



FORMAL METHODS / SYSTEMS RESEARCH | OCTOBER 24, 2026

OpenSSH-Lean

Source-Linked Formal Analysis and Reproducible Crash Discovery

Brian Mingus
Cognatory, LLC

Pinned C source, reusable Lean models, and source-linked evidence in one research system.

Code: github.com/mingusb/openssh-lean

Collection edition. The complete nine-page manuscript follows.

AI-generated conceptual artwork; separate from scientific figures.

brianmingus.com

OpenSSH-Lean: Source-Linked Formal Analysis and Reproducible Crash Discovery

Brian Mingus

Cognatory, LLC
brian@cognatory.com

Code: github.com/mingusb/openssh-lean

Abstract. OpenSSH-Lean is a source-linked formal analysis system for a pinned OpenSSH C build. We extended Cerberus to translate the unchanged sources of all 14 programs into Lean 4 and store their models in a content-addressed corpus. A streamed index locates selected fragments; typed loading reconstructs them with original-C source spans. A durable service retrieves indexed semantic rules, persists proof work, checks completed proofs in Lean, and replays certificates. Native observations support specification mining. An evidence catalog binds model identities, source witnesses, formal obligations, native traces, and executable reproducers. In an `scp` evaluation, a generated Bash program reproduces a native `SIGSEGV`, while a Lean theorem proves reservation failure in an abstract stack model under a stated shortfall hypothesis. The catalog records the translation, reachability, and platform obligations connecting these results.

Keywords: C semantics · Lean 4 · OpenSSH · crash discovery · source correspondence · reproducible evidence

1 Introduction

OpenSSH’s programs share a large C codebase but expose different entry points. OpenSSH-Lean pins one source revision and build configuration, extends Cerberus’s C semantics [11,5] for the GNU C and cross-unit behavior used by OpenSSH, and produces linked Lean models for all 14 programs. Upstream C remains unchanged, and the native executables run normally.

The system stores translated objects by content and indexes their structural addresses and original-C spans. A query selects typed fragments and applicable semantic rules. The proof service persists attempts, checks completed proofs in Lean, and replays certificates in a fresh worker. Native interface captures provide candidates for specification mining. A separate catalog admits crash observations through fresh native replay and binds their reproducers, source witnesses, and formal obligations.

The `scp` recursive-directory case exercises these components. An automated depth search generates a Bash reproducer for a native `SIGSEGV`; source checks identify the recursive C call; and a Lean theorem establishes a modeled stack result under explicit hypotheses. The model corpus and proof service remain reusable across all 14 programs.

2 The OpenSSH Cohort and Source Identity

The pinned cohort is OpenSSH 10.6p1, built for Linux x86_64 without OpenSSL or zlib. The upstream C and headers remain byte-identical to the recorded source revision. Configured parsing covers hundreds of translation units. The maintained Cerberus and Lean adaptation adds more than fifty thousand code lines. Its changes cover GNU syntax and attributes, integer and floating representations, cross-unit type and symbol identity, variable arguments, object sizes, lifetimes, and memory operations. The count excludes generated payloads and does not represent changes to OpenSSH C.

The translator runs once for the pinned source, configuration, and semantic dependencies (Fig. 1). Its 14 linked models occupy a multi-gigabyte archive of unique objects. A streamed structural index addresses millions of fragments. Each index record stores a model identity, section, structural address, byte range, and digest. The typed loader reconstructs selected terms with the producer’s datatypes. Queries reuse those terms without repeating translation or decoding the entire cohort.

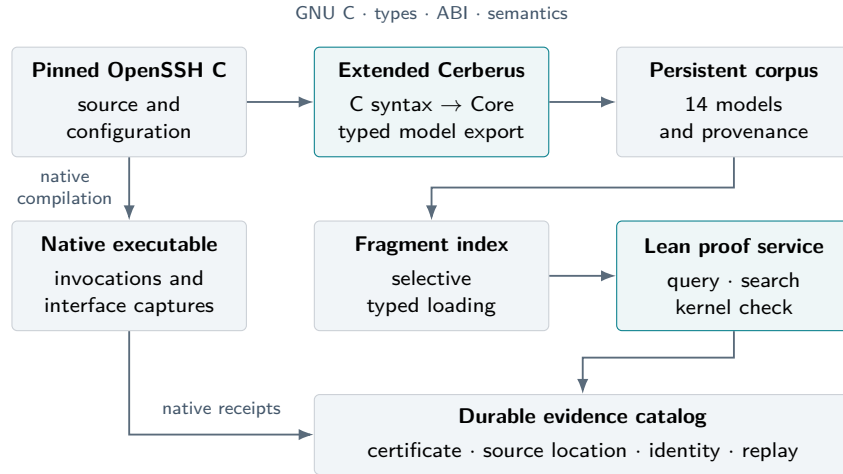


Fig. 1. One pinned source cohort feeds reusable formal queries and native observations.

