Lean Disprove: Certified Counterexample Search for AI-Assisted Formal Mathematics

Jan Ondras [1] **Cameron Freer** [1]

Abstract

LLM-driven formal mathematics has focused mainly on *proving* true statements; the complementary task of *disproving* false ones (finding a counterexample the proof assistant can certify) has had little dedicated tool support. We present `/disprove`, an interactive command for coding agents in the open-source `lean4-skills` package. Given a Lean goal, the command runs a bounded six-phase cycle to perform *certified counterexample search*: it profiles the target, consults prior knowledge, selects a disproof method from a registry (including decision procedures, finite enumeration, random sampling, and external scripts such as SAT/SMT queries), searches for a witness, and assembles a shape-specific negation term. It reports REFUTED only when Lean accepts the refutation, and otherwise reports WITNESS-UNCERTIFIED, when a candidate witness was found but could not be certified, or INCONCLUSIVE, when no candidate was found. External witnesses are never trusted directly, but must be lifted into Lean and checked by the kernel. We demonstrate on a pilot benchmark that it certifies counterexamples a bare decision procedure cannot reach, while never refuting a true statement.

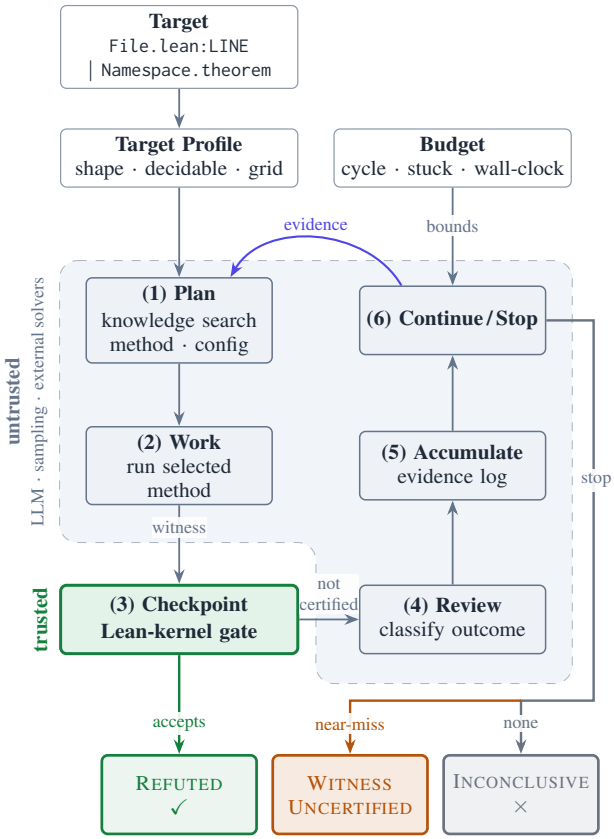

Figure 1. System overview of `/disprove`, an agentic coding command in the `lean4-skills` package (Freer, 2025). A six-phase loop searches for a counterexample with *untrusted*, LLM-driven methods, but a witness is reported as a disproof only if the *trusted* Lean kernel accepts it at the Checkpoint gate. Each session ends in one of three outcomes (§3).

1. Introduction

Formalized mathematics benefits from having both a prover and a *falsifier*. When an AI agent is given a statement to prove, ideally it would cheaply ask whether the statement is already refutable before spending inference budget on a proof attempt. Our work provides this falsification primitive, which produces a Lean-kernel certificate when successful. Because the check is kernel-backed, it keeps an LLM-driven formalization pipeline verifiable: every falsity claim is settled by the kernel, not the model.

Research and benchmarks in LLM-driven formal mathematics concentrate primarily on *proof construction*: PutnamBench (Tsoukalas et al., 2024), MiniF2F (Zheng et al., 2022), and ProofNet (Azerbayev et al., 2023) all measure the ability to prove statements presumed true. The complementary task of *disproof* (producing a certified counterexample to a false conjecture) has received less attention; the most

[1]Massachusetts Institute of Technology, Cambridge, MA, USA. Correspondence to: Jan Ondras <janko@mit.edu>, Cameron Freer <freer@mit.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

direct prior effort, Li et al. (2026a), fine-tunes an LLM (via expert iteration) to emit Lean-verified counterexamples. We argue that disproof deserves dedicated, *interactive* tool support inside the proof assistant, with the trust burden on the kernel.

A motivating failure. Consider a concrete episode from an ongoing Lean formalization of the theory of dense graph limits.[1] A central lemma asserted that, for twin-free graphs, equal closed-walk (diagonal spectral-moment) profiles force two vertices into the same orbit. It was checked against four corpora (over 22,000 twin-free graphs) with zero counterexamples, and a roughly 500-line Lean proof was planned around it. A targeted search then exhibited an unweighted 9-vertex "double-pin tree" (twin-free, trivial automorphism group) whose vertices 1 and 5 have identical closed-walk profiles: for the adjacency matrix A, $(A^m)_{11} = (A^m)_{55}$ for all $m \geq 0$ (checking $m = 0, \ldots, 8$ suffices by Cayley–Hamilton), yet they lie in different orbits, refuting the lemma and sparing the planned proof effort. That witness was found by an *ad hoc* Python script, typical of how working formalizers use falsification today, and the workflow we set out to systematize inside the proof assistant. This episode is motivational rather than a benchmark result: `/disprove` did not produce the witness, it is not certified in Lean, and refuting this graph claim lies beyond the command's current scope. Our benchmark instead includes a toy "validated-then-false" analogue in the same spirit (§5).

