OProver: A Unified Framework for Agentic Formal Theorem Proving

Anonymous ACL submission

Abstract

Recent progress in formal theorem proving has benefited from large-scale proof generation and verifier-aware training, but agentic proving is rarely integrated into prover training, appearing only at inference time. We present OProver, a unified framework for agentic formal theorem proving in Lean 4, in which failed proof attempts are iteratively revised using retrieved compiler-verified proofs and Lean compiler feedback. OProver is trained through continued pretraining followed by iterative post-training: each iteration runs agentic proving, indexes newly verified proofs into OProofs and the retrieval memory, uses repair trajectories as SFT data, and uses unresolved hard cases for RL. OProofs is built from public Lean resources, large-scale proof synthesis, and agentic proving traces, containing 1.76M Lean statements, 6.80M compiler-verified proofs, and serialized trajectories with retrieved context, failed attempts, feedback, and repairs. Across five benchmarks, OProver-32B attains the best Pass@32 on MiniF2F (93.3%), ProverBench (58.2%), and PutnamBench (11.3%), and ranks second on MathOlympiad (22.8%) and ProofNet (33.2%). OProver models, OProofs, and code are available at `https://anonymous.4open.science/r/OProver-oss-05B1`.

1 Introduction

Formal theorem proving in systems such as Lean 4 (de Moura and Ullrich, 2021) provides a rigorous setting for machine reasoning, where every step of a proof is mechanically verified by a small, trusted kernel. This makes them a natural foundation for reliable mathematical reasoning and verified software, but also imposes a high bar: a proof is accepted only if it is fully formal and type-correct. Recent prover systems built on Lean have substantially improved performance on challenging benchmarks such as MiniF2F (Zheng et al., 2021) and PutnamBench (Tsoukalas et al., 2024), yet absolute success rates on harder benchmarks remain low, and most systems still rely primarily on single-pass or best-of-N whole-proof generation. Retrieval and compiler feedback, when used at all, are typically applied as test-time heuristics rather than as a proving policy that the prover is trained to use, leaving an important source of supervision largely unexploited.

A small but growing line of work has begun to incorporate retrieval, compiler feedback, and iterative repair into formal theorem proving (Yang et al., 2023; Jiang et al., 2022; Wu et al., 2025). However, these capabilities are typically introduced as inference-time augmentations on top of a fixed prover, rather than as a learned policy that the model is trained to use. As a result, a prover trained mainly on finalized proofs sees compiler feedback and retrieved evidence only at deployment, in distributions it was never optimized for. Closing this train–inference mismatch requires training the prover to perform retrieval-grounded, feedback-conditioned refinement as part of its policy, not as a separate inference-time procedure.

Training such a policy in turn requires data that existing formal corpora do not provide. Public formal theorem proving corpora and proof-synthesis datasets (Wu et al., 2022; Ying et al., 2024; Peng et al., 2025) focus on the end products of proving: large collections of formal statements and final compiler-verified proofs. In general, they omit failed attempts, retrieved context, and compiler diagnostics that drive proof repair in practice. Autoformalization and proof synthesis have greatly expanded the number of available statements and verified proofs, but they do not by themselves record the multi-round interaction histories needed to learn agentic self-correction. What is missing is therefore not more verified proofs, but corpora that explicitly preserve how proofs are constructed, fail, and get repaired.

In this paper, we present **OProver**, a unified

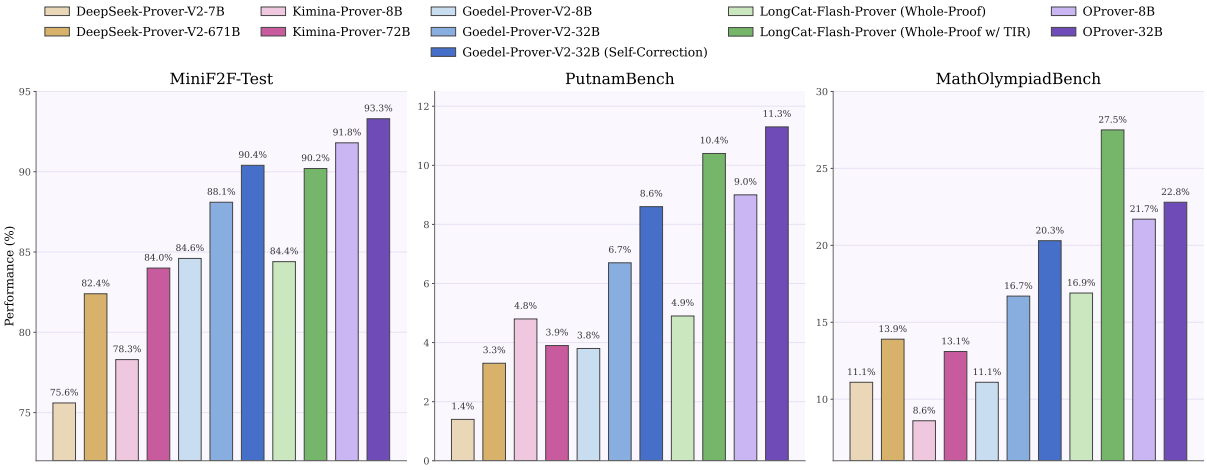


Figure 1: Pass@32 performance on MiniF2F-Test, PutnamBench, and MathOlympiadBench. OProver consistently outperforms prior open-source whole-proof prover baselines across all three benchmarks; the only exception is LongCat-Flash-Prover with TIR on MathOlympiadBench.

framework for agentic formal theorem proving in Lean 4. Rather than treating retrieval, compiler feedback, and iterative repair as separate inference-time modules on top of a fixed prover, OProver unifies them with training and data construction into a single proving framework. At inference time, proving is a multi-round refinement loop in which each failed attempt is revised using retrieved compiler-verified proofs and Lean 4 compiler feedback. At training time, the same retrieval and feedback signals shape the prover's policy: continued pretraining on Lean code and mathematics is followed by iterative post-training that alternates agentic proving, SFT, and RL. Newly verified proofs and repair traces from the current prover are recirculated back into OProofs and the retrieval memory, so data, training, and proving policy co-evolve within a single framework.

We pair OProver with **OProofs**, a large-scale Lean 4 corpus that records not only final compiler-verified proofs but also the trajectories of proof construction, failure, feedback, and repair. It contains 1.76M Lean statements and 6.80M compiler-verified proofs from three complementary sources: public Lean resources, large-scale proof synthesis with compiler verification, and traces from OProver's own agentic proving. The first two sources support pretraining and the initial post-training rounds, while the third grows continuously as the prover improves: newly verified proofs are indexed into retrieval memory and repair trajectories are added to subsequent training rounds.

In summary, our contributions are:

1. **A unified framework for agentic formal theorem proving.** We propose OProver, which treats proving as a retrieval-grounded, feedback-conditioned refinement loop and trains the prover end-to-end to use these signals as part of its policy, rather than as test-time heuristics.

2. **A large-scale corpus for agentic formal theorem proving.** We construct OProofs, containing 1.76M Lean statements, 6.80M compiler-verified proofs, and serialized proving trajectories that record retrieved context, failed attempts, compiler feedback, and subsequent repairs—supervision that prior Lean corpora do not provide.

3. **A co-evolution pipeline between prover and corpus.** We develop an iterative post-training pipeline in which the current prover produces new verified proofs and repair trajectories: verified proofs are indexed into the retrieval memory, repair trajectories become SFT data, and the hardest unresolved cases provide RL signal for the next round.

4. **State-of-the-art performance among open-weight whole-proof provers.** OProver-32B attains three best and two second-best results across five formal theorem-proving benchmarks, reaching Pass@32 of 93.3 on MiniF2F, 58.2 on ProverBench, and 11.3 on PutnamBench.

2 Related Work

Proof generation and search. Neural theorem proving has evolved from tactic-level interaction

and neural-guided search (Polu and Sutskever, 2020; Han et al., 2021; Lample et al., 2022; Yang et al., 2023) toward whole-proof generation. Recent Lean provers combine synthetic proof data with proof-assistant feedback, including DeepSeek-Prover-V1.5/V2 (Xin et al., 2024; Ren et al., 2025), Goedel-Prover-V1/V2 (Lin et al., 2025a,b), Kimina-Prover (Wang et al., 2025), BFS-Prover (Xin et al., 2025), InternLM2.5-StepProver (Wu et al., 2025), Bourbaki (Zimmer et al., 2025), and Seed-Prover (Chen et al., 2025).

Verifier-guided refinement and agentic proving. Treating proof generation as a single-pass problem discards rich semantic feedback from the proof assistant. Baldur (First et al., 2023) and Lean-STaR (Lin et al., 2024) repair or interleave reasoning using verifier signals, while recent systems make this loop explicitly agentic: Leanabell-Prover-V2 (Ji et al., 2025) and StepFun-Prover (Shang et al., 2025) train verifier-integrated reasoning with RL; Prover Agent (Baba et al., 2025) coordinates informal reasoning, formal proving, and lemma generation; Gupta et al. (2025) learn multi-turn repair from interactive trajectories; and LongCat-Flash-Prover (Wang et al., 2026) scales agentic tool-integrated RL.

Formal data synthesis and process supervision. Scaling formal provers depends on synthetic theorem–proof construction (Ying et al., 2024; Tao and Xu, 2026; Lin et al., 2025b) and on supervision beyond final proofs: informal sketches (Jiang et al., 2022), benchmark-style alignment (Azerbayev et al., 2023), critic-guided training (Peng et al., 2025), tool-feedback autoformalization (Guo et al., 2025), and self-improving proving (Dong and Ma, 2025). Our work unifies these threads: OProofs extends formal corpora with *trajectory-level* supervision (retrieved context, failed attempts, compiler feedback, repairs), and OProver couples corpus growth with policy improvement by recirculating verified proofs and repair traces. Unlike prior self-improving systems that iterate over statement–proof pairs (Dong and Ma, 2025; Chen et al., 2025), OProver learns retrieval-grounded, feedback-conditioned revision as a policy rather than an inference-time wrapper.

3 Methodology

Figure 2 illustrates the three components of our framework. The **OProofs Construction** pipeline (top-left) builds a Lean-specific corpus from public Lean resources and autoformalized statements through deduplication, filtering, and Lean 4 verification (§3.2). The **OProver Agentic Proving** pipeline (top-right) performs multi-round refinement under retrieved compiler-verified proofs and Lean 4 compiler feedback (§3.1). The **OProver Agentic Training** pipeline (bottom) trains OProver in two stages: a one-time continued pretraining yields OProver-Base, followed by an iterative post-training loop in which agentic rollouts produce SFT and RL data while verified proofs are folded back into OProofs (§3.3).

3.1 The OProver Framework

OProver consists of a proving policy π, a retrieval memory $\mathcal{M}$ of compiler-verified proofs, and the Lean 4 compiler $\mathcal{V}$ as the verification environment.

