

IETF KEYTRANS WG

RuKT

Key Transparency in Rust

Guy Fischman

18 March 2026



Motivation

Why build this implementation?

Zero-trust git forge

We need users to verify the authenticity of signing keys for people they share code with -- KT provides exactly this guarantee.

Stress-testing the protocol

The protocol has strong theoretical properties but limited real-world implementation experience. We wanted to stress-test it against production scenarios: messaging apps, enterprise key management, and developer tooling.

Why Rust?

Rust was chosen for memory safety guarantees in security-critical infrastructure and for performance -- the transparency log is on the critical path for every code push.

Full protocol coverage

This implementation covers the full protocol: combined tree (log + prefix), fixed-version and greatest-version search, contact and owner monitoring, third-party auditing, binary ladder, credentials, and both cipher suites (Ed25519 and P-256).

Protocol Wins — Clear and Implementable

What went well:

- Combined tree structure is elegant
The log and prefix tree composition maps cleanly to RocksDB.
- Binary ladder is brilliant
This compresses version discovery to $O(\log n)$ with near-zero computation time.
- Implicit binary search tree for timestamps
Reusing the log structure makes monotonicity verification nearly free.
- Batching is natural
Log entries map perfectly to batched writes for efficient Merkle computation.
- Cipher suite abstraction is clean
Swapping cryptographic primitives required no architectural changes.
- Proof composition is well-designed
Clear ordering and deduplication rules make implementation straightforward.

Protocol Wins — Scalable

Performance results that matter:

Search Latency vs Tree Size

10 users	398 us
100 users	403 us
500 users	398 us
1,000 users	399 us
2,000 users	413 us

VRF evaluation dominates — not tree traversal.

Sub-millisecond Core Operations

Search (greatest version)	~400 us
Search (fixed version)	~210 us
Monitor (contact, 1 label)	~190 us
Credential issuance	~415 us
Audit (100 entries)	~64 us
Tree head	~130 ns

Protocol Wins — Scalable (cont.)

Throughput at scale:

Concurrent Read Throughput (4 threads)

Parallel Searches	Throughput	Per-Search
1	2,200/s	450 us
10	7,200/s	138 us
50	8,600/s	115 us
100	9,000/s	110 us

Batch Write Throughput

Batch Size	Throughput	Per-Update
1	17/s	59 ms
10	145/s	69 ms
50	492/s	100 ms
100	671/s	149 ms
500	913/s	548 ms
1,000	968/s	1.03 s

Challenges — Protocol Complexity

Where the spec was hard to implement:

1

Traversal session state management (Section 11.3)

CombinedTreeProof ordering and deduplication rules proved the most bug-prone part of the implementation.

2

Distinguished log entry computation (Section 7.1)

The recursive algorithm creates a hard-to-test dependency between search and monitoring. Pseudocode would improve clarity.

3

Owner monitoring state machine (Section 8.3)

Two-phase monitoring with conditional recursion creates subtle, difficult pagination edge cases.

4

Maximum lifetime / pruning interaction (Section 6.2)

Three different expired-entry cases each require distinct, complex handling that is easy to implement incorrectly.

Challenges — Scalability Concerns

Where performance may be a concern at scale:

Update latency is dominated by batcher timeout (~58ms floor)

Adjusting the timeout trades latency for batching efficiency. Merkle and crypto computations remain the primary bottleneck, not data size.

Fixed-version search cost scales poorly

Fixed-version searches become slower than greatest-version searches at ~50 versions. Use `maximum_lifetime` to bound costs for high-rotation deployments.

VRF is the dominant per-operation cost

Ed25519 VRF (~123us) is significantly faster than P-256 (3x slower). Cipher suite selection impacts performance across all operations.

Sequential Merkle construction is the write bottleneck

Prefix tree inserts are inherently serial. While crypto operations are parallelised, the core Merkle chain limits single-entry throughput.

Interesting Benchmark Results

Surprising findings:

01

Constant search latency

Scaling from 32 to 1M entries adds only 66us. VRF evaluation, not tree traversal, remains the primary cost.

03

Cheap contact monitoring

At ~190us per label, the $O(\log n)$ direct-path traversal confirms the protocol's lightweight design.

05

Optimal batch size

50 is the "sweet spot" for performance. Throughput gains stall beyond 500 while latency increases linearly.

02

Version search efficiency

Fixed-version searches are ~2x faster, but deep version histories (50+) favor greatest-version searches.

04

I/O-bound auditing

Pre-computed blobs turn the audit read path into a streamlined, sequential DB scan (64us for 100 entries).

06

Cipher suite impact

P-256 is 3x slower than Ed25519. Use Ed25519 by default to avoid performance penalties unless compliance dictates otherwise.

Protocol Design Validation — Comparative Analysis

How does this protocol's design compare to simpler alternatives?

Combined proof deduplicates shared path elements

Proof size grows $O(\log n)$; scaling from 32 to 1M users adds only 374 bytes. The full search response remains ~1 KB at 1M users.

Greatest-version vs fixed-version search

Fixed-version searches are ~2x faster than greatest-version. Applications should cache and request specific versions when possible.

Binary ladder scales $O(\log n)$ with version count

The binary ladder compresses proof requirements from $O(n)$ to $O(\log n)$. This innovation makes greatest-version search practical even for billion-user deployments.

Proof payload grows slowly with versions

Payloads remain manageable at 349 bytes for a single version and 9.7 KB for 100 versions. This supports even high-rotation labels efficiently.

Million-User Scalability

Does this scale to WhatsApp (1B users) or large enterprise (1M users)?

