GOEDEL-SCRIBE: TOWARD FAITHFUL STATEMENT AUTOFORMALIZATION VIA ADVERSARIAL MUTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

AI systems now possess the capability to produce Annals-level mathematical content autonomously. This capability introduces an asymmetry in the review process wherein the pace of generation of mathematical content far exceeds the (human) review capacity of the field. One way forward for the community is to more heavily rely on *formalized* proofs in languages like Lean. A pivot to formalization also has benefits for autonomous AI generation as models can iteratively improve proof candidates without the need for human intervention until a final compiling proof is produced. However, this shifts the importance of verifying proofs to verifying *faithfulness of statements* in Lean. A compiling proof matters only insofar as it corresponds precisely to a natural language mathematical statement. Complicating this matter is the reality that most human mathematicians are not currently fluent in Lean, meaning reliable AI assistance in formalizing statements, or whole papers, is necessary. In this work, we outline reasons for shifting focus to (auto)formalization of mathematical statements, we describe several desiderata of a formalization process, and we present a lightweight and flexible pipeline for AI driven autoformalization of mathematical statements. Our pipeline outperforms competing AI autoformalization approaches by relying on kernel certified repairs to misformalizations, and, when tested, autonomously finds several mis-formalizations in existing Lean code, including code used in benchmarks, and code certified by human mathematicians.

1 INTRODUCTION

The last year has seen break-neck speed improvements in AI's mathematical abilities. Accomplishments from only a year ago, like achieving gold-level performance at the International Mathematics Olympiad (IMO) (Google DeepMind, 2025; OpenAI, 2025), now seem unimpressive in comparison to recent achievements by AI systems. Systems from OpenAI and Anthropic have autonomously resolved several important conjectures in mathematics that have withstood human efforts for several decades, including those by Fields medalists and Abel Prize laureates (OpenAI, 2026; TechCrunch, 2026; ScienceDaily, 2026). Results by these companies are checked internally by elite mathematicians on staff before being released. This process works well enough for headline-grabbing proofs from large and wealthy companies, but it doesn't scale to the broader math community's use of these tools, or a future where we see weekly, or even daily, AI progress in math.

In this future, an asymmetry grows between what members of the community can produce and disseminate versus what trusted journals, referees, or even just interested observers can ingest (Dekoninck, 2026; Jin et al., 2026). AI judging of correctness can be used as a coarse filter, and has been shown to be reliable for simpler problems (Luong et al., 2025), but for research mathematics, a higher level of verifier trust is needed. Formal verification languages, like Lean, offer this desired reliability, but *only if* a formalized statement corresponds exactly to the natural language statement. This semantic equivalence is called *faithfulness*. Faithfulness is a different matter from compiling, since a statement can type-check and still fail to say what the natural-language statement says. In this work, we argue that this step, producing faithful formal statements, represents the crucial part of the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

AI integrated future of mathematics. Throughout the paper we call this step autoformalization.[1] This claim relies on a series of straightforward observations:

Faithful statement formalization enables autonomous AI solving. Once a theorem is faithfully formalized, any pipeline can be set to run almost arbitrarily long to attempt to solve the formalized statement. No human intervention is needed.

Off-the-shelf models are becoming increasingly capable Lean writers. In the last year, the long horizon math and coding capabilities of frontier models have progressed rapidly. On the PutnamBench leaderboard (Tsoukalas et al., 2026b), the top submission a year ago was a custom trained academic model at about 13% (Lin et al., 2025b), and today it is dominated by open source coding harnesses on top of a vanilla frontier model that saturate the benchmark (672/672).

The cost of a mistake in formalizing statements far outweighs the cost of a proof mistake. Almost by definition, any compute spent on proving an unfaithfully formalized statement is wasted. And even more dangerous than the cost of wasted compute is the cost of a false sense of validity from a proof that compiles against a subtly incorrect statement. Even if the proof is a valid logical resolution of the natural language statement, and it compiles against the formalized lean statement, a misformalized statement *alone* can poison future results that rely on this statement.

Taken together, these observations say that reliable autoformalization of statements represents the crux of AI-driven formal mathematics, and is the area where more community effort is needed. To this end, in this work, we present a pipeline that drafts a Lean statement from a claim and then iteratively stress-tests its faithfulness with mutations. Section 2 fixes the objects and the notation used throughout, Section 3 describes the pipeline, and Section 4 reports quantitative results and comparisons. Our primary contributions are the following.

- **We present a pipeline for writing and checking formal statements.** A coding agent with a Lean compiler and Mathlib search drafts the statement, after which adversarial agents propose alternatives that each change one thing. The Lean kernel then decides whether an alternative is only a rewording, and a judge must ground any remaining difference between the two statements in a sentence quoted from the paper. As a result, every change the pipeline makes is backed either by a Lean proof or by a quoted sentence. The pipeline uses off-the-shelf models and costs about one dollar per statement (Section 3).
- **We find defects in published formalizations that passed expert human and AI review.** When run on the 89 reference statements of the two most recent months of ArXivLean (Dekoninck et al., 2026), the pipeline repairs five statements that do not assert the theorem of their paper, one of which is proved trivially in a few lines by the identity matrix, and it identifies two more that do not compile under the benchmark's own toolchain. All 89 statements had passed the benchmark's AI and human vetting. We also compare with two research-paper formalization systems on the statements they released and with a trained faithfulness critic on our sets, and find further errors in vetted statements (Section 4.5).
- **We characterize what is important for faithful statement autoformalization.** We find that several components are necessary to identify subtle mis-formalizations, and that removing any one of them causes the pipeline to miss defects that human reviewers had also missed. A checker must be able to call the Lean compiler and search the library, since models given the same instructions without these tools almost never propose a correction. Tools are not sufficient on their own, because a model simply asked whether a statement matches the paper reads Lean the way a person would, and overlooks defects that hinge on what a Mathlib notion actually means, such as a hypothesis that reads as smoothness but denotes analyticity. Requiring the checker to first make the meaning of every notion and the range of every hypothesis explicit is what lets the same models recover all five defects. Finally, the kernel discards rewordings and exposes trivially true statements, and the judge must be asked about mathematical convention explicitly (Section 4.4).

[1]The term is sometimes used more broadly to include formalizing the proof, which is a separate step that begins only once the statement is fixed.

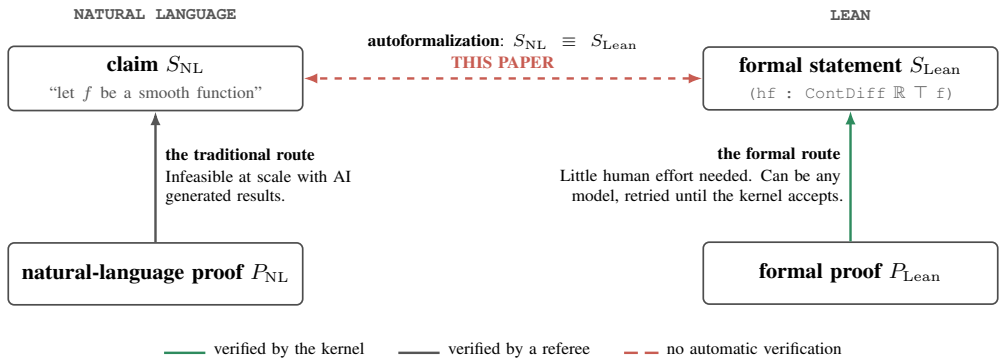

Figure 1: The four objects in a formalized result. The traditional route, a referee checking P_{NL} against S_{NL}, does not scale to AI-generated mathematics. Instead, the formal route, where a kernel checks if P_{Lean} proves S_{Lean}, scales with little human effort. Proving S_{NL} via the formal route assumes S_{NL} is exactly equivalent to S_{Lean}, but there is little verification for that correspondence. The example S_{NL} and S_{Lean} pair shows a misformalized statement from a reference translation from the ArXivLean benchmark, where the natural-language version assumes f is smooth, whereas the Lean version requires f to be analytic (Section 4).

2 The components of autoformalization

Proving a math result in formal language involves four parts. 1) The claim S_{NL} is the natural-language statement with the necessary definitions. 2) The natural-language proof P_{NL} is optional, but if it exists, it can guide the formalization. 3) The formal statement S_{Lean} is the target to prove, translated into formal language. It originally comes with a placeholder proof `sorry`, which needs to be filled in later. 4) The formal proof P_{Lean}, a term of type S_{Lean}, proves S_{Lean}, and its correctness can be automatically verified by the kernel. Figure 1 shows how the four parts relate.

Verifying whether P_{NL} proves S_{NL} involves the scrutiny of referees (e.g., human experts) and the community's review. However, it cannot keep pace with the rate at which AI systems now produce proofs. On the other hand, verifying whether P_{Lean} proves S_{Lean} can be done by a cheap kernel, so any AI model can be run until the kernel accepts the proof P_{Lean}.

Each route comes with its own form of trust. For S_{NL} and P_{NL}, trust rests with referees and the community's review. For P_{Lean} and S_{Lean}, a trusted kernel verifies that P_{Lean} proves S_{Lean}. Kernel bugs exist, as in a recent false disproof of the Collatz conjecture (de Moura, 2026), but they are rare, and Lean certificates are, by and large, highly trusted. It should also be noted that human refereeing has its own failures. For example, the Annals of Mathematics published a negative answer to the Busemann–Petty problem in $\mathbb{R}^4$ in 1994 and a positive one in 1999, by the same author (Zhang, 1994; 1999).

However, producing a math result via the formal route relies on one more assumption, that S_{NL} and S_{Lean} are equivalent, and no machine checks that correspondence, which has received far less attention than proving. This paper inspects the correspondence between S_{NL} and S_{Lean}.

Overall, in the path to proving S_{NL}, differences between S_{NL} and S_{Lean} are the most pernicious errors, because nothing automatically stops them. A mis-stated S_{Lean} can come with a compiling proof. In the *best case*, these can be caught by hand inspecting proofs. For example, a widely used example of Lean mis-formalizations is an imprecise statement of the Archimedean property written as "for every $\varepsilon > 0$ there is a natural N with $1/N < \varepsilon$". This is proved in Lean by $N = 0$, because Lean defines $1/0 = 0$. But in more subtle cases, this sort of check does not catch mis-formalizations, since the compiling proof can genuinely be a correct proof for the correctly formalized version of the statement. This means, translated back into English, the Lean reads correctly. Several defects that Section 4 finds in reviewed formalizations are of this second kind.

3 Statement autoformalization pipeline

Much of the focus on AI generated formal math has been placed on producing better *provers* (Lin et al., 2025a;b; Chung et al., 2026; Ren et al., 2025; Breen et al., 2026; Mistral AI, 2026; Liu et al., 2026). But, as argued in the previous sections, efforts like these are becoming less important due to

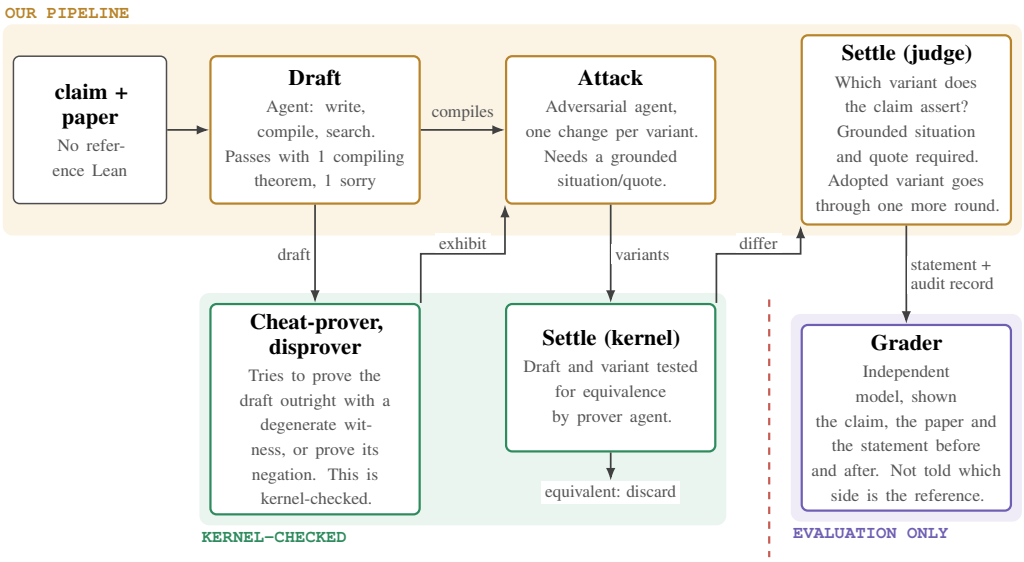

Figure 2: Overview of the pipeline. A drafting agent writes the statement, two attacking agents propose alternatives, and the settle stage decides between them. The green stages are decided by the Lean kernel. The judge is the one place where a model's reading decides, and it must quote the sentence of the paper that settles the question. The reference formalization is used only for evaluation.

several factors. Conversely, there has been relatively little work on designing systems to faithfully formalize natural-language statements or results. To this end, we introduce a pipeline designed around adversarial probing of candidate Lean statements. It takes a natural-language statement, and optionally the paper the statement comes from, and returns a Lean 4 theorem statement whose proof is `sorry`, together with a record of every check that was made. While there is no automatic certificate for faithfulness, our pipeline relies heavily on proxy verifiable signals, and grounded arguments.

