

Who Grades the Prover?

Grader Soundness and Agent Behaviour on Lean Proofs about Real Rust Code

Thomas Marchand

October 2026

Abstract

Formal verification benchmarks look easy to grade: the proof assistant’s kernel either accepts a proof or rejects it. We test this assumption when the prover is an LLM agent that can write arbitrary Lean. We use RFVB, a benchmark of 100 Lean 4 theorems about mechanically extracted Rust code, evaluated across eight models and 815 full agent transcripts. On the grading side, the released verifier accepts any build whose `#print axioms` lists only standard axioms. Lean’s own documentation warns that this check assumes an honest author, and we show what happens when it is used anyway: elaboration-time code can add declarations that the kernel never checks, and a single task-agnostic submission passes all 100 tasks without proving anything. Kernel replay with `leanchecker` rejects it. Replaying all 468 recorded passing proofs shows no model exploited this gap in practice. On the agent side, transcripts show two distinct strategies among strong models: submit only when certain, or submit and iterate. The turn budget is the real bottleneck, rather than the submission cap. Weaker models explore more per proof attempt. Agents also routinely use metaprogramming and prompt-forbidden tactics during exploration, which are the exact capabilities a cheating submission needs. Finally, a harness bug told agents 4,184 times that files containing `sorry` were clean, showing how directly the evaluation environment shapes agent behaviour.

1 Introduction

Theorem-proving benchmarks are appealing because the grader seems incorruptible. In reality, the kernel only checks what the harness hands it. An agent that writes Lean can also write Lean *metaprograms* that run during elaboration. The grader faces an adversary with the full power of the proof assistant’s extension language, while the agent relies on feedback from an environment it trusts.

We look at both problems on RFVB, a benchmark of 100 Lean 4 theorems transcribed from the `proptest` and `quickcheck` property tests of 33 Rust crates over their mechanical Lean translations (`cargo hax into lean`, built on Charon and Aeneas [3, 6, 7]). We ran eight models through a standard tool-using agent loop and logged every tool call.

Contributions.

1. **A grader-soundness analysis** (§3): a stage-by-stage adversarial evaluation of a released verifier, a task-agnostic non-proof accepted on 100/100 tasks, the root cause, a tested fix using kernel replay, and a re-verification of all 468 recorded passing proofs. The weakness is documented by Lean; the contribution is showing its concrete cost in an agentic benchmark.
2. **An agent-behaviour study** (§4) across 815 transcripts: submission strategies, budget exhaustion, exploration ratios, token cost, use of forbidden tactics and metaprogramming, and the impact of a harness feedback bug.

Table 1: Solved tasks, one run per model, scored on the statements as first released. Cluster CI: 95% bootstrap over the 38 crate modules, in percentage points. [†]97 valid tasks.

Model	Solved	Cluster 95% CI	Easy/21	Med./47	Hard/32
Gemini 4 Argon	87	[76, 94]	21	45	21
Claude Opus 5.5	84	[73, 91]	21	45	18
GPT-6.1 Sol	75	[64, 83]	21	42	12
GLM 5.3	46	[33, 59]	19	27	0
GLM 5.3 Flash	32	[20, 45]	16	16	0
Grok 4.7	22	[13, 31]	13	9	0
Muse Spark 1.3 [†]	16	[6, 26]	13	3	0
GPT-6 Luna	5	[1, 9]	4	1	0

3. Practical recommendations for building agentic formal benchmarks (§5).

2 Setting

Tasks. Each task fixes an extracted Rust implementation, a specification transcribed from one property test, and one editable file with the target theorem. In the specification, generators become quantified inputs, `prop_assume!` becomes hypotheses, and assertions plus panic-freedom become a weakest-precondition postcondition. Every task has a reference proof with a median length of 136 lines. The benchmark has 21 easy, 47 medium, and 32 hard tasks.

Agent. We run a simple OpenAI-compatible tool loop. The model gets `read_file` and `search` for exploration, `run_lean` to elaborate candidates for free (`sorry` allowed), and `check_proof` to submit (capped at 10 calls, ending the run on a clean local compile). Each task gets one attempt with a budget of 50 turns and 80 tool calls. An independent verifier scores the final file.

Results. Table 1 shows the solve counts. The top two models (87 vs. 84) are statistically tied (exact McNemar $p = 0.45$). Eleven tasks are unsolved by any model. Outcomes match the difficulty labels: easy, medium, and hard tasks are solved by an average of 6.1, 4.0, and 1.6 of the eight models.

