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 (same-size equivocation yields transferable evidence, and local pinning rejects inconsistent extensions), 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 published alongside every signed head; 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 lack distribution infrastructure, and we are unaware of a deployed transparency log designed to carry kernel-replay attestations together with theorem-level assumption boundaries.[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 $[\![(a_0, \ldots, a_4)]\!] = \sum_i a_i 2^{51i} \bmod p$ 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.

$$\forall a\, b.\ \mathsf{bnd}\, a \Rightarrow \mathsf{bnd}\, b \Rightarrow \exists c.\ \mathsf{mul}\, a\, b = \mathsf{ok}\, c \wedge \mathsf{bnd}\, c \wedge [\![c]\!] = [\![a]\!] \cdot [\![b]\!].$$

[1] The 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 $E : -x^2 + y^2 = 1 + d\,x^2y^2$ over $\mathbb{F}_{2^{255}-19}$,

$$(x_1, y_1) + (x_2, y_2) \;=\; \left(\frac{x_1y_2 + x_2y_1}{1 + d\,x_1x_2y_1y_2},\; \frac{y_1y_2 + x_1x_2}{1 - d\,x_1x_2y_1y_2}\right),$$

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 $\mathsf{accept}(A, m, R, s)$ 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 32-byte R component exactly as it appears in the signature (raw bytes; no canonicity of them is presupposed). 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 part of the open frontier recorded in §9).

T1 (byte apex). $\mathsf{accept}(A, m, R, s) \Leftrightarrow \mathsf{compress}([s]B - [k]A) = r_1$. 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 $\mathsf{compress}([s]B - [k]A)$ *are* the canonical encoding of the group element $[k](-A) + [s]B$; that is, $\mathsf{compress}$ agrees on this input with the mathematical canonical-encoding function. T1 and T2 give $\mathsf{accept} \Leftrightarrow \mathsf{enc}([k](-A) + [s]B) = r_1$.

T3 (injectivity / point equation). Canonical encodings are injective on $E(\mathbb{F}_{2^{255}-19})$: if a valid curve point P has $\mathsf{enc}(P) = r_1$ 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). $\mathsf{accept}(A, m, R, s) \Leftrightarrow \mathsf{decompress}(R) = [k](-A) + [s]B$, with the extracted $\mathsf{decompress}$ proven to realize the mathematical inverse of enc: exact byte parsing, the $(p+3)/8$-power square root, and sign-bit root selection (for $x \neq 0$ the two roots x and $p - x$ differ in parity since p is odd, so the stored sign bit selects correctly; at $x = 0$ the roots coincide and a set sign bit is rejected, per RFC 8032—the theorem, an *iff* over the extracted code, covers this branch by construction).

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). Rekor 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: the Merkle inclusion core is roughly 25 lines of standard-library code (Appendix B); the full standalone consumer—head signature, consistency, mirror audit—is ≈150 lines (§5.4), atop an Ed25519 backend. 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*:*

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

MTH, ProveIncl/VerifyIncl *and* ProveCons/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, and two accepted heads of *equal* tree size with different roots are, together, transferable publicly verifiable evidence of equivocation. Unequal-size split views are not exposed by the head pair alone; they are exposed by the public leaf mirror (§5.4), from which any party recomputes every prefix root (itself operator-published, hence witness-dependent; §9), or by an external witness. (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 can deny (a certificate the operator does not itself mark proven never counts) but can never grant. (Proposition 2.)

4.4 The residual trust, isolated

Goals G1–G3 reduce the operator's trusted role to a single sentence, which we state at its honest width: *"the operator executed the declared replay procedure against the exact pinned source and dependency state, using the declared toolchain, and bound the resulting kernel outputs faithfully to the correct theorem entries of the attestation."* Checkout, dependency state, theorem-to-entry binding, and output parsing are all inside this observation pipeline—the first deployed run failed on precisely such a defect (§8). 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 $\mathsf{allowed}(c)$ 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 *up to naming* from the theorem statements—Lean's foundational three, plus, for the apex tiers, placeholders for exactly those primitives the theorem deliberately leaves opaque (the hash, the wire format); the placeholder *names* themselves are fixed by the fork's extracted surface and read off from Appendix C.
- 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," once the fork's extraction naming is fixed, 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 takes no positive input from the operator's opinion: labels act, if at all, only as a conservative veto.