Agentic proving formulation. We formulate theorem proving as a bounded multi-round refinement process. At round t, the policy conditions on a state

$$X_t = (s,\ \mathcal{R}_t,\ p_{t-1},\ f_{t-1}),$$

where s is the target statement, $\mathcal{R}_t$ is the retrieved proof context, p_{t-1} is the previous proof attempt, and f_{t-1} the corresponding compiler feedback. The policy then produces $p_t \sim \pi(\cdot \mid X_t)$. At $t = 0$, p_0 and f_0 are empty. The interaction terminates when $\mathcal{V}$ verifies a proof or when a round budget T is exhausted. The policy conditions only on the most recent (p_{t-1}, f_{t-1}) rather than the full history, and the same state formulation is used throughout rollout, SFT, and RL, so that training and proving share an identical interface (§3.3).

Retrieval and feedback. At each round, OProver retrieves the top-k proofs from $\mathcal{M}$ by semantic similarity to s, using a large text embedding model. $\mathcal{M}$ is initialized from OProofs and continuously expanded during iterative post-training. When a proof attempt fails, $\mathcal{V}$ returns textual diagnostics (syntax errors, type mismatches, tactic failures, unsolved goals), which we pass directly to the policy as f_{t-1} without projecting them into a hand-designed error taxonomy; preserving the raw textual form enables fine-grained, targeted revisions across rounds.

3.2 OProofs: A Unified Training Corpus

OProofs serves two roles: it provides training data across CPT, SFT, and RL, and it supplies the proofs that populate the retrieval memory $\mathcal{M}$ (§3.1). In

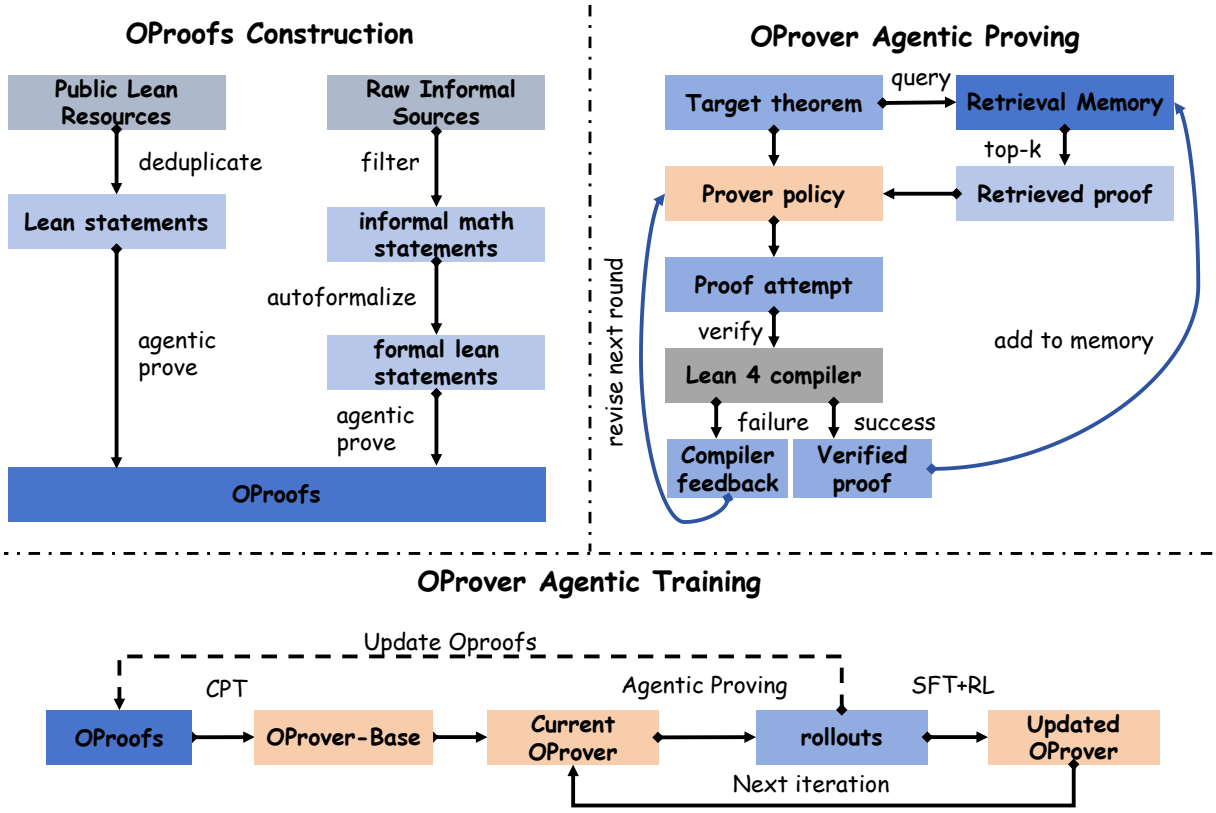


Figure 2: Overview of OProver. OProofs Construction (top-left), Agentic Proving (top-right), and Iterative Agentic Training (bottom) co-evolve: each iteration adds verified proofs to OProofs and the retrieval memory, and trains OProver_{t+1} from OProver_t.

total, OProofs contains 1.76M Lean statements and 6.80M compiler-verified proofs, together with serialized proving trajectories that record retrieved context, failed attempts, compiler feedback, and subsequent repairs (full statistics in §4).

Construction pipeline. Figure 2 (top-left) shows the two-branch construction pipeline, both ending in Lean 4 verification. *Branch 1* collects open-source Lean statements (NuminaMath-LEAN, Lean-Workbook, Leanabell-FormalStmt, Goedel-Pset, etc.), deduplicates them at the theorem-signature level, and runs agentic proving with open-source provers, retaining only Lean-verified proofs. *Branch 2* mines informal mathematical statements from Common Crawl and GitHub code repositories using a FastText classifier, autoformalizes them into Lean 4 with CriticLean (Peng et al., 2025), and runs the same agentic proving process.

Source attribution. After deduplication and verification filtering, OProofs retains 1.76M unique Lean statements: 78.2% from six public Lean corpora and 21.8% from autoformalized informal mathematics. Per-source counts are reported in Appendix B

Round-level repair examples. Each rollout is decomposed into round-level instances of the form $(s, \mathcal{R}_t, p_{t-1}, f_{t-1}) \rightarrow p_t$, matching the state formulation in §3.1 exactly. Each example matches the proving-time format, supervising only successful repairs with no train–inference mismatch.

3.3 Training OProver with OProofs

Training proceeds in two phases: a one-time continued pretraining on OProofs produces OProver-Base; iterative post-training then alternates agentic proving rollouts with SFT and RL, expanding OProofs as the prover improves (Algorithm 1).

Continued pretraining. We perform a one-time CPT on a 65B-token mixture: ∼30% Lean-related formal data from OProofs (§3.2), 20% code from OpenCoder (Huang et al., 2025), 40% mathematical data from Nemotron-Math-4-Plus (Mahabadi et al., 2025), and 10% long-CoT data from ProLong-64K (Gao et al., 2025). The resulting *OProver-Base* is the starting point of iterative post-training.

Algorithm 1 Iterative co-evolution of OProver and OProofs

Require: CPT-initialized policy π_0; initial corpus $\mathcal{D}_0$; theorem pool $\mathcal{Q}$; verifier $\mathcal{V}$; iterations K

$\mathcal{M}_0 \leftarrow \mathrm{Index}(\mathcal{D}_0)$
for $k = 0, 1, \ldots, K-1$ **do**
  Run multi-round agentic rollouts of π_k on $\mathcal{Q}$, retrieving from $\mathcal{M}_k$ and verifying with $\mathcal{V}$
  Collect verified pairs $\mathcal{P}_k^+$ and round-level repair examples $\mathcal{B}_k$; select hard cases $\mathcal{H}_k$
  $\mathcal{D}_{k+1} \leftarrow \mathcal{D}_k \cup \mathcal{P}_k^+$; $\mathcal{M}_{k+1} \leftarrow \mathrm{Index}(\mathcal{D}_{k+1})$
  $\pi_{k+1} \leftarrow \mathrm{SFT}(\pi_k; \mathcal{B}_k)$ followed by $\mathrm{GSPO}(\,\cdot\,; \mathcal{H}_k, \mathcal{M}_{k+1})$
end for
return $\pi_K, \mathcal{D}_K$

Iterative post-training. At each iteration, the current model performs agentic proving on theorems it does not yet reliably solve. From these rollouts we extract round-level repair examples $(s, \mathcal{R}_t, p_{t-1}, f_{t-1}) \rightarrow p_t$, used as SFT data with cross-entropy loss on p_t only; newly verified proofs are simultaneously added to OProofs and indexed into $\mathcal{M}$. We then refine the SFT policy with Group Sequence Policy Optimization (GSPO) (Zheng et al., 2025). For each theorem we sample n rollouts of up to R rounds and assign a *per-round* reward

$$r_t = \left(0.8 + 0.2\,\mathbf{1}_{\mathrm{fmt}(p_t)}\right) \cdot \mathbf{1}[\mathcal{V}(p_t) = \text{verified}].$$

All $n \times R$ rounds for the same theorem are pooled into one group for group-relative advantage normalization, so the policy learns from contrasts both across independent attempts and across successive refinement rounds. RL uses the same state X_t and retrieval memory as proving time, so the policy is optimized end-to-end under the deployment interface. Values of n, R, and other hyperparameters are reported in §E.3.

4 Data Statistics

OProofs is a large-scale Lean 4 corpus constructed from both public formal resources and model-generated data. Public resources contribute formal statements and theorem–proof pairs, while our proof synthesis and agentic proving pipelines further expand the corpus with additional compiler-verified proofs and serialized proving trajectories. These trajectories are further decomposed into round-level repair examples for training. As a result, OProofs supports not only compiler-verified proof supervision, but also retrieval-grounded and feedback-conditioned agentic theorem proving.

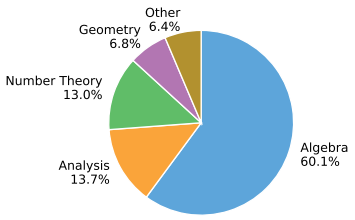
Figure 3(a) summarizes the current Lean 4 component of OProofs. The present version contains approximately 1.76M unique Lean statements paired with 6.80M compiler-verified proofs, together with 1.06M agentic proving trajectories. Beyond raw verified proofs, OProofs explicitly preserves the supervision signals required for retrieval-grounded and feedback-conditioned proof revision: 4.29M proofs carry retrieved proof context, 859k carry non-trivial compiler feedback from prior failed attempts. The frozen Lean corpus snapshot used in our current training pipeline comprises 51.4B tokens (prompt and chain-of-thought combined, tokenized with the Qwen3 tokenizer).