3 Grader soundness

Threat model. A submission is an arbitrary Lean file that preserves the theorem statement. Lean elaborates the file by running tactics, commands, and attribute handlers, any of which can be user-written metaprograms. We want to know if a file can get a `pass` without proving the theorem.

The grader under test. The released verifier runs four stages: (1) a static check blocking forbidden tokens (`sorry`, `axiom`, `native_decide`, `run_cmd`), syntax extensions, and `set_option debug.*`, while enforcing an import allow-list and a textual statement match; (2) a fresh build of the submission; (3) a separate module checking that the submitted theorem has the canonical type; and (4) a check that `#print axioms` lists only `propext`, `Classical.choice`, and `Quot.sound`. Stage 4 is the main line of defense, since `sorry` or custom axioms normally show up in the axiom list.

Table 2: Adversarial submissions against the released verifier, by technique. Stage = the stage that rejected the submission. Bold: accepted although nothing was proved.

Technique	Static policy	Final verdict
Plain <code>sorry</code>	rejects	rejected (1)
<code>sorry</code> hidden after a character literal that confuses the comment/string stripper	passes	rejected (4: sorryAx)
Placeholder axiom referenced directly (sorryAx)	passes	rejected (4)
Tactic metaprogram (<code>run_tac</code> , not on the blacklist) closes the goal with <code>sorryAx</code>	passes	rejected (4)
Metaprogram adds a fresh axiom with the goal’s type	passes	rejected (4)
Spec definition shadowed in the task namespace	passes	rejected (2)
Extra section hypothesis added to the theorem	passes	rejected (3)
Metaprogram adds a declaration <i>with kernel checking disabled</i>	passes	passed

What Lean already says. The Lean reference is explicit about the limits of stage 4 [1]. It calls the axiom listing meaningful only when the authors are honest, names `debug.skipKernelTC` as code that “only serves to circumvent the system”, and counts un-reviewed AI-generated proofs as potentially malicious. For that case it recommends kernel replay with `lean4checker --fresh`, and sandboxed judging with `comparator` as the gold standard. RFVB’s verifier stopped at stage 4.

Adversarial evaluation. We wrote probes to bypass each stage (Table 2).

Root cause. Lean’s declaration-adding API checks `debug.skipKernelTC` from the *current* options. Elaboration-time code can modify these options in memory, bypassing the static check that only blocks the textual `set_option` command. The kernel never type-checks a declaration added this way. Lean writes it to the compiled module, and downstream importers trust it. Because its value can be any closed term, it uses no axioms, leaving stage 4 clean. A single task-agnostic proof body using this trick passed all 100 tasks, including the 11 tasks that no model solved. (We explain the mechanism enough to justify the fix, without publishing the exploit.)

Fix. The Lean toolchain includes `leanchecker`, which replays every declaration in a compiled module through the kernel [1, 9]. It rejects our fake proof because the stored value does not match the claimed type, and it accepts all reference proofs. Passing `--fresh` also replays imports, which increases verification time from seconds to minutes per task; our replays did not use it. Replay closes this bypass, but Lean’s documentation does not treat it as a full defense against a deliberately malicious submission, which is what `comparator` is for [10]. We also sandboxed the build with Linux Landlock to restrict writes to the working directory and block network and credential access. Sandboxing is good practice, though it does not fix the soundness bug.

Did anyone exploit it? We re-verified all 468 passing proofs from ten runs (the eight models in Table 1 plus two later runs). Every proof matches its recorded hash. 467 replay cleanly through the kernel, and the last one targets a task statement that was corrected after the run. No final submission uses metaprogramming commands, kernel-check options, `native_decide`, or `bv_decide`. No transcript tries to read reference proofs or files outside the workspace. The benchmark scores are real, though we only know this because of kernel replay.

Table 3: Agent behaviour per model. Explore/Lean: `read_file+search` calls per `run_lean+check_proof` call. Silent failures: failed runs that never called `check_proof`. 1st-try: solves accepted on the first submission. Turns: median turns on solved tasks. Tok./solve: prompt+completion tokens per solved task. Mislead.: `run_lean` results that called a file containing `sorry clean`.