A source witness identifies an operation by structural address within a typed fragment and resolves its annotation against the recorded C bytes. Cabs is Cerberus’s C syntax tree; Core is its typed semantic representation. The witness checks tree membership, exact source span, and Core ancestry. A behavior-preservation relation between these representations is a separate proof obligation, as in verified compilation [9].

2.1 Implementation contributions

We use *cloc* [7] to count nonblank, noncomment source lines. Table 1 reports changes to the maintained Cerberus fork against its upstream baseline. Table 2 describes the surrounding project implementation. Counts are rounded; k denotes a thousand lines. Generated payloads, copied reference sources, and draft experiments are excluded from Table 2’s maintained-code counts.

Table 1. Cerberus framework changes by language. C and header changes belong to framework support and tests; upstream OpenSSH C remains unchanged.

Language	Added	Edited	Removed	Contribution
Lem	32k	800	400	GNU C translation, types, builtins, arrays, variable arguments, and memory semantics.
Lean	11k	400	100	Typed export, memory integration, inventories, termination proofs, caching, and checks.
Python	2.9k	0	0	Code generation, contract processing, and consistency checks.
C	2.9k	< 10	0	ABI fixtures, native caches, libc support, and oracle tests.
OCaml	1.6k	300	80	Lexer/parser integration, Cabs export, printing, and memory models.
C headers	750	< 10	< 10	Cache interfaces, libc declarations, and test templates.
Core	520	100	< 10	Core standard-library semantics.
make	150	20	< 10	Build integration and model source lists.
Bourne shell	90	30	10	Build, oracle, freshness, and synchronization checks.
Bash	< 10	10	70	Process-control compatibility.
Ott	< 10	0	0	Core syntax and grammar definitions.
Source total	52k	1.7k	700	About 52k net added code lines.
JSON	9.1k	0	0	Generator contracts and capture scenarios.
TOML	50	< 10	0	Lean package and build configuration.

Edited lines occur at both endpoints; net growth is added minus removed. The source total includes maintained templates, regression tests, and framework support. Declarative JSON and TOML inputs are shown separately.

Table 2. Current project implementation outside the Cerberus fork. This is a source inventory; the fork table measures changes from a baseline.

Language	Code	Contribution
Python	50k	Translation workflow, model archives, indexing, source lookup, specification mining, scheduling, catalogs, and tests.
Lean	2.1k	Typed fragment loading, operation selection, proof queries, memory lemmas, and worker automation.
C	850	Streaming JSON framing, native oracles, feature fixtures, and capture templates.
Bourne shell	260	Build, bootstrap, and verification wrappers.
PowerShell	210	Windows workflow and recovery wrappers.
TOML	20	Lean package configuration.
YAML	20	Source-integrity checks in continuous integration.
Total	53k	Maintained project source.

The initial project snapshot already contained the core implementation. Since that snapshot, the selected code has grown by about 11k lines, chiefly Python orchestration and Lean proof support.

3 Candidate Search and Checked Properties

A proof job supplies a Lean proposition, selected model fragments, and optional rules, tactics, and imported modules. A reachability proposition can quantify an input, initial state, transition trace, and final outcome. The operation inventory locates syntax sites for formulating such propositions. A native-signal property also includes the C, compiler, and platform assumptions that connect the modeled outcome to `SIGSEGV`.

3.1 Specification mining from native observations

A specification states how an operation behaves under specified inputs and state. *Specification mining* infers candidate rules from code or executions; classic work mines API state machines and likely program invariants [1,8]. Here, source-derived API declarations guide native observation scenarios. Captures retain arguments, return values, `errno`, and declared byte spans with interface hashes.

The demonstration fits XGBoost [6] to native API observations, converts the model into a Decision Program, a compact branching evaluator [12], and emits a Lean evaluator. Its Lean `capturedRowsMatch` theorem checks equality on those captured rows; it does not prove a contract for every admissible input. A universal contract requires a separately submitted proof query for its quantified premises and semantic connection to C.

3.2 Inventory and proof search

The Core inventory enumerates expression occurrences. A separate exact-symbol self-call scan identifies recursive call sites in archived bodies and records their structural addresses. Semantic rules use actual Cerberus operations, including `CerbMem.loadM` for a local null-load control. A long-lived Lean worker retains imported declarations and proof environments. Its discrimination-tree index retrieves rules by conclusion shape; each completed proof is checked against the original proposition. Unresolved metavariables and admitted proofs are rejected [13,10].