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 $\mathsf{allowed}(c)$. 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 $\mathsf{obs}(c)$ for the observed cone recorded in the leaf, define

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

Deviation in *either* direction—an unexpected axiom, or a missing boundary axiom—falsifies clean. The second direction matters for these *oracle* boundaries: a missing boundary axiom signals that the theorem no longer consumes a primitive it deliberately left opaque. The verifier does not attempt to distinguish the readings of that drift (a genuinely strengthened proof; a changed theorem; a hash oracle discharged by a placeholder rather than kept opaque; stale policy): it refuses to classify, and rejects. 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:

$$\mathsf{h}_{\mathsf{leaf}}(d) = \mathsf{H}(\texttt{0x00} \,\|\, d), \qquad \mathsf{h}_{\mathsf{node}}(x, y) = \mathsf{H}(\texttt{0x01} \,\|\, x \,\|\, y).$$

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

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

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

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

and the consumer's root-reconstruction function $\mathsf{Root}(v, m, n, P)$ 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 $\mathsf{ConsRec}$ 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:

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

where k is the largest power of two below n, $(x, y) = \mathsf{ConsRec}(n_0, k, C', b, r)$ in the third case, and $(x', y') = \mathsf{ConsRec}(n_0 - k, n - k, C', \bot, r)$ 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 $\mathsf{ConsRec}(n_0, n_1, C, \top, r_0) = (r_0, r_1)$. 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 $(n, \mathsf{MTH}(D), t)$ with Ed25519; a *receipt* for a leaf is its index, its sibling path, and a signed head. The signed payload is not the bare triple

but the canonical JSON serialization (sorted keys, fixed separators, UTF-8—injective on the field set) of the head record, which additionally carries a protocol version tag (`pacta.transparency.signed_tree_head.v1`) and the log identity; a head signature therefore transfers neither across logs nor across protocol versions.

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; freshness, however, is an availability policy, not an append-only property—the construction detects rollback relative to a persisted pin, but does not prove that a consumer sees the newest issued head (an operator can re-issue fresh timestamps over a frozen tree). 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 $\mathsf{Root}(v, m, n, P)$ for the consumer's root reconstruction: it is defined by $\mathsf{Root}(v, m, 1, [\,]) = v$ and, for $n > 1$ with k the largest power of two below n and $P = P' \| [s]$,

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

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

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

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]; it guarantees that whenever a leaf preimage and a node preimage are compared, they already differ as strings, so equal hash values across the two types constitute a collision.

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

$$\mathsf{Root}(\mathsf{h}_{\mathsf{leaf}}(D[m]),\, m,\, n,\, \mathsf{Path}(m, D)) = \mathsf{MTH}(D).$$

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

$$\mathsf{Root}(\mathsf{h}_{\mathsf{leaf}}(D[m]),\, m,\, k,\, \mathsf{Path}(m, D[0{:}k])) = \mathsf{MTH}(D[0{:}k]),$$

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

Both soundness theorems below rest on a single collision-extraction fact, which we isolate first. Fix the honest Merkle tree T of a leaf list D. A *hash-fold over T* is a computation shaped by a connected sub-tree S of T containing T's root: at every internal node of T lying in S it emits $\mathsf{h}_{\mathsf{node}}$ of its two children's values; each child lying outside S is an *input*, consumed as an opaque value; at every leaf of T lying in S it emits $\mathsf{h}_{\mathsf{leaf}}$ of an input leaf value. All inputs may be adversarial; only the shape is T's. Three instantiations recur below: the inclusion reconstruction $\mathsf{Root}(\mathsf{h}_{\mathsf{leaf}}(\cdot), m, n, \cdot)$ (S is the root path of leaf m; the consumed inputs are the path's siblings); the new-root component of the consistency verifier $\mathsf{ConsRec}$ (§5.3) (S reaches down to the perfect subtrees covering $[0, n_0)$; the consumed inputs are the proof nodes and, on the leftmost spine, the pinned root); and the honest computation of $\mathsf{MTH}(D')$ for any D' with $|D'| = |D|$ (S is all of T; the inputs are the leaves of D').

Lemma 2 (Root binding)**.** *Let F be a hash-fold over the honest Merkle tree T of a leaf list D, and suppose F's output equals $\mathsf{MTH}(D)$. Then either (i) at some node of S, F's hash argument differs from T's while the two hash values agree—an explicit SHA-256 collision—or (ii) F's computation coincides with T node-for-node: every value F emits,* every input it consumes*, and every leaf input it takes equals, respectively, the corresponding node value of T and the corresponding leaf of D.*

Proof. Top-down induction on S, maintaining at each visited node the invariant that F's value there equals T's. At the root both equal $\mathsf{MTH}(D)$ by hypothesis. At an internal node of S where the invariant holds, both values are $\mathsf{h}_{\mathsf{node}}$ of an argument pair (65-byte preimages); if the pairs differ we are in case (i); if they coincide, each child's value is pinned: a child inside S inherits the invariant and we recurse, while a child outside S is a consumed input now known to equal T's node value there—no descent needed. At a leaf of S the invariant reads $\mathsf{h}_{\mathsf{leaf}}(d') = \mathsf{h}_{\mathsf{leaf}}(D[j])$: either $d' = D[j]$, or the two leaf preimages differ and we are in case (i). If case (i) never fires, the accumulated equalities at every node of S are exactly claim (ii). Because F's shape is T's, every comparison above is leaf-to-leaf or node-to-node; Lemma 1 additionally ensures that even a cross-type value coincidence would be a collision of distinct strings, which matters in the deployed protocol, where the same hash function commits leaves and nodes across trees of attacker-influenced sizes [3, 4]. □