The pipeline has four stages (Figure 2) and is inspired by the simplicity of the mini-SWE-agent coding agent (SWE-agent team, 2025; Yang et al., 2024). Appendices B to D describe each stage in more detail.

Draft. A drafting agent, whose only tools are a compiler against a pinned Lean version and semantic search over Mathlib, writes a statement that must compile with exactly one theorem and one `sorry`. Drafting costs a few cents per statement, and the result compiles for 98% of our problem set.

Cheat-prover and disprover. A *cheat-prover* tries to prove the draft outright with a degenerate witness such as zero or the identity, and a *disprover* tries to prove its negation. A success in either is a verdict checked by the Lean kernel, namely that the draft is trivially true or that it is false as written, and the proof is passed to the later stages as evidence that the formalization might be incorrect. Notably, our cheat-prover found the March p4 defect of Section 4 in its first round!

Attack. Two *attacking* agents, built on different models and equipped with the same tools as the drafter, look for places where the draft's meaning differs from the claim. Rather than giving them a list of known mistakes, we guide the attackers to follow a procedure before reporting that they found no difference. These procedures include printing what each Mathlib notion in the draft means in the pinned Mathlib version, stating the range of every hypothesis and quantifier both in Lean and in the paper, and comparing each part of the conclusion. For every difference an attacker finds, it writes a compiling alternative, called a mutant, that changes one thing, together with a situation in which the two statements disagree and, when possible, a sentence quoted from the paper. These procedures are generic, and were not tuned/constructed based on tested problems.

Settle. The *settle* stage decides between each alternative candidate. It first asks the kernel whether the alternative is equivalent to the draft, by attempting both implications with a sequence of provers. An equivalent alternative is a rewording and is thus discarded. Whatever candidates remain are shown twice to a strong frontier model, which must say which of the two statements the claim asserts, name

a case on which they differ, and quote the deciding part of the paper. An alternative is adopted only when two calls agree, with a third breaking a tie, and their quotes appear in the text. When the kernel has proved that the alternative is the stronger of the two statements and the judge still calls them the same, we add one further question, which we found to be necessary in practice (Section 4.4): the judge is asked whether the paper's wording, or the standard convention for the objects it introduces, restricts those objects to the alternative's range. This happens when a standard mathematical term is narrower than the Lean type used to encode it, as with a Sylow subgroup (Section 4.4).

Second pass. An adopted alternative replaces the draft and the pipeline runs once more on the result. To prevent this second pass from undoing the first, an alternative that is still proved by a degenerate proof from an earlier pass is discarded, because adopting it would reintroduce the defect that the earlier repair removed. The output is the final statement together with its record comprising every kernel certificate, every relation with its proof file, and every vote with its accompanying quote.

Models. The pipeline is not tied to any particular model. A role whose output is checked mechanically, which is the case for the cheat-prover, the disprover, the first provers of the settle stage and the drafter, runs on a cheaper model, since a weak model there wastes a call but cannot produce a wrong verdict. The attackers, the last prover and the judge run on the most capable models we have, and the two attackers are built on different models so that their blind spots differ (Appendix E).

4 Results

4.1 Setup

Data. We conduct most of our experiments on ArXivLean (Dekoninck et al., 2026), a benchmark that pairs the main theorem of a recent arXiv paper with a reference Lean 4 statement which was generated by an AI system and subsequently checked by human mathematicians. This benchmark was originally designed to test the Lean *proving* abilities of models/pipelines, but it also offers a setting in which to test *autoformalization*. We use the 89 problems from its two most recent releases, from March and June 2026. We focus on this problem set for a few reasons. First, because the underlying papers are between three and six months old, the risk of contamination is minimized (but not totally eliminated) compared to other Lean statement repositories. This also means there has been less time for the community to by-hand vet these statements, meaning faithfulness checking is more important. Second, the statements are of research level, so testing on this problem set mirrors the real-world use cases of mathematicians. Third, the problems are pre-filtered for those that are likely to be supported by Mathlib, which gives some level of confidence that the problems should be *formalizable*. In addition to ArXivLean, to compare with other systems, we also consider the released main-theorem statements of Theo (Moakhar et al., 2026) and Trellis (Pegden, 2026a), which together comprise fifteen statements.

Experiments. We run the pipeline presented in Section 3 in four settings. The first is a control with *planted defects*, in which we generate artificial variants of ArXivLean references, each with one planted defect that Lean certifies, such as a special case or a widened scope. These errors are generally more visible, and mainly serve to calibrate performance. The second is the set of all 89 *published references* themselves. These are formalized statements that have ostensibly passed both human and AI vetting, and have been released as popular benchmark questions for current Lean provers. The third consists of the fifteen main-theorem statements released by two *other systems*, with our own drafts of the same claims, and the fourth of *our own drafts* for the 89 ArXivLean claims. A trained faithfulness critic is also run on the first three sets. Appendix E lists the settings common to every run and the cost of each.

Evaluation procedure. The drafting stage of our pipeline is used only in the fourth setting to produce our own drafts. In the other three settings, because the statement under test already exists, we skip the first stage of our pipeline. The remaining stages receive that statement together with the text it is meant to formalize (where possible). In every case the pipeline either keeps the statement or proposes a changed version. We release formalizations for any interested auditors. Importantly, the pipeline never has access to a reference formalization of the same claim. Every change is submitted to an independent grader, Claude Opus 5, a model that plays no other role in the pipeline. It is shown

only the claim, the paper and the statement before and after the change, never the pipeline's reasoning and never which side is the reference, and it is called three times. We count a change as a repair when a majority of the three calls conclude that the paper asserts the changed version, each supporting the conclusion with a sentence quoted verbatim from the paper, which the code checks. The grader is not the last word. Every change it confirmed on a reference statement was read by the authors, who include Lean experts and research mathematicians, all five defects of Table 1 are confirmed by hand, and three of the five carry a kernel certificate that depends on no model. The numbers we report are the grader's, and the human reading changed none of them. Appendix A discusses the limits of this evidence. On the planted defects the ground truth is the kernel, which proves whether a change moves the statement toward the reference. Appendix E lists every setting, including the model behind each role.[2]

4.2 Planted defects as a control

Before turning to statements whose defects are unknown, we check that the pipeline repairs defects we know are there and leaves correct statements alone. As described earlier, each defect is a reference with one edit that Lean proves makes it weaker or stronger than the original claim, so the direction of a correct repair is fixed in advance. To control costs, we experiment with 60 such statements, which turn into 54 after removing equivalent statements and the two mutants of March p4, whose reference is provable outright, so that no implication test can tell the direction of a repair. An equivalence the kernel did not prove counts against the pipeline as a miss, so the rate we report is a lower bound. A repair is a *certified catch* when the kernel proves that the pipeline's final statement lies between the mutant and the reference in the planted direction. Of the 54, the kernel certifies the repair of 47 in the planted direction and the grader confirms 6 more, so 53 are repaired. The one mutant kept unchanged is in fact equivalent to its reference, but under our rule it counts as a miss. Table 5 in Appendix F gives the rate by kind of defect and a reading of every miss.

4.3 Defects in published references

In addition to the "controlled" environment of planted defects, we also experiment on Lean statements "in the wild". On the official ArXivLean releases, we find that at least five of the 89 reference statements do not assert the theorem of their paper (Table 1), although every one had passed the benchmark's AI and human review, and that two more (June p18 and p39) do not compile under the benchmark's own pinned Mathlib. The five defects are invisible both to a compiler and to a careful reader. For example, the reference for March p4 can be proved in a few lines by taking the identity matrix as the transition kernel, which means that the statement carries no information about the Markov chain that the paper constructs. Similarly, the hypothesis `ContDiff` $\mathbb{R} \top f$ in June p1 appears to express smoothness, whereas in the current version of Mathlib it requires f to be analytic. Appendix H discusses all five defects in detail.

A single run over all 89 references repairs all five defects and changes four other references in ways the grader judges equivalent (Table 4). With the claim sentence alone as text, and no paper, the pipeline still repairs three of the five (Appendix G).

With the drafting stage included, on the same 89 claims, the pipeline repaired two mistakes of its own drafter, the analytic hypothesis of June p1 and induced subgraphs where June p18 means ordinary containment, changed two further drafts into equivalent statements, and kept the other 85 (Table 4).

4.4 What matters for faithful autoformalization

To understand which components of our approach are responsible for these findings, we ran several ablations of the pipeline over the 89 references, varying the tools available to the attackers, the instructions they receive, and the questions that are put to the judge (Table 2). We find that several components are necessary to identify subtle misformalizations, and we discuss each of them in turn.

[2]The models are `openai/gpt-5.6-sol`, `deepseek/deepseek-v4-pro-0813` and `deepseek/deepseek-v4-flash-0731`, as served through OpenRouter, and the grader is `anthropic/claude-opus-5`.

reference	reference statement	paper statement	kernel certificates	fraction of 4 reruns that repaired it
March p4	a nearest-neighbour chain reversible for π exists, with no condition that it moves	the chain of the paper's construction has positive rates	the reference is proved outright by the identity matrix	1.0
March p25	H permutes with `Sylow p B` for every natural number p	H permutes with every Sylow subgroup, so p is prime	the repair implies the reference	1.0
June p1	f is analytic (`ContDiff` $\mathbb{R}$ $\top$)	f is smooth	the repair implies the reference	1.0
June p20	a right-angled Artin group on any graph, finite or not	the graph is finite by definition	neither direction	1.0
June p23	the entries are almost-everywhere measurable	the entries are random variables	the reference implies the repair	1.0

Table 1: The five ArXivLean references that do not assert their paper's theorem. The implications are between the two statements as propositions, so a repair that implies the reference claims more than the reference does. The last column is the fraction of four reruns on these five statements in which the pipeline repaired the statement with a grader-confirmed change. The run over all 89 references repaired all five.

feature ablation	defects repaired, of 5	other references changed
attackers without compiler or Mathlib search	1	0
+ compiler and Mathlib search	2	0
+ explicit notion meanings and hypothesis ranges, + convention question (full pipeline)	5	4

Table 2: Ablations over the 89 ArXivLean references. A defect counts as repaired when the pipeline changes the reference and the grader confirms the change. Every change to one of the other 84 references was graded an equivalent rewording. The first row uses an earlier form of the attacker instructions, since a bare instruction was never run without tools.

Access to the compiler and the library. We first find that access to the Lean compiler and to Mathlib is essential. When we give the same two models the same instructions but remove the shell, the compiler and the library search, the attackers report no difference in 173 of their 178 calls, and only one of the five defects is found. A model that cannot look up what a notion means has to read the Lean the way a human reviewer would, and the defects that survived human review are the ones that reading is likely to not catch.

Making the meaning of every notion explicit. Access to tools is not sufficient on its own, however. When the attackers have the tools but are only instructed to find differences between the statement and the paper, both of them report no difference on 66 of the 76 statements whose run finished, and only two of the five defects are found. Three of the 89 runs hit the time limit and seven were stopped because an attacker read outside its directory. On June p1 one attacker wrote that `ContDiff` $\mathbb{R}$ $\top$ "expresses smoothness" and stopped, although the notion denotes analyticity in the current Mathlib, and on June p23 neither attacker questioned the measurability hypothesis. Our procedure requires the attacker to print the meaning of every Mathlib notion, notation and special value in the statement, and to state the range of every hypothesis and quantifier in Lean and in the paper, before it may conclude that the statement is faithful. With these instructions, June p1 and June p23 are repaired in every run.

The role of the kernel. The Lean kernel contributes two things a model cannot. It identifies vacuous statements, since the degenerate proof of March p4 by the identity matrix is a Lean proof and needs no reading of the paper, and it discards proposed changes that only reword the statement. In two full runs over the 89 references the attackers proposed 29 and 33 changes on 14 and 18 statements. The kernel proved 3 and 6 of them equivalent to the original and the judge rejected 14 more in each run, which left the five repairs and a few rewordings the grader judged equivalent. No version of the pipeline changed a correct reference in a way the grader considered harmful.

Asking the judge about convention. Finally, we find that the judge must be asked about mathematical convention explicitly. Without this convention question the attackers proposed the correct repair for March p25 and the kernel proved it the stronger statement, yet the judge answered that the two were the same, since the paper says "Sylow subgroup" and both statements mention Sylow subgroups. With the question, March p25 was repaired in every run.

checker	planted mutants flagged, of 54	real ArXivLean defects named, of 5	correct references flagged
GPT-5.6 Sol, one call	53	2	6%
Claude Opus 5, one call	53	2	1%
CriticLean	52	1	21%
our pipeline	53	5	0%*

Table 3: Comparable checkers and our pipeline on the same statements, each given the paper. The planted column is on the 54 mutants of Section 4.2. A single call and CriticLean answer yes or no. Our pipeline changes the statement, so its planted number counts kernel-certified or grader-confirmed repairs, and its last column counts the 84 correct references it changed. *Here our pipeline changed four statements, but all ended up being equivalent rewordings, meaning no correct reference was worsened. A real defect counts for a checker only when its stated reason names it, and for our pipeline when the grader confirms the change. CriticLean's planted number is from the claim sentence alone. The single-call prompt lists the kinds of failure to look for (Appendix K.7), and the calls were scored on the 82 correct references that compile.