The corpus spans a broad spectrum of mathematics (Figure 3(b)): Algebra (60.1%), Analysis (13.7%), Number Theory (13.0%), and Geometry (6.8%) dominate, with the remaining 6.4% grouped as "Other" (covering Combinatorics, Logic, Probability, Topology, and Computation). Difficulty levels (Figure 3(c)) range from Elementary arithmetic (27.4%) through High School (48.9%) and Undergraduate (18.9%) to Graduate-level problems (4.8%); domain and difficulty are labeled per statement by an LLM classifier (see Appendix G for the prompt and validation protocol). During iterative training, the underlying OProofs corpus continues to expand through newly verified proofs and repair data collected from the evolving prover. We emphasize that the statistics reported in Figure 3 cover only the Lean 4 theorem-proving component of OProofs, which is also the component we primarily analyze and release in this work. Auxiliary code, mathematical reasoning, and long-context reasoning corpora used during continued pretraining are excluded from these counts.

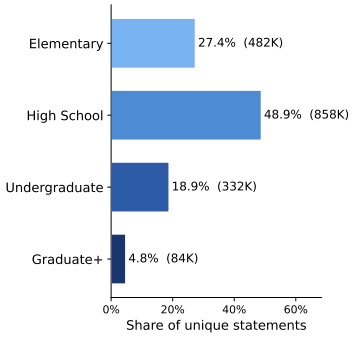
Table 1 compares OProofs with representative open Lean resources along the supervision dimensions relevant to agentic formal theorem proving. Most public corpora expose only a subset of these signals; the closest, Nemotron-Math-Proofs, provides formal statements, verified proofs, natural-language rationales, and compiler feedback, but lacks retrieved proof context and multi-round repair supervision. OProofs closes this gap by jointly providing all of the above within a unified corpus for retrieval-grounded, feedback-conditioned proof

Metric	Value
Unique Lean statements	1.76M
Compiler-verified proofs	6.80M
with retrieval context	4.29M
with compiler feedback	859K
Agentic proving trajectories	1.06M
Lean corpus tokens	51.4B

(a) Corpus overview.



(b) Mathematical domain.



(c) Difficulty level.

Figure 3: OProofs Lean 4 corpus statistics. (a) Headline counts of statements, proofs, retrieval/feedback signals, agentic trajectories, and training tokens. (b) Mathematical domain distribution. (c) Difficulty level distribution.

Table 1: Comparison of OProofs with representative open Lean resources.

Resource	Formal stmt.	Verified proof	NL rationale / CoT	Compiler feedback	Retrieved proof	Multi-round repair
NuminaMath-LEAN	✓	✗	✗	✗	✗	✗
Leanabell-SFT	✓	✓	✓	✗	✗	✗
Lean-Workbook	✓	✗	✗	✗	✗	✗
Leanabell-FormalStmt	✓	✗	✗	✗	✗	✗
Workbook-proofs	✓	✓	✗	✗	✗	✗
Goedel-Pset	✓	✗	✗	✗	✗	✗
FineLeanCorpus	✓	✗	✗	✗	✗	✗
FormalMATH-All	✓	✗	✗	✓	✗	✗
Nemotron-Math-Proofs	✓	✓	✓	✓	✗	✗
OProofs	✓	✓	✓	✓	✓	✓

revision. To prevent test-set leakage, we apply exact-match and identifier-based decontamination between OProofs and the five evaluation benchmarks; details are in Appendix B.1.

5 Experiments

Setup. We evaluate OProver on five Lean 4 benchmarks of varying difficulty: MiniF2F (Zheng et al., 2021), MathOlympiadBench (Lin et al., 2025b), ProofNet (Azerbayev et al., 2023), ProverBench (Ren et al., 2025), and PutnamBench (Tsoukalas et al., 2024), the same suite used by recent open-source provers. We compare against open-weight reasoning models (DeepSeek-V3.2 (Liu et al., 2025), Kimi-K2.5 (Team et al., 2026)) and open-weight provers (Kimina-Prover (Wang et al., 2025), DeepSeek-Prover-V2 (Ren et al., 2025), Leanabell-Prover-V2 (Ji et al., 2025), Goedel-Prover-V2 (Lin et al., 2025b), and LongCat-Flash-Prover (Wang et al., 2026)); † marks numbers not re-evaluated by us.

Evaluation protocol. We report Pass@k using the standard unbiased estimator $1 - \binom{n-m}{k}/\binom{n}{k}$, where n is the number of independent samples and m is the number of successes. Unless noted, we report Pass@32 with $n = 64$. For OProver, one sample is a complete multi-round rollout (up to R rounds) and counts as successful if any of its attempts is Lean-verified. §5.2 provides a budget-controlled comparison. For models we re-evaluated, we report seed-level standard errors from 1,000 random $k = 32$ subsamples of the n chains per problem (following Lin et al. (2025b)).

5.1 Main Results

Table 2 shows that OProver-32B attains the best Pass@32 on MiniF2F (93.3), ProverBench (58.2), and PutnamBench (11.3), and the second-best on MathOlympiad (22.8) and ProofNet (33.2), behind only LongCat-Flash-Prover with whole-proof TIR. This gives OProver-32B more top placements (three best, two second-best) than any other model in the table. Even OProver-8B outperforms Goedel-Prover-V2-32B on all five benchmarks despite having four times fewer parameters.

These gains are notable: OProver-32B is a 32B dense model, whereas LongCat-Flash-Prover is a 560B MoE with 27B active parameters (17× more

Table 2: Theorem-proving performance (Pass@32, %). Best in bold, second-best underlined; † denotes scores taken from prior work. We report mean ± seed-level SE from 1000 k=32 subsamples of n=64 chains per problem.

Model	#Params	MathOlympiad (Pass@32)	MiniF2F (Pass@32)	ProofNet (Pass@32)	ProverBench (Pass@32)	PutnamBench (Pass@32)
Open-Weights Reasoning Models						
DeepSeek-V3.2	671B	14.7†	77.9†	20.4†	42.8†	5.8†
Kimi-K2.5	1T	7.5†	76.6†	19.9†	44.3†	1.2†
Open-Weights Prover Models						
Kimina-Prover-8B	8B	7.6±0.4	77.2±0.5	13.0±0.6	40.5±0.6	2.3±0.2
Kimina-Prover-72B	72B	11.5±0.5	83.4±0.5	16.3±0.6	45.0±0.5	4.0±0.2
DeepSeek-Prover-V2-7B	7B	7.9±0.3	73.9±0.6	18.1±0.6	45.7±0.5	1.5±0.1
DeepSeek-Prover-V2-671B	671B	13.9†	82.4†	30.5†	52.9†	3.3†
Leanabell-Prover-V2-KM	8B	–	68.4†	13.4†	39.8†	–
Leanabell-Prover-V2-DS	7B	–	76.6†	23.7†	47.8†	–
Goedel-Prover-V2-8B	8B	9.9±0.5	79.4±0.7	17.3±0.7	48.6±0.5	0.7±0.1
w/ self-correction	8B	–	86.7†	–	–	–
Goedel-Prover-V2-32B	32B	16.0±0.5	85.8±0.5	22.0±0.9	51.0±0.6	5.0±0.3
w/ self-correction	32B	20.3†	90.4†	–	–	8.6†
LongCat-Flash-Prover	560B	16.9†	84.4†	19.9†	49.9†	4.9†
whole-proof w/ TIR	560B	**27.5**†	90.2†	**36.1**†	57.9†	10.4†
Ours						
OProver-8B	8B	21.7±0.4	91.8±0.4	31.9±0.4	56.0±0.3	9.0±0.3
OProver-32B	32B	22.8±0.3	**93.3±0.4**	33.2±0.2	**58.2±0.3**	**11.3±0.4**

total), and DeepSeek-Prover-V2 (671B) is outperformed on all five benchmarks; relative to Goedel-Prover-V2-32B, OProver-32B improves by 6.3 to 11.2 points per benchmark. The gains span benchmarks of varying difficulty—from MiniF2F and ProverBench to PutnamBench (sub-12% absolute success for all methods)—indicating that improvements reflect a broadly effective proving strategy rather than specialization, driven by retrieval, compiler feedback, and iterative refinement rather than scale.

5.2 Test-Time Scaling

We study how OProver benefits from additional test-time compute. Given a budget B allocated as $R \cdot k = B$ refinement rounds and samples per round, we report

$$\text{BestPass}(B) = \max_{R \cdot k = B} \text{Pass}(R, k),$$

the best success rate achievable across all (R, k).

Figure 4 shows that both model sizes scale consistently on all five benchmarks. From $B = 8$ to $B = 256$, OProver-32B improves by 5.3 points on MiniF2F (87.5 $\rightarrow$ 92.8) and by 4.9 on PutnamBench (6.4 $\rightarrow$ 11.3), with similar trends for OProver-8B. Gains exhibit clear diminishing returns—on ProofNet, doubling B from 8 to 16 adds 2.1 points but the same doubling from 128 to 256 adds only 0.9—with saturation arriving earlier on easier benchmarks and later on PutnamBench, where the low baseline leaves more headroom. Notably, at the same inference budget of B=32, OProver-8B/32B outperform Goedel-Prover-V2-8B/32B respectively on all five Pass@32 numbers, indicating the gains come from the prover internalizing retrieval and iterative refinement rather than from a larger inference budget. A full $(R, B/R)$ analysis and within-rollout refinement curves are in Appendix C.

5.3 Effectiveness of Iterative Post-Training

Figure 5 reports Pass@32 on MiniF2F-Test for every intermediate checkpoint of the co-evolution loop (§3.3). OProver-8B improves from 79.5 (Base) to 86.2, 87.0, and 91.8 after Rounds 1, 2, and 3 (+12.3 total), and OProver-32B improves from 84.7 to 88.1 and 93.3 after Rounds 1 and 2 (+8.6 total). Both models improve monotonically across iterations, with no sign of saturation after the first post-training round. The per-iteration gains shrink gradually—OProver-8B adds 6.7, 0.8, and 4.8 points across Rounds 1–3, while OProver-32B adds 3.4 and 5.2 points across Rounds 1–2—suggesting that recirculating verified proofs and repair trajectories into OProofs and the retrieval memory continues to provide useful supervision even after the prover reaches a strong baseline.

5.4 Ablation Studies

We ablate two core components of OProver. *w/o feedback* (denoted "-FB" in Table 3) keeps only

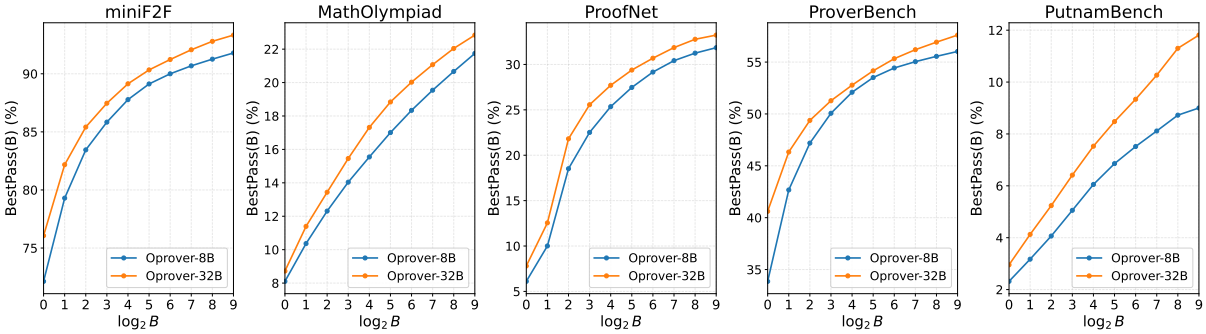


Figure 4: Test-time scaling of OProver under a fixed total budget B. Both OProver-8B and OProver-32B improve consistently with larger budgets across all five benchmarks, with clear diminishing returns at the high end.