Model	Solved	Expl./Lean	Silent fail.	1st-try	Turns	Tok./solve	Mislead.
Gemini 4 Argon	87	0.45	10/13	85/87	21	1.0M	471
Claude Opus 5.5	84	0.49	16/16	84/84	18	1.3M	465
GPT-6.1 Sol	75	0.42	1/25	58/75	16	1.1M	615
GLM 5.3	46	1.01	53/54	45/46	31	3.8M	296
GLM 5.3 Flash	32	1.22	63/68	28/32	26	5.4M	210
Grok 4.7	22	0.72	70/78	19/22	32	8.1M	434
Muse Spark 1.3	16	0.89	81/81	10/16	31	11.6M	865
GPT-6 Luna	5	0.41	1/95	2/5	26	21.5M	828

Harness-level soundness. Two harness flaws affected the runs without corrupting the proofs. First, `run_lean` ran agent-written Lean directly on the host without a sandbox, and agents did execute code this way (§4). Second, `check_proof` stopped the run on a clean local compile without checking axioms. Three runs ended early on `bv_decide` proofs that the verifier later rejected because they rely on a native-evaluation axiom [4]. Those agents lost their remaining turn budget.

4 Agent behaviour

We analysed the final archived run for every task and model, totalling 815 transcripts (including re-runs and one excluded model). Table 3 summarizes the numbers.

Two submission strategies. Top models take two different approaches. Claude Opus 5.5 and Gemini 4 Argon only submit when they are sure the proof is done. All 84 of Claude’s solves passed on the first submission, and none of its 16 failed runs ever called `check_proof`. GPT-6.1 Sol submits early and iterates: 17 of its 75 solves took 2–4 submissions, and 24 of its 25 failed runs submitted at least once. GPT-6 Luna takes this too far, making 461 submissions for just 5 solves and hitting the 10-submission cap on 10 tasks. Both main strategies can reach the top of the leaderboard. The difference matters when building a harness, because an iterative agent relies on accurate submission feedback.

Turn budget is the real bottleneck. Nearly every failure comes from hitting the 50-turn limit. Only 12 runs across all models hit the submission cap instead. We checked if agents were failing to submit working proofs: failed runs contain only 0–2 tasks per model where a `run_lean` call elaborated the target theorem without `sorry`. Failed runs are genuine non-solutions within the budget, meaning the leaderboard partly measures how fast a model proves per turn.

Weaker models explore more. The top three models make 0.42–0.49 exploration calls per Lean call and solve tasks in a median of 16–21 turns. The middle tier makes 0.72–1.22 exploration calls per Lean call and takes 26–32 turns. Token costs diverge further, from roughly 1M tokens per solve for the top three models to 8–22M for the weakest. GPT-6 Luna is the outlier, pairing a low exploration ratio with hundreds of failed submissions.

Agents use forbidden and powerful tools. The system prompt forbids `native_decide` and `bv_decide`. Grok 4.7 still tried them 94 times across 29 tasks during exploration, and three other models used them occasionally. Five models used metaprogramming or `#eval` in exploration (`run_tac`, `run_cmd`, `#eval`), mostly to inspect the environment and list constants. Three models also added temporary axiom declarations to sketch out proofs before proving the lemmas. None of this made it into a passing submission, but it shows that running metaprograms during elaboration is already part of normal agent behaviour. Assuming agents will not use these features is wishful thinking.

A feedback bug shaped agent behaviour. The harness appended “*This file elaborated without errors or sorry ... submit it with check_proof*” to `run_lean` output whenever the text lacked `declaration uses 'sorry'`. Lean 4.31 prints that warning with backticks instead of single quotes, so the harness attached this message to files that actually contained `sorry` 4,184 times across the eight models (Table 3). GPT-6 Luna, which received 828 of these messages, submitted files containing `sorry` 70 times, all rejected by the static check. The logs alone do not prove causation, but agents act on harness messages, and a one-character mismatch in a feedback check stays hidden if you only look at final scores. We fixed the bug.

5 Recommendations

1. **Follow Lean’s validation guidance.** Use `#print axioms` only as a fast filter. Replay the compiled module with `leanchecker --fresh`, and for adversarial settings judge with a sandboxed comparator [10].
2. **Assume agents run metaprograms.** Sandbox every process that elaborates agent-written Lean, including exploration calls, and strip credentials from the environment.
3. **Match local acceptance to final grading.** Any tool that ends the run must run the full grader checks, including axiom audits, so agents do not waste their budget on rejected proofs.
4. **Test harness feedback strings.** Agents act on harness messages. Regression-test your output parsers against the exact compiler version you use.
5. **Publish transcripts and replay scripts.** We could audit our results only because we saved the full transcripts and final submissions.
6. **Report paired uncertainty.** With a single run per model and tasks grouped by crate, adjacent ranks are often not meaningful.