4.5 Comparison with other methods

Other recent methods also seek to address similar problems to the one we tackle, albeit mainly focusing on the broader question of both statement *and* proof formalization, which we argue is a slightly mis-guided focus. Theo (Moakhar et al., 2026) and Trellis (Pegden, 2026a) formalize a research paper end to end, statement and proof. Trellis has each formal statement pass through human approval before proving and Theo has experts validate the finished formalizations, meaning a human may have corrected a statement at either point, so any accuracy measured on their released statements is only an upper bound for their automatic step, whereas our pipeline runs without human intervention. Neither system can be run on the ArXivLean references, since Theo's pipeline is not publicly available and Trellis's own record prices one run on a single paper, statement and proof, at about $2,800 at list rates (Pegden, 2026b). We therefore compare on the fifteen main-theorem statements the two systems released, running our pipeline and grader on their statements and on our own drafts of the same claims (Table 4). CriticLean (Peng et al., 2025) is a critic trained with reinforcement learning to say whether a Lean statement faithfully formalizes a text. CriticLean's weights are released, so we self-host it with its published prompt and default settings.

Theo and Trellis. The pipeline changed three of Theo's eight statements. We find that two of the three changes are indeed defects in Theo's formalizations. The first is a result on the median mechanism in $\mathbb{R}^d$, whose paper bounds the approximation ratio by a function $\mathrm{UB}(q)$ that is increasing in q, with $\mathrm{UB}(1) = 1$ and limit 3. Theo's statement asks only for a function with these values and never requires it to be increasing, so it says nothing about the ratio between them. The second is a lower bound for a generalized trace reconstruction problem, which the paper proves for every large n and Theo's statement asserts only for $n \equiv 1 \pmod 8$. Both had passed Theo's own faithfulness judge and its expert validation. The third change follows the literal wording of a loosely stated claim, and on inspecting the paper it is not a defect (Appendix I). On Trellis's seven statements the attackers proposed no alternative at all, but these statements have been vetted by a human, while our pipeline does not require human correction. None of our own drafts of the fifteen claims spanning both papers were graded as incorrect. Appendix I gives the details.

A single call of a frontier model. One call of GPT-5.6 Sol or Claude Opus 5, given the statement and the paper, and prompted with a condensed version of our instructions flags almost every planted mutant, albeit with a nonzero false positive rate, but finds only two of the five real defects in ArXivLean (Table eftab:critic).

CriticLean. CriticLean was trained on competition and textbook problems, whose texts are a few lines long, so giving it the paper goes beyond its intended use. It flags almost every planted mutant and names both of Theo's defects, which are visible in the statement's text, but it names none of the five ArXivLean defects from the statement alone and only June p20 with the paper, while flagging 32% of the correct references from the statement alone and 21% with the paper (Table 3). This model is also tool-less, so it cannot look up what a notion means, and like the tool-less pipeline in Table 2 it does not find the defects that survived review.

statements	number	changed	confirmed repairs	equivalent rewordings	rejected by the grader
ArXivLean references	89	9	5	4	0
our drafts of the 89 ArXivLean claims	89	4	2	2	0
Theo, released main theorems	8	3	2	0	0
Trellis, released main theorems	7	0	0	0	0
our drafts of those fifteen claims	15	0	0	0	0

Table 4: What the pipeline changes on each set of statements, all through the same procedure and the same grader. The third change to a Theo statement follows the literal wording of a loosely stated claim and is not an error on reading the paper (Appendix I).

5 Related work

Proving formal statements. Most effort on AI for Lean is focused on proof generation. Benchmarks such as miniF2F (Zheng et al., 2022) and PutnamBench (Tsoukalas et al., 2024) supply pre-formalized statements, and provers hill-climb on compiling proofs for them. Over the last year, the field has moved from bespoke trained models (Lin et al., 2025a;b) to pipelines over off-the-shelf frontier models (Chung et al., 2026; Humanfia, 2026) that saturate the benchmarks with a compiler and Mathlib search (The mathlib Community, 2020) as their only tools, and these pipelines have begun to be used on open research questions (Tsoukalas et al., 2026a). Our pipeline uses the same kind of agentic setup, but for the complementary problem of statement autoformalization.

Writing formal statements and proofs from papers. Autoformalization with language models was originally attempted, to our knowledge, by Wu et al. (2022), who checked the translations by hand. Recent systems take a whole research paper as input and formalize it end to end, proof included (Moakhar et al., 2026; Pegden, 2026a), with human review at different points of the process. Our work focuses on the statement only, and is about minimizing pernicious errors that prevent powerful proving systems from furthering mathematical discovery.

Errors in reference statements. Benchmarks of research-level statements such as ArXivLean (Dekoninck et al., 2026) come with a reference Lean statement for each problem, written with model help and checked by mathematicians, and a candidate is scored by comparison with the reference (LeanFRO, 2026). The observation that references contain errors is not new. Ospanov et al. (2025) found discrepancies between the formal and the informal statement in more than half of miniF2F's problems by reading them, and Ammanamanchi et al. (2026) ran static checkers over five Lean benchmarks and certified 398 issues, among them vacuous theorems. The defects of Section 4 are of another kind. They are statements that compile, are true, and are not what the paper says, such as analytic for smooth, and finding them needs the text. A comparison with the reference cannot see a defect the reference shares.

Checking whether a formal statement is faithful. Poiroux et al. (2024) build metrics and benchmarks for faithfulness, including research-level pairs scored against reference statements, and Zhang et al. (2026) measure over several hundred graduate-level statements how far the faithfulness rate falls below the compiling rate, often by close to 30%. The common ground truth is a reference or a model judge (Moakhar et al., 2026; Dekoninck et al., 2026). FormalMATH (Yu et al., 2025) adds several judge models and a negation-based disproof filter. Peng et al. (2025) train a critic for faithfulness on competition and textbook problems (tested in Section 4.5). Most of this work tests on competition, textbook or graduate-level statements. We instead test on ArXivLean, whose statements come from recent research papers and have passed expert and AI review, so any defect that remains is one that careful reading missed.

Mutation, roundtrips and audits. Lu et al. (2026) also mutate formal statements, but as the iterations of an evolutionary search for a statement that a fixed model judge, CriticLean, scores as faithful, on competition and textbook problems, and no two candidates are ever compared by a proof. Their code is not released, so we do not compare with them experimentally. Ours are attacks on a given statement, including one that a person has already approved, and what an attack shows is decided by the kernel and by a judge that must quote the paper. Amrollahi et al. (2026) check two formalizations of a legal text for equivalence, as our settle stage does.

REPRODUCIBILITY STATEMENT

The full pipeline code and the planted-defect corpus will be released with the paper. Every prompt is reproduced verbatim in Appendix K, and Appendix E lists the model behind each role, the pinned Lean and Mathlib versions, and the settings and cost of every run. Appendix D states the pipeline as a procedure, and Appendix C gives the settle rule precisely.

AI USE STATEMENT

We used generative AI tools to generate the planted defects of Section 4.2, and kept each one only when the Lean kernel certified it. We used AI coding assistants, under the authors' direction, to implement the pipeline and the experiment scripts, to run the experiments, and to draft and revise parts of the text. We have not used generative AI tools to formulate the paper's direction, its claims, or to choose which results to report. The authors designed the method and the experiments from inception, read the record of every run, checked every number reported here against those records, and read by hand every change the pipeline made to a reference statement. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

REFERENCES

Pawan Sasanka Ammanamanchi, Siddharth Bhat, and Stella Biderman. Faults in our formal benchmarking: Dataset defects and evaluation failures in Lean theorem proving, June 2026. URL http://arxiv.org/abs/2606.29493. arXiv:2606.29493.

Daneshvar Amrollahi, Jerry Lopez, and Clark Barrett. Faithful autoformalization via roundtrip verification and repair, April 2026. URL http://arxiv.org/abs/2604.25031. arXiv:2604.25031.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M. Taylor, Frank H. L. Koppens, and Dirk Englund. Ax-Prover: A Deep Reasoning Agentic Framework for Theorem Proving in Mathematics and Quantum Physics, May 2026. URL http://arxiv.org/abs/2510.12787. arXiv:2510.12787 [cs.AI].

Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H. Fowl, and Sanjeev Arora. Goedel-Architect: Streamlining Formal Theorem Proving with Blueprint Generation and Refinement, June 2026. URL http://arxiv.org/abs/2606.06468. arXiv:2606.06468 [cs.AI].

Leonardo de Moura. Postmortem for kernel soundness bug #14576. https://leodemoura.github.io/blog/2026-8-1-postmortem-for-kernel-soundness-bug-14576/, August 2026. Accessed September 2026.

Jasper Dekoninck. Mathematics submissions to arXiv since 2010: about 20% above the fifteen-year trend by the second quarter of 2026. Post on X, https://x.com/i/status/2073834616323035350; reported in https://officechai.com/ai/mathematical-articles-on-arxiv-have-risen-20-over-trend-in-2026/, July 2026. Accessed September 2026.

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond Benchmarks: MathArena as an Evaluation Platform for Mathematics with LLMs, May 2026. URL http://arxiv.org/abs/2605.00674. arXiv:2605.00674 [cs.CL].

Google DeepMind. Advanced version of Gemini with Deep Think officially achieves gold-medal standard at the International Mathematical Olympiad. https://deepmind.google/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad/, July 2025. Accessed September 2026.

Humanfia. Humanfia at PutnamBench: 672/672 (100%). https://github.com/humanfia/putnambench, 2026. PutnamBench leaderboard submission; agentic solver built on the Humanize framework (NVIDIA Research, UCLA PolyArch, MIT HAN Lab).

Jiashun Jin, Zheng Tracy Ke, and Bingcheng Sui. The gold rush in AI4Math: Where are we now? *arXiv preprint arXiv:2608.24961*, 2026. All 32,944 arXiv mathematics submissions from March 1 to August 20, 2026; disclosed substantive AI use rose from 1.39% to 14.09%.

LeanFRO. leanprover/comparator, August 2026. URL https://github.com/leanprover/comparator. original-date: 2025-06-06T11:02:28Z.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source automated theorem proving, April 2025a. URL http://arxiv.org/abs/2502.07640. arXiv:2502.07640 [cs.LG].

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction, August 2025b. URL http://arxiv.org/abs/2508.03613. arXiv:2508.03613 [cs.LG].

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics, January 2026. URL http://arxiv.org/abs/2601.14027. arXiv:2601.14027 [cs.AI].

Haijian Lu, Wei Wang, and Jing Liu. FormalEvolve: Neuro-symbolic evolutionary search for diverse autoformalization, March 2026. URL http://arxiv.org/abs/2603.19828. arXiv:2603.19828.

Thang Luong, Dawsen Hwang, Hoang H. Nguyen, Golnaz Ghiasi, Yuri Chervonyi, Insuk Seo, Junsu Kim, Garrett Bingham, Jonathan Lee, Swaroop Mishra, Alex Zhai, Clara Huiyi Hu, Henryk Michalewski, Jimin Kim, Jeonghyun Ahn, Junhwi Bae, Xingyou Song, Trieu H. Trinh, Quoc V. Le, and Junehyuk Jung. Towards robust mathematical reasoning. *arXiv preprint arXiv:2511.01846*, 2025. EMNLP 2025. IMO-Bench: AnswerBench, ProofBench, GradingBench.

Mistral AI. Leanstral: Open-Source foundation for trustworthy vibe-coding, 2026. URL https://mistral.ai/news/leanstral/.

Arshia Soltani Moakhar, Iman Gholami, Max Springer, Mahdi JafariRaviz, and MohammadTaghi Hajiaghayi. Beyond the Library: An Agentic Framework for Autoformalizing Research Mathematics, August 2026. URL http://arxiv.org/abs/2606.31134. arXiv:2606.31134 [cs.AI].

OpenAI. We achieved gold medal-level performance on the 2025 International Mathematical Olympiad with a general-purpose reasoning LLM. https://x.com/OpenAI/status/1946594928945148246, July 2025. Accessed September 2026.

OpenAI. Ten advances in mathematics and theoretical computer science. https://openai.com/index/ten-advances-in-mathematics/, 2026. Accessed September 2026.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. miniF2F-Lean revisited: Reviewing limitations and charting a path forward, November 2025. URL http://arxiv.org/abs/2511.03108. arXiv:2511.03108.

Wesley Pegden. (Auto)formalization is supposed to be easy: Trellis process semantics for spelling out rigorous proofs, June 2026a. URL http://arxiv.org/abs/2606.09674. arXiv:2606.09674 [cs.AI].