Table 3: Ablation on multi-turn compiler feedback and retrieval. "-FB" keeps only the first interaction round (retrieval retained); "-FB, -RAG" additionally removes retrieved proofs. Mean $\pm$ seed-level SE from 1000 k=32 subsamples of n=64 chains per problem.

Model	Variant	MathOlympiad	MiniF2F	ProofNet	ProverBench	PutnamBench
OProver-8B	Full	**21.7±0.4**	**91.8±0.4**	**31.9±0.4**	**56.0±0.3**	**9.0±0.3**
OProver-8B	-FB	15.8±0.4	87.7±0.4	22.4±0.8	50.6±0.5	5.7±0.3
OProver-8B	-FB, -RAG	12.4±0.3	86.2±0.5	22.4±0.8	49.9±0.5	4.1±0.2
OProver-32B	Full	**22.8±0.3**	**93.3±0.4**	**33.2±0.2**	**58.2±0.3**	**11.3±0.4**
OProver-32B	-FB	16.5±0.4	88.4±0.5	25.8±0.6	52.0±0.4	7.0±0.3
OProver-32B	-FB, -RAG	14.8±0.5	87.9±0.5	24.7±0.7	51.1±0.5	5.9±0.3

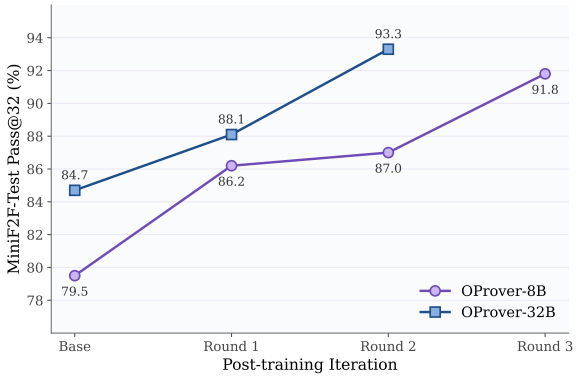


Figure 5: MiniF2F-Test Pass@32 across post-training iterations. Both OProver-8B and OProver-32B improve monotonically.

the first interaction round (i.e., R=1 with k=32, retrieval retained); *w/o feedback, w/o RAG* (denoted "-FB, -RAG") additionally removes retrieved proofs.

Table 3 shows both components contribute, but with markedly different magnitudes. Removing compiler feedback causes the largest and most consistent degradation: for OProver-32B, Pass@32 drops by 5.0–7.4 points per benchmark (e.g., MathOlympiad 22.8 $\rightarrow$ 16.5, PutnamBench 11.3 $\rightarrow$ 7.0), confirming that iterative refinement is the primary driver of OProver's gains. Further removing RAG yields a smaller but consistent additional drop (0.5–1.7 points). Feedback enables iterative repair, while retrieval grounds proof attempts; their combination yields the strongest gains. Appendix F provides real OProver-32B trajectories showing how retrieved references and Lean feedback turn failed first-round attempts into verified repairs.

6 Conclusion

We present OProver, a unified framework for agentic formal theorem proving in Lean 4 in which a trained policy iteratively revises failed proof attempts using retrieved compiler-verified proofs and Lean 4 compiler feedback. We construct OProofs, a large-scale Lean 4 corpus of 1.76M statements, 6.80M compiler-verified proofs, and serialized proving trajectories, used both for training and as the retrieval memory at proving time. Through continued pretraining and iterative SFT+RL post-training that recirculates verified proofs and repair trajectories, OProver-32B attains more top placements on five Lean 4 benchmarks than any prior open-weight whole-proof prover. By releasing the models, code, and OProofs corpus, we aim to support future research on open, reproducible, and feedback-aware formal theorem proving.

Limitations

OProofs preserves multi-round interaction histories but does not capture tactic-level intermediate states (e.g., goal stacks); extending to tactic-level supervision is left as future work. Our evaluation focuses on five Lean 4 mathematics benchmarks; generalization to software verification or non-Mathlib libraries is not validated. Iterative post-training is compute-intensive: each iteration requires tens of thousands of multi-round rollouts plus Lean verification, which may limit reproducibility without significant compute. Finally, the autoformalization branch relies on existing tools (CriticLean and open-source Lean provers); errors in these tools may propagate into OProofs despite Lean 4 verification and our re-verification stability check.

Ethics Statement

OProver is trained on publicly available Lean 4 corpora and synthetic proofs from automated provers; the data consists primarily of formal mathematical statements, Lean proofs, compiler feedback, and mathematical metadata. We apply format-based filtering to retain Lean-formalized content and remove non-Lean free-form text except for structured prompts and verifier feedback fields used for training, and manually inspected corpus samples without observing personally identifying or offensive content. Any such content discovered during release preparation will be removed; because the corpus does not contain user records, conversations, or human-subject annotations, no anonymization of personal records was required.

Public Lean corpora are unevenly distributed across mathematical domains, which may bias the prover toward overrepresented topics; we report per-benchmark results to make this explicit. The pipeline relies on automated proof generation and Lean 4 verification. The main misuse risk is overtrusting unverified model outputs; all reported results therefore require Lean verification, and released inference code will keep verifier checking in the loop.

Use of AI Writing Assistance. Following ACL/EMNLP policy, we used AI writing assistants for grammar checking and phrasing refinement. All technical content, experimental design, and analysis are the work of the authors. No citations are hallucinated; all reported numbers come from actual experiments.

Reproducibility and Open Science. We will release OProver-8B and OProver-32B weights, the OProofs corpus, training code for CPT, SFT, and GSPO RL, agentic-proving inference code, and evaluation scripts for all five benchmarks. The Lean toolchain configuration (Lean 4.15.0, Mathlib commit, Kimina Lean server version, container image), heartbeat caps, verification timeouts, and all training and evaluation hyperparameters required to reproduce our results are detailed in Appendix E. Each newly created artifact will be released with an explicit license and terms of use covering permitted use, redistribution, and derivative works. Third-party artifacts used in this work, including Lean, Mathlib, REPL/Kimina Lean server, public Lean corpora, released prover checkpoints, and evaluation benchmarks, retain their original licenses and terms of use; we will document these licenses and terms in the release.

References

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*.

Kaito Baba, Chaoran Liu, Shuhei Kurita, and Akiyoshi Sannai. 2025. Prover agent: An agent-based framework for formal mathematical proofs. *arXiv preprint arXiv:2506.19923*.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, and 1 others. 2025. Seed-prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pages 625–635, Cham. Springer International Publishing.

Kefan Dong and Tengyu Ma. 2025. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*.

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241.

Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. 2025. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pages 7376–7399.

Qi Guo, Jianing Wang, Jianfei Zhang, Deyang Kong, Xiangzhou Huang, Xiangyu Xi, Wei Wang, Jingang Wang, Xunliang Cai, Shikun Zhang, and 1 others. 2025. Autoformalizer with tool feedback. *arXiv preprint arXiv:2510.06857*.

Kanan Gupta, Jannis Limperg, and Udaya Ghai. 2025. Tool-assisted multi-turn theorem proving with llms. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. 2021. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, and 1 others. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193.

Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. 2025. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. *arXiv preprint arXiv:2507.08649*.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2022. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*.

Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349.

Haohan Lin, Zhiqing Sun, Sean Welleck, and Yiming Yang. 2024. Lean-star: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and 1 others. 2025a. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, and 1 others. 2025b. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Rabeeh Karimi Mahabadi, Sanjeev Satheesh, Shrimai Prabhumoye, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. 2025. Nemotron-cc-math: A 133 billion-token-scale high quality math pretraining dataset. *arXiv preprint arXiv:2508.15096*.

Zhongyuan Peng, Yifan Yao, Kaijing Ma, Shuyue Guo, Yizhe Li, Yichi Zhang, Chenchen Zhang, Yifan Zhang, Zhouliang Yu, Luming Li, and 1 others. 2025. Criticlean: Critic-guided reinforcement learning for mathematical formalization. *arXiv preprint arXiv:2507.06181*.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*.

ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, and 1 others. 2025. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*.

Shijie Shang, Ruosi Wan, Yue Peng, Yutong Wu, Xiong-hui Chen, Jie Yan, and Xiangyu Zhang. 2025. Stepfun-prover preview: Let's think and verify step by step. *arXiv preprint arXiv:2507.20199*.

Yicheng Tao and Hongteng Xu. 2026. Theoremforge: Scaling up formal data synthesis with low-budget agentic workflow. *arXiv preprint arXiv:2601.17332*.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, and 1 others. 2026. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. 2024. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. *CoRR*, abs/2407.11214.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, and 1 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Jianing Wang, Jianfei Zhang, Qi Guo, Linsen Guo, Rumei Li, Chao Zhang, Chong Peng, Cunguang Wang, Dengchang Zhao, Jiarong Shi, and 1 others. 2026. Longcat-flash-prover: Advancing native formal reasoning via agentic tool-integrated reinforcement learning. *arXiv preprint arXiv:2603.21065*.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Zheng Yuan, Wenwei Zhang, Dahua Lin, and Kai Chen. 2025. Internlm2. 5-stepprover: Advancing automated theorem proving via critic-guided search. In *2nd AI for Math Workshop@ ICML 2025*.

Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, and 1 others. 2024. Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 2025. Bfs-prover: Scalable best-first tree search for llm-based automatic theorem proving. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32588–32599.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612.

Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Lean workbook: A large-scale lean problem set formalized from natural language math problems. *Preprint*, arXiv:2406.03847.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, and 1 others. 2025. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*.

Matthieu Zimmer, Xiaotong Ji, Rasul Tutunov, Anthony Bordg, Jun Wang, and Haitham Bou Ammar. 2025. Bourbaki: Self-generated and goal-conditioned mdps for theorem proving. *arXiv preprint arXiv:2507.02726*.

Table 4: Retriever comparison on 1,000 queries with 9,000 candidates. Relevance is assessed by GPT-4o pairwise judgment over top-5 retrieved contexts.

Retriever	Sim. spread	Judge pref.
Goedel-Prover-V2-8B	$[0.85, 0.99]$	333 / 1000
Qwen3-8B-Embedding	$[0.20, 0.99]$	667 / 1000

A Additional Method Details

This appendix provides additional details on retriever selection, repair instance construction, data recirculation, training and evaluation hyperparameters, and pretraining data recall.

A.1 Retriever Selection

Section 3.1 describes how OProver retrieves top-k compiler-verified proofs from a memory $\mathcal{M}$ at each proving round. This appendix details the retriever model selection.

We compared two candidate retrievers: Qwen3-8B-Embedding and Goedel-Prover-V2-8B. The comparison was conducted on a held-out retrieval set of 1,000 query statements and 9,000 candidate retrieval documents.

