Goedel-Code-Prover: Hierarchical Proof Search for Open State-of-the-Art Code Verification

Zenan Li[1,*], **Ziran Yang**[2,*], **Deyuan (Mike) He**[3], **Haoyu Zhao**[3], **Andrew Zhao**[3], **Shange Tang**[2], **Kaiyu Yang**[4], **Aarti Gupta**[3], **Zhendong Su**[1], **Chi Jin**[2]
[1]ETH Zürich, [2]Princeton Language and Intelligence,
[3]Department of Computer Science, Princeton University, [4]MiroMind
*Equal contribution
{`zenan.li@inf.ethz.ch`, `zirany@princeton.edu`, `chij@princeton.edu`}

Abstract

Large language models (LLMs) can generate plausible code but offer limited guarantees of correctness. Formally verifying that implementations satisfy specifications requires constructing machine-checkable proofs, a task that remains beyond current automation. We propose a hierarchical proof search framework for automated code verification in Lean 4 that decomposes complex verification goals into structurally simpler subgoals before attempting tactic-level proving. Central to our approach is a principled decomposition score that combines constructive justification with structural effectiveness. Crucially, this score serves as both the training reward and the inference-time ranking criterion, ensuring strict alignment between optimization and deployment. We train *Göedel-Code-Prover-8B*, a single unified policy for both decomposition and completion, via supervised initialization followed by hybrid reinforcement learning, where a continuous decomposition reward drives planning exploration while supervised replay stabilizes proof generation. On three Lean-based code verification benchmarks comprising 427 tasks, our 8B-parameter model achieves a 62.0% prove success rate, a 2.6× improvement over the strongest baseline, surpassing neural provers up to 84× larger. We further observe consistent inference-time scaling: success rates improve monotonically with search iterations and sampling budget, with our trained model achieving greater efficiency than frontier off-the-shelf models of comparable scale.

1 Introduction

Large language models (LLMs) have rapidly advanced code generation, achieving strong performance on programming benchmarks and increasingly serving as real-world coding assistants (Chen et al., 2021; Li et al., 2022; Alharbi & Alshayeb, 2026; Li et al., 2026). Yet while LLM-based coding agents routinely generate unit tests and reproduction scripts to validate their outputs, empirical studies show that such testing cannot certify the absence of subtle logical errors, boundary-case violations, or specification mismatches (Jesse et al., 2023; Liu et al., 2023; Al-Kaswan et al., 2026). As LLMs are deployed in safety-critical environments, bridging the gap between "code that appears to work" and "code that is provably correct" becomes increasingly urgent.

Formal verification offers a principled solution by expressing specifications (see Figure 1 for an example) in a formal language and constructing machine-checked proofs of correctness. Lean 4 (Moura & Ullrich, 2021), a modern interactive theorem prover, provides a powerful platform for such verification tasks, with a growing ecosystem of libraries (e.g., `Mathlib` (mathlib Community, 2020) and `CSLib` (Barrett et al., 2026)). Figure 1 illustrates a typical verification task: given a function `FindSingleNumber` that returns the unique element from a list, the specification encodes a precondition and a postcondition as Lean predicates, and the top-level theorem asserts that the implementation satisfies this specification. The

```
-- Precondition: every element appears 1 or 2 times; exactly one is unique
def FindSingleNumber_precond (nums : List Int) : Prop :=
  let numsCount := nums.map (fun x => nums.count x)
  numsCount.all (fun c => c = 1 ∨ c = 2) ∧ numsCount.count 1 = 1

-- Postcondition: result is the unique element; others appear twice
def FindSingleNumber_postcond (nums : List Int) (result : Int)
    (h : FindSingleNumber_precond nums) : Prop :=
  nums.length > 0 ∧ (filterlist result nums).length = 1 ∧
  ∀ (x : Int), x ∈ nums → x = result ∨ (filterlist x nums).length = 2
```

```
-- Verification goal (proof to be synthesized automatically)
theorem FindSingleNumber_spec (nums : List Int)
    (h : FindSingleNumber_precond nums) :
    FindSingleNumber_postcond nums (FindSingleNumber nums h) h := by
  sorry
```

Figure 1: A code verification task in Lean 4. The **blue** block defines the specification: a precondition constraining valid inputs and a postcondition characterizing correct outputs. The **red** block states the top-level verification goal, where `sorry` marks the proof obligation to be synthesized automatically. The complete proof synthesized by our framework is shown in Appendix A.

`sorry` placeholder marks the proof obligation to be discharged. Even for this seemingly simple problem, constructing a complete proof requires discovering auxiliary lemmas about list filtering, XOR properties over integers, and inductive reasoning over the input structure, none of which are suggested by the specification itself. Writing such proofs by hand remains labor-intensive and demands substantial expertise, limiting widespread adoption of formal verification.