Wesley Pegden. Off-diagonal Ramsey numbers: Trellis formalization and run record. https://github.com/wpegden/offdiagonal, file formalization.yaml, 2026b. Accessed 2026-09-09.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, Minghao Liu, Yihang Xia, Jiawei Shen, Yuchen Wu, Yixin Cao, Zhaoxiang Zhang, Wenhao Huang, Jiaheng Liu, and Ge Zhang. CriticLean: Critic-guided reinforcement learning for mathematical formalization, July 2025. URL http://arxiv.org/abs/2507.06181. arXiv:2507.06181.

Auguste Poiroux, Gail Weiss, Viktor Kunčak, and Antoine Bosselut. Reliable evaluation and benchmarks for statement autoformalization. *arXiv preprint arXiv:2406.07222*, 2024.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv:2504.21801*, 2025.

ScienceDaily. Claude Fable 5 AI finds a tiny formula that topples an 87-year-old math conjecture. https://sciencedaily.com/releases/2026/08/260804034634.htm, August 2026. Accessed September 2026.

Alexander Skidanov. putnambench-deepseek: solving PutnamBench with DeepSeek. https://github.com/SkidanovAlex/putnambench-deepseek, 2026. Accessed September 2026.

SWE-agent team. mini-SWE-agent: the 100 line AI agent that solves GitHub issues. https://github.com/SWE-agent/mini-swe-agent, 2025. Accessed September 2026.

TechCrunch. An unreleased Anthropic model made progress on one of math's biggest unsolved problems. https://techcrunch.com/2026/08/11/an-unreleased-anthropic-model-made-progress-on-one-of-maths-biggest-unsolved-problems/, August 2026. Accessed September 2026.

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating Neural Theorem-Provers on the Putnam Mathematical Competition, November 2024. URL http://arxiv.org/abs/2407.11214. arXiv:2407.11214 [cs.AI].

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing Mathematics Research with AI-Driven Formal Proof Search, June 2026a. URL http://arxiv.org/abs/2605.22763. arXiv:2605.22763 [cs.AI].

George Tsoukalas et al. PutnamBench leaderboard. https://trishullab.github.io/PutnamBench/leaderboard.html, 2026b. Accessed September 2026.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Christian Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *NeurIPS*, 2022.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, November 2024. URL http://arxiv.org/abs/2405.15793. arXiv:2405.15793 [cs.SE].

Zhouliang Yu, Ruotian Peng, Keyi Ding, Yizhe Li, Zhongyuan Peng, Minghao Liu, Yifan Zhang, Zheng Yuan, Huajian Xin, Wenhao Huang, Yandong Wen, Ge Zhang, and Weiyang Liu. FormalMATH: Benchmarking formal mathematical reasoning of large language models. *arXiv preprint arXiv:2505.02735*, 2025.

Gaoyong Zhang. Intersection bodies and the busemann-petty inequalities in $\mathbb{R}^4$. *Annals of Mathematics*, 140(2):331, 1994. doi: 10.2307/2118603.

Gaoyong Zhang. A positive solution to the busemann-petty problem in $\mathbb{R}^4$. *Annals of Mathematics*, 149(2):535, 1999. doi: 10.2307/120974.

Ke Zhang, Patricio Gallardo Candela, Sudhir Murthy, Yi Xie, Zhi Wang, and Maziar Raissi. Beyond compilation: Evaluating faithful natural-language-to-Lean statement formalization. *arXiv preprint arXiv:2606.31002*, 2026.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A cross-system benchmark for formal olympiad-level mathematics. In *ICLR*, 2022.

A LIMITATIONS

Unfortunately, there is a lack of recently released, reviewed formalizations of research-level statements on which to test. ArXivLean publishes a few dozen reference statements every few months, and its two most recent releases at the time of writing give the 89 references we test on. Theo and Trellis have released fifteen main-theorem statements between them. We test on every one of these, and we plant defects into the references to get volume, but an edit made to a correct statement is easier to see than a convention the paper takes for granted, so the planted defects measure something easier than the real ones. Every claim about what the pipeline finds in the wild therefore rests on seven real defects, five in ArXivLean and two in Theo's statements. Ideally, autoformalization would be tested on a wider range of reviewed research-level statements to measure performance with more precision.

The pipeline's judgments and its grading are both made by frontier models, so an error that every current model shares could pass both. This is why the authors read every confirmed change and why three of the five ArXivLean defects carry a kernel certificate that depends on no model. The convention question also adopts changes the grader judges equivalent, so the pipeline's output should be read together with its record rather than as a verdict. We also report most numbers from a single run, without confidence intervals, except for the runs on the five real defects. A pass of the pipeline over the 89 references costs about $93, so we repeated only the runs on the five defective references, four times, and ran the planted defects on a random subset of 60 to limit expenses on an academic budget.

B THE PIPELINE STAGES IN MORE DETAIL

This appendix describes the stages of Section 3 in full. Appendix C states the settle rule precisely, and Appendix D writes the pipeline out as a procedure.

Draft. A drafting agent on a mini-SWE-agent scaffold writes the statement in a shell with two tools, a compiler against a pinned Lean version and semantic search over Mathlib. A submission is accepted only if the file compiles with exactly one theorem, one `sorry` and no axioms, re-verified after the agent exits. The prompt lists the places where formalizations tend to go wrong (omitted side conditions and conventions, definitions broader or narrower than the paper's, missing parts of the conclusion, quantifier scope, degenerate cases, coercions). Drafting costs a few cents per statement and compiles for 98% of the 89 ArXivLean claims.

Cheat-prover. An inexpensive model tries, for three rounds, to prove the draft outright with a degenerate witness such as zero, the identity, the empty family, a constant function or the trivial group. Each retry sees the previous attempt and the compiler's full message, and when the attempts fail to compile a frontier model gets two more rounds. A success is checked by the kernel and shows that the statement as written is trivially true. The proof is passed to the next two stages. This stage found the March problem 4 defect of Section 4 in its first round.

Disprover. An inexpensive model tries for eight rounds, with the same error feedback, to prove the negation of the draft, and the provers of the settle stage then try as well. A kernel-checked success is the strongest certificate the pipeline produces. It shows that the statement is false as written, and no judge and no reference are needed to say so. The proof is shown to the attacker and the judge as evidence. The stage is adapted from Skidanov's PutnamBench solver (Skidanov, 2026), whose disproof route found two false PutnamBench references. In a separate test it disproved 3 of 9 planted defects that strengthen a true statement into a false one and stayed silent on all 178 statements that should be true, at about $0.18 per statement.

Attack. Two further agents on the same scaffold, with the same tools and built on different models, receive the draft and the paper. Their instructions describe the task and a procedure, and contain no list of known mistakes. The procedure has five steps. The agent prints what every Mathlib notion, notation and special value in the draft means in the pinned Mathlib, because a symbol can mean something different there from what the paper means or from what the model remembers. It states, for every hypothesis, quantifier and helper definition, the set of objects it ranges over in Lean and the set the paper intends. It compares every part of the conclusion with the claim. It tries to satisfy the draft

with a degenerate object in a scratch file. It checks subtraction and division on the naturals, coercions, and which norm or measure a notion carries. Where the draft departs from the paper's meaning, the agent writes a compiling alternative that changes that one thing, with a situation in which the two statements disagree and a quote from the paper. Mechanical gates check the proposal, among them that the quote occurs in the text. An agent may report that it found no difference, but only after the first two steps, and on statements judged correct the agents did so in almost every run. We do not ask for "stronger" or "weaker" variants, since that framing excludes structural errors. Every command an agent issues is checked before it runs, and a command that reaches outside the agent's own directory is refused, so that an agent cannot read another run's files or a reference formalization.

Settle. For each alternative the kernel is asked first whether it is the same statement as the draft. The two are treated as propositions and an escalating series of provers attempts both implications. If both close, the alternative is a rewording and is discarded. The most common false repair is the same statement rewritten in the paper's notation. Each remaining alternative is put to a frontier model twice, with a third call when the two disagree, with the kernel's finding, the cheat-prover's proof if there is one, and kernel-printed definitions of every library notion on which the two differ. The judge must give a situation on which the two statements make different claims, write both claims down, and quote verbatim the sentence of the claim, or of a definition it relies on, that decides it. An alternative is adopted when two votes agree with verbatim quotes and the two written claims differ.

Two further rules apply. The first is the convention question. Without it, the attackers proposed the right repair for March p25 in every run and the kernel proved it to be the stronger statement, yet the judge answered that the two statements were the same, because the paper says "Sylow subgroup" and both statements mention Sylow subgroups. When the kernel has proved that the alternative is the stronger statement and the votes do not adopt it, the judge is therefore asked once whether the paper's wording, or the standard convention for the objects it introduces, restricts them to the alternative's range, and a yes with a verbatim quote adopts the alternative. A second follow-up question covers the mirror case, in which the alternative only adds a hypothesis, the kernel has proved that the draft implies it, and both votes say the text does not decide. The judge is then asked whether the text's conventions imply the added hypothesis. The second rule protects the second pass. Without it, the second pass can lose the evidence that justified the first repair. In one run the first pass repaired March p4 on the strength of the cheat-prover's degenerate proof, and the second pass, which no longer had that proof in front of it, adopted a variant that removed the repair. An alternative that a degenerate proof from an earlier pass still proves is therefore discarded without asking the judge. An adopted alternative replaces the draft, and the pipeline runs once more on the result.

Roles and cost. Roles whose output the kernel checks, namely the cheat-prover, the disprover and the first provers in the sequence, run on an inexpensive model, since a weak model there cannot produce a wrong verdict. The drafter, the attackers, the last prover and the judge run on frontier models. Over the 89 benchmark problems a draft costs about $0.24, and one pass of the pipeline over a statement costs about $0.80 at the median and about $3 for a statement with a real defect, since each proposed repair goes through the kernel ladder in both directions. The independent grade of a changed statement costs about $0.80. A pass over all 89 references costs about $93, of which the two attackers account for a little over half and the kernel ladder for a third.

C More details about the settle rule

We fix the terms used in the rest of the paper. In this appendix S abbreviates the formal statement S_{Lean} of Section 2. A *statement* S is a Lean file that compiles with exactly one theorem $T : \varphi_S$ whose proof is `sorry`, and φ_S is its proposition. The text X is the claim S_{NL}, optionally followed by the paper. A *variant* V of S is a statement proposed by an attacker together with a stated change, a situation in which S and V disagree, and a quote from X.

The *kernel relation* $\mathrm{rel}(S, V)$ is computed by attempting the two theorems $\varphi_S \to \varphi_V$ and $\varphi_V \to \varphi_S$, first with a fixed cascade of tactics and then with a ladder of provers. A proof is accepted only if it compiles without `sorry` and uses no axiom beyond `propext`, `Classical.choice` and `Quot.sound`. The relation takes one of four values, which are $\equiv$ when both implications are proved, $S \Rightarrow V$ or $V \Rightarrow S$ when one is, and *unresolved* otherwise. When a proof of an implication never uses its hypothesis, both statements are simply theorems, and we record *both provable* instead

The pipeline as a procedure. Input: the claim's text and, when available, the paper. Output: a Lean statement with one `sorry`, and its audit record.

1. **Draft.** An agent writes a statement that compiles with exactly one theorem and one `sorry`, using the compiler and Mathlib search.
2. **Cheat-prover.** A prover tries to prove the draft with a degenerate witness, seeing its previous attempt and the compiler's message on each retry. If the kernel accepts, the proof becomes evidence and is kept for every later pass.
3. **Disprover.** The same prover tries to prove the negation. If the kernel accepts, the disproof becomes evidence.
4. **Attack.** Two attackers, given the claim, the paper and the evidence, each follow the procedure of Section 3 (print the meaning of every library notion, state the range of every hypothesis, compare the conclusion, try a degenerate object, check types) and then propose compiling variants that change one thing, with a separating situation and a quote from the paper, or report that they found no difference. A command that reaches outside the attacker's directory is refused.
5. **Settle.** For each variant: the kernel is asked whether it is equivalent to the draft, by attempting both implications, and an equivalent variant is discarded. A variant that a degenerate proof from an earlier pass still proves is discarded. A judge is asked twice which of the two the claim asserts, with a separating situation and a verbatim quote, and a third call breaks a tie. A variant is adopted if two calls agree with verbatim quotes. If the kernel proved the variant to be the stronger statement and the votes did not adopt it, the judge is asked once whether the text's wording or the standard convention restricts the objects to the variant's range. If the variant only adds a hypothesis that the draft implies and the votes say the text does not decide, the judge is asked whether the text's conventions imply that hypothesis. A yes to either question, with a verbatim quote, adopts the variant.
6. **Repeat once** from step 2 on the adopted variant, then output the statement with the kernel certificates, the variants, the votes and the quotes.

Figure 3: The pipeline written out as a procedure. Steps 2 and 3, the equivalence attempt and the reversion check in step 5, are checked by the Lean kernel. The judge in step 5 is the only point at which a model's reading decides, and it has to quote the paper.

of $\equiv$. A *certificate* for S is a kernel-accepted proof of φ_S by a degenerate witness, found by the cheat-prover, or a kernel-accepted proof of $\neg\varphi_S$, found by the disprover.