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 $\mathsf{Root}(\mathsf{h}_{\mathsf{leaf}}(d), m, n, P) = \mathsf{MTH}(D)$, $\mathcal{E}(D, m, d, P)$ outputs a SHA-256 collision.*

Proof. $F = \mathsf{Root}(\mathsf{h}_{\mathsf{leaf}}(d), m, n, \cdot)$ applied to P is a hash-fold over the honest tree T_D whose sub-tree S is the root path of leaf m, with consumed inputs the entries of P and leaf input d; by hypothesis its output is $\mathsf{MTH}(D)$. Apply Lemma 2. Case (ii) includes the claim that the leaf input equals $D[m]$, contradicting $d \neq D[m]$; so case (i) fires. $\mathcal{E}$ recomputes T_D ($O(n)$ hashes), replays the fold to locate the disagreeing pair, and outputs it. □

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 $|D_0| = n_0 \le n_1 = |D_1|$ 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.* $\mathsf{ConsRec}(n_0, n_1, C, \top, \mathsf{MTH}(D_0)) = (\mathsf{MTH}(D_0), \mathsf{MTH}(D_1))$*, $\mathcal{E}'(D_0, D_1, C)$ outputs a SHA-256 collision.*

Proof. ConsRec returns a pair; acceptance equates its second component with $\mathsf{MTH}(D_1)$ and its first with $\mathsf{MTH}(D_0)$. Reading the four cases, the second component emits $\mathsf{h}_{\mathsf{node}}$ at every split it traverses of the size-n_1 tree and bottoms out on consumed values—it is a hash-fold over the honest tree T_1, with consumed inputs the proof nodes and, on the leftmost spine, the pinned root—while the first component reuses a sub-list of those same values, namely the ones 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 values 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 and every input it consumed—each proof node, and the pinned root where the fold bottoms out on it—equals the corresponding node of T_1. Assume the latter; the consumed values are now known to be genuine nodes of the honest tree T_1.

Step 2 (the prefix roots collide). The consumed values covering $[0, n_0)$ sit at 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 (degenerately, when the old tree is itself a perfect subtree of T_1, the "fold" is the consumed pinned root alone)—yields the root of $D_1[0{:}n_0]$, i.e. the first component equals $\mathsf{MTH}(D_1[0{:}n_0])$. But acceptance also equates the first component with $\mathsf{MTH}(D_0)$. Hence $\mathsf{MTH}(D_0) = \mathsf{MTH}(D_1[0{:}n_0])$ 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 the honest computation of $\mathsf{MTH}(D_1[0{:}n_0])$ is a hash-fold over T_{D_0} (S the whole tree; leaf inputs the leaves of $D_1[0{:}n_0]$). Its output is $\mathsf{MTH}(D_1[0{:}n_0]) = \mathsf{MTH}(D_0)$ 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. □

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 $(n_1, r_1), \ldots, (n_t, r_t)$, then $n_1 \le \cdots \le n_t$, and for any leaf lists D_i the operator can exhibit with $\mathsf{MTH}(D_i) = r_i$, $|D_i| = n_i$, 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.*

consumer conclusion	established by	remaining assumption
leaf bytes sit at index m under head h	inclusion proof (Thm 2)	SHA-256 collision resistance; authentic head
head was authorized under the log key	Ed25519 verification	correct key pin; EUF-CMA
new local head extends the old one	consistency proof (Thm 3)	SHA-256 collision resistance
equal-size heads conflict: equivocation	two valid signatures, unequal roots (Prop 1)	correct key pin
observed cone matches local policy	set equality (Prop 2)	semantic identity of the named declarations at the pinned commit
the kernel produced the observation	operator replay attestation	replay-pipeline honesty, or independent replay (§4.4)
deployed binary matches verified source	*not established*	reproducible build / binary attestation
signing binary is the claimed implementation	*not established*	execution provenance (§7)

Table 1: The end-to-end claim matrix. Every consumer conclusion, what establishes it, and what remains assumed. The last two rows are deliberate non-claims (§1, §9).