The scheduler stores expansions and attempts as separate durable work items. Resource exhaustion enlarges the attempt’s Lean heartbeat quantum and reschedules it; it is not a negative result. Completed certificates carry the model and fragment hashes, proposition, proof term, dependencies, and source references. Fresh-worker replay reconstructs their environment. This reuses translation, typed terms, and checked lemmas across queries.

Listing 1 shows two entry points. In four body-bearing Core sections, inventory commits each section and digest after Lean checks its expression-node count. A submitted proof job replays ancestors before each durable expansion or tactic step. The worker accepts a proof only after checking the original proposition, kernel elaboration, and axioms. Certificates enter the catalog through an outbox; source mappings attach when references resolve. The listing is Lean-style pseudocode.

```

for row in index.executableCore do
  let key := (row.section, row.sha256)
  if not (inventory.done key) then
    let scan ← lean.scan row.section row.bytes row.sha256
    require scan.coverageChecked
    require scan.visited = scan.completeNodeCount
    inventory.commit key scan.sites

for job in submittedProofJobs do
  while let some task := job.nextQueued do
    let state ← job.replayAncestors task.node
    let result ← match task.kind with
    | expand      => lean.choices state indexedRules
    | step tactic => lean.step state tactic
    job.commit task result
    if result.status == proved then
      job.outbox.add (certificateWithResolvedMappings result)
  catalog.ingestPending job.outbox

```

Listing 1. Body-bearing Core inventory and resumable proof search. The loops have separate callers; `proved` denotes the worker’s checked result.

4 Native Crash Discovery and the Evidence Catalog

A machine-readable strategy supplies the native executable, arguments, environment, and input template. The runner starts a fresh native process for each probe and records the input hash and outcome. For the `scp` recursive-directory strategy, it varies directory-record depth: doubling finds a crashing input and bisection narrows the observed bracket. The selected input is captured with a fatal-signal trace and exact faulting instruction site.

The system renders a Bash reproducer from the selected profile and executes it without human intervention. Catalog admission requires a fresh run of the pinned native `scp` to produce a decoded fatal synchronous `SIGSEGV`. The record binds the input and executable identities, source-link witnesses, actual fault site, and recursive trace to that observation. Admission is automatic.

Canonical text exports reconstruct the catalog database, and replay checks the identities of stored assets. New entries also pin the renderer and parser dependencies used to produce and interpret their evidence.

Formal discovery records enter with importance unassessed. Later append-only enrichment can record an importance assessment and rationale; assessment does not gate automatic catalog admission.

The source witness checks the recursive `sink` call in archived Cabs against unchanged C bytes and tracks its source region into the selected Core call. A Lean membership theorem confirms the exact-symbol self-call in the archived `scp` body. The native trace records recursive `sink` frames; the catalog associates that trace with the source witness.

```
def reserveFault (frameBytes : Nat) : Nat → Nat → Bool
| _, 0 => false
| available, depth + 1 =>
  if available < frameBytes then true
  else reserveFault frameBytes (available - frameBytes) depth

theorem scp_abstract_stack_fault_if_depth_exceeds
  (depth available : Nat)
  (withinSoftLimit :
    available ≤ observedSoftStackLimitBytes)
  (depthShortfall :
    observedSoftStackLimitBytes < depth * scpReservedBytes) :
  reserveFault scpReservedBytes available depth = true := by
  apply reserveFault_of_shortfall
  omega
```

Listing 2. Selected checked Lean declarations, with line breaks adjusted for print. Supporting constants and a generic shortfall lemma are omitted.

The formal result uses a small stack model. Listing 2 shows its transition and checked case theorem. The theorem applies a generic shortfall lemma: when available stack space is less than the aggregate reservation demanded by a call depth, some reservation fails. The case uses measured frame reservation, an observed stack bound, and the selected input depth.

Repeated native runs have faulted at different instructions, including a library instruction, while retaining hundreds of recursive `sink` frames. Stack layout can change the final instruction, so the catalog records each run’s exact site instead of naming one universal crash PC. The original C recursion edge and its exact bytes are checked independently of those compiled instructions.

An append-only formal link connects the Lean theorem to the native reproducer. The link records five semantic obligations: protocol records induce the modeled recursive calls; the Cabs-to-Core translation preserves the relevant behavior; the ELF prologue implements the modeled reservation; the process stack budget matches the assumption; and a modeled failed reservation yields native `SIGSEGV`. The native trace demonstrates one reachable invocation; Figure 2 shows its evidence beside those proof obligations.