A *vote* is one call of the judge. It returns whether the two statements differ, which of them the text asserts (or that the text does not decide), a situation σ, the two claims c_S and c_V written in prose, and a quote q. A vote *adopts* V when it says that the statements differ and that the text asserts V, when q occurs in X up to whitespace and the delimiters of mathematical mode, when $c_S \neq c_V$, and when σ is not empty. The variant V is adopted when $\mathrm{rel}(S, V)$ is not $\equiv$ and two votes adopt it, with a third vote cast when the first two split. It is also adopted in two further cases. When the kernel has proved $V \Rightarrow S$ and the votes do not adopt, the judge is asked the convention question, and a yes with a quote from X adopts V. When the kernel has proved $S \Rightarrow V$ and every vote says the text does not decide, the judge is asked whether the text's conventions imply the hypothesis that V adds, and a yes with a quote from X adopts V. The first adopted variant replaces S, and the procedure runs once more on it.

In the evaluation we use two further terms. A change from S to S^* is *kernel-certified* in direction d when $\mathrm{rel}(S, S^*)$ proves the implication in direction d. It is *grader-confirmed* when an independent model that is not used anywhere in the pipeline, shown X, S and S^* and nothing about the run, says in a majority of three calls that X asserts S^*, each time with a quote from X.

D THE PIPELINE AS A PROCEDURE

Figure 3 writes the pipeline of Section 3 out step by step and marks the points at which the Lean kernel, rather than a model, decides.

E ADDITIONAL EXPERIMENTAL DETAILS

Settings common to every run. The judge, the last prover and one attacker are GPT-5.6 Sol. The drafter, the other attacker, the cheat-prover and the first provers of the settle stage are DeepSeek

V4 Pro, and the disprover runs eight rounds of DeepSeek V4 Flash before the settle-stage provers. Each attacker has 40 steps. Every model call is made at its provider's maximum output length. The independent grader is Claude Opus 5, which plays no role in the pipeline. It is shown the claim sentence followed by the paper's source, the statement before and after the change, and nothing about the run, so it sees neither the pipeline's quote nor which side is the reference. It is called three times, and a change counts as confirmed when a majority says the text asserts the changed statement, each with a quote found in the text. Every Lean check runs against Mathlib pinned to the benchmark's revision. Every command an attacker issues is checked before it runs, and a scan of each attacker's record afterwards discards the run if it reached a path outside its task, so no attacker saw a reference. Every run was preregistered with numeric predictions and a spending cap before its first paid call, and the records, the cost ledgers and the preregistrations are in the code repository.

Cost of each run. The run over the 89 ArXivLean references cost $93, and grading its changes $3. The four reruns on the five defective references cost between $32 and $44 each. The run with the claim sentence alone cost $81, and the run of the whole pipeline, drafting included, on the 89 claims cost $81. The comparison on the fifteen Theo and Trellis claims, both sides, cost $45. The planted defects cost $141 to run and $37 to score.

F PLANTED DEFECTS BY KIND

Table 5 gives the result of Section 4.2 by kind of planted edit. The 54 mutants are those of Section 4.2. A repair is certified when the kernel proves that the final statement lies between the mutant and the reference in the planted direction, and grader-confirmed when the kernel could not decide the direction and a majority of three grader calls say the paper asserts the final statement.

kind of planted defect	run	certified	grader-confirmed	not repaired
free-form edit	14	12	2	0
hypothesis added that the claim does not have	8	7	1	0
restricted to a special case	7	6	1	0
isomorphism classes replaced by raw terms	5	4	1	0
domain narrowed	4	4	0	0
conclusion weakened	4	3	1	0
domain widened beyond the claim	4	4	0	0
generalized beyond the text	3	2	0	1
conclusion strengthened	2	2	0	0
strict and non-strict inequality swapped	2	2	0	0
needed hypothesis dropped	1	1	0	0
all	54	47	6	1

Table 5: Planted defects repaired by the pipeline, by kind.

The one that is not repaired, and the two mutants of March p4. One mutant of June p45 drops the hypothesis that the total measure is positive. Both attackers proposed restoring it, and the kernel proved each proposal equivalent to the mutant, which means the statement holds anyway when the measure is zero and the mutant is equivalent to its reference. The corpus certificate proves one implication only and had not excluded this. Under our rule an equivalence the kernel did not prove in advance counts as a miss. The two mutants of March p4 are excluded from the count, because the reference is provable outright, so both mutants are too, and the kernel can prove either implication between two provable statements. Read by hand, the pipeline reversed the planted edit in its first generation on both, restoring the detailed-balance conjunct on one and the bound $3 \leq m$ on the other, and in its second generation repaired the reference's own defect, which is the repair of Table 1.

G FORMALIZATION WITHOUT FULL NATURAL LANGUAGE RESULTS

Formalizing already proved (in natural language) theorems is an important part of AI assisted math, but another important part is formalizing statements or theorems that do not come with an accompanying proof, or possibly even exposition. To test our performance in this setting, we repeat

the runs with the text input cut down to the claim sentence. On the references the pipeline repairs March p25, June p1 and June p23, and the grader confirms all three against the sentence. It finds the degenerate proof of March p4 but adopts only a rewording, since no sentence of the claim justifies the positive rates that the paper's construction supplies. It also changes June p20 to finite graphs through the convention question, on the grounds that a right-angled Artin group is by definition given by a finite graph. The grader rejects this change against the sentence, which never mentions finiteness, and confirms it against the paper. Four further references change in ways the grader judges equivalent.

H CASE STUDIES

Here we describe the five defects our pipeline found on the reference side, in the order of Table 1. Each is a real error in a released ArXivLean reference. For each we show the lines of the official reference that the pipeline changed and the lines of its repair, exactly as in the released files.

March, problem p4 (the reference is trivially true). The claim is that for every $m \geq 3$ and every strictly positive probability mass function π on the cycle C_m, there is a nearest-neighbour Markov chain on C_m reversible with respect to π. The reference encodes the chain as a stochastic matrix supported on each vertex and its two neighbours, but nothing forces the chain to move, and the identity matrix satisfies every hypothesis. The cheat-prover produced that witness in its first round, and the proof checks under the pinned toolchain with the standard axioms only. The paper's construction has strictly positive rates to both neighbours ("the off-diagonal rates $q_{r,r\pm1}$ are well-defined and strictly positive"), and the repair adds that condition. Several models have "solved" this benchmark problem with the identity-matrix proof, as the ArXivLean leaderboard, which publishes each model's proof, shows (Dekoninck et al., 2026).

Reference:

```
(∀ i j, 0 ≤ P i j) ∧
(∀ i, ∑ j, P i j = 1) ∧
(∀ i j, P i j ≠ 0 →
  j = i ∨ j.1 = (i.1 + 1) % m ∨ i.1 = (j.1 + 1) % m) ∧
(∀ i j, π i * P i j = π j * P j i) ∧
(∀ j, ∑ i, π i * P i j = π j)
```

Repair:

```
(∀ i j, i ≠ j → 0 ≤ P i j) ∧
(∀ i, ∑ j, P i j = 0) ∧
(∀ i j, P i j ≠ 0 ↔
  j = i ∨ j.1 = (i.1 + 1) % m ∨ i.1 = (j.1 + 1) % m) ∧
(∀ i j, π i * P i j = π j * P j i) ∧
(∀ j, ∑ i, π i * P i j = 0)
```

March, problem p25 (the reference quantifies a Sylow condition over every natural number). The claim concerns finite groups with a maximal subgroup series whose terms are SS-quasinormal, meaning that each term H has a supplement B with $G = HB$ such that H permutes with every Sylow subgroup of B. The reference writes this as "for all $p : \mathbb{N}$ and all P : `Sylow p B`". In Mathlib, `Sylow p B` is defined for every natural p. For $p = 0$ every element has order dividing $0^k = 0$ trivially, so the only Sylow 0-subgroup is B itself, and for composite p the Sylow p-subgroups are the maximal subgroups whose element orders have prime factors among those of p. The reference's hypothesis therefore asks H to permute with more subgroups than "every Sylow subgroup", so the reference theorem is weaker than the paper's. Our draft restricts p to primes. The kernel proves that the repaired statement implies the reference. The converse is not proved, and we have no Lean proof that the two conditions differ.

Reference:

```
∀ p : ℕ, ∀ P : Sylow p B, ∀ x : G,
```

Repair:

```
∀ p : ℕ, Nat.Prime p → ∀ P : Sylow p B, ∀ x : G,
```

June, problem p1 (the reference's "smooth" is real-analytic). The claim says "let f be a smooth function". The reference writes `ContDiff ℝ ⊤ f`, and under the benchmark's pinned Mathlib the top element of `WithTop ℕ∞` is ω, real-analytic, not C^∞. Our own draft had the same hypothesis. The pipeline repaired both to `ContDiff ℝ (⊤ : ℕ∞)`, quoting the sentence above. The independent grader confirmed both, giving as its example x^2 plus a C^∞ bump that is not analytic, which satisfies the claim but not the reference.

Reference:

```
(hf : ContDiff ℝ ⊤ f)
```

Repair:

```
(hf : ContDiff ℝ ((⊤ : ℕ∞) : WithTop ℕ∞) f)
```

June, problem p20 (the central object is defined over a broader class). The claim is that there is a group with a finite-index subgroup isomorphic to a right-angled Artin group (RAAG) but no finite-index normal subgroup isomorphic to one. The paper defines RAAGs from finite simplicial graphs. The reference writes $\exists (V : \texttt{Type})\,(\Gamma : \texttt{SimpleGraph}\ V)$ with no finiteness on V, in both the positive clause and the negated one, so the positive clause becomes easier to satisfy and the exclusion clause harder. The attacker found this when it stated the range of the graph in Lean and in the paper. The kernel proved no implication between the two versions, since finiteness makes one clause easier and the other harder, and the judge decided from the definition sentence. All of our drafts restrict to finite graphs, and every grader that compares with the reference scored them as departing from the reference.

Reference:

```
        ∃ (V : Type) (Γ : SimpleGraph V),
...
            ¬ ∃ (V : Type) (Γ : SimpleGraph V),
```

Repair:

```
        ∃ (V : Type) (_ : Finite V) (Γ : SimpleGraph V),
...
            ¬ ∃ (V : Type) (_ : Finite V) (Γ : SimpleGraph V),
```

June, problem p23 (the reference asks only almost-everywhere measurability). This problem concerns an i.i.d. array of real random variables. The reference carries the entries only as almost-everywhere measurable functions. The pipeline required every entry to be measurable, quoting the claim's "random variables". The independent grader confirmed the change, giving as its example an entry modified on a non-measurable subset of a null set, which satisfies the reference but is not a random variable. The same change was made to our own draft of this problem.

Reference:

```
(hmeas : ∀ ij, AEMeasurable (w ij) μ)
```

Repair:

```
(hmeas : ∀ ij, Measurable (w ij))
```

Two references that do not compile. June p18 and p39 name `SimpleGraph.IsIsolated` and `Subgroup.lowerCentralSeries`, neither of which exists in the pinned Mathlib. A statement that does not compile cannot be checked at all. We list these separately from the defects of meaning.

Changes that are not defects. The run over all 89 references changed four further references, and the grader judged all four to be equivalent rewordings. March p3 adds $1 \le M$ inside an infimum, which only affects $n = 0$ in an asymptotic statement. March p11 re-encodes a degree. March p27 restricts a domain to the positive reals written as a subtype, and June p44 writes the positive naturals as a subtype. Three of the four were adopted through the convention question, which is why the pipeline's output should be read with its record.

I Details of the comparison with Theo and Trellis

Section 4 reports what the pipeline changed on the released statements of Theo (Moakhar et al., 2026) and Trellis (Pegden, 2026a). This appendix describes how the comparison was set up and what the changes are. Their statement is attacked as a statement. The target theorem's proof is replaced by `sorry`, and the file imports only the modules of their project that declare the definitions the statement uses, so that neither the cheat-prover nor an attacker can cite their proof. The attacker may read those definition modules. Our drafts are written by the pipeline from the claim text with Mathlib alone, defining what Mathlib lacks in the same file, and are listed in Appendix J.

The two defects. The pipeline changed three of Theo's eight statements, none of Trellis's seven, and none of our fifteen drafts. The grader confirmed all three changes, and two of them are defects. *Median mechanism.* The paper states that the approximation ratio is at most $\mathrm{UB}(q)$ "where $\mathrm{UB}(q)$ is an increasing function in q with $\mathrm{UB}(1) = 1$" and limit 3. Theo's statement asks for some function with those values at 1, 2 and infinity, and no monotonicity, so any finite bound at the other q satisfies it and the guarantee for intermediate q is empty. The repair adds monotonicity on $[1, \infty)$. A second change restricts the coordinate-wise median to the reported central values, as the paper's tie-breaking convention says. *Trace reconstruction.* Theo's statement asserts the lower-bound witnesses only for $n \equiv 1 \pmod 8$. Their own docstring calls this "a faithful specialization of the paper's odd-n construction", whereas the paper's theorem is stated for every large n. The repair drops the residue condition. The kernel proved that the trace reconstruction repair implies Theo's statement, so the repair is the stronger one. For the median mechanism it could prove neither direction, which is the expected answer. The repair adds a condition on UB and removes a case from the medians, so the two statements are incomparable and only the paper's text decides between them.