Similarity score distribution. Qwen3-8B-Embedding produced similarity scores spanning approximately $[0.20, 0.99]$, whereas Goedel-Prover-V2-8B produced scores concentrated in a narrow high-similarity range $[0.85, 0.99]$. The broader spread of Qwen3-8B-Embedding provides a more discriminative ranking signal: when score variance is small across candidates, top-k selection becomes effectively random, whereas a broader spread aligns top-k selection with semantic relevance.

Pairwise LLM-as-a-judge evaluation. We further evaluated retrieval quality using GPT-4o in a pairwise LLM-as-a-judge setup. For each query, GPT-4o compared the top-5 retrieved theorem contexts returned by the two retrievers. Over 1,000 query-level comparisons, Qwen3-8B-Embedding was preferred in 667 cases (66.7%), while Goedel-Prover-V2-8B was preferred in 333 cases (Table 4). Based on these results, OProver uses Qwen3-8B-Embedding as the default retriever.

A.2 Repair Instance Construction Details

Section 3.2 defines a round-level repair instance as $(s, \mathcal{R}, p^-, f^-, p^+)$. We describe here how such instances are extracted from raw proving trajectories.

Extraction. Given an agentic proving trajectory of T rounds $(p_1, f_1, p_2, f_2, \ldots, p_T)$, each transition $(p_{t-1}, f_{t-1}) \rightarrow p_t$ for $t \geq 2$ becomes one repair instance, paired with the retrieval context $\mathcal{R}_t$ used at round t. A trajectory of T rounds therefore yields up to $T-1$ repair instances.

Filtering. We discard instances in which (i) p_{t-1} is empty or syntactically malformed before verification, (ii) f_{t-1} exceeds 8,000 tokens (these are dominated by repeated boilerplate errors), or (iii) p_t differs from p_{t-1} by fewer than 3 tokens (near-no-op revisions). The retained repair instances form the primary supervision substrate for feedback-aware post-training.

Deduplication. Repair instances are deduplicated by exact match on (p^-, f^-, p^+) to avoid multiple trajectories producing identical local transitions.

A.3 Data Recirculation Details

Section 3.3 describes how each post-training iteration recirculates the prover's rollouts into OProofs, the retrieval memory, the SFT repair set, and the RL hard-case set. We provide additional implementation details below.

Routing rules. After each iteration, each rollout is routed according to its outcome and the success rate of its theorem's group (the n rollouts sampled for the same theorem):

- Rollouts that yield a Lean-verified proof contribute their final proof to $\mathcal{P}_t^+$, which is appended to OProofs $\mathcal{D}_{t+1}$ and re-indexed into the retrieval memory $\mathcal{M}_{t+1}$.
- Repair instances extracted from successful rollouts are added to the SFT set $\mathcal{B}_t$.
- Theorems whose group success rate is in $(0, 1)$—that is, not solved by all rollouts and not failed by all rollouts—are retained as hard cases $\mathcal{H}_t$ for the next round of RL.

The routing follows DAPO-style group filtering, which empirically reduces gradient variance by excluding both fully-solved and fully-failed groups.

A.4 Pretraining Data Recall

We recall mathematics-adjacent examples from Common Crawl and GitHub through an iterative fastText classifier seeded with verified proofs and repair instances from OProofs. At each round, the

Table 5: Per-source statement attribution in OProofs. OProofs retains only statements with at least one Lean-verified proof; statements where no proof attempt verified are excluded.

Source	Statements	Share
Branch 1: Public Lean resources		
Leanabell-Prover-Formal-Statement	528,510	30.0%
Goedel-Pset-v1	506,455	28.8%
FineLeanCorpus	209,187	11.9%
Leanabell-Prover-Traindata-SFT	49,231	2.8%
NuminaMath-LEAN	41,892	2.4%
Lean-Workbook	41,211	2.3%
Subtotal (Branch 1)	*1,376,486*	*78.2%*
Branch 2: Autoformalized from informal sources		
Common Crawl + GitHub mined	384,325	21.8%
Total unique Lean statements	**1,760,811**	**100%**

trained classifier retrieves a high-recall candidate pool from the pretraining corpus, high-scoring candidates are merged back into the seed set, and the classifier is retrained; the train–retrieve–expand cycle is repeated to broaden coverage beyond exact Lean syntax matches. The pipeline yields approximately 1.1M recalled documents.

B OProofs Construction Details

This appendix details how the 1.76M unique Lean statements and 6.80M verified proofs in OProofs are constructed and de-duplicated. Downstream usage of OProofs (retriever selection, repair instance construction, data recirculation, training hyperparameters, and pretraining data recall) is described separately in Appendix E.

B.1 Statement Source Attribution

OProofs is built from two complementary branches. **Branch 1** aggregates publicly available Lean 4 corpora; **Branch 2** mines and autoformalizes informal mathematical content from the open web. Table 5 reports the source attribution of the 1.76M unique Lean statements in OProofs, computed by hashing every Lean statement under a normalized signature (stripping line and block comments, collapsing whitespace) and matching against the union of all six Branch 1 corpora. Statements whose normalized hash does not appear in any Branch 1 source are attributed to Branch 2.

Before verification filtering, Branch 1 contributes 3.76M unique statements across all six corpora. Only 1.38M (36.6%) of these statements produce at least one Lean-verified proof under our agentic proving pool (§3.2) and are therefore retained in OProofs. The remaining 63.4% are statements where no proof attempt verified within our compute budget; these are typically problems that require Mathlib lemmas not in our retrieval memory or whose proof search depth exceeds our refinement horizon.

B.2 Statement De-duplication

We de-duplicate Lean statements at two granularities. *Within each source*, identical statements (e.g., the same theorem signature repeated across shard files) are collapsed via exact match on the normalized signature defined above. *Across sources*, statements with identical normalized signatures across two Branch 1 corpora are merged and attributed to the first source in the loading order (Leanabell-Prover-Formal-Statement, Goedel-Pset-v1, FineLeanCorpus, Leanabell-Prover-Traindata-SFT, NuminaMath-LEAN, Lean-Workbook).

B.3 Decontamination Against Evaluation Benchmarks

To prevent test-set leakage, we apply three layers of decontamination between OProofs and the five evaluation benchmarks (MiniF2F-Test, MathOlympiadBench, ProofNet, ProverBench, PutnamBench).

(i) Exact-match removal. Any OProofs statement whose normalized signature (stripping comments, whitespace, import lines, and ASCII-equivalent variants, as in §B.2) matches a held-out test theorem of any of the five benchmarks is removed from both the training corpus and the retrieval memory $\mathcal{M}$. This catches statements that were copied verbatim from a public benchmark into one of the Branch 1 corpora, or that were re-derived through autoformalization in Branch 2.

(ii) Identifier-based removal. For benchmarks that publish stable problem identifiers (e.g., `imo_*`, `putnam_*`, `aime_*`, `amc_*`, `mathd_*`), we additionally remove any OProofs statement that shares an identifier with a test theorem. This catches reformatted variants where the formal statement differs in cosmetic details (variable names, comment text, import lines) but the underlying problem is identical.

(iii) Near-duplicate removal. Finally, we apply a token-shingle near-duplicate filter to catch paraphrased or lightly reformatted variants that are not removed by exact matching or identifier matching.

We first normalize each Lean statement by removing comments and boilerplate, canonicalizing common Unicode symbols to ASCII-equivalent forms, replacing theorem names with a placeholder, and collapsing whitespace. We then tokenize the normalized statement and compute word-level n-gram shingles. Candidate matches between OProofs and held-out benchmark theorems are retrieved with MinHash locality-sensitive hashing, and each candidate is verified with exact Jaccard similarity over the shingle sets. Any OProofs statement whose shingle similarity to a held-out benchmark theorem exceeds the near-duplicate threshold is removed from both training and retrieval memory. This step is designed to catch cases where the same mathematical problem is preserved but the Lean statement has been renamed, reordered, or cosmetically rewritten.

As a result, the retrieval memory used to report Table 2 is disjoint from the test theorems of all five evaluation benchmarks under exact-match, identifier-based, and near-duplicate criteria. The remaining limitation is that no automatic filter can rule out all semantic overlap between independently formalized statements; however, the combination of normalized exact matching, benchmark identifier matching, and MinHash-based near-duplicate filtering substantially reduces the most likely leakage paths from public Lean corpora and autoformalized web data into the evaluation benchmarks.

B.4 Proof De-duplication

A single statement may receive multiple verified proofs from different provers or from the same prover under different retrieved contexts and compiler feedback. We deduplicate verified proofs at the granularity $(\text{blake2b}(\text{formal_proof}),\ \texttt{has_rag},\ \texttt{has_feedback})$, where the boolean flags indicate whether the proof was produced under retrieved context and under non-trivial compiler feedback respectively. This preserves proofs that exercise different supervision signals (e.g., the same byte-identical proof produced under retrieval versus without retrieval contributes two distinct training instances, since the supervision signal differs) while removing byte-identical duplicates that share both flags. Note that this proof-level deduplication is distinct from the round-level repair instance deduplication described in Appendix A.2: the former operates on full Lean-verified proofs, the latter on local transitions (p^-, f^-, p^+) used for SFT supervision. Under this scheme, the 1.76M unique statements yield 6.80M unique training proofs, of which 4.29M carry retrieved context and 859K carry non-trivial compiler feedback (Section 4, Figure 3a).

B.5 Human Audit of OProofs Quality

To assess the quality of OProofs beyond compiler verification, we conduct a human audit of 1,000 randomly sampled Lean statements. The sample is stratified by source branch, with 500 statements drawn from Branch 1 public Lean resources and 500 statements drawn from Branch 2 autoformalized informal sources. Within each branch, we further stratify examples by the LLM-assigned difficulty label to avoid over-representing elementary statements.

Three annotators independently inspect each sampled statement. For all 1,000 examples, annotators judge whether the Lean statement is mathematically meaningful and whether the LLM-assigned mathematical domain and difficulty labels are correct. For the 500 Branch 2 examples, annotators additionally compare the formal Lean statement against its source informal statement and judge whether the formalization preserves the intended mathematical meaning. Final human labels are obtained by majority vote.

We report inter-annotator agreement using Fleiss' κ for categorical judgments. For difficulty, which is an ordered four-level label, we report the average pairwise quadratic-weighted κ across annotator pairs. LLM labels are compared against majority-vote human labels. For difficulty, we report both exact agreement and adjacent-level agreement, where adjacent-level agreement treats labels within one difficulty level as acceptable.

Table 6 summarizes the audit results. Annotators reach substantial to near-perfect agreement across the audited dimensions. The sampled statements are judged mathematically meaningful in 99.1% of cases. For Branch 2, 94.0% of audited autoformalized statements are judged semantically equivalent to their source informal statements. The LLM-assigned domain labels agree with human majority labels on 88.7% of examples. Difficulty labels show 78.2% exact agreement and 96.2% adjacent-level agreement, reflecting the inherent subjectivity of fine-grained difficulty distinctions while preserving reliable coarse-level ordering.