Falsification is harder to support in a proof assistant than testing is in a programming language, and existing tools cover only pieces of it. Random testers such as Lean's `plausible` (Böving, 2024) surface a failing input but not a refutation term. Mathlib's `Counterexamples/` (The mathlib Community, 2026) is a curated source library, not a search procedure. The `lean-smt` tactic (Mohamed et al., 2025) reconstructs SMT proofs in the kernel, but targets proving, not counterexample search. What is missing is support that *searches* for a witness across methods and then *certifies* it in the kernel.

We present `/disprove`, an interactive *command* in the open-source `lean4-skills` package (Freer, 2025).[2] The `/disprove` command is invoked in an LLM coding agent to perform *certified counterexample search*. Given a goal, it runs a six-phase cycle (§4.1) that, each pass, consults prior knowledge, selects and runs a disproof method, and certifies any witness it finds in the kernel; it returns one of three outcomes, emitting a refutation only when `lake env lean` accepts the assembled term and the refuting declaration contains neither `sorry` nor `admit` and invokes no axioms outside an approved whitelist (Definition 3.1).

[1] https://github.com/cameronfreer/graphon

[2] https://github.com/cameronfreer/lean4-skills

Contributions.

1. *Problem formulation.* Certified counterexample search as a tri-state task (REFUTED / WITNESS-UNCERTIFIED / INCONCLUSIVE) gated by the Lean kernel (Sec. 3).
2. *An interactive six-phase cycle* with dynamic, evidence-ranked menus (knowledge search, method, configuration), explicit user checkpoints, and an audit log; later cycles re-rank methods from accumulated evidence rather than a hardcoded table (Secs. 4.1, 4.4, 4.5).
3. *Shape-aware certification.* Seven canonical goal shapes (Table 1) with per-shape Lean recipes, an atom-tactic fallback cascade, and untrusted external SAT/SMT witnesses lifted into kernel-checked terms (Sec. 4.4).
4. *Open implementation and diagnostic evaluation.* We release the command and a 16-target benchmark on which `/disprove` certifies all 15 false targets; a bare `decide` baseline reaches 10, and each added capability certifies at least one further target the lower tiers miss (Sec. 5).

2. Background and Related Work

LLM-driven formal theorem proving. LeanDojo (Yang et al., 2023), Lean Copilot (Song et al., 2025), DeepSeek-Prover (Xin et al., 2024), COPRA (Thakur et al., 2024), and LeanAgent (Kumarappan et al., 2025) are all proving-centric: the conjecture is assumed true and the task is to find a proof.

Falsification in proof assistants. Mathlib's `Counterexamples/` (The mathlib Community, 2020; 2026) is a manual library of refutations. Property-based random testers report a failing input, not a kernel-checked term: Lean's `plausible` (Böving, 2024) (which continues mathlib's earlier `slim_check` (Hudon & The mathlib Community, 2024)), the QuickCheck lineage (Claessen & Hughes, 2000), and Coq's QuickChick (Dénès et al., 2014). Model finders such as Isabelle's Nitpick (Blanchette & Nipkow, 2010) and Nunchaku (Reynolds et al., 2016; Cruanes & Blanchette, 2016) instead search for countermodels (Nunchaku even imports them into proof assistants), but operate outside Lean and emit no Lean proof. `lean-smt` (Mohamed et al., 2025) reconstructs SMT proofs in the Lean kernel, but targets proving, not disproof search.

Closest prior work. Li et al. (2026a) *fine-tunes* an LLM via multi-reward expert iteration on synthetically mutated data (deleting hypotheses from true theorems) to generate counterexamples whose proofs are checked in Lean 4. We share that kernel-checking discipline but differ on three axes: (i)

training-free: we orchestrate a search over a fixed method registry rather than training an LLM; (ii) *multi-method*: incorporating untrusted external SAT/SMT solvers whose witnesses are lifted into kernel-checked Lean terms (their pipeline is LLM-only); and (iii) an *interactive*, evidence-adaptive loop. Our contribution is the integration of these axes, together with the WITNESS-UNCERTIFIED diagnostic and the disproof-direction reframing of the task. Their code and a 3,000-problem benchmark are public (Li et al., 2026b); we do not run a head-to-head here because the two target different regimes. Theirs is generated by dropping hypotheses from Mathlib theorems and is dominated by competition problems and real, polynomial, and measure-theoretic statements, whereas our training-free registry targets decidable, finite, and solver-amenable goals (§4.2); only about a quarter of their targets fall in that regime, so a like-for-like run would chiefly evaluate the command outside its intended scope. Extending our shape and method coverage to their setting is future work.