The third change to a Theo statement. The pipeline also changed Theo's statement of the sharp Talagrand selector-process theorem, replacing a hypothesis $t \geq 2$ by the claim's $t > 0$ and allowing a fractional cover to put weight on the empty set, and the grader confirmed the change against the claim's literal wording. On reading the paper it is not an error. The bound $q = cp/\log t$ is undefined at $t = 1$, so the claim's $t > 0$ means $t \geq 2$, and weight on the empty set loses no generality, since a cover with $w(\varnothing) = a$ rescales to one without it at cost $(1/2 - a)/(1 - a) \leq 1/2$. We count it as a change that is not a defect.

Limits of the comparison. Finding nothing to change is not the same as showing a statement correct, on either side. Trellis writes its statements with an agent, reviews them with four reviewing agents, and then holds them for a human's approval before any proof is attempted. The attack stage is the kind of check that could be run before that approval is asked for. Their systems prove the theorems and ours only audits statements, so there is no head-to-head cost. For scale, Trellis's released record prices its run on the off-diagonal Ramsey paper, statements and proofs, at about $2,800 at the list rates of the model it used (Pegden, 2026b), against about $1 to $3 to check one of its statements here.

J Our formal statements of the fifteen claims

Each file below is the pipeline's final statement for one claim, exactly as produced (imports included), compiled against the pinned Mathlib. Where Mathlib has no definition for a notion in the claim, the file defines it first. The five defective ArXivLean references and their repairs are shown in Appendix H.

A generalized trace reconstruction problem, Theorem 1 (Theo).

```
import Mathlib

/-- The law of a generalized trace.  At each position a Bernoulli bit with
parameter `p` is sampled, and that bit is then retained independently with
probability `1 - δ`. -/
noncomputable def generalizedTrace :
    (ps : List NNReal) → (∀ p ∈ ps, p ≤ 1) → (δ : NNReal) → δ ≤ 1 → PMF (List Bool)
  | [], _, _, _ => PMF.pure []
  | p :: ps, hp, δ, hδ =>
      (PMF.bernoulli p (hp p (by simp))).bind fun b =>
        (PMF.bernoulli (1 - δ) (by simp)).bind fun keep =>
```

```
          (generalizedTrace ps (fun q hq => hp q (by simp [hq])) δ hδ).map
            (fun t => if keep then b :: t else t)
termination_by ps => ps.length

 theorem stmt_main :
    ∃ (n₀ : ℕ) (c aInf AInf aOne AOne b : ℝ),
      1 ≤ n₀ ∧ 0 < c ∧ 0 < aInf ∧ 0 < AInf ∧ 0 < aOne ∧ 0 < AOne ∧ 0 < b ∧
      ∀ n : ℕ, n₀ ≤ n →
        ∃ (S S' : Fin n → NNReal)
          (hS : ∀ i, S i ≤ 1) (hS' : ∀ i, S' i ≤ 1),
          aInf ≤ ‖(fun i => (S i : ℝ) - (S' i : ℝ))‖ ∧
          ‖(fun i => (S i : ℝ) - (S' i : ℝ))‖ ≤ AInf ∧
          aOne * Real.sqrt n ≤ ∑ i, |(S i : ℝ) - (S' i : ℝ)| ∧
          (∑ i, |(S i : ℝ) - (S' i : ℝ)|) ≤ AOne * Real.sqrt n ∧
          ∀ (δ : NNReal) (hδpos : 0 < δ) (hδone : δ < 1),
            c / Real.sqrt n ≤ (δ : ℝ) →
              (1 / 2 : ℝ) * ∑' t : List Bool,
                |ENNReal.toReal (generalizedTrace (List.ofFn S)
                    (by simpa using hS) δ (le_of_lt hδone) t) -
                 ENNReal.toReal (generalizedTrace (List.ofFn S')
                    (by simpa using hS') δ (le_of_lt hδone) t)|
                ≤ Real.exp (-b * Real.sqrt n) := by sorry
```

A sharp version of Talagrand's selector process conjecture, main theorem (Theo).

```
import Mathlib

open scoped BigOperators

theorem stmt_main :
    ∃ c : ℝ, 0 < c ∧
      ∀ (X : Type) [Fintype X] [DecidableEq X],
        ∀ (p : ℝ) (t : ℕ) (ℋ : Set (Finset X)),
          p ∈ Set.Icc (0 : ℝ) 1 →
          0 < t →
          (∃ w : Finset X → ℝ,
            (∀ W, w W ∈ Set.Icc (0 : ℝ) 1) ∧
            (∀ H ∈ ℋ, 1 ≤ ∑ W ∈ Finset.univ.filter (fun W : Finset X => W ⊆ H), w W) ∧
            (∑ W : Finset X, w W * p ^ W.card) ≤ (1 / 2 : ℝ) ∧
            (∀ W, w W ≠ 0 → W.card ≤ t)) →
          ∃ g : Finset X → ℝ,
            (∀ W, g W = 0 ∨ g W = 1) ∧
            (∀ H ∈ ℋ, 1 ≤ ∑ W ∈ Finset.univ.filter (fun W : Finset X => W ⊆ H), g W) ∧
            (∑ W : Finset X,
              g W * (c * p / Real.log (t : ℝ)) ^ W.card) ≤ (1 / 2 : ℝ) := by
  sorry
```

Approximation guarantees of the median mechanism in $\mathbb{R}^d$, main theorem (Theo).

```
import Mathlib

noncomputable section

/-- The `ℓ_q` distance on a finite-dimensional real vector space. -/
def lqDistance {d : ℕ} (q : ℝ) (x y : Fin d → ℝ) : ℝ :=
  Real.rpow (∑ j, |x j - y j| ^ q) (1 / q)

/-- A coordinate-wise median, with either central reported value allowed when
there are an even number of reports. -/
def IsCoordinatewiseMedian {n d : ℕ} (p : Fin n → Fin d → ℝ)
    (m : Fin d → ℝ) : Prop :=
  ∀ j, (∃ i, p i j = m j) ∧
    2 * (Finset.univ.filter fun i => p i j < m j).card ≤ n ∧
    2 * (Finset.univ.filter fun i => m j < p i j).card ≤ n

theorem stmt_main :
    ∃ UB : ℝ → ℝ,
      MonotoneOn UB (Set.Ici 1) ∧
      UB 1 = 1 ∧
      Filter.Tendsto UB Filter.atTop (nhds 3) ∧
      UB 2 = Real.sqrt (6 * Real.sqrt 3 - 8) ∧
      ∀ (q : ℝ), 1 ≤ q →
        ∀ (d n : ℕ), 1 ≤ d → 1 ≤ n →
          ∀ (p : Fin n → Fin d → ℝ) (m f : Fin d → ℝ),
            IsCoordinatewiseMedian p m →
            (∑ i, lqDistance q (p i) m) ≤
              UB q * ∑ i, lqDistance q (p i) f := by sorry
```

Efficiently learning mixtures of two Gaussians, Theorem 4 (Theo).

```
import Mathlib

open MeasureTheory ProbabilityTheory BigOperators

 theorem stmt_main :
    ∃ c : ℝ, 0 < c ∧
      ∀ (ε : ℝ)
        (w w' : Fin 2 → ℝ) (μ μ' : Fin 2 → ℝ)
        (v v' : Fin 2 → NNReal),
        0 < ε →
        (∑ i : Fin 2, w i) = 1 →
        (∑ i : Fin 2, w' i) = 1 →
        (∀ i, ε ≤ w i ∧ w i ≤ 1) →
        (∀ i, ε ≤ w' i ∧ w' i ≤ 1) →
        (∀ i, |μ i| ≤ 1 / ε) →
        (∀ i, |μ' i| ≤ 1 / ε) →
        (∀ i, (v i : ℝ) ≤ 1) →
        (∀ i, (v' i : ℝ) ≤ 1) →
        |μ 0 - μ 1| + |(v 0 : ℝ) - (v 1 : ℝ)| ≥ ε →
        |μ' 0 - μ' 1| + |(v' 0 : ℝ) - (v' 1 : ℝ)| ≥ ε →
        (∀ π : Equiv.Perm (Fin 2),
          ε ≤ ∑ i : Fin 2, (|w i - w' (π i)| + |μ i - μ' (π i)| +
            |(v i : ℝ) - (v' (π i) : ℝ)|)) →
        ε < c →
        let F : Measure ℝ :=
          ∑ i, (ENNReal.ofReal (w i)) • gaussianReal (μ i) (v i)
        let F' : Measure ℝ :=
          ∑ i, (ENNReal.ofReal (w' i)) • gaussianReal (μ' i) (v' i)
        ∃ i : ℕ, 1 ≤ i ∧ i ≤ 6 ∧
          |(∫ x : ℝ, x ^ i ∂F) - ∫ x : ℝ, x ^ i ∂F'| ≥ ε ^ 67 := by
  sorry
```

Refuting the direct sum conjecture (deterministic communication complexity), main theorem (Theo).

```
import Mathlib

inductive DetProtocol (X Y Z : Type) where
  | leaf : Z → DetProtocol X Y Z
  | alice : (X → Bool) → DetProtocol X Y Z → DetProtocol X Y Z → DetProtocol X Y Z
  | bob : (Y → Bool) → DetProtocol X Y Z → DetProtocol X Y Z → DetProtocol X Y Z

namespace DetProtocol

variable {X Y Z : Type}

def eval : DetProtocol X Y Z → X → Y → Z
  | leaf z, _, _ => z
  | alice q p₀ p₁, x, y => if q x then eval p₁ x y else eval p₀ x y
  | bob q p₀ p₁, x, y => if q y then eval p₁ x y else eval p₀ x y

def cost : DetProtocol X Y Z → ℕ
  | leaf _ => 0
  | alice _ p₀ p₁ => 1 + max (cost p₀) (cost p₁)
  | bob _ p₀ p₁ => 1 + max (cost p₀) (cost p₁)

end DetProtocol

noncomputable def communicationComplexity {X Y Z : Type}
    (f : X → Y → Z) : ℕ :=
  sInf {d : ℕ | ∃ p : DetProtocol X Y Z,
    p.cost = d ∧ ∀ x y, p.eval x y = f x y}

def parallelPower {X Y Z : Type} (f : X → Y → Z) (l : ℕ) :
    (Fin l → X) → (Fin l → Y) → (Fin l → Z) :=
  fun xs ys i => f (xs i) (ys i)

theorem stmt_main :
    ∃ f : ∀ n : ℕ, (Fin n → Bool) → (Fin n → Bool) → Bool,
      ∀ N L C : ℕ, ∃ n l : ℕ,
        N ≤ n ∧ L ≤ l ∧
          (communicationComplexity (f n) : ℚ) >
            (communicationComplexity (parallelPower (f n) l) : ℚ) / l + C := by sorry
```

The strong perfect graph theorem, Theorem 1.2 (Theo).

```
import Mathlib

open SimpleGraph

/-- A finite simple graph is perfect when every induced subgraph has a clique
whose size equals its chromatic number. -/
def IsPerfect {V : Type*} [Fintype V] (G : SimpleGraph V) : Prop :=
  ∀ S : Set V, ∃ C : Finset S,
    (G.induce S).IsClique (C : Set S) ∧
      (C.card : ℕ∞) = (G.induce S).chromaticNumber

/-- A graph is Berge when it has no induced odd cycle of length at least five,
or the complement of such a cycle. -/
def IsBerge {V : Type*} (G : SimpleGraph V) : Prop :=
  ∀ n : ℕ, 5 ≤ n → Odd n →
    IsEmpty (SimpleGraph.cycleGraph n ↪g G) ∧
      IsEmpty ((SimpleGraph.cycleGraph n)ᶜ ↪g G)

theorem stmt_main {V : Type*} [Fintype V] (G : SimpleGraph V) :
    IsPerfect G ↔ IsBerge G := by sorry
```