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. □

Proposition 2 (Verdict integrity)**.** *Fix a consumer with local allowed-set table* $\mathsf{allowed}(\cdot)$*. For every attestation leaf a and certificate c in it, the consumer's* cleanliness verdict *is the predicate* $\mathsf{clean}(c) \Leftrightarrow \mathsf{obs}_a(c) = \mathsf{allowed}(c)$*, a function of the leaf's observed cones and the consumer's table only; the operator's embedded pass/fail labels are not an input to it. The consumer's* acceptance policy *consults those labels at most negatively: no operator assertion can upgrade any verdict or acceptance.*

Proof. By construction of the consumer tooling: the verdict function takes $(\mathsf{obs}_a, \mathsf{allowed})$ and ignores the label fields in every branch; a certificate lacking an observed cone is mapped to $\mathsf{unverifiable}$, not to a verdict. The acceptance policy applies the operator's proven/failed `status` label only as a veto—a certificate the operator does not itself mark proven can never count—and a veto cannot upgrade; hence labels can deny but never grant. □

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*. Table 1 decomposes the end-to-end chain: each consumer conclusion, the mechanism that establishes it, and the assumption that remains. The architecture does not pretend to eliminate trust; it decomposes trust into independently visible components—including two rows it deliberately does *not* establish.

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 deployed operator *enforces and records* the invariant that the signing binary is built from the attested artifact rather than an unrelated third implementation; a consumer can check that the claimed source is attested in the signed tree, but—an Ed25519 signature reveals nothing about the program that produced it—cannot independently establish that this binary produced a given signature. Establishing that would require reproducible builds or execution attestation (§9). 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 deployed[2] with twelve leaves, produced by three full replay runs (one attestation per fork per run; 58–64 Lean files and ≈1,800 s per fork, under hard memory caps and core

[2]Service: https://ltl.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.

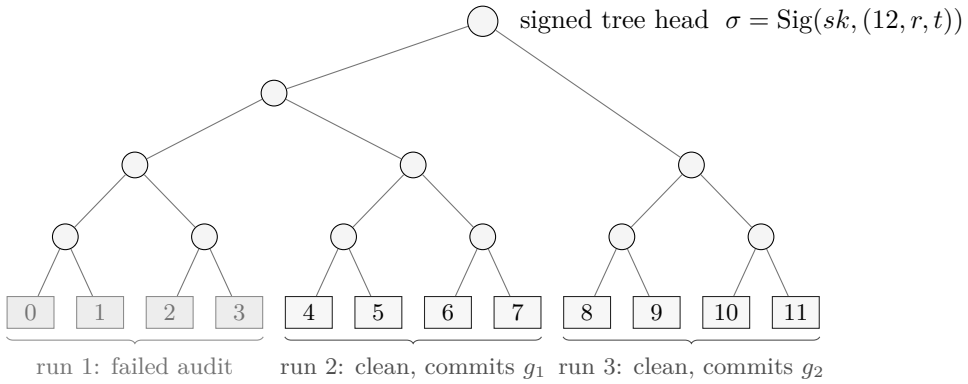

Figure 1: The deployed twelve-leaf log. Grey leaves 0–3 record the first run's audit failure (retained, not erased); leaves 4–7 and 8–11 are two clean runs, at commit generations g_1 and g_2 across a subject-history rewrite. The interior is the exact RFC 9162 shape of §5.3 for $n = 12$ (root split $8 \mid 4$; r denotes the root value). Every value in the figure is recomputable from the public leaves.

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.
- **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

fork	Lean files	apex cone (axioms, total)	SHA-512 in the boundary
upstream `dalek`	64	11	3-call streaming (`new/update/finalize`)
Solana (`anza`)	58	7	one `ed_sigs.sha512_hash3`
RISC Zero	63	8	one `verifying.sha512_hash3`
Betrusted	63	8	one `verifying.sha512_hash3`

Table 2: The four subject implementations. Each replay re-checks 16 certificates in ≈1,800 s under memory caps and core pinning. The apex-cone count is the full allowed axiom set at the signature tiers—Lean's three standard axioms plus 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.

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 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 as to make their mechanization direct; we expect the principal work to be specification alignment and proof engineering rather than new cryptographic argument: (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, and adversarial reviews by both Claude and GPT (OpenAI) shaped the final manuscript; 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. Cappos. 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. doi:10.1109/EuroSP57164.2023.00016. 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—where the fork's API surfaces it—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 (namespace `CurveFieldProofs` elided; 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	`verify_accepts_iff`
T2	canonical half-lift	`verify_accepts_iff_point`
T3	injectivity / point eq.	`verify_accepts_iff_point_eq`
T4	constructive full lift	`verify_accepts_iff_decompress`

All four are proven under the wire-format hypotheses $\mathcal{W}$ of §2; a consumer reasoning about the underlying group element relies on their conjunction (the ladder up to T4), and all four cones are audited against the same per-fork boundary of Appendix C.