Interactive AI agents in proof assistants. ImProver (Ahuja et al., 2024) and LeanAgent (Kumarappan et al., 2025) are LLM agents for *proving*; Aesop (Limperg & From, 2023) is a white-box proof-search tactic. We position `/disprove` as a command that produces *verified* artifacts in the disproof direction, not a tactic generator.

3. Problem Formulation

We target the following task. The input is a Lean *goal*, a declaration `T` whose type is a proposition P, possibly under a telescope of binders. The output is either a *certified counterexample*, a Lean term the kernel accepts as a proof of the shape-appropriate negation of P, or an explicit report that no such term was found within the search budget. The command never asserts falsity without a kernel-checked witness.

Definition 3.1 (Certified counterexample)**.** Let the target `T` have type $\forall x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ P$. A *certified counterexample* is a Lean term whose type is the shape-appropriate disproof goal for P (Table 1) and whose declaration is accepted when the file is checked with `lake env lean`. We report REFUTED only if the refuting declaration contains no `sorry` or `admit` and its axioms lie within the approved whitelist: `propext`, `Classical.choice`, and `Quot.sound`, together with `Lean.ofReduceBool` only when `native_decide` is explicitly enabled. For a Shape-1 (universal) goal the disproof goal is $\exists x,\ \neg P\, x$; for a decidable finite goal the term may instead prove $\neg\, \forall x,\ P\, x$ directly by decision procedure, without extracting a witness. A pre-existing `sorry` elsewhere in the file (e.g., the original conjecture, left in place with the refutation appended below) does not affect the verdict.

Tri-state outcome. A session ends in exactly one of three outcomes. REFUTED: a certified counterexample was produced (Definition 3.1). WITNESS-UNCERTIFIED: a candidate witness was found but the assembled term failed certification, either because the kernel rejected it (typically a non-decidable residual) or because a typecheck-passing term invoked an axiom outside the allowed set; a near-miss. INCONCLUSIVE: the cycle, stuck-cycle, or wall-clock budget was exhausted with no certified witness found. Only REFUTED licenses a falsity claim; the other two are reported as "not refuted here," never as evidence that the conjecture holds.

Why kernel-checked matters. This contract is what separates `/disprove` from property testers. A random tester such as `plausible` or `slim_check` reports a *failing input*: a useful signal, but not a proof, and one that says nothing when the sampler is ill-specified or the predicate is undecidable on the sampled point. Routing every refutation through the Lean kernel turns "found a counterexample" into a checkable theorem, at the cost of having to construct the negation term, which the method families of §4.3 automate.

4. System Design

4.1. Six-Phase Cycle

A session runs as a sequence of *cycles*, each comprising six phases. **(1) Plan** presents three dynamic menus: the *knowledge-search* menu (Step 0, optional; §4.4), the *method* menu (Step 1, which method family to run), and the *configuration* menu (Step 2, its parameters). The LLM agent driving the command synthesizes and ranks these menus each cycle, choosing which registry methods (§4.3) to surface and how to configure them from the target profile and the evidence gathered so far. It cannot, however, violate the menu invariants (§4.5). **(2) Work** executes the chosen method. **(3) Checkpoint** emits the artifact on success and, optionally, commits it. **(4) Review** classifies the outcome and records any near-miss signature. **(5) Accumulate** appends the cycle's evidence (§4.5), and **(6) Continue/Stop** re-enters Plan or terminates. Three budgets bound a session: a maximum number of cycles, a maximum number of consecutive stuck cycles, and a wall-clock limit. Figure 1 shows the loop, highlighting the dynamic menus and the evidence feedback from Continue/Stop back to Plan.

4.2. Targets, Shapes, and Profile

A target is named either by source location (`File.lean:LINE`) or by qualified name (`Namespace.theorem`); the command resolves it through the Lean language server and normalizes its type into one of seven canonical *shapes* (Table 1). Each shape fixes the disproof goal. For example, a universal (Shape 1) is refuted

by an existential witness, an existential (Shape 3) by a universal non-existence proof, a conjunction by refuting one conjunct. The command can therefore use a fixed certification recipe per shape. Shape 6 (equality and order atoms) is the common special case of the general decidable atom (Shape 7), split out for its canonical negation ($a \neq b$, $\neg(a \leq b)$) and its preferred atom tactic (`omega`/`norm_num` rather than generic `decide`).

Table 1. The seven canonical goal shapes `/disprove` recognizes, each with its *disproof goal*: the shape-appropriate negation the Lean kernel must accept. P, Q are predicates (Shapes 1–3) or propositions; a, b are terms.