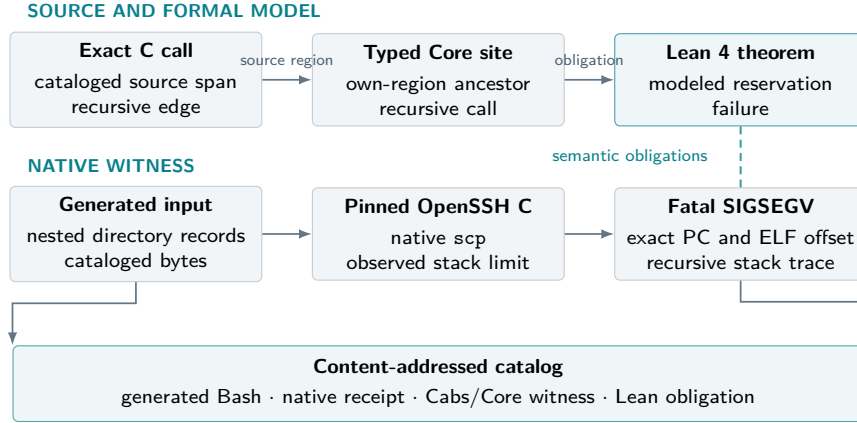


Fig. 2. Source witnesses, a checked model theorem, and a reproduced native fault are connected by recorded semantic proof obligations.

5 Evaluation and Interpretation

Table 3 reports scale and observed outcomes. The archive indexes every model section across all 14 targets. Repeated warm source-location lookups completed in well under a millisecond. In a separate single-proof control, rule indexing more than halved warm search time.

Table 3. Engineering scale and observed witness outcomes in the pinned build.

Measure	Recorded result
Framework adaptation	More than fifty thousand added code lines; upstream OpenSSH C unchanged.
Model cohort	All 14 programs; a multi-gigabyte object archive and millions of indexed fragments.
Local Lean rule control	One local proof completed with both settings; indexing more than halved warm search time.
Native depth search	Dozens of fresh probes established an observed bracket between noncrashing and crashing inputs.
Selected and replayed fault	Generated nested-directory input; fatal SIGSEGV ; hundreds of recursive sink frames in both captured and replay executions.

The pinned Linux x86_64 cohort excludes OpenSSL and zlib. The native trace establishes a crashing invocation under recorded conditions; the Lean theorem establishes an abstract stack result. The catalog records them as separate evidence and names the additional obligations for an end-to-end theorem.

6 Related Work and Next Steps

Cerberus provides C semantics [11]; CompCert establishes compiler preservation [9]; RefinedC and VST-Floyd offer other C proof interfaces [14,4]. KLEE and directed greybox fuzzing generate or steer native tests [3,2]. Our strategy varies one protocol parameter and binds a native crash, reproducer, C source, typed model context, and conditional theorem.

Future cohorts cover OpenSSL, zlib, other platforms, and interface contracts. Strategies synthesized from Cabs, protocol constants, typed Core, and mined contracts can widen native search. Entry-to-fault proofs and translation validation can close semantic premises; shared summaries across 14 programs can limit rechecking.

7 Conclusion

OpenSSH-Lean translates unchanged OpenSSH C into source-linked models shared by all 14 programs. Its index, typed loader, and durable proof service make those models available for repeated formal queries. Specification mining produces candidate interface relationships and captured-row checks. The evidence catalog connects checked results to source witnesses and native observations. The **scp** case records a reproduced crash beside a checked abstract stack theorem and the obligations that connect them.

Acknowledgments. Brian Mingus directed the work, with AI assistance in implementation and manuscript preparation.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 4–16. ACM (2002). doi:10.1145/503272.503275
2. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344. ACM (2017). doi:10.1145/3133956.3134020
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association (2008), USENIX proceedings
4. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning* **61**(1–4), 367–422 (2018). doi:10.1007/s10817-018-9457-5
5. Cerberus Lean contributors: Cerberus Lean, <https://github.com/0athTech/cerberus-lean>, software repository; accessed October 2026
6. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 785–794 (2016). doi:10.1145/2939672.2939785
7. cloc contributors: cloc: Count lines of code, <https://github.com/AlDanial/cloc>, version 2.06 used for the source audit
8. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* **27**(2), 99–123 (2001). doi:10.1109/32.908957
9. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* **52**(7), 107–115 (2009), <https://compcert.org/publi.html>
10. Limperg, J., From, A.H.: Aesop: White-box best-first proof search for Lean. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM (2023). doi:10.1145/3573105.3575671
11. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages* **3**(POPL), 67:1–67:32 (2019). doi:10.1145/3290380
12. Mingus, B.: Decision programs, <https://github.com/mingusb/decision-programs>, software repository; accessed October 2026
13. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: *Automated Deduction—CADE 28*. pp. 625–635. Springer (2021). doi:10.1007/978-3-030-79876-5_37
14. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: Automating the foundational verification of C code with refined ownership types. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. pp. 158–174. ACM (2021). doi:10.1145/3453483.3454036