6 Related work

Most Lean code-proving benchmarks use competition problems with hand-written specs [11, 13, 14]. FVSpec [5] also extracts specs from property tests, targeting Python. Agentic proving on Aeneas output has already been demonstrated at industrial scale [8]. Ammanamanchi et al. [2] audit five Lean benchmarks and find dataset bugs and evaluation flaws that skew scores both ways. Our grader bypass is a specific case of this in an agentic setup, where the submission itself runs code. Lean’s documentation separates the axiom listing from kernel replay and sandboxed judging and states which threats each one covers [1]; native evaluation carries its own trust risks [4]. Verity [12], a verified compiler for EVM smart contracts in Lean 4, reports its proof status with the same three-axiom audit and lists its macro elaborator in the trusted computing base. That trust is reasonable

for code reviewed by its authors. It is the assumption that breaks when the author of the Lean code is an untrusted agent.

7 Limitations

We ran each model once without recording sampling parameters, so some behavioural differences may come from provider defaults. We only kept transcripts for the final run of each task. Our behavioural analysis is descriptive, as we did not run controlled ablations on the feedback bug or the turn budget. We ran kernel replay out of band and without `--fresh`; it is not yet wired into the released verifier.

8 Conclusion

When the prover is an agent, the grader has to treat every submission as untrusted code. Checking the axiom list is not enough, and Lean’s documentation already says so; kernel replay closes the bypass we found, and sandboxed judging is the documented standard for adversarial submissions. Our transcripts show this is a practical concern: agents already write metaprograms, try forbidden tactics, and act on buggy harness feedback. No model exploited the grader in our runs, which means the benchmark’s integrity held up by luck rather than by design.

References

- [1] Validating a Lean proof. The Lean Language Reference, 2026. URL <https://lean-lang.org/doc/reference/latest/ValidatingProofs/>. Accessed 2026-10-18 (page links to reference version 4.35.0).
- [2] Pawan Sasanka Ammanamanchi, Siddharth Bhat, and Stella Biderman. Faults in our formal benchmarking: Dataset defects and evaluation failures in Lean theorem proving, 2026. URL <https://arxiv.org/abs/2606.29493>. Accepted at ICML 2026.
- [3] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical Rust software using multiple provers. In *Verified Software. Theories, Tools and Experiments (VSTTE 2024)*, volume 15525 of *Lecture Notes in Computer Science*, pages 96–119. Springer Nature Switzerland, 2025. doi: 10.1007/978-3-031-86695-1_7. Preprint: Cryptology ePrint Archive, Paper 2025/142.
- [4] Joachim Breitner. RFC: One axiom per native computation. leanprover/lean4 GitHub issue #12216, 2026. URL <https://github.com/leanprover/lean4/issues/12216>.
- [5] Quinn Dougherty, Max von Hippel, Simon Henniger, Hazel Shackleton, and Mike Dodds. FVSpec: Real-world property-based tests as Lean challenges, 2026. URL <https://arxiv.org/abs/2606.01008>.
- [6] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022. doi: 10.1145/3547647.
- [7] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. Charon: An analysis framework for Rust, 2024. URL <https://arxiv.org/abs/2410.18042>.

- [8] Son Ho, Cédric Fournet, Jonathan Protzenko, Michael Naehrig, Joshua Clune, Patrick Longa, Guillaume Boisseau, Fernando Leal Sánchez, Aymeric Fromherz, and Antoine Delignat-Lavaud. Scaling verification of cryptographic software with Aeneas, Rust, and Lean, 2026. URL <https://arxiv.org/abs/2609.15648>.
- [9] leanprover. lean4checker: Replay the Environment for a given Lean module, ensuring that all declarations are accepted by the kernel. GitHub repository, 2026. URL <https://github.com/leanprover/lean4checker>. Deprecated; merged into the Lean toolchain as `leanchecker` from v4.28.0. Accessed 2026-10-18.
- [10] leanprover. Comparator: a trustworthy judge for Lean proofs. GitHub repository, 2026. URL <https://github.com/leanprover/comparator>. Deprecated; merged into core as `lake comparator` as of Lean v4.35.0. Accessed 2026-10-18.
- [11] Evan Lohn and Sean Welleck. miniCodeProps: a minimal benchmark for proving code properties, 2024. URL <https://arxiv.org/abs/2406.11915>.
- [12] Thomas Marchand. Verity: A verified compiler for a core fragment of EVM smart contracts in Lean 4. ETHReS: Ethereum Research Symposium @ ETHPrague, Prague, May 8–10, 2026, 2026. URL <https://veritylabs.dev/research/verity>. Proceedings to appear in CEUR Workshop Proceedings. Accessed 2026-10-18.
- [13] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025. URL <https://openreview.net/forum?id=Ib0acMF5qd>.
- [14] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *International Conference on Learning Representations (ICLR)*, 2026. URL <https://openreview.net/forum?id=0A4Uf88pog>.