler, an extraction pipeline, and one key.

Acknowledgments

The author designed the system, directed the verification effort, and is solely accountable for every claim in this paper. Claude (Anthropic) was used as an assistant in developing the proof corpora, tooling, and text; all proofs, measurements, and claims have been reviewed by the author and are independently re-checkable from the public artifacts and the referenced check scripts.

References

- [1] B. Laurie, A. Langley, E. Käsper. Certificate Transparency. RFC 6962, 2013.
- [2] B. Laurie, E. Messeri, R. Stradling. Certificate Transparency Version 2.0. RFC 9162, 2021.
- [3] S. A. Crosby, D. S. Wallach. Efficient Data Structures for Tamper-Evident Logging. *USENIX Security*, pp. 317–334, 2009.
- [4] B. Dowling, F. Günther, U. Herath, D. Stebila. Secure Logging Schemes and Certificate Transparency. *ESORICS, LNCS 9879*, pp. 140–158, 2016.
- [5] Z. Newman, J. S. Meyers, S. Torres-Arias. Sigstore: Software Signing for Everybody. *ACM CCS*, pp. 2353–2367, 2022.
- [6] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, J. Cappel. in-toto: Providing farm-to-table guarantees for bits and bytes. *USENIX Security*, pp. 1393–1410, 2019.
- [7] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, M. J. Freedman. CONIKS: Bringing Key Transparency to End Users. *USENIX Security*, pp. 383–398, 2015.
- [8] G. C. Necula. Proof-Carrying Code. *ACM POPL*, pp. 106–119, 1997.
- [9] V. Cheval, J. Moreira, M. Ryan. Automatic verification of transparency protocols. *IEEE EuroS&P*, pp. 107–121, 2023. arXiv:2303.04500.
- [10] G. Barthe, B. Grégoire, S. Heraud, S. Zanella Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. *CRYPTO, LNCS 6841*, pp. 71–90, 2011.
- [11] S. Ho, J. Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6 (ICFP): 711–741, 2022.
- [12] L. de Moura, S. Ullrich. The Lean 4 Theorem Prover and Programming Language. *CADE-28, LNCS 12699*, pp. 625–635, 2021.
- [13] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, B. Beurdouche. HACl*: A Verified Modern Cryptographic Library. *ACM CCS*, pp. 1789–1806, 2017.
- [14] J. Protzenko et al. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. *IEEE S&P*, pp. 983–1002, 2020.
- [15] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, A. Chlipala. Simple High-Level Code for Cryptographic Arithmetic—With Proofs, Without Compromises. *IEEE S&P*, pp. 1202–1219, 2019.
- [16] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.-Y. Yang. High-speed high-security signatures. *J. Cryptographic Engineering* 2(2): 77–89, 2012.
- [17] S. Josefsson, I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, 2017.

- [18] D. J. Bernstein, T. Lange. Faster addition and doubling on elliptic curves. ASIACRYPT, LNCS 4833, pp. 29–50, 2007.
- [19] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, C. Peters. Twisted Edwards curves. AFRICACRYPT, LNCS 5023, pp. 389–405, 2008.
- [20] A. Pnueli. The temporal logic of programs. IEEE FOCS, pp. 46–57, 1977.

A Leaf schema

Each leaf is the canonical JSON serialization (sorted keys, no insignificant whitespace, UTF-8) of an attestation. Below is leaf 8 of the deployed log—the re-attestation of the upstream fork. The 16-certificate array is elided to its first (foundational) and last (apex) entries; long values (hashes, timestamps, version strings, paths) are shortened, and omitted fields are marked, with ellipses. The field names and values shown, and the axiom lists, are verbatim, and the unelided leaf is one jq invocation away in the public mirror.

```
{ "type": "pacta.attestation", "schema_version": 1,
  "attestation": {
    "provider": "local-pacta-provider",
    "issued_at": "2026-07-07T...Z",
    "subject": { "component": "dalek-ed25519-verified",
      "repo_commit": "33fb8bb2311c70ead2e83c0...",
      "repo_url": "...", "verified_backend": "serial/u64",
      ... },
    "environment": {
      "lean_version": "Lean (version 4.30.0-rc2, ...)",
      "lake_version": "...", "env_script": "...",
      "lean_project_dir": ... },
    "machine_protection": { "lean_guard": ...,
      "note": "All Lean compiles route through the
        repo's lean-guard (memory cap, core pinning,
        timeout, single-flight lock) ..." },
    "replay": { "checked_files": 64, "failed_files": [],
      "check_ok": true, "axiom_ok": true, ... },
    "certificates": [
      { "name": "CurveFieldProofs.fieldImplementation",
        "status": "proven", "axiom_status": "clean",
        "observed_axioms": ["propext",
          "Classical.choice", "Quot.sound"],
        "expected_axioms": [...] },
      ... (14 more) ...
      { "name":
        "CurveFieldProofs.verify_accepts_iff_decompress",
        "status": "proven", "axiom_status": "clean",
        "observed_axioms": ["propext", "Classical.choice",
          "Quot.sound", "ed25519.Signature", "sha2.Sha512",
          "verifying.sha512.finalize_bytes",
          "verifying.sha512_new", "verifying.sha512_update",
          "ed25519.Signature.to_bytes",
          "signature.error.Error",
          "signature.error.Error.new"] } ],
    "signature": { "scheme": "openssl-ed25519", ... } } }
```