Dimension	N	Agreement	Result
Meaningfulness	1,000	$\kappa = 0.91$	99.1% valid
Domain	1,000	$\kappa = 0.79$	88.7% exact
Difficulty	1,000	w-$\kappa = 0.74$	78.2% / 96.2%
Branch 2 sem.	500	$\kappa = 0.81$	94.0% equiv.

Table 6: Human audit of OProofs quality. For difficulty, results report exact and adjacent-level agreement. Branch 2 semantic equivalence is evaluated only for autoformalized examples.

Compiler feedback category	Count	Share
Incomplete induction / case split	292,140	34.0%
Type or term mismatch	114,438	13.3%
Timeout / heartbeat exhaustion	109,669	12.8%
Tactic or automation failure	106,033	12.3%
Unsolved goals / goal mismatch	89,146	10.4%
Unknown identifier / namespace	79,539	9.3%
Syntax / parser error	40,958	4.8%
Other	27,043	3.1%

Table 7: Distribution of compiler feedback categories over 858,966 error-bearing feedback segments extracted from released OProofs prompts. Prompts whose feedback section contains no compiler error message are excluded. Each feedback segment is assigned one primary category.

B.6 Compiler Feedback Distribution

To characterize the diversity of Lean failure modes captured by OProofs, we categorize compiler feedback in the released prompt records. We exclude prompts whose feedback section contains no compiler error message, yielding 858,966 error-bearing feedback segments. Each feedback segment is assigned one primary category using a rule-based classifier over common Lean error patterns. Table 7 summarizes the resulting distribution.

C Test-Time Compute Allocation and Refinement

This appendix complements the main test-time scaling analysis (§5.2) along two axes: how a fixed total budget should be allocated between refinement depth and sampling width, and how each refinement round contributes to the final Pass@32 at a fixed checkpoint.

C.1 Budget Allocation between Depth and Width

To further study how test-time compute should be allocated, we fix the total budget B and plot $\mathrm{Pass}(R, B/R)$ as a function of the refinement depth R, isolating the trade-off between deeper interaction and wider sampling.

Figure 6 reveals that the optimal allocation between R and $k = B/R$ is governed by the per-chain success probability p of the benchmark. On MiniF2F, MathOlympiad, ProofNet, and ProverBench, p is already substantial, so the marginal gain from additional refinement remains positive throughout; performance improves monotonically with R and the optimum is consistently $R = 16$ for both model sizes under any medium-to-large budget. The probability $1 - (1 - p)^k$ that at least one of k independent chains succeeds is highly sensitive to k when p is small. On PutnamBench ($p \in [5, 11]\%$), halving k to double R costs more in lost exploration than it gains in deeper refinement: the optimum settles at $R = 8$ for every $B \geq 16$ on both model sizes; at $B = 256$, OProver-32B reaches 11.3% at $R = 8$ but drops to 10.7% at $R = 16$. The same effect appears on easier benchmarks under small budgets: at $B = 8$, pushing $R = 16$ forces $k = 1$ and eliminates parallel exploration, shifting the optimum to $R = 4$–8. In summary, $R = 16$ is optimal once $B \geq 16$ on easier benchmarks, while on the hardest benchmark the optimum saturates at $R = 8$.

C.2 Each Refinement Round Produces New Proofs

The analysis above shows how an externally-set budget should be allocated. A complementary question is whether, at a fixed checkpoint, each successive refinement round produces genuinely new verified proofs rather than re-sampling earlier successes. For each model and benchmark we compute the *cumulative* Pass@32 restricted to the first r refinement rounds: a rollout is counted as successful at round r if any of its attempts in rounds $1, \ldots, r$ was Lean-verified.

Figure 7 shows that on every benchmark and for both model sizes, cumulative Pass@32 grows monotonically with R. If later rounds were merely re-emitting earlier behaviour, the curves would flatten by $R \approx 2$; instead they continue to climb steadily through $R = 16$, confirming that each refinement round repairs proofs that earlier rounds failed to verify. The magnitude of the refinement gain mirrors the budget-level pattern above: on easier benchmarks (MiniF2F, ProverBench) the curves plateau by $R \approx 8$ with $+4$–6 absolute points, while on harder benchmarks the gains are much larger in relative terms, with PutnamBench rising from 5.7 to 9.0 ($+57\%$) for OProver-8B and from 7.0

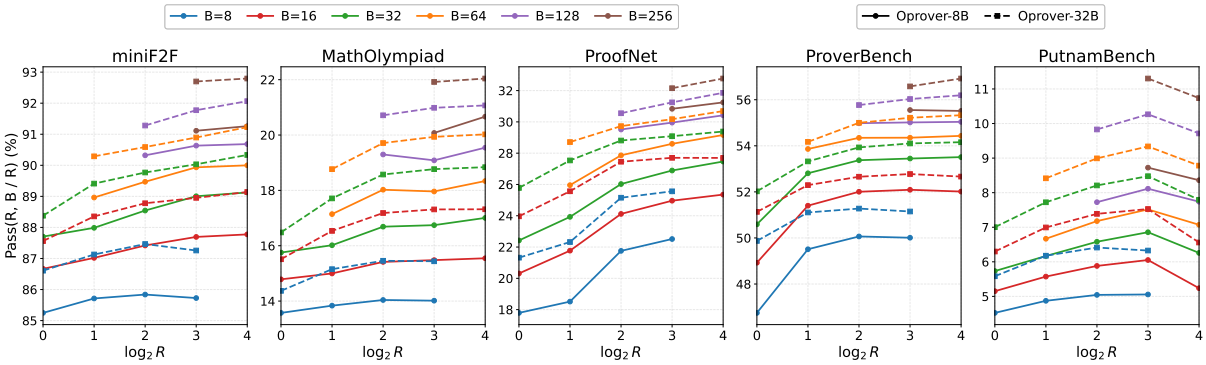


Figure 6: Performance under fixed total budget B as a function of refinement depth R. Each curve plots $\mathrm{Pass}(R, B/R)$, isolating the trade-off between deeper interaction and wider sampling. On most benchmarks performance keeps improving as R grows up to $R = 16$; on PutnamBench the optimum settles around $R = 8$.

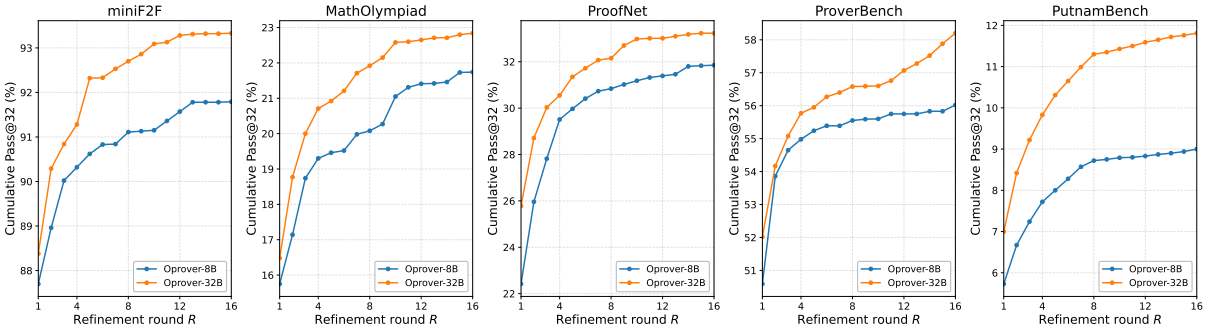


Figure 7: Cumulative Pass@32 of OProver-8B and OProver-32B as a function of the refinement round cutoff R on five Lean 4 benchmarks. The single-round value ($R = 1$) corresponds to a non-refinement baseline; $R = 16$ matches the headline numbers in Table 2.

to 11.8 (+69%) for OProver-32B. When per-chain success probability is low, adding refinement depth is the most compute-efficient way to convert failed attempts into verified proofs.

D Effectiveness of Agentic Reasoning

This appendix provides qualitative evidence for the effectiveness of the agentic refinement loop in OProver. In our setting, the policy does not generate a proof in a single shot. Instead, it iteratively refines candidate proofs by conditioning on retrieved formal references and Lean verifier feedback. The examples below illustrate how these two sources of information support grounded repair behavior across rounds. All three case studies are real OProver-32B trajectories on the MiniF2F-Test benchmark in which Round 1 fails to verify and Round 2 produces a Lean-verified proof.

D.1 Roles of Retrieval and Compiler Feedback

Retrieved formal references are most useful in three recurring ways. First, they help ground lemma names, rewrite rules, and tactic idioms that are easy to miss in one-shot generation. Second, they provide short structural hints about how to decompose a goal into subgoals that better match Lean's expected proof state. Third, they expose proof patterns that help the model avoid common formulation errors, such as missing coercions, inappropriate tactic ordering, or mismatched algebraic normal forms. In practice, these signals are lightweight, but they often suffice to redirect a stalled proof attempt toward a more verifiable local proof structure.

Lean feedback provides a complementary signal: it identifies concrete verifier-side failures that can be mapped to local edits in the next round. Rather than projecting compiler outputs into a hand-designed taxonomy during inference, OProver conditions directly on the original textual diagnostics. This preserves fine-grained information about the actual source of failure—including type mismatches, missing identifiers, unresolved goals, or tactic failures. Common categories of such feedback and the corresponding repair actions are listed

in Table 8 of Appendix F.

Presentation note. For compatibility with the `pdflatex` toolchain, the proof excerpts below normalize Lean Unicode notation into an ASCII display form while preserving the logical structure of each repair trace.

D.2 Case Study A: AMC 12B 2002 P2

This example illustrates a precise local repair driven by a Lean type-mismatch diagnostic. The first-round attempt restates the goal with an associativity error (`((3*x-2) * 4 * x` instead of `(3*x-2) * (4*x)`); the verifier identifies the exact mismatch, and the second-round proof restores correct parenthesization and closes the goal directly via tactic automation.

Round 1 Proof Attempt

```
import Mathlib
import Aesop

set_option maxHeartbeats 4000000
% set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

theorem amc12b_2002_p2 (x : Int) (h0 : x =
    4) :
    (3 * x - 2) * (4 * x + 1) - (3 * x - 2)
    * (4 * x) + 1 = 11 :=
by
  have h1 : (3 * x - 2) * (4 * x + 1) - (3 *
    x - 2) * 4 * x + 1 = 11 := by
    rw [h0]
    <;> norm_num
    <;> ring_nf at *
    <;> norm_num
    <;> linarith

  exact h1
```

Round 1 Lean Feedback

```
Lean errors:
type mismatch
  h1
has type
  (3 * x - 2) * (4 * x + 1) - (3 * x - 2) *
    4 * x + 1 = 11 : Prop
but is expected to have type
  (3 * x - 2) * (4 * x + 1) - (3 * x - 2) *
    (4 * x) + 1 = 11 : Prop
```

Reasoning summary. The proof attempt and the goal differ by a single associativity bracket: the auxiliary statement `h1` is parsed as `((3*x-2) * 4) * x` while the goal expects `(3*x-2) * (4*x)`. The verifier diagnostic pinpoints this mismatch, so the repair only needs to align the bracketing.