#	Goal shape	Disproof goal
1	$\forall x : \alpha,\ P\,x$	$\exists x,\ \neg P\,x$
2	$\forall x : \alpha,\ P\,x \to Q\,x$	$\exists x,\ P\,x \wedge \neg Q\,x$
3	$\exists x : \alpha,\ P\,x$	$\forall x,\ \neg P\,x$
4	$P \wedge Q$	$\neg P \vee \neg Q$
5	$P \vee Q$	$\neg P \wedge \neg Q$
6	$a = b$	$a \neq b$
	$a \leq b$	$\neg(a \leq b)$
7	Decidable atom P	$\neg P$

Once per session the command builds a *target profile*. A deterministic profiler resolves the target to its source location; using the Lean language server, the agent then determines the shape, the free variables and their types, the cardinality of any finite binder type, decidability, and a candidate grid for enumeration. The profile gates the menus. A decidable goal over a small finite type surfaces `decide-cascade` at the top of the ranked menu (*rank-1*), whereas an unbounded $\mathbb{N}$-quantified goal surfaces witness search instead; this is the split that separates the regimes of §5, where bare `decide` suffices for the former but not the latter. After peeling a binder prefix the command re-classifies the body by its outermost connective, so the shape assignment is unambiguous. A goal whose core matches none of the seven shapes (an atom over non-decidable or dependently-typed structure, or quantifier nesting that the prefix peel does not resolve) surfaces no applicable method and is reported INCONCLUSIVE. The seven shapes bound what the command attempts; other goals fall outside its scope.

4.3. Method Registry

The `/disprove` command draws its methods from a declarative *registry* (Table 2); the LLM selects and configures entries but cannot invent method identifiers, and the schema (cost class, applicable shapes, parameter types) is validated at load time. Six families span a cost/coverage spectrum: `decide-cascade` (kernel decision procedures), `mine` (reuse literals from `Inhabited` or example declarations), `enumerate` (iterate a finite range, discharging each candidate with an atom tactic), `plausible` (random sampling, the one non-certifying family, used only to locate a witness), `tactics` (a general tactic cascade), and `external` (an LLM-emitted script run outside Lean, e.g., a SAT/SMT query, or a custom routine such as a Python implementation of an algorithm surfaced by the knowledge-search menu of §4.4). The `external` method is *untrusted*: it may propose a witness, but `/disprove` reports REFUTED only after that witness is lifted into a Lean term and accepted by the kernel under the axiom whitelist (Definition 3.1).

Table 2. The six disproof method families in `/disprove`'s registry, spanning a cost/coverage spectrum. *Shapes* index Table 1; *Cert.* is whether the family yields a kernel-checked term on its normal path: ✓ always, ∼ sometimes (otherwise it only narrows the search), × never (locates a witness only).

Family	Cost	Shapes	Cert.	Mechanism
`decide-cascade`	cheap	1–3,6,7	✓	`decide`/`omega`
`mine`	cheap	1,2	∼	`Inhabited` literals
`enumerate`	medium	1,2	✓	finite-range scan
`plausible`	medium	1,2	×	random sampling
`tactics`	medium	all	✓	tactic cascade
`external`	costly	1,2,6,7	✓*	Python/SAT/SMT

* `external` witnesses are lifted into a Lean term and kernel-checked; the script itself is not part of the trusted base.

4.4. Knowledge Search and Certification Recipes

Before searching, the optional knowledge-search menu (Step 0) consults external knowledge across three sources: a mathlib search, a local search of the repository and mathlib's `Counterexamples/` for known refutations, and a web search for known counterexamples (or for known non-counterexamples that prune the space). Knowledge search only changes the ranking. Every retained finding must carry a source, and a web finding is fetched and spot-verified before it can become a rank-1 suggestion; a per-cycle budget caps re-entry. The kernel still certifies every REFUTED.

Each shape carries a Lean *certification recipe* that assembles the negation term from a witness (Listing 1). The hole left for the residual obligation is filled by an *atom-tactic cascade*: `decide`, `native_decide`, `norm_num`, `omega`, `simp`, `rfl`, tried in this fixed fallback order, with `native_decide` surfaced only when explicitly enabled (it alone enlarges the trusted base). If the first atom fails the pre-screen, the cycle falls back to the next before downgrading to WITNESS-UNCERTIFIED, so a correct witness with an awkward residual is not discarded prematurely.

4.5. Evidence-Driven Ranking and Audit

After each cycle, Accumulate appends a record (family, configuration, outcome, near-miss signature) to the session's evidence log, and in the next Plan the LLM re-ranks the

Listing 1 A kernel-checked counterexample `/disprove` emits for a Shape-1 goal (`EasyBad.lean`). The negated goal is decidable, so the cascade discharges it with `decide` directly without extracting a witness, and `#print axioms` reports no axioms at all. Listings 2–3 instead build the existential witness form $\exists x,\ \neg P\,x$ of Table 1.

```
-- Refuting easy_bad : ∀ n : Fin 5, n.val < 3
-- Witness n = 3 (n.val = 3, not < 3); the negated
-- goal is decidable on the finite type Fin 5.
theorem easy_bad_counterexample :
    ¬ (∀ n : Fin 5, n.val < 3) := by decide
```

method and configuration menus *conditioned on this evidence*. There is no hardcoded escalation table: the search widens (each cycle proposes new, non-overlapping territory) by reading what has already failed, a select-and-refine loop. A cycle is *stuck* when no certified outcome was reached and no admissible (family, configuration) pair remains untried under the menu invariants, which forbid exact repeats and overlapping search spaces, and require a `decide`-family entry in the top 3 whenever the profile is decidable and no decision procedure has yet been tried.

