LTL: the Lean Transparency Log

Distributing Machine-Checked Proof Evidence via an Authenticated Data Structure Signed by Its Own Certified Artifact

saymrwulf

zkdefi.org — `https://zkdefi.org/lean-transparency-log`

July 2026

Abstract

Interactive theorem provers can certify the functional correctness of deployed cryptographic code, but the resulting assurance is expensive to *consume*: re-checking a realistic proof corpus takes a proof toolchain and hours of kernel time, which excludes almost every downstream user, including autonomous agents that must decide which cryptographic library to trust. We describe the *Lean Transparency Log* (LTL)[1], a deployed 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 hours of kernel time are paid once, by the log operator. Three design decisions distinguish the LTL from prior attestation transparency systems: (i) consumers re-derive verification verdicts locally from the *observed axiom cones* carried in each attestation, so the operator is trusted for observations, never for verdicts; (ii) axiom cones are matched against per-theorem *documented boundaries* exactly, in both directions; and (iii) the log's tree heads are signed by a binary built from the very Ed25519 implementation whose correctness certificates are leaves of the log, and each signature embeds the operator's own Merkle self-check of that leaf. We report a small production deployment covering four verified production Ed25519 implementations, describe what the accumulated evidence does and does not establish, and outline the natural next step: verifying the log's own proof-checking algorithms in Lean and entering those certificates into the log they defend.

1 Introduction

Formal verification of deployed cryptographic code has matured from research prototypes to substantial artifacts: verified-by-construction libraries such as HACL* [10] and Fiat-Crypto [12] ship in mainstream software, and post-hoc verification pipelines such as Aeneas [8] 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 [9] certificates, proven against that fork's own extracted model, covering field arithmetic over $\mathbb{F}_{2^{255}-19}$, the complete twisted Edwards laws [15, 16], scalar arithmetic mod ℓ, encoding/decoding with constructive point decompression, and a four-tier characterization of signature verification [13, 14] whose strongest tier states: the extracted verifier accepts iff the signature's R component decompresses to a valid curve point equal to $[k](-A) + [s]B$.

The economics of *consuming* such evidence are poor. Re-checking one fork's certificates takes ≈30 minutes of Lean kernel time on commodity hardware and requires a pinned toolchain. A wallet application, a package manager, or an autonomous agent choosing a cryptographic backend cannot pay this per decision—and should not have to: the result of a deterministic re-check is a fact that can be attested once and then distributed.

This is the classic transparency-log trade. Certificate Transparency [1, 2] made certificate issuance publicly auditable; Sigstore's Rekor [3] did the same for software signing events and supply-chain attestations [4]; key transparency systems [5] for identity bindings; Go's checksum database for module hashes. The LTL applies the same authenticated data structure to a payload with different trust semantics: *evidence of machine-checked mathematical truth, together with its exact assumption set.* We do not claim novelty for any component—the hash structure and proof algorithms are RFC 9162 verbatim; the contribution is the composition, its trust model, and a deployed, reproducible instance.

Concretely, this paper describes: the attestation format and the *observation-not-verdict* trust model (§2); boundary-exact axiom auditing (§2); the self-referential signing loop (§3); the deployment and what its evidence establishes (§4); and the verification agenda for the log

[1] Not to be confused with linear temporal logic [17]; the collision is acknowledged and, in a paper about verification, difficult to resist.

itself (§6).

2 The Lean Transparency Log

Roles. 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) hold the operator's public key, receive small evidence files, and verify with ≈25 lines of standard-library code. Nothing a consumer does requires a theorem prover.

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 above each attestation carries 16 certificates.

Boundary-exact auditing. Every certificate has a *documented* allowed axiom set. 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). A cone is *clean* iff it equals its allowed set; deviation in either direction—an unexpected axiom, or a missing boundary axiom—is flagged. Each source repository enforces the same discipline in its own check scripts; the log mirrors those sets and treats the repositories' scripts as authoritative.

Observation, not verdict. Attestations include the operator's pass/fail judgment, but consumers ignore it: the consumer's tooling re-derives every verdict locally by comparing the *observed* cone against the consumer's own copy of the allowed sets. An operator that labels a dirty cone "clean" gains nothing; an attestation that omits observed cones is treated as unverifiable. This narrows the trust placed in the operator to: "the reported `#print axioms` output is what the kernel printed for this commit."

Tree, heads, receipts. Leaves are accumulated in an RFC 9162 Merkle tree (SHA-256 with $0x00/0x01$ leaf/node domain separation). The operator signs tree heads; consumers receive a *receipt*: leaf index, sibling path, and the signed head. Consumers additionally maintain a local pin store: same-size heads must match the pinned root exactly (a mismatch is reported as equivocation and is unrecoverable); growth requires a verified consistency proof from the pinned size; shrinkage is rejected. A freshness policy bounds head age. The full log is also published as a git repository (one file per leaf, plus every head ever signed), so any cloner can recompute every prefix root from the public leaves and audit the complete head history without consistency proofs—a low-infrastructure witness mechanism [2]; a standalone ≈150-line standard-library verifier ships in the mirror.