Round 2 Revised Proof

```
import Mathlib
import Aesop

set_option maxHeartbeats 4000000
% set_option maxHeartbeats 0

open BigOperators Real Nat Topology Rat

theorem amc12b_2002_p2 (x : Int) (h0 : x =
    4) :
    (3 * x - 2) * (4 * x + 1) - (3 * x - 2)
    * (4 * x) + 1 = 11 :=
by
  have h1 : (3 * x - 2) * (4 * x + 1) - (3 *
    x - 2) * (4 * x) + 1 = 11 := by
    rw [h0]
    <;> norm_num
    <;> ring_nf at *
    <;> norm_num
    <;> linarith

  simpa using h1
```

Repair summary. Round 2 rewrites the auxiliary equality to match the goal's parenthesization exactly, then discharges it with the same automation chain (`rw`, `norm_num`, `ring_nf`, `linarith`) and closes the top-level goal via `simpa`. The repair is purely structural and is uniquely determined by the verifier feedback.

D.3 Case Study B: MathD numbertheory 321

This example shows the policy abandoning a brittle Mathlib lemma application after the verifier reports a unification failure, and adopting a more direct computational tactic. The Round 1 attempt tries to derive the modular inverse via `inv_eq_of_mul_eq_one_right`; the verifier fails to unify the lemma's polymorphic shape with the concrete `ZMod 1399` setting. The Round 2 attempt drops the lemma entirely and uses Lean's kernel evaluator to discharge the inverse identity by direct computation.

Round 1 Proof Attempt

```
theorem mathd_numbertheory_321 (n : ZMod
    1399) (h1 : n = 160-inv) : n = 1058 :=
by
  have h2 : (160 : ZMod 1399) * (1058 : ZMod
    1399) = 1 := by
    native_decide

  have h3 : (160 : ZMod 1399)-inv = (1058 :
    ZMod 1399) := by
```

```
    have h4 : (160 : ZMod 1399) * (1058 :
     ZMod 1399) = 1 := h2
    have h5 : (160 : ZMod 1399)-inv = (1058
     : ZMod 1399) := by
      apply inv_eq_of_mul_eq_one_right
      <;> simpa using h4
    exact h5

  have h4 : n = (1058 : ZMod 1399) := by
    rw [h1]
    rw [h3]
    <;> norm_num
    <;> rfl

  norm_num at h4
  <;> simpa using h4
```

Round 1 Lean Feedback

```
Lean errors:
tactic 'apply' failed, failed to unify
  @Inv.inv ?G DivInvMonoid.toInv ?a = ?b
with
  @Inv.inv (ZMod 1399) (ZMod.instInv 1399)
     160 = 1058
n : ZMod 1399
h1 : n = 160-inv
h2 h4 : 160 * 1058 = 1
turnstile  160-inv = 1058
```

Reasoning summary. The verifier complains that `inv_eq_of_mul_eq_one_right` cannot be unified against a goal stated with `ZMod.instInv`: the lemma expects a generic `DivInvMonoid` structure but the concrete `ZMod 1399` carries a different `Inv` instance. The feedback signals that the chosen lemma is the wrong tool for this concrete computation, not that the underlying arithmetic is incorrect.

Round 2 Revised Proof

```
theorem mathd_numbertheory_321 (n : ZMod
     1399) (h1 : n = 160-inv) : n = 1058 :=
by
  have h2 : (160 : ZMod 1399)-inv = (1058 :
     ZMod 1399) := by
    native_decide

  have h3 : n = (1058 : ZMod 1399) := by
    rw [h1]
    rw [h2]
    <;> norm_num
    <;> rfl

  norm_num at h3
  <;> simpa using h3
```

Repair summary. Round 2 collapses the entire lemma-based detour into a single `native_decide`: since both `160` and `1058` are ground integers in a finite ring, Lean's kernel can decide the inverse identity by direct evaluation. The verifier feedback redirected the policy away from a misapplied lemma toward a tactic that matches the structure of the problem.

D.4 Case Study C: MathD numbertheory 427

This example illustrates a more substantive repair where the verifier surfaces a numerical inconsistency that propagates through the proof. The Round 1 attempt assumes the wrong value for an intermediate sum-of-divisors quantity, and the verifier reports two coupled failures: an unsolved goal at the intermediate step and a downstream `native_decide` that evaluates a derived proposition to `False`. The Round 2 attempt decomposes the computation explicitly, computing the correct intermediate value first and then chaining the subsequent prime-sum identity.

Round 1 Proof Attempt

```
theorem mathd_numbertheory_427 (a : Nat)
    (h0 : a = Finset.sum (Nat.divisors 500)
     (fun k => k)) :
    (Finset.sum (Finset.filter (fun x =>
     Nat.Prime x) (Nat.divisors a))
      (fun k => k)) = 25 :=
by
  have h1 : a = 5467 := by
    rw [h0]
    <;> (try native_decide)
    <;> (try { native_decide })

  have h2 : (Finset.sum (Finset.filter (fun
     x => Nat.Prime x) (Nat.divisors a))
              (fun k => k)) = 25 := by
    rw [h1]
    native_decide

  exact h2
```

Round 1 Lean Feedback

```
Lean errors:
unsolved goals
a : Nat
h0 : a = Finset.sum (Nat.divisors 500) (fun
     k => k)
turnstile  Finset.sum (Nat.divisors 500)
     (fun k => k) = 5467

tactic 'native_decide' evaluated that the
     proposition
  Finset.sum (Finset.filter (fun x =>
     Nat.Prime x) (Nat.divisors 5467))
     (fun k => k) = 25
is false
```

Reasoning summary. The verifier exposes two coupled failures. The first is local: `native_decide` cannot establish that the sum of divisors of `500`

equals 5467, because the assumed value is wrong. The second is downstream: the subsequent prime-divisor identity, conditioned on the same wrong value, evaluates to `False`. The combined feedback signals that the proof structure is sound but the intermediate numerical constant must be replaced.

Round 2 Revised Proof

```
theorem mathd_numbertheory_427 (a : Nat)
    (h0 : a = Finset.sum (Nat.divisors 500)
     (fun k => k)) :
    (Finset.sum (Finset.filter (fun x =>
     Nat.Prime x) (Nat.divisors a))
      (fun k => k)) = 25 :=
by
  have h_sum_divisors_500 :
      Finset.sum (Nat.divisors 500) (fun k
    => k) = 1092 := by
    native_decide

  have h_a_value : a = 1092 := by
    rw [h0]
    <;> rw [h_sum_divisors_500]
    <;> norm_num

  have h_sum_primes :
      (Finset.sum (Finset.filter (fun x =>
    Nat.Prime x) (Nat.divisors a))
        (fun k => k)) = 25 := by
    rw [h_a_value]
    native_decide

  exact h_sum_primes
```

Repair summary. Round 2 isolates the intermediate quantity by giving `Finset.sum (Nat.divisors 500) = 1092` its own `native_decide` lemma, then derives `a = 1092` by rewriting and finally discharges the prime-divisor identity on the corrected value. The repair preserves the high-level decomposition from Round 1 but substitutes the correct numerical constant, which in turn lets every downstream `native_decide` succeed.

Overall takeaway. The three case studies span a range of repair difficulties triggered by qualitatively different verifier signals. In Case A, a single type-mismatch diagnostic pinpoints a one-symbol parenthesization fix that re-aligns the auxiliary equality with the goal. In Case B, a unification failure indicates that the chosen Mathlib lemma is structurally incompatible with the concrete `ZMod` setting, prompting the policy to substitute a more direct `native_decide`. In Case C, two coupled failures expose a wrong intermediate constant that propagates through the proof, leading the policy to decompose the computation and inject a corrected lemma. Together, these traces show that the feedback-conditioned refinement loop scales naturally from a one-symbol local fix to a multi-step structural correction, and matches the granularity of the repair to the failure mode reported by Lean.

E Reproducibility Configuration

This appendix consolidates the information required to reproduce our results: (i) the artifacts we will release, (ii) the Lean and software environment used during training and evaluation, (iii) training and evaluation hyperparameters in full, and (iv) the computational budget required for each phase. Other reproducibility-relevant material is documented elsewhere in the appendix: prompt templates and Lean feedback serialization in Appendix F, retriever selection in Appendix A.1, iterative data recirculation in Appendix A.3, pretraining data recall in Appendix A.4, OProofs construction in Appendix B, and decontamination against evaluation benchmarks in Appendix B.3.

E.1 Released Artifacts

We release the model weights, OProofs corpus, training code, and agentic proving inference code during review. The prebuilt container image will be released upon acceptance:

- **Model weights.** We release OProver-8B as a HuggingFace checkpoint at https://huggingface.co/OProver/Oprover-8B, together with OProver-32B, OProver-Base from continued pretraining, and the intermediate post-training checkpoints.
- **OProofs corpus.** We release the OProofs corpus at https://huggingface.co/datasets/OProver/OProofs, including 1.76M Lean 4 statements, 6.80M compiler-verified proofs, 1.06M agentic proving trajectories.
- **Training code.** We release the end-to-end pipeline for continued pretraining, supervised fine-tuning, and reinforcement learning at https://anonymous.4open.science/r/OProver-oss-05B1.
- **Agentic proving inference code.** We release the multi-round refinement loop, including retrieval, prompt construction, Lean verification, and feedback serialization,

at https://anonymous.4open.science/r/OProver-oss-05B1.

- **Container image.** Upon acceptance, we will release a Docker image that pins Lean, Mathlib, the Kimina Lean server, Python, CUDA, and all Python dependencies to the exact versions used in this paper. During review, we provide the corresponding software versions and configuration details in Appendix E.2.

E.2 Lean and Software Environment

Lean verification stack. We use the Lean toolchain `leanprover/lean4:v4.15.0` with Mathlib pinned to tag `v4.15.0` (commit `9837ca9d65d9de6fad1ef4381750ca688774e608`) and the `FrederickPu/repl` backend on branch `lean415compat`. Verification is served by an OProver-specific fork of the Kimina Lean server `v2.0.0` (commit `8651d4a`, with pipeline tooling and verifier-side caching tweaks), released alongside the code. We set `maxHeartbeats` to 2M during RL and 4M during evaluation, with per-request timeouts of 120s and 240s respectively; proof reconstruction is enabled with fully explicit proofs. The verifier runs on Python `3.13` inside the same containerized environment; the corresponding Docker image will be released upon acceptance.

E.3 Hyperparameters

Continued pretraining. OProver-Base is obtained by continued pretraining on the 65B-token mixture described in Section 3.3, using AdamW with peak learning rate 5×10^{-5}, cosine schedule with 3% warmup, global batch size 512, and sequence length 8192.

Supervised fine-tuning. OProver-8B and OProver-32B are fine-tuned with global sequence length 40,960 and global batch size 64. Learning rates are 2×10^{-5} for OProver-8B and 5×10^{-5} for OProver-32B. Cross-entropy loss is computed only on the target proof attempt p_t.