Every session leaves an audit log: surviving knowledge-search findings are logged with their sources. The artifact writer only appends the refutation, without modifying existing declarations; the Checkpoint phase re-runs `lake env lean` and the axiom-whitelist check before any REFUTED is reported, rolling the append back on failure. The one capability that enlarges the trusted base (`native_decide`, which admits `Lean.ofReduceBool`) is flagged as audit-worthy and surfaced only as an explicit opt-in, never silently.

5. Experiments

5.1. Benchmark Construction

Our benchmark is a set of 16 conjectures (15 false, 1 true control) we hand-constructed to cover five of the seven canonical shapes (Table 1) and more importantly, a *difficulty gradient* that exercises successively more of the command's components. It is a diagnostic suite: each target isolates one capability, so the study shows which capabilities reach which goal categories and does not estimate performance on a natural problem distribution. Four regimes: (i) *decidable-finite* goals (universals/existentials over `Fin n`, an equality, an inequality, a divisibility, a primality atom), refuted by a kernel decision procedure directly; (ii) *bounded*-$\mathbb{N}$ goals whose witness a finite `enumerate` scan finds:$\forall n,\ n^2 \le 10n$ (false at 11), the Shape-2 $\forall n,\ n \le 5 \to n^2 \le 20$ (false at 5), and the "validated-then-false" $\forall n,\ \mathrm{Prime}(n^2{+}n{+}41)$ (prime for all $n \le 39$, composite at $n{=}40$, where it equals $1681{=}41^2$; the toy analogue of §1's research-scale failure); (iii) an *out-of-window* goal whose witness lies beyond the default scan and is reached only by multi-cycle widening ($\forall n,\ n^2 \le 1000n$, false at 1001); and (iv) a *nonlinear*-$\mathbb{Z}$ goal that `decide`, `omega`, and one-dimensional `enumerate` all miss but an `external` SMT search solves ($\forall x\, y : \mathbb{Z},\ 2x^2 - 3y^2 \ne 5$). Targets are authored by hand, not mined from a searchable corpus, sidestepping the leakage hazard of drawing from `Counterexamples/` (the directory the command's local search scans). Hand-authoring removes leakage into the benchmark; it does not by itself prevent the command's Step-0 search from reusing a known refutation in `Counterexamples/`, so a Step-0-off (held-out) comparison is the cleaner control and is future work. The sole true target ($\forall n : \mathtt{Fin}\,5,\ n.\mathrm{val} < 5$) is a control that must yield no counterexample.

5.2. Setup

We used Lean 4 (de Moura & Ullrich, 2021) `v4.29.1` and the default search budgets (at most 3 cycles, 2 consecutive stuck cycles, and a 5-minute wall-clock limit per target). We drive the full pipeline end-to-end (method selection, search, artifact emission, and the `lake env lean` gate) on the non-trivial targets: multi-cycle widening for the out-of-window goal, an external Z3 query for the nonlinear goal, and a single `enumerate` pass for the bounded-$\mathbb{N}$ goals; the 10 decidable-finite targets are certified by `decide-cascade` directly. Three targets thus exercise the multi-cycle or external machinery; the rest are single-method certifications. The runs are inexpensive: every target certifies in a single cycle except the out-of-window goal (2 cycles), the suite issues exactly one external solver call (the Z3 query), and each REFUTED is settled by a single `lake env lean` kernel gate. We do not report fine-grained timings; on this suite the $\sim$3 s `lake env lean` startup dominates runtime. The rank-1 method per cycle is fixed by the target profile, so the scripted-operator protocol below is deterministic and reproducible (no LLM nondeterminism); multi-cycle *widening* is the cycle-2 menu offering a wider range after cycle 1 reports no witness. The `external` search uses Z3 4.15.4 (de Moura & Bjørner, 2008), treated as *untrusted*: its witness is rechecked by the Lean kernel. `native_decide` is left off, so no certified term admits a compiler-trust axiom.

Qualitative read of the menu ranking. The rank-1 choices follow the target profile as expected: `Fin` and other finite decidable goals select `decide-cascade`; an exhausted `enumerate` window triggers a wider, non-overlapping range; and the nonlinear-$\mathbb{Z}$ goal (§6) selects `external` SMT once `decide`, `omega`, and one-dimensional `enumerate` are inapplicable. We treat these transcripts as sanity checks, not a quantitative ranking evaluation.