$O(\log n)$ confirmed at 1M users

Search grows only 24% (276→342 us)
from 32 to 1M users. All operations stay
sub-millisecond.

Billion-user deployments are
feasible

From 1M to 1B adds only 10 more tree
levels. Projected search at 1B users:
~390–410 us.

Use `maximum_lifetime` to bound
search cost

and `RMW` to bound monitoring cost, not
search cost.

GDPR and Right to Be Forgotten

How does an append-only transparency log handle data deletion?

Maximum Lifetime (Section 6.2)

Expired entries are skipped during traversal, with searches for purely expired versions returning a clean error. There is zero measurable overhead on normal searches.

Deletable Openings (Section 10.6)

Openings are stored separately, allowing deletion via a single near-instant (1.3 us) `delete_cf()` call. The log remains immutable, but the data becomes inaccessible.

Error paths are faster than success

"Unavailable" and "expired" responses short-circuit before VRF proof generation. These paths take approximately 135 us compared to 279 us for successful searches.

Log integrity is preserved

Deletion removes the ability to prove a key was registered, not the log entry itself. Auditors can still verify the consistency of the log.

Recommendations for the Working Group

Based on implementation experience:

1

Add pseudocode for the traversal session / CombinedTreeProof construction

This is the most complex part to implement correctly. The current prose description in Sections 11.3.x leaves room for misinterpretation.

2

Consider a "fast path" for single-version labels

Most messaging app users will have 1-3 key versions. The binary ladder for version 0 is trivially $[0, 1]$ -- a special case could avoid the general ladder machinery.

3

Document the batcher pattern

The protocol naturally supports batching (multiple updates per log entry) but doesn't discuss it. Our experience shows batching is critical for write throughput and should be a recommended implementation pattern.

4

Clarify the interaction between `maximum_lifetime`, RMW, and distinguished nodes

With concrete examples. The three features interact in non-obvious ways during search and monitoring.

5

Consider recommending Ed25519 as the default cipher suite

The 3x performance advantage over P-256 in VRF operations compounds across all protocol operations.

Implementation Summary

Component	Lines of Rust	Notes
Combined Tree (log + prefix)	~1,200	Core data structure
Search (fixed + greatest)	~300	Traversal + binary ladder
Monitoring (contact + owner)	~250	State machine
Auditing	~100	Pre-computed blobs
Cryptography (VRF, signing)	~350	Ed25519 + P-256
Batcher	~200	4-phase pipeline
Service (gRPC)	~150	tonic-based
Integration tests	~2,000	20+ test scenarios
Benchmarks	~600	40+ benchmark scenarios

Stack

Rust, RocksDB, tonic (gRPC), ed25519-dalek, p256, criterion

Repository

<https://github.com/guyfischman/RuKT>

Appendix: Full Benchmark Results

4 worker threads, release mode

Cryptographic Primitives

VRF Prove (Ed25519)	123 us
VRF Verify (Ed25519)	121 us
VRF Prove (P-256)	358 us
VRF Verify (P-256)	458 us
Ed25519 Sign	14 us
Ed25519 Verify	29 us
HMAC Commitment	1.2 us
SHA-256 Log Leaf	177 ns
SHA-256 Log Parent	387 ns

Binary Ladder (pure computation)

Version 0	13 ns
Version 100	63 ns
Version 10,000	99 ns
Version 1,000,000	146 ns

Appendix: Full Benchmark Results (cont. 1)

4 worker threads, release mode

Scale Scenarios — Batch Throughput

Batch throughput (1)	17 ops/s
Batch throughput (10)	145 ops/s
Batch throughput (50)	492 ops/s
Batch throughput (100)	671 ops/s
Batch throughput (500)	913 ops/s
Batch throughput (1000)	968 ops/s

Scale Scenarios — Concurrent Search

Concurrent search (1)	2,200/s
Concurrent search (10)	7,200/s
Concurrent search (50)	8,600/s
Concurrent search (100)	9,000/s

Value Size Impact on Update Latency

32 bytes	59 ms
256 bytes	59 ms
1024 bytes	59 ms
4096 bytes	59 ms

Appendix: Full Benchmark Results (cont. 2)

4 worker threads, release mode

Protocol Operations (500-user tree)

Update (new user)	59 ms
Update (key rotation)	60 ms
Search greatest (100)	431 us
Search greatest (1000)	446 us
Search fixed v0 (100)	211 us
Search fixed v0 (1000)	212 us
Search greatest (10 vers)	1.7 ms
Search greatest (50 vers)	3.7 ms
Search fixed rand (10 vers)	1.4 ms
Search fixed rand (50 vers)	4.6 ms

Protocol Operations (cont.)

Monitor contact (1)	192 us
Monitor contact (10)	1.2 ms
Monitor owner	2.9 us
Audit 10 entries	63 us
Audit 100 entries	64 us
Credential	415 us
Tree size RPC	117 ns
Full tree head	132 ns
Full tree head + cons	128 ns

Appendix: Full Benchmark Results (cont. 3)

4 worker threads, release mode

Git Forge Scenario

Verify developer key	392 us
Get credential	392 us
Onboard new developer	59 ms
CI verify 10 devs	3.9 ms

Enterprise Key Rotation (50 users x 10 rotations)

Search after rotations	3.5 ms
Search fixed (random ver)	648 us
Audit 100 entries	166 us

GDPR / Privacy (100-user tree, maximum_lifetime = 10s)

Search (lifetime enabled)	279 us
Delete opening (forget)	1.3 us
Search after forget	135 us
Search expired entry	140 us