The `observed_axioms` field is the exact output of `#print axioms` for that theorem. Operator labels (`replay.check_ok`, per-certificate `status` and `axiom_status`) are recorded for the audit trail, but the cleanliness verdict is `obs = allowed` computed against the consumer's own table in every case, with a missing cone mapped to unverifiable (Proposition 2). The `status` label is consulted only *negatively*: a certificate the operator itself does not mark proven can never count toward acceptance, so labels can deny but never grant.

B The consumer verifier

The consumer-side inclusion check, in full (Python, standard library only); this is the recursive form proved in §6 and is equivalent to the iterative algorithm of RFC 9162 §2.1.3.2.

```
import hashlib
def H(b): return hashlib.sha256(b).digest()
def h_leaf(d): return H(b'\x00' + d)
def h_node(x, y): return H(b'\x01' + x + y)
def largest_pow2_below(n):
    k = 1
    while 2 * k < n: k *= 2
    return k
def root(v, m, n, path):
    if n == 1:
        if path: raise ValueError
        return v
    if not path: raise ValueError
    *rest, s = path
    k = largest_pow2_below(n)
    if m < k:
        return h_node(root(v, m, k, rest), s)
    return h_node(s, root(v, m - k, n - k, rest))
def verify_inclusion(leaf, m, n, path, head_root):
    return m < n and root(h_leaf(leaf), m, n, path) == head_root
```

Signature verification of the head (Ed25519) and the pin-store logic of §5.4 complete the consumer; the deployed ≈ 150 -line standalone verifier in the mirror additionally checks consistency proofs and recomputes prefix roots from the public leaves.

C Allowed axiom sets

Foundational certificates (12 of 16) must carry exactly Lean’s three standard axioms:

```
propext    Classical.choice    Quot.sound
```

The four signature-tier certificates additionally carry a per-fork enumerated boundary: an opaque SHA-512 oracle and opaque wire-format types (the signature type, its byte accessors, and the error type). The boundary is not identical across forks—it reflects each fork’s actual extracted surface—and auditing is exact against the fork’s own set. The three distinct boundaries in the deployed corpus, verbatim from the repositories’ check scripts, are as follows (the three standard axioms above, plus):

Upstream curve25519-dalek (11 axioms total; this fork exposes SHA-512 as three streaming operations):

```
ed25519.Signature    sha2.Sha512
verifying.sha512_finalize_bytes
verifying.sha512_new    verifying.sha512_update
ed25519.Signature.to_bytes
signature.error.Error    signature.error.Error.new
```

RISC Zero and Betrusted forks (8 axioms total; identical to each other—SHA-512 is a single hash3 oracle):

```
ed25519.Signature    verifying.sha512_hash3
ed25519.Signature.to_bytes
signature.error.Error    signature.error.Error.new
```

Solana (anza) fork (7 axioms total; its own `ed_sigs` namespace, and `R/s` byte accessors rather than a whole-signature encoder):

```
ed25519.Signature    ed_sigs.sha512.hash3
ed25519.Signature.r_bytes  ed25519.Signature.s.bytes
```

A consumer’s local table (§5.2) contains exactly these sets. That a boundary differs by fork is itself audited: an upstream-shaped cone appearing under the `anza` label, or vice versa, fails clean in the “unexpected axiom” direction.

D The four verification tiers: Lean theorem names

The lifting ladder T1–T4 and the mathematical facts it turns on are stated in §2. For reproducibility we record here the verbatim Lean theorem name backing each tier in the upstream corpus (the forks use the same names against their own extractions); a reader can `#print axioms` any of these to reproduce the cones of Appendix C.

tier (§2)	Lean theorem
T1 byte apex	<code>verify_accepts_iff</code>
T2 canonical half-lift	<code>verify_accepts_iff_point</code>
T3 injectivity / point eq.	<code>verify_accepts_iff_point_eq</code>
T4 constructive full lift	<code>verify_accepts_iff_decompress</code>

All four are proven under the wire-format hypotheses $\mathcal{W}$ of §2; their conjunction is what an R4 consumer relies on, and all four cones are audited against the same per-fork boundary of Appendix C.