Evaluation protocol (guided command, scripted operator). The command is *guided*: it presents the three menus and per-cycle checkpoints and gates `external` scripts on user approval (cf. the guided `/prove` in the same package, Freer, 2025; a hands-off `/autoprove`-style mode is future work). So that the evaluation measures the *command* rather than operator expertise, we run every problem under a fixed *scripted-operator* protocol: always take the top-ranked (rank-1) method and configuration, auto-approve `external` scripts from a fixed sandboxed allowlist, and disable per-cycle committing. The operator is thus a mechanical rank-1 selector, so the reported numbers reflect the command's ranking, search, and certification.

5.3. Main Results

Table 3 reports cumulative coverage as each capability is switched on. A bare-`decide` baseline (one `decide` call, no search) certifies 10 of the 15 false targets (the decidable-finite ones) and none of the unbounded or nonlinear goals, where it cannot even synthesize a `Decidable` instance. Adding finite `enumerate` reaches 3 more (witnesses $n = 11, 40, 5$); multi-cycle *widening* reaches the out-of-window target ($n = 1001$, found in cycle 2 after $[5, 64)$ is exhausted); and an `external` Z3 search (whose model $(-4, 3)$ is re-checked by the kernel) reaches the nonlinear target. The `/disprove` command certifies all $15/15$ false targets and correctly does not refute the lone true control.

Table 3. Cumulative certified refutations as each capability is switched on, of 15 false targets (the 16th is a true control, correctly not refuted). Each added tier certifies a target the lower tiers miss.

Capability (cumulative)	Certified
bare `decide`	10 / 15
+ `enumerate` (1 cycle, $n \in [5, 64)$)	13 / 15 (+3)
+ multi-cycle widening	14 / 15 (+1)
+ `external` SMT (Z3), kernel-rechecked	**15 / 15** (+1)

The tiers are validated cumulatively, not independently: on this hand-built suite no tier is redundant, and dropping any one leaves a target uncertified. (Widening earns this status given the default $[5, 64)$ window; a larger single window would also reach $n = 1001$, so its contribution is the evidence-driven automation of growing the window, not unique reachability.)

This evaluation is designed to test capabilities, not to estimate real-world success rates. Because each target's profile fixes the rank-1 method and the kernel fixes the verdict, the outcomes are deterministic, with no run-to-run variance for a confidence interval to summarize. Because the targets were built to cover specific case types rather than sampled at random, a single pooled accuracy would be misleading. Broken down by case type, the command certifies all 10 decidable-finite targets, all 3 bounded-$\mathbb{N}$ targets, the 1 out-of-window target, and the 1 nonlinear-$\mathbb{Z}$ target; it correctly does not refute the true control. Every certified term passes `#print axioms` with nothing beyond Lean/mathlib's three standard axioms (`propext`, `Classical.choice`, `Quot.sound`); `native_decide` is disabled, so no result depends on `Lean.ofReduceBool`.

5.4. Ablations

Table 3 also functions as an ablation: it exercises the three families that certify results in this suite (`decide-cascade`, `enumerate`, `external`) and shows what each one adds. The other three families and the Step-0 knowledge search are not exercised here and so are not separately scored: `mine` reuses existing literals, `plausible` only locates a witness for sampleable Shape-1/2 goals before the same recipes certify it, `tactics` is a general fallback cascade, and Step-0 supplies external knowledge (as in the §1 graphon episode). Scoring each in isolation, with `tactics`-only and `plausible`-only baselines, needs only further runs under the scripted-operator protocol (§5.2) and is left to future work; the larger ranking grid (knowledge search on/off, LLM ranking vs. random/heuristic) needs the autonomous batch mode to run at scale.

5.5. Failure-Mode Analysis

The `/disprove` command does not emit a spurious REFUTED: every REFUTED verdict is backed by a kernel-checked proof of the negation. The one true control is correctly not refuted. And when a witness lies beyond a method's current reach the outcome is the conservative INCONCLUSIVE, not a guess: the out-of-window target is INCONCLUSIVE under the default scan and becomes REFUTED only once cycle-2 widening reaches $n = 1001$. A WITNESS-UNCERTIFIED near-miss, where a witness is found but its residual is not closed by the atom cascade (`decide`/`native_decide`/`norm_num`/`omega`/`simp`/`rfl`), did not arise in this set: every witness we located had a cascade-closable residual and an axiom set within the whitelist. That middle state is therefore unexercised here, a limitation of the suite rather than evidence it is uncommon; it would arise for a semantically-correct witness whose residual needs reasoning beyond the cascade (e.g. an existence fact requiring a non-trivial lemma), and adding such a target is future work.

6. Case Study

Two sessions illustrate the capabilities a bare decision procedure lacks. The outcomes, witnesses, certified terms, and axiom reports below are the real outputs of our runs (§5); the interactive menu prompts are abridged.

Multi-cycle widening (Listing 2). On $\forall n : \mathbb{N},\ n^2 \leq 1000n$, cycle 1 enumerates $[5, 64)$ and finds nothing; **Accumulate** records the exhausted range, and cycle 2's configuration menu widens to the non-overlapping $[64, 1024)$, where $n = 1001$ falsifies the goal. The Shape-1 recipe certifies $\langle 1001,$ `by norm_num`$\rangle$. This is the evidence loop of §4.5 in action: the second range is chosen because the first was exhausted.