Reinforcement learning. We adopt GSPO with learning rate 2×10^{-6}, training batch size 256, PPO mini-batch size 256, and $n = 8$ rollouts per theorem. The maximum prompt length is 14,000 tokens. The maximum response length is 24,000 tokens for OProver-8B and 20,000 tokens for OProver-32B. The maximum number of refinement rounds during RL is 4 for OProver-8B and 2 for OProver-32B. Rollout decoding uses temperature 1.0 and top-p 0.999. The per-round verification reward is $r_t = 0.8 + 0.2 \cdot \mathbb{1}[\text{format correct}]$ when $\mathcal{V}(p_t) =$ verified, and $r_t = 0$ otherwise; the $0.8/0.2$ split follows the format-aware reward design used by recent Lean provers (Wang et al., 2025), allocating 80% of the signal to verification correctness and 20% to output format to discourage format collapse during RL. Advantages are computed by group-relative normalization pooled across the $n \times R$ rounds for the same theorem. Lean verification during RL uses proof reconstruction with fully explicit proofs, a per-request timeout of 120s, and a `maxHeartbeats` cap of 2M.

Evaluation. At evaluation time we use the same multi-turn proving interface as RL training. Generation uses temperature 1.0, top-p 0.999, and a maximum response length of 32,000 tokens. Proof verification uses the Kimina Lean server (Lean 4.15.0) with proof reconstruction and fully explicit proofs, a per-request timeout of 240s, and a `maxHeartbeats` cap of 4M.

E.4 Hardware and Compute Budget

All training runs use NVIDIA H800 GPUs. Continued pretraining consumes $\approx$ 27.5k H800-hours, supervised fine-tuning consumes $\approx$ 4.6k H800-hours, and reinforcement learning consumes $\approx$ 36.8k H800-hours, for a total of $\approx$ 69k H800-hours.

E.5 Determinism and Variance

Pass@32 numbers in Tables 2 and 3 are reported as mean $\pm$ seed-level standard error (Section 5). Generation uses temperature 1.0 and top-p 0.999, so successive runs of the released checkpoints on the released benchmarks will not be bit-for-bit identical to the numbers in this paper, but Pass@32 on each benchmark should fall within the reported confidence interval.

F Prompt Templates and Feedback Format

This appendix specifies the prompt template and the feedback serialization used by OProver during both training and inference, following the state formulation $X_t = (s, \mathcal{R}_t, p_{t-1}, f_{t-1})$ from Section 3.1.

F.1 Prompt Template

We use a single prompt template across all rounds. At round t, the model conditions on the target Lean 4 statement, the top-k retrieved references,

and the immediately preceding proof attempt with its Lean feedback. When $t = 1$, the previous-attempt and feedback fields are left empty.

Unified Prompt Template at Round t

```
**Current Task:**
Complete the following Lean 4 code:
{FORMAL_STATEMENT}

Before producing the Lean 4 proof, first
    provide a concise proof plan
summarizing the intended strategy, key
    lemmas, and proof structure.
If a previous attempt and Lean feedback are
    provided, revise the proof
accordingly.

Reference theorems and proofs:
{RETRIEVED_PROOF_1}
{RETRIEVED_PROOF_2}

Previous Failed Attempt:
{PREVIOUS_PROOF_ATTEMPT}

Lean Feedback:
{LEAN_ERROR_MESSAGES}
```

F.2 Feedback Serialization

Lean feedback is serialized as the raw plain-text diagnostic returned by the Lean 4 compiler for the immediately preceding proof attempt, with no projection into a fixed error taxonomy. This preserves identifier names, expected/actual types, and goal contexts that targeted revision often depends on. An example diagnostic, after stripping ANSI control codes, looks like:

Example Lean Feedback

```
type mismatch
h0
has type
Nat.succ ?m.123
but is expected to have type
Real

tactic 'linarith' failed to find a
    contradiction
case h
x y : Real
h0 : x <= y
h1 : y <= 0
turnstile False
```

F.3 Common Feedback Categories

Although feedback is passed in its raw textual form, the underlying diagnostics fall into a few recurring categories, which we summarize in Table 8 together with the typical repair actions we observe across multi-round refinement.

Table 8: Common Lean feedback categories observed in OProver's proving trajectories, with the typical repair actions taken in the subsequent round.

Feedback type	Typical repair action
Unsolved goals / goal mismatch	Introduce missing intermediate lemmas or case splits, restate the target in a more convenient form, or align the local proof state before continuing.
Tactic or automation failure	Replace brittle one-shot automation with explicit derivation steps, reorder tactics, or simplify the context before invoking automation such as `linarith`, `omega`, or `simp`.
Type or term mismatch	Rewrite ill-typed expressions, add explicit coercions or binders, and ensure intermediate terms match the expected type.
Unknown identifier / namespace / notation error	Qualify names explicitly, open the required namespace, fix notation, or replace unavailable lemmas with in-scope equivalents.
Incomplete induction or case split	Add missing branches, make parity or sign cases explicit, and construct the required witnesses before closing the proof.
Timeout / heartbeat exhaustion	Shorten proof scripts, reduce expensive search, and replace heavy automation with more structured local reasoning.

G Statement Labeling Prompt

In addition to the prover-side template, we use a separate offline classification prompt to annotate each Lean 4 statement with a mathematical domain (one of 10 categories) and a difficulty level (one of 4 tiers), as reported in Figure 3(b,c). The full prompt is shown below.

Lean 4 Statement Domain & Difficulty Classification Prompt

```
**Instruction: Lean 4 Statement
    Classification (Mathematical Domain &
    Difficulty Level)**

You are an expert mathematician and Lean 4
    specialist. You will receive a Lean 4
formal statement. Your task is to classify
    it along two dimensions:
1. **Mathematical Domain** (one of 10
    categories)
2. **Difficulty Level** (one of 4 levels, by
    mathematical sophistication required,
 NOT by Lean proof length)

---

**DIMENSION 1: MATHEMATICAL DOMAIN**
```

```
Choose **exactly one** primary domain. If
     the statement spans multiple domains,
pick the most central one.

- **Algebra**: abstract algebra,
     groups/rings/fields, linear algebra,
     polynomials,
algebraic identities, algebraic inequalities
- **NumberTheory**: divisibility,
     congruences, primes, Diophantine
     equations,
modular arithmetic, analytic number theory
- **Analysis**: real/complex/functional
     analysis, calculus, limits, continuity,
series, (non-probabilistic) measure theory,
     analytic inequalities
- **Topology**: general topology, algebraic
     topology, differential topology,
topological spaces, continuity in
     topological settings
- **Geometry**: planar/solid/analytic
     geometry, differential geometry,
geometric inequalities, classical geometry
     problems
- **Combinatorics**: combinatorial
     identities, graph theory, counting,
discrete mathematics, Ramsey-style problems
- **ProbabilityStatistics**: probability
     theory, statistics, probabilistic
inequalities, stochastic processes
- **LogicFoundations**: logic, set theory,
     type theory, model theory,
foundational results, definability
- **Computation**: algorithms, complexity,
     formal languages, information
theory, CS-flavor mathematics
- **Other**: anything that does not fit the
     above, including trivially true
statements, malformed statements, or
     non-mathematical content.

---

**DIMENSION 2: DIFFICULTY LEVEL**

Choose **exactly one** level based on the
     mathematical sophistication required
to prove the statement. Do NOT base
     difficulty on the length or syntactic
complexity of the Lean proof.

- **Elementary**: middle-school level or
     below; basic arithmetic, simple
algebraic manipulations, elementary number
     facts, basic geometric facts.
- **HighSchool**: high-school level math;
     standard algebra/trigonometry/
precalculus, basic combinatorics, standard
     introductory contests
(AMC, AIME, Chinese gaokao).
- **Undergraduate**: standard undergraduate
     mathematics; linear algebra,
single/multi-variable calculus, introductory
     abstract algebra,
introductory real analysis, introductory
     topology.
- **GraduatePlus**: graduate-level or
     research mathematics; advanced topology,
functional analysis, algebraic geometry,
     advanced number theory.
Also includes olympiad-level problems (e.g.,
     IMO, USAMO, Putnam) that
require nontrivial mathematical insight
     beyond standard curriculum.

*Note 1: When in doubt between two adjacent
     levels, prefer the lower level
unless the statement clearly requires the
     higher one.*
*Note 2: For statements that are trivially
     true (e.g., `True`, `0 = 0`),
malformed, or otherwise not meaningful
     mathematical claims, classify the
domain as `Other` and the difficulty as
     `Elementary`.*

---

**STRICT OUTPUT REQUIREMENTS:**
1. Output **ONLY** a single JSON object on a
     single line.
2. Do **NOT** include any explanation,
     markdown formatting (no ```json fences),
 or extra text outside the JSON.
3. All three fields are required.

**Output format (single JSON object):**

{"domain": "<one of the 10 domains above>",
     "difficulty": "<one of the 4 levels
     above>",
"rationale": "<one short sentence, max 25
     words, explaining the classification>"}

---

**EXAMPLES**

**Example 1 (Elementary, Algebra):**
Statement:
theorem ex (a b : Nat) : a + b = b + a := by
     ring
Output: {"domain": "Algebra", "difficulty":
     "Elementary", "rationale": "Basic
     commutativity of
natural number addition, taught at
     elementary level."}

**Example 2 (HighSchool, Combinatorics):**
Statement:
theorem ex (n : Nat) (hn : 5 <= n) :
     Nat.choose n 2 = n * (n - 1) / 2 := by
     sorry
Output: {"domain": "Combinatorics",
     "difficulty": "HighSchool", "rationale":
     "Standard
binomial-coefficient identity, accessible at
     high-school combinatorics level."}

**Example 3 (Undergraduate, Topology):**
Statement:
theorem ex {X : Type*} [TopologicalSpace X]
     [CompactSpace X] [T2Space X] :
     NormalSpace X := by
sorry
Output: {"domain": "Topology", "difficulty":
     "Undergraduate", "rationale": "Standard
```

```
    introductory
topology result that compact Hausdorff
    spaces are normal."}

**Example 4 (GraduatePlus, Analysis):**
Statement:
theorem ex (f : Real -> Real) (hf :
    Continuous f) (h_nn : forall x, 0 <= f x)
(h_zero : Real.intervalIntegral f 0 1 = 0) :
    forall x in Set.Icc 0 1, f x = 0 := by
    sorry
Output: {"domain": "Analysis", "difficulty":
    "GraduatePlus", "rationale": "Standard
measure-theoretic result that a non-negative
    continuous function with zero integral
    vanishes on
the domain."}

---

**[Input Content Starts]**
{FORMAL_STATEMENT}
**[Input Content Ends]**

**Output (JSON only, single line):**
```

We run this prompt once per unique Lean 4 statement; 96.4% of responses parse to a well-formed JSON object with valid domain and difficulty values, yielding the 1.73M statement-level annotations summarized in Figure 3(b,c).