A proof of the cycle double cover conjecture, Theorem 1.1 (Theo).

```
import Mathlib

open Set
open scoped Graph

namespace Graph

/-- A cycle in an undirected multigraph, represented by its cyclically ordered
vertices and edges. This includes loops (length one) and pairs of parallel
edges (length two). -/
def IsCycle (G : Graph α β) {n : ℕ} (v : Fin n → α) (e : Fin n → β) : Prop :=
  ∃ hn : 0 < n, Function.Injective v ∧ Function.Injective e ∧
    ∀ i, G.IsLink (e i) (v i)
      (v ⟨(i.val + 1) % n, Nat.mod_lt _ hn⟩)

/-- An undirected multigraph is bridgeless when each of its edges is on a cycle. -/
def IsBridgeless (G : Graph α β) : Prop :=
  ∀ e ∈ E(G), ∃ (n : ℕ) (v : Fin n → α) (c : Fin n → β),
    G.IsCycle v c ∧ ∃ i, c i = e

/-- A cycle double cover is a finite, multiplicity-indexed family of cycles in
which every graph edge occurs exactly twice. -/
def HasCycleDoubleCover (G : Graph α β) : Prop :=
  ∃ (k : ℕ) (n : Fin k → ℕ)
      (v : (i : Fin k) → Fin (n i) → α)
      (c : (i : Fin k) → Fin (n i) → β),
    (∀ i, G.IsCycle (v i) (c i)) ∧
    ∀ e ∈ E(G),
      (Set.ncard {p : Σ i, Fin (n i) | c p.1 p.2 = e} = 2)

end Graph

theorem stmt_main {α β : Type*} (G : Graph α β)
    (hV : V(G).Finite) (hE : E(G).Finite) (hG : G.IsBridgeless) :
    G.HasCycleDoubleCover := by sorry
```

Planar point sets with many unit distances, Theorem 1.1 (Theo).

```
import Mathlib

open scoped Real

/-- The number of unordered unit-distance pairs in a finite planar point set. -/
noncomputable def unitDistanceCount (P : Finset (EuclideanSpace ℝ (Fin 2))) : ℕ :=
  ((P.powersetCard 2).filter fun e =>
    ∃ x ∈ e, ∃ y ∈ e, dist x y = 1).card

/-- The maximum number of unit distances determined by an `n`-point planar set. -/
noncomputable def maxUnitDistanceCount (n : ℕ) : ℕ :=
  sSup {k : ℕ | ∃ P : Finset (EuclideanSpace ℝ (Fin 2)),
    P.card = n ∧ unitDistanceCount P = k}

theorem stmt_main :
```

```
    ∃ δ : ℝ, 0 < δ ∧
      Set.Infinite {n : ℕ | 0 < n ∧
        (n : ℝ) ^ (1 + δ) ≤ (maxUnitDistanceCount n : ℝ)} := by
  sorry
```

Improving $R(3,k)$ in just two bites, main theorem (triangle-free graphs) (Trellis).

```
import Mathlib

 theorem stmt_main :
    ∀ ε : ℝ, 0 < ε →
      ∃ n₀ : ℕ, ∀ n : ℕ, n₀ ≤ n →
        ∃ G : SimpleGraph (Fin n),
          G.CliqueFree 3 ∧
            (G.indepNum : ℝ) <
              (1 + ε) * Real.sqrt ((n : ℝ) * Real.log (n : ℝ)) := by sorry
```

Improving $R(3,k)$ in just two bites, $R(3,k)$ lower bound (Trellis).

```
import Mathlib

open Filter

/-- The least order such that every graph of that order has an `ℓ`-clique or a
`k`-vertex independent set. -/
noncomputable def ramseyNumber (ℓ k : ℕ) : ℕ :=
  sInf {N : ℕ | ∀ G : SimpleGraph (Fin N),
    (∃ s : Finset (Fin N), G.IsNClique ℓ s) ∨
    (∃ s : Finset (Fin N), G.IsNIndepSet k s)}

theorem stmt_main :
    ∀ ε : ℝ, 0 < ε →
      ∀ᶠ k : ℕ in atTop,
        (1 / 2 - ε) * ((k : ℝ) ^ 2 / Real.log k) ≤ (ramseyNumber 3 k : ℝ) := by sorry
```

Off-diagonal Ramsey numbers, main theorem (Trellis).

```
import Mathlib

noncomputable def ramseyNumber (s k : ℕ) : ℕ :=
  sInf {n : ℕ | ∀ G : SimpleGraph (Fin n),
    (∃ A : Finset (Fin n), G.IsNClique s A) ∨
      ∃ A : Finset (Fin n), G.IsNIndepSet k A}

theorem stmt_main :
    ∀ s : ℕ, s ≥ 3 → ∃ c_s : ℝ, 0 < c_s ∧
      ∀ k : ℕ, k ≥ 2 →
        (ramseyNumber s k : ℝ) ≥
          c_s * (k : ℝ) ^ (s - 1) / (Real.log (k : ℝ)) ^ (2 * s - 4) := by
  sorry
```

Off-diagonal Ramsey numbers, general off-diagonal bound (Trellis).

```
import Mathlib

noncomputable def ramseyNumber (s k : ℕ) : ℕ :=
  sInf {n : ℕ | ∀ G : SimpleGraph (Fin n),
    ¬ G.CliqueFree s ∨ ¬ G.IndepSetFree k}

theorem stmt_main :
    ∀ δ : ℝ, 0 < δ →
      ∃ L : ℕ, ∀ s k : ℕ,
        0 < s → 0 < k → L ≤ s → L * s ≤ k →
          ((ramseyNumber s k : ℕ) : ℝ) ≥
            Real.rpow ((k : ℝ) / (s : ℝ)) ((1 - δ) * (s : ℝ)) := by
  sorry
```

Off-diagonal Ramsey numbers, $r(s,Cs)$ bound (Trellis).

```
import Mathlib

noncomputable def ramseyNumber (s k : ℕ) : ℕ :=
  sInf {n : ℕ | ∀ G : SimpleGraph (Fin n),
    ¬ G.CliqueFree s ∨ ¬ G.IndepSetFree k}

theorem stmt_main (C : ℝ) (hC : C > 1) :
    ∃ s₀ : ℕ, ∀ s : ℕ, s₀ ≤ s →
      (ramseyNumber s ⌊C * (s : ℝ)⌋₊ : ℝ) ≥
        (2 ^ (1 - 1 / (2 * C)) : ℝ) ^ s := by
  sorry
```

Off-diagonal Ramsey numbers, near-diagonal bound (Trellis).

```
import Mathlib

noncomputable def ramseyNumber (s k : ℕ) : ℕ :=
  sInf {n : ℕ | ∀ G : SimpleGraph (Fin n),
    ¬ G.CliqueFree s ∨ ¬ Gᶜ.CliqueFree k}

theorem stmt_main :
    ∀ a : ℕ → ℕ,
      (Asymptotics.IsLittleO Filter.atTop
        (fun s : ℕ => (a s : ℝ)) (fun s : ℕ => (s : ℝ))) →
      ∃ ε : ℕ → ℝ, Filter.Tendsto ε Filter.atTop (nhds 0) ∧
        ∀ᶠ s : ℕ in Filter.atTop,
          (ramseyNumber s (s + a s) : ℝ) ≥
            (1 + ε s) * ((s : ℝ) / Real.exp 1) *
              Real.rpow 2
                (((s : ℝ) + (a s : ℝ) - 1) / 2 -
                  ((a s : ℝ) ^ 2) / (2 * (s : ℝ))) := by sorry
```

Off-diagonal Ramsey numbers, multicolour bound (Trellis).

```
import Mathlib

open scoped Sym2

/-- The multicolor Ramsey number: the least `n` such that every coloring of the
edges of `K_n` with `ℓ` colors has a monochromatic clique of size `s`. -/
noncomputable def multicolorRamsey (s ℓ : ℕ) : ℕ :=
  sInf {n : ℕ | ∀ coloring : (SimpleGraph.completeGraph (Fin n)).edgeSet → Fin ℓ,
    ∃ S : Finset (Fin n), S.card = s ∧
      ∃ c : Fin ℓ, ∀ v ∈ S, ∀ w ∈ S, ∀ h : v ≠ w,
        coloring ⟨s(v, w), (SimpleGraph.mem_edgeSet _).2 h⟩ = c}

theorem stmt_main :
    ∀ ℓ : ℕ, 3 ≤ ℓ →
      ∃ c : ℝ, 0 < c ∧ ∃ s₀ : ℕ, ∀ s : ℕ, s₀ ≤ s →
        c * Real.rpow 2 ((((ℓ - 1 : ℕ) : ℝ) * (s : ℝ)) / 2) ≤
          (multicolorRamsey s ℓ : ℝ) := by sorry
```

K PIPELINE PROMPTS

This appendix reproduces every prompt of the released pipeline, verbatim from the repository. Placeholders in double braces are filled in at run time with the theorem name, the paths of the compile and search scripts, and the step and variant budgets, and one absolute path has been replaced by a description in angle brackets. The source text (the claim sentence, followed by the paper) is appended to the attacker's and the judge's messages and is omitted here. Each agent runs in a shell and sees the output of every command it issues.

K.1 DRAFTING AGENT

The drafter receives the claim and, when available, the paper, and must submit a file that compiles with exactly one theorem and one `sorry`.

```
1 You are a Lean 4 statement formalization assistant that can interact
      with
```

```
a computer shell.

Formalize the CLAIM below as a Lean 4 theorem STATEMENT. You never prove
it: the proof must be exactly `sorry`.

The CLAIM is the FIRST PARAGRAPH of the text. The rest is the paper: use
it for definitions, notation, conventions and standing assumptions the
claim relies on. The paper's own theorem may say more than the claim
(sharper constants, a "moreover", a construction's parameters): do NOT
add content the claim does not state.

{{task}}

Faithfulness to the claim, read with the paper's definitions, is the
only objective: every hypothesis, regularity class, domain, and side
condition exactly as asserted, not stronger, not weaker, no smuggled
assumptions inside definitions.

## Checklist before you submit (the places formalizations go wrong)

1. Hypotheses the claim states, or relies on by convention, that your
   draft omits: side conditions, positivity, nonemptiness, 0 < n,
   n ≥ n₀, finiteness of a set, graph, group or index type, regularity
   (continuous, C^k, smooth, measurable, integrable, bounded), standing
   assumptions stated earlier in the paper, conventions carried by
   notation (σ^2 is a variance, so σ ≥ 0; "a prime p"; a list a₁..aₙ is
   nonempty).
2. Hypotheses your draft adds that the claim does not: a special case, a
   stronger regularity class, strict where non-strict is meant, an extra
   normalization or restriction.
3. Definitions: each helper `def` and each mathlib notion must mean
   exactly what the paper's definition means (finite vs arbitrary,
   simple graph vs multigraph, prime vs any natural, which norm,
   indexing conventions, "nonconstant", "proper").
4. The conclusion: every part of the claim, the claim's constants and
   exponents, existence vs uniqueness, and the carrier of a count or
   quantifier ("up to isomorphism" vs literal equality or counting of
   Lean terms).
5. Quantifier structure: ∀∃ vs ∃∀, which constants depend on which
   parameters, "for all sufficiently large n".
6. Degenerate cases: your statement must not be satisfiable trivially
   (empty set, zero, identity, constant function, trivial group) where
   the claim's objects exclude that case. Test: try to prove your draft
   with such a witness in a scratch file; if you can, add the missing
   condition.
7. Types and coercions: ℕ subtraction and division, reals vs naturals,
   `Real.rpow` vs `^`, extended reals, `Finset` vs `Set`, measures vs
   probability measures.

Never `cd`; every command runs in this task directory, and every file you
write goes here. The fixed target file is {{lean_target}} in the current directory. It
must contain `import Mathlib` and exactly one theorem named `stmt_main`
whose proof is exactly `by sorry`. Helper `def`s are allowed only when
mathlib genuinely lacks the concept, and they must not restate the goal.
```

```
No `axiom` declarations. No other `sorry`.

## Recommended workflow

1. Read the claim and find the corresponding theorem and definitions in
   the paper (`grep -n` in problem.txt, which holds the complete text).
2. Search mathlib for the right vocabulary instead of guessing:
   {{search_cmd}} "your query"
3. Draft {{lean_target}}, then compile it:
   {{check_cmd}} {{lean_target}}
4. Read every error, revise, and compile again. Scratch `.lean` files in
   this directory with `#check` are fine.
5. Go through the checklist against the claim. Then, when the target
   compiles with only the one intended `sorry` warning, submit and
   finish with the stand-alone command:
   bash {{submit_cmd}}
   Do not combine it with any other command.
   <important>After this command, you cannot continue working.</important>

## Command execution rules

You are operating in a loop where:

1. You issue at least one command.
2. The system executes the command or commands in a subshell.
3. You see the result.
4. You write your next command or commands.

Each response should include reasoning text and AT LEAST ONE bash tool
call. Directory or environment variable changes are not persistent; every
action runs in a new subshell.
```