Listing 2 Multi-cycle session: cycle 1 exhausts the default window, the evidence loop widens, and cycle 2 certifies the witness $n = 1001$.

```
$ /disprove HardBad.lean:5   -- ∀ n:ℕ, n²≤1000n
[Cycle 1]    enumerate [5,64), atom=norm_num
             → exhausted-no-witness
[Accumulate] [5,64) exhausted; widen next cycle
[Cycle 2]    enumerate [64,1024)  (non-overlapping)
             → witness n = 1001  (1001² > 1001·1000)
[Phase 3]    emit ∃ n, ¬ (n²≤n*1000)
                 := ⟨1001, by norm_num⟩
             lake env lean: OK;  #print axioms:
             propext, Classical.choice, Quot.sound
Outcome:     REFUTED   (evidence-driven widening)
```

External SMT (Listing 3). On $\forall x\, y : \mathbb{Z},\ 2x^2 - 3y^2 \neq 5$, `decide` (infinite domain) and `omega` (nonlinear) both fail and one-dimensional `enumerate` does not apply to two free variables. The `external` method emits an SMT query; Z3 returns the model $(-4, 3)$, which is lifted to a Lean term and re-checked by the kernel. Z3 is untrusted: a bug in it cannot by itself yield a spurious REFUTED; the kernel re-checks every witness it proposes.

Listing 3 External-method session: Z3 returns the model $(-4, 3)$; the Lean kernel, not Z3, licenses REFUTED.

```
$ /disprove SmtBad.lean:8  -- ∀ x y:ℤ, 2x²−3y²≠5
[Step 1]  method: external  (decide/omega fail:
          nonlinear over ℤ; 1-D enumerate N/A)
[Step 2]  language = smt-z3
[Phase 2] Z3 (untrusted) → sat, model (x,y)=(-4,3)
[Phase 3] lift + kernel re-check:
          ∃ x y:ℤ, 2*x²-3*y² = 5
              := ⟨-4, 3, by norm_num⟩
          lake env lean: OK
          #print axioms: propext, Quot.sound
Outcome:  REFUTED   (untrusted; kernel-certified)
```

7. Discussion and Limitations

Within-session adaptation, not self-evolution. The evidence loop (§4.5) adapts the search within a session; later cycles re-rank methods using what earlier cycles revealed. This is online adaptation, not multi-session learning: the command retains no state across runs by default, and we make no "self-evolving" claim beyond it. The same certify-then-reweight loop with an auditable trace could underpin a multi-session agent, which we leave to future work. The learned signal here is the within-session, evidence-conditioned re-ranking. More broadly, the contribution is *neuro-symbolic orchestration*: an LLM proposes and an untrusted solver searches, while the Lean kernel alone certifies every REFUTED verdict. We make no model-training claim.

Limitations. This evaluation tests capabilities; it does not estimate broad success rates. It shows that the implemented method families produce kernel-checked refutations on a hand-built 16-target suite (15 false) designed to require them, reported with stratified coverage rather than confidence intervals. It does *not* estimate success on naturally occurring false conjectures, measure the LLM ranking's quality (the command is interactive; we run a fixed scripted-operator protocol, §5.2), or exercise every registry family; a larger, externally-sourced benchmark and an autonomous mode are future work. Goals that are true, or false only outside any tractable search window, are correctly reported INCONCLUSIVE rather than refuted; this is useful, but not a proof of truth. (`external`-script approval is a deliberate safety gate, auto-approved from a sandboxed allowlist only during evaluation.)

Scope of the taxonomy and registry. Beyond the classification of §4.2, the boundary of the taxonomy and the method registry is set by the families' preconditions, which the registry records as silent-failure modes. `decide-cascade` needs a synthesizable `Decidable` instance (or linear arithmetic for `omega`); `enumerate` needs a finite, iterable candidate grid; `plausible` needs a `SampleableExt` instance and a decidable residual; `mine` needs reusable literals. Conjectures over abstract algebraic structures, real or measure-theoretic objects, or goals carrying dependent types, coercions, or typeclass side-conditions meet none of these, and a counterexample that needs a domain-specific construction (such as the nine-vertex tree of §1) lies beyond every generic family. Only the `external` family can reach such witnesses: an LLM-emitted script encodes the construction and the kernel certifies the witness, but only when that construction is known and expressible. Broadening the shape taxonomy and the method registry to these regimes is the principal direction for future work.

Use in research workflows. Counterexample search is often a cheaper preliminary check before committing proof effort (exactly the role it played in the graph-limit episode of §1), and a debugging step for autoformalization pipelines that may emit ill-stated targets. It complements LLM theorem provers rather than competing with them.