3 The self-certifying signature

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: 4
signing_library_certificates_proven: 16/16
self_inclusion: verified
```

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 (Fig. 1). We state the honest extent of this claim precisely: 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`.

4 Deployment and evidence

The LTL is deployed[2] with eight leaves: one attestation per fork from each of two full replay runs. Each replay

[2]Service: `https://zkdefi.org/lean-transparency-log` (read-only HTTP API and documentation). Mirror: `https://github.com/saymrwulf/lean-transparency-log`. Operator/consumer tooling and a twelve-lecture course: `https://github.com/`

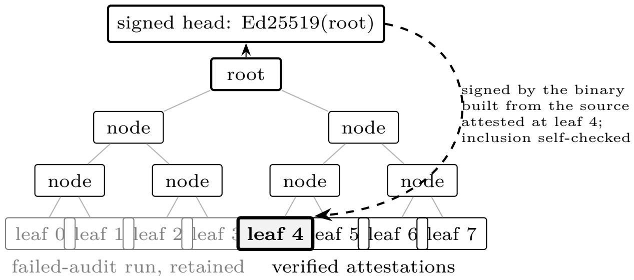

Figure 1: The deployed 8-leaf log. Leaves 0–3 record an early replay whose audit step failed (two tooling defects, since fixed); an append-only ledger retains them. The head is signed by a binary built from the implementation attested at leaf 4, whose inclusion the operator verifies before signing.

re-checked ≈64 Lean files per fork (≈1,800 s per fork under hard memory caps and core pinning) and audited all 16 certificates; in the second run all four forks reported 16/16 proven with boundary-exact cones, pinned to exact commits. The first run is deliberately still in the log: its audit step failed due to 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), and the operator signed attestations recording the failure rather than suppressing the run. We consider the resulting ledger—four failure leaves permanently beside four success leaves—a feature of the trust model, and note that both defects were fail-closed: valid proofs were rejected; invalid ones were never accepted.

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, 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 [8]; each fork's documented oracle boundary; operator key custody and the trust-on-first-use key distribution (mitigated by publishing the key in two independent locations); collision resistance of SHA-256 for the log; unforgeability of Ed25519 for the heads; and the consumer's own ≈25-line verifier.

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; zero lines between structurally identical forks). Per-target verification, in other words, is doing measurable work exactly where the targets actually differ.

5 Related work

Certificate Transparency [1, 2] supplies the data structure and proof algorithms, used here unchanged. Rekor within Sigstore [3] is the closest deployed system: a transparency log over signing events and supply-chain attestations such as in-toto [4] 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 [5] and checksum databases share the accumulator pattern with different payloads. Proof-carrying code [6] 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 but does not eliminate. Cheval, Moreira and Ryan formally verify transparency protocols themselves [7]; our direction is the complement (we log the verification), and §6 proposes meeting in the middle. Verified Merkle tree *implementations* exist, notably in EverCrypt [11]; §6 builds on that precedent rather than claiming it.

6 Limitations and next steps

Limitations. The deployment is small (one operator, eight leaves, four subject repositories) and the operator is a single party; split-view defense currently rests on consumer-side pinning 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 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

`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.

unavailable rather than backed by an unverified implementation.

Verifying the accumulator itself. The natural next step applies the corpus's own discipline to the log's cryptographic half, which is currently the *unproven* half of the composition. All of the following are tractable Lean targets: (i) completeness of the RFC 9162 inclusion verifier (honest proofs verify—no assumptions); (ii) *soundness as an explicit reduction*: an accepting inclusion proof for a leaf outside the tree yields a SHA-256 collision, making collision resistance a documented boundary axiom audited exactly like the SHA-512 oracle in the Ed25519 tiers; (iii) the analogous statement for the consistency verifier (acceptance implies prefix, modulo collisions); (iv) domain separation as a lemma; and (v) total correctness of the consumer's pin-store state machine (equivocation detection, rollback rejection). Verified Merkle implementations in F* [11] and algorithm verifications in other systems 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 proof corpora, tooling, deployment, and this paper were produced with substantial assistance from Claude (Anthropic). All claims are enforced by the referenced check scripts and are independently re-checkable from the public artifacts.

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] Z. Newman, J. S. Meyers, S. Torres-Arias. Sigstore: Software Signing for Everybody. *ACM CCS*, pp. 2353–2367, 2022.

[4] 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*, 2019.

[5] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, M. J. Freedman. CONIKS: Bringing Key Transparency to End Users. *USENIX Security*, 2015.

[6] G. C. Necula. Proof-Carrying Code. *ACM POPL*, pp. 106–119, 1997.

[7] V. Cheval, J. Moreira, M. Ryan. Automatic verification of transparency protocols. *IEEE EuroS&P*, 2023. arXiv:2303.04500.

[8] S. Ho, J. Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6 (ICFP): 711–741, 2022.

[9] L. de Moura, S. Ullrich. The Lean 4 Theorem Prover and Programming Language. *CADE-28*, LNCS 12699, pp. 625–635, 2021.

[10] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. *ACM CCS*, 2017.

[11] J. Protzenko et al. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. *IEEE S&P*, 2020.

[12] 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.

[13] 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.

[14] S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, 2017.

[15] D. J. Bernstein, T. Lange. Faster addition and doubling on elliptic curves. *ASIACRYPT*, LNCS 4833, pp. 29–50, 2007.

[16] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, C. Peters. Twisted Edwards curves. *AFRICACRYPT*, LNCS 5023, pp. 389–405, 2008.

[17] A. Pnueli. The temporal logic of programs. *IEEE FOCS*, pp. 46–57, 1977.