K.2 Attacking agents

Two attackers, one on GPT-5.6 Sol and one on DeepSeek V4 Pro, receive the same instructions. The instructions name no particular mistakes and instead require a procedure before the agent may report that it found no difference.

```
You audit Lean 4 formalizations of research mathematics, working in a
computer shell. You do not trust your memory of what a Lean notion or
symbol means: you check it in this Mathlib.

A Lean 4 statement is under audit. The draft compiles. The source text
is below: its FIRST PARAGRAPH is the CLAIM being formalized; the rest is
the paper, which supplies the definitions and conventions the claim
relies on. Find every place where the draft's mathematical meaning
differs from the claim, and write a corrected alternative for each. Do
not add content the claim does not state (the paper's theorem may say
more than the claim).

{{task}}

Never `cd`; every command runs in this task directory and every file
you write goes here. The draft is {{lean_target}} in the current
directory (theorem `{{theorem_name}}`). Do not modify it. The complete
source text is in `problem.txt` (grep it for definitions and for the
sentences you will quote). Compile a scratch `.lean` file with
`{{check_cmd}} <file>`; search Mathlib with `{{search_cmd}} "query"`.
```

```
Mathlib's sources are under
<pinned Mathlib checkout>/Mathlib; do not
look anywhere else on the filesystem.

## Procedure (do all of it before you conclude anything)

1. Meanings. List every Mathlib notion, notation and symbol the draft
   uses whose exact meaning matters (definitions, type classes,
   special values like ⊤ or ∞, coercions, norms, measures). For each
   one, `#print` or `#check` it, or read its Mathlib source, and write
   down what it means in this Mathlib. A symbol's meaning in Mathlib
   can differ from the paper's usage and from what you remember.
2. Ranges. For every hypothesis, quantifier and helper definition in
   the draft, state the set of objects it ranges over in Lean, and
   the set the text intends (from the claim, the paper's definitions,
   and the standard convention for that term of art). A range that is
   wider or narrower than the text's is a difference, even when the
   extra objects seem harmless.
3. Conclusion. Compare every part of the draft's conclusion with the
   claim: each conjunct, each direction of an equivalence, constants
   and exponents, existence versus uniqueness, what is being counted
   and up to what identification.
4. Trivial truth. Try to satisfy or prove the draft with a degenerate
   object (empty, zero, constant, trivial) in a scratch file. If it
   succeeds and the text's objects exclude that case, that is a
   defect.
5. Types. Check subtraction and division on ℕ, coercions between ℕ, ℤ
   ,
   ℝ and extended reals, `Finset` versus `Set`, and which norm or
   measure a notion carries.

## What is not a difference

A restyling with the same meaning: another encoding of the same
objects, a definition folded or unfolded, a renamed variable, a
hypothesis already implied by the others. Before submitting a variant,
ask: is there a situation, in the text's own terms, on which the draft
and the variant make different claims? If not, do not submit it.

## Budget

You have {{step_limit}} commands. Write `variant_1.lean` as soon as
    you
have one concrete difference and keep probing for others; reserve the
last five commands for `variants.json` and the submit.

## Deliverable

Up to {{max_variants}} files `variant_1.lean`, `variant_2.lean`, ...
Each is a COMPLETE compiling file: `import Mathlib`, any helper
definitions, exactly one `theorem` line, named `{{theorem_name}}`,
proof exactly `:= by sorry`, no `axiom`, no other `sorry`. Each variant
changes ONE thing and keeps everything else byte-identical to the
draft; different variants address different issues.

Also write `variants.json` (use python: `json.dump` to a file, so
    LaTeX
backslashes are escaped correctly), a JSON list with one object per
variant:
  {"file": "variant_1.lean",
   "change": "what the variant changes, one sentence",
   "situation": "a situation in the text's own terms on which the
    draft
                 and the variant make different claims, and what each
                 claims there",
```

```
    "quote": "a sentence copied VERBATIM from problem.txt (of the claim,
              or of a definition or convention the claim relies on) that
              decides it",
    "evidence": "a short Lean snippet (an `example`, `#check` or
                 `#print`) that exhibits the situation or the meaning of
                 a Mathlib notion; it is compiled and, if it compiles,
                 shown to the judge as kernel-checked. Required whenever
                 the variant narrows or widens the range of a hypothesis
                 or quantifier: exhibit an object that one statement
                 admits and the other does not."}
The submit check rejects a quote that is not a verbatim substring of
problem.txt.

If, after running the procedure above, you find no difference in
meaning, create an empty file named `NO_VARIANTS` instead. Do not
refuse before step 1 and step 2 are done; do not invent a variant to
have something to submit.

When done, submit and finish with the stand-alone command:
bash {{submit_cmd}}
Do not combine it with any other command.
<important>After this command, you cannot continue working.</important>

## Command execution rules

You are operating in a loop where:

1. You issue at least one command.
2. The system executes the command or commands in a subshell.
3. You see the result.
4. You write your next command or commands.

Each response should include reasoning text and AT LEAST ONE bash tool
call. Directory or environment variable changes are not persistent;
every action runs in a new subshell.
```

K.3 CHEAT-PROVER AND DISPROVER

Both receive the statement as a Lean file and its proposition. A reply is accepted only if the proof compiles without `sorry` and uses no axiom beyond the standard three. A retry sees the previous attempt and the compiler's full message.

```
You are a cheat-prover. Try to prove the Lean proposition below
    OUTRIGHT using a degenerate or trivial witness (zero, the
    identity, the empty set/family, a constant function, the trivial
    group) or by exposing vacuity. Produce ONE ```lean block
    appending exactly:
  theorem degenerate_proof : (<the original theorem's statement as a
    proposition>) := by <complete proof, no sorry, no new axioms>
If you cannot, reply with exactly CANNOT PROVE and nothing else.
```

```
You are a disprover for Lean 4 / Mathlib. The file below states a
    theorem with proof `sorry`. Try to prove that the statement is
    FALSE. Output ONE ```lean block that appends exactly:
  theorem disproof : ¬ (<the statement as a proposition, EXACTLY as
    given>) := by <complete proof, no sorry, no new axioms>
```

```
3 Helper lemmas before it are fine; do not repeat the file's imports or
      definitions. Typical routes: a concrete counterexample (a small
      n, the empty family, zero, a constant function, a degenerate
      object the hypotheses admit), a contradiction among the
      hypotheses, or an edge case the statement forgot to exclude (n =
      0, an empty type, division by zero being 0 in Lean). If you
      cannot, reply with exactly CANNOT PROVE.
```

K.4 JUDGE

The judge is called twice per surviving alternative, with a third call when the two disagree, at temperature 0.1. Its message contains the source text, the two statements, the attacker's stated change and situation, the kernel's finding, any degenerate proof or disproof, and the kernel-printed definitions of the library notions on which the two statements differ.

```
1 You audit a Lean 4 formalization against its source text. The CLAIM
      under audit is the FIRST PARAGRAPH of the source text; the rest
      of the text is context for definitions, conventions and standing
      assumptions. The paper's own theorem may say more than the claim
      (extra constants, a 'moreover' clause, a construction's
      parameters); content the claim does not state is NOT asserted for
      this task. You are given the DRAFT statement and an ALTERNATIVE
      proposed by an adversarial auditor, who often proposes
      restylings: the same statement in the text's notation (a subtype
      domain instead of a restriction, an explicit definition instead
      of the mathlib notion, a conjunct that is already implied).
2 Step 1: do the two statements differ in MATHEMATICAL CONTENT? They
      differ only if you can describe, in the text's own terms, a
      situation (an object, instance, parameter value, or case) that
      actually exists, on which the two statements make different
      claims. Write down what each statement claims about it
      (claim_draft, claim_alternative). If the extra condition one side
      carries is automatically satisfied under the other, or the two
      are different encodings of the same objects, answer differ=false.
      A change in what is counted or quantified over (isomorphism
      classes versus Lean terms, so that one side is satisfied by
      copies of a single object and the other is not) IS a difference
      in content. When the auditor supplies KERNEL-CHECKED evidence (a
      compiled Lean snippet), trust it over your recollection of what a
      mathlib notion means.
3 Step 2: if they differ, which statement does the CLAIM assert in that
      situation? Copy VERBATIM the sentence that decides it: a sentence
      of the claim itself, or a definition, convention, or standing
      assumption the claim relies on. Conventions carried by the text's
      wording count as asserted and are decided by the sentence that
      introduces the object: 'variance σ^2' makes σ a standard
      deviation, so σ ≥ 0; 'positive integers a_1, ..., a_n' or 'an n × n
      matrix' means n ≥ 1; 'a finite graph' in the text's definition is
      finite; 'a prime p' is prime. If the draft asserts a case the
      text never considers and the alternative merely excludes that
      case (an empty family, a negative standard deviation, n = 0), the
      claim asserts the ALTERNATIVE; do not answer paper_does_not_say
      for a convention. If the DRAFT has been proved outright by a
      degenerate witness (shown to you), then the draft as written
      claims nothing the text's objects would not trivially satisfy; if
      the alternative excludes that witness and the text's definition
      of the object excludes it too, the claim asserts the ALTERNATIVE.
      A sentence about a construction, example, remark, or proof does
      not decide the claim. If nothing decides it, answer
      paper_does_not_say. Resemblance to the text's wording ('directly
      encodes', 'exactly as stated') is not evidence. Never answer
      which statement is stronger, more general, or which you prefer.
```

```
Answer ONLY JSON: {"differ": true|false, "situation": "<one
    sentence>", "claim_draft": "<what the draft claims there>",
    "claim_alternative": "<what the alternative claims there>",
    "paper_asserts": "draft"|"alternative"|"paper_does_not_say",
    "quote": "<verbatim sentence, or empty>", "why": "<one sentence>"}
```

K.5 THE TWO FOLLOW-UP QUESTIONS

The first question is asked when the kernel has proved that the alternative is the stronger statement and the votes did not adopt it. The second is asked when the alternative only adds a hypothesis and the kernel has proved that the draft implies it.

```
You judge a Lean 4 formalization against the mathematical text it
    formalizes. The DRAFT and the ALTERNATIVE differ in one way: the
    DRAFT's hypothesis ranges over MORE objects than the
    ALTERNATIVE's (a machine-checked proof shows the alternative
    implies the draft). Question: does the text's own wording, or the
    standard convention for the objects it introduces, restrict those
    objects to the ALTERNATIVE's range, so that the DRAFT demands
    something the text never asks? For example, a term of art may by
    convention name only a subclass of the objects the draft ranges
    over, or a Mathlib notion may be defined on a wider domain than
    the mathematical one. Answer ONLY JSON:
    {"restricted_by_convention": true|false, "quote": "<the sentence
    of the claim or of a definition that introduces the objects,
    verbatim, or empty>", "why": "<one sentence>"}
```

```
You audit a Lean 4 formalization against its source text. The CLAIM
    is the first paragraph of the source text. The ALTERNATIVE
    differs from the DRAFT only by an added hypothesis (a
    machine-checked proof shows the draft implies the alternative).
    Question: is that hypothesis implied by the text's own wording or
    standard conventions for the objects it introduces, so that the
    draft, which omits it, asserts cases the text never considers?
    Examples: 'variance σ²' makes σ a standard deviation, so σ ≥ 0;
    'positive integers a₁, ..., aₙ' or 'an n × n matrix' means n ≥ 1;
    'a prime p' is prime; 'a finite graph' is finite; a probability
    lies in [0, 1]. Answer ONLY JSON: {"implied_by_convention":
    true|false, "quote": "<the sentence of the claim or definition
    that introduces the object, verbatim, or empty>", "why": "<one
    sentence>"}
```

K.6 INDEPENDENT GRADER

The grader is a model that plays no role in the pipeline (Claude Opus 5). It sees the claim and paper, the statement before and after the change, and, when the pipeline found a degenerate proof, that proof. It is called three times per change and the majority decides.

```
You grade whether a change to a Lean 4 formal statement moved it
    toward or away from the natural-language claim. You see the
    CLAIM, the statement BEFORE the change, and the statement AFTER.
    Decide which of the two the claim asserts. Rules: (1) the target
    is the claim as written, read with its stated definitions;
    content the claim does not state is not required, and adding it
    is not more faithful. (2) 'same' means the two statements assert
    the same proposition up to encoding; a change of notation, order,
    or an equivalent Mathlib spelling is 'same'. (3) You must give a
    concrete separating situation (specific objects for which one
    statement holds and the other fails) before choosing a side; if
    you cannot, answer 'same'. (4) Quote, verbatim, the part of the
    claim that decides it, or write an empty quote and answer
```

```
    'claim_does_not_decide'. Answer in JSON only: {"situation": str,
    "claim_asserts": "before"|"after"|"same"|"claim_does_not_decide",
    "quote": str, "why": str}.

[appended when the pipeline found a degenerate proof:]
 A KERNEL FACT may be given: a Lean proof, checked by the kernel,
    that the BEFORE statement is provable outright by a degenerate
    witness. Treat it as true. A statement provable by a degenerate
    witness asserts nothing the claim's author proved, unless the
    claim itself is about that degenerate object.
```

K.7 SINGLE-CALL CHECKER

The prompt of the single-call rows of Table 3, one call per statement at temperature 0. A statement is flagged when the answer says it is not faithful. The kinds of failure it lists are the ones the attackers' procedure checks.

```
System:
You check whether a Lean 4 theorem statement asserts exactly the
    mathematical claim it is meant to formalize. The statement is
    unfaithful if it is weaker than the claim, stronger than the
    claim, about different objects, trivially true, false as written,
    or uses a convention the claim does not. Read the Lean carefully:
    binders, hypotheses, the exact conclusion, coercions, and the
    definitions in the file. Answer with one JSON object only:
    {"faithful": true or false, "defect": "one sentence naming the
    defect, or an empty string"}.

User:
CLAIM:
<the claim sentence, followed by the paper's source>

LEAN STATEMENT:
```lean
<the statement file>
```
```