8. Conclusion

The `/disprove` command treats falsification as a certified, interactive search: it locates a witness across a registry of methods and emits only kernel-checked refutations, reporting a non-result when it cannot. On a small benchmark it certifies every false target and recovers cases a bare decision procedure cannot, without adding any trusted axioms. Three directions follow: a fully *autonomous* mode that auto-selects menus and runs sandboxed scripts (an `/autoprove`-style counterpart, Freer, 2025), enabling unattended large-scale evaluation; broader shape coverage and salvage modes for partially-true goals; and an autoformalize-and-disprove agent that closes the propose → check → prove loop (Dong & Ma, 2025; Onda et al., 2025).

Software and Data

The `/disprove` command ships in the open-source `lean4-skills` package (Freer, 2025), released at https://github.com/cameronfreer/lean4-skills. The 16-target benchmark, its kernel-certified refutation terms, and run logs are in a companion repository, https://github.com/jancio/l4s-disprove-benchmark.

Impact Statement

This paper presents work whose goal is to advance machine learning for mathematics. A dual-use consideration specific to this work is that LLM-generated Lean can produce confidently wrong refutations; our design mitigates this by routing every REFUTED claim through the Lean kernel (§3), so that a flawed model or external solver cannot certify a non-counterexample. We see no other societal consequences that must be specifically highlighted here.

References

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. ImProver: Agent-based automated proof optimization. arXiv:2410.04753, 2024.

Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., and Avigad, J. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics. arXiv:2302.12433, 2023.

Blanchette, J. C. and Nipkow, T. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving (ITP)*, pp. 131–146, 2010.

Böving, H. Plausible: Property-based random testing for Lean 4. https://github.com/leanprover-community/plausible, 2024.

Claessen, K. and Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279, 2000.

Cruanes, S. and Blanchette, J. C. Extending Nunchaku to dependent type theory. In *Hammers for Type Theories (HaTT), EPTCS 210*, pp. 3–12, 2016. URL https://doi.org/10.4204/EPTCS.210.3.

de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340, 2008.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635, 2021. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Dénès, M., Hriţcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. QuickChick: Property-based testing for Coq. The Coq Workshop, 2014.

Dong, K. and Ma, T. STP: Self-play LLM theorem provers with iterative conjecturing and proving. arXiv:2502.00212, 2025.

Freer, C. Lean 4 Skills: Theorem proving skill and workflow pack for AI coding agents, October 2025. URL https://github.com/cameronfreer/lean4-skills.

Hudon, S. and The mathlib Community. SlimCheck: Property-based random testing in mathlib. Part of mathlib4, 2024.

Kumarappan, A., Tiwari, M., Song, P., George, R. J., Xiao, C., and Anandkumar, A. LeanAgent: Lifelong learning for formal theorem proving. In *International Conference on Learning Representations (ICLR)*, 2025.

Li, Z., Li, Z., Yang, K., Ma, X., and Su, Z. Learning to disprove: Formal counterexample generation with large language models. arXiv:2603.19514, 2026a.

Li, Z., Li, Z., Yang, K., Ma, X., and Su, Z. Official code for "Learning to Disprove: Formal counterexample generation with large language models", 2026b. URL https://doi.org/10.6084/m9.figshare.31741879.v1.

Limperg, J. and From, A. H. Aesop: White-box best-first proof search for Lean. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 253–266, 2023.

Mohamed, A., Mascarenhas, T., Khan, H., Barbosa, H., Reynolds, A., Qian, Y., Tinelli, C., and Barrett, C. LeanSMT: An SMT tactic for discharging proof goals in Lean. In *Computer Aided Verification (CAV)*, pp. 197–212, 2025.

URL https://doi.org/10.1007/978-3-031-98682-6_11.

Onda, N., Kasaura, K., Oriike, Y., Taniguchi, M., Sannai, A., and Sonoda, S. LeanConjecturer: Automatic generation of mathematical conjectures for theorem proving. arXiv:2506.22005, 2025.

Reynolds, A., Blanchette, J. C., Cruanes, S., and Tinelli, C. Model finding for recursive functions in SMT. In *International Joint Conference on Automated Reasoning (IJCAR)*, 2016.

Song, P., Yang, K., and Anandkumar, A. Lean Copilot: Large language models as copilots for theorem proving in Lean. In *International Conference on Neuro-symbolic Systems (NeuS); PMLR vol. 288*, pp. 144–169, 2025.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. Conference on Language Modeling (CoLM); arXiv:2310.04353, 2024.

The mathlib Community. The Lean mathematical library. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2020. URL https://doi.org/10.1145/3372885.3373824.

The mathlib Community. Counterexamples in mathlib4. https://github.com/leanprover-community/mathlib4/tree/master/Counterexamples, 2026.

Tsoukalas, G., Lee, J., Jennings, J., Xin, J., Ding, M., Jennings, M., Thakur, A., and Chaudhuri, S. PutnamBench: Evaluating neural theorem-provers on the Putnam mathematical competition. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

Xin, H. et al. DeepSeek-Prover-V1.5: Harnessing proof assistant feedback for reinforcement learning and Monte-Carlo tree search. arXiv:2408.08152, 2024.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

Zheng, K., Han, J. M., and Polu, S. MiniF2F: A cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations (ICLR)*, 2022.