

sorryaudit

Transitive Sorry Taint Detection for AI-Assisted Lean 4 Proofs

Anshuman Singh | Apart Research Secure Program Synthesis Hackathon 2026 |
github.com/anshuman-dev/sorryaudit

Abstract

As AI tools generate increasing volumes of Lean 4 proof code, a structural gap has emerged between proofs that compile and proofs that are actually sound. `sorry` in Lean 4 is a proof placeholder that silences all errors while marking a theorem as depending on `sorryAx`. AI proof assistants use it constantly for steps they cannot complete. The critical problem is transitive: a theorem with a complete proof is still unsound if any theorem in its dependency chain uses `sorry`. Lean only warns at the point of `sorry`, not at every downstream dependent. `sorryaudit` makes this gap visible by running Lean's own `#print axioms` mechanism across an entire codebase and reporting the real trust status of every theorem.

1. Problem

Lean 4's `sorry` tactic allows any proof goal to be discharged immediately. It is intended as a temporary placeholder during development. AI proof tools use it as a fallback when they cannot complete a proof step. The resulting file compiles without errors. Standard CI passes. Nothing in the toolchain tells you how much of the codebase is actually proved.

Consider the following output from an AI proof assistant:

```
theorem list_reverse_length (l : List Nat) : l.reverse.length = l.length := by
  induction l with
  | nil => simp
  | cons h t ih => sorry    -- AI left this incomplete

theorem reverse_preserves_length_doubled (l : List Nat) :
  (l.reverse ++ l).length = 2 * l.length := by
  have h := list_reverse_length l
  simp [List.length_append, h]; omega
```

`reverse_preserves_length_doubled` has no `sorry` in its body. It compiles. It looks proved. But it is unsound, because it depends on `list_reverse_length` which depends on `sorryAx`. Lean does not warn here. The only way to know is to query the kernel directly.

The fellowship reading list documents this problem concretely. "Lies, Damned Lies, and Proofs" shows AI systems introducing unnecessary axioms, using `sorry` for hard steps, and rewriting code under verification rather than verifying it. "Who Watches the Watchers?" found a real heap buffer overflow in formally verified Lean 4 code at the boundary between the verified core and the unverified C++ runtime. `sorryaudit` addresses the proof layer of this problem.

2. Approach

sorryaudit runs two passes and merges the results.

2.1 Syntactic Pass

Parses every .lean file in the project. Strips line comments and string literals (including multi-line continuations via re.DOTALL) before scanning, so keywords inside comments or log strings do not produce false positives. Detects sorry, admit, native_decide, and Unsafe.cast by line. Assigns hits to theorems using line range boundaries, treating theorem, lemma, def, elab, macro, and notation declarations as boundaries. Fast: no Lean installation required for this pass alone.

2.2 Semantic Pass

For each file, generates a standalone Lean script that imports the module, opens its namespaces, and appends #print axioms for every theorem in that file. Runs the script through lean (or lake env lean for Lake projects). Parses lines matching " depends on axioms: [...]". If sorryAx appears in that list, the theorem is UNSOUND regardless of what its own proof text contains.

For Lake projects: detects the presence of lakefile.lean or lakefile.toml, runs lake build to compile dependencies, infers module names from file paths relative to the project root, and invokes lake env lean so that import resolution works against built .olean files. The standalone query file is created at the project root to avoid namespace resolution failures that occur when appending queries to submodule files.

2.3 Merge and Report

Semantic result takes precedence where available. Syntactic covers the rest.

Verdict	Condition	Trust Score
UNSOUND	Depends on sorryAx	0.0
SUSPECT	Non-standard axioms beyond the core set	0.4-0.85
WARN	native_decide or Unsafe calls present	0.6
TRUSTED	Only propext, Classical.choice, Quot.sound, funext	1.0

3. Results

3.1 AI-Generated Proof File (demo/AIGenerated.lean)

A file simulating the output of an AI proof assistant: 8 theorems, some proved correctly, some with direct sorry, one with transitive taint only.

```
Theorems: 8 | Overall score: 45.0%
Trusted: 3 | Unsound: 4 | Warn: 1
```

UNSOUND:

```
x list_reverse_length      (direct sorry in inductive case)
x reverse_preserves_length_doubled (no sorry in body - caught transitively)
x hard_number_theory       (direct sorry)
x composite_has_factor      (transitive via hard_number_theory)
```

WARN:

```

! small_prime                                (native_decide)

TRUSTED:
v list_length_nonneg
v exists_nat_gt
v add_comm_example

```

The key finding: `reverse_preserves_length_doubled` is caught as UNSOUND by the semantic pass. Running with `--no-semantic` causes it to appear as TRUSTED, raising the score from 45.0% to 57.5%. This confirms the semantic pass is catching real transitive taint that pattern matching alone cannot find.

3.2 lean-tcb (Real Open Source Project)

lean-tcb is a Lean 4 library for auditing trusted computing bases, linked in the fellowship's own resource list. 117 theorems across 23 files.

```

Theorems: 117 | Overall score: 98.8%
Trusted: 115  | Unsound: 1   | Warn: 1

UNSOUND:
x sorryThm : 1 + 1 = 3   (depends on sorryAx)

WARN:
! nativeThm              (native_decide)

```

`sorryThm` is a theorem lean-tcb includes deliberately to test its own soundness checker. `sorryaudit` identified it independently using `#print axioms` without prior knowledge it was there. 115 of 117 theorems are fully trusted.

4. Usage

```

# Install Lean 4
curl https://elan.lean-lang.org/elan-init.sh -sSf | sh

# Run on any Lean 4 project
python3 sorryaudit/main.py <path-to-project>

# Flags
--no-semantic    syntactic pass only, no Lean required
--no-build       skip lake build (if project already compiled)
--json           machine-readable JSON output
--tree           compact per-file view
--dot            Graphviz DOT for visualization

```

5. Limitations

- The semantic pass resolves theorems where the module can be imported in the Lake environment. Theorems in separate Lake build targets require that target to be built first.
- The syntactic pass is a conservative over-approximation. Edge cases in string literal parsing (interpolated strings, raw string literals) could produce false positives.
- Trust scores are heuristic. The score is useful as a relative metric across a codebase, not as an absolute formal guarantee.

- Cross-file transitive analysis in the syntactic pass depends on theorem definition order within a file, not on the full import graph.

6. Future Work

- CI integration: fail a build when trust score drops below a threshold or when any theorem in a specified set becomes UNSOUND.
- Trust score tracking over time: graph how the verified surface area changes as AI tools contribute more proof code.
- Coq and Isabelle support: admit in Coq and sorry in Isabelle are direct analogues.
- Repair loop: feed sorry-tainted proofs back to an LLM with full context, re-run the audit, track which sorries get genuinely closed.