Recent work has explored leveraging LLMs to automate proof synthesis in interactive theorem provers (ITPs) such as Lean 4 and Isabelle (Li et al., 2024; Yang et al., 2024a). State-of-the-art systems have achieved remarkable success in formal mathematics, reaching competition and olympiad level performance (Ren et al., 2025; Lin et al., 2025; Xin et al., 2025b; Hubert et al., 2025; Wang et al., 2025). These methods typically operate in a *decompose-and-prove* paradigm: the model first decomposes a complex theorem into simpler intermediate lemmas (or subgoals), then proves each lemma individually using tactic generation.

Despite this progress, these methods do not transfer directly to code verification due to two fundamental mismatches. The first problem is *ungrounded decomposition*. In mathematics, correctness is central to the discipline: mathematicians routinely write proofs by hand, and the resulting reasoning pervades textbooks, papers, and online discussions. LLMs absorb this rich corpus during pretraining, acquiring the intuition needed to propose meaningful proof decompositions even before formal verification. No comparable corpus exists for program correctness. Most software developers treat testing as a sufficient proxy for correctness (Myers et al., 2004; Ammann & Offutt, 2017) and never articulate formal reasoning about their code. In the few domains where formal guarantees are required, practitioners rely on automated tools such as SMT solvers (De Moura & Bjørner, 2008; Barbosa et al., 2022) and model checkers (Clarke, 1997; Jhala & Majumdar, 2009) rather than hand-written proofs, producing little natural-language proof reasoning in the process. Consequently, LLMs lack the informal-to-formal bridge that enables effective decomposition in mathematics: when applied to code verification, their proposed subgoals are frequently imprecise, semantically invalid, or no simpler than the original goal.

The second issue is *compound domain shift*: even when a decomposition is valid, proving the resulting subgoals faces a dual challenge. On one hand, program verification introduces a fresh conceptual universe with every program (*concept proliferation*): each function definition brings new computational behavior, control flow patterns, and data-structure invariants

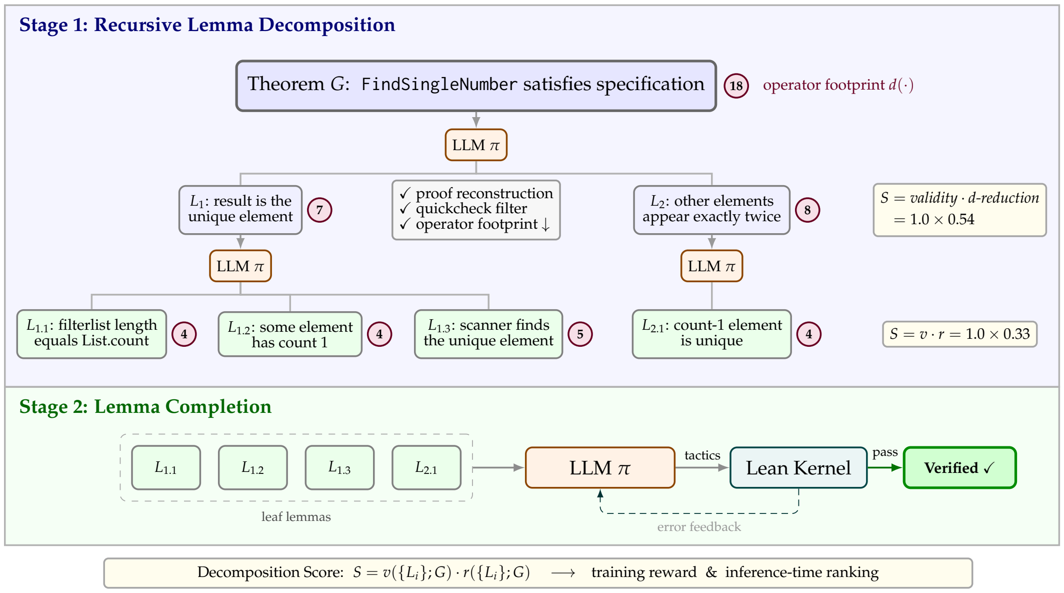
Stage 1: Recursive Lemma Decomposition
Theorem G: FindSingleNumber satisfies specification
18
operator footprint $d(\cdot)$
LLM π
L_1: result is the unique element
7
✓ proof reconstruction
✓ quickcheck filter
✓ operator footprint ↓
L_2: other elements appear exactly twice
8
$S = validity \cdot d\text{-}reduction = 1.0 \times 0.54$
LLM π
LLM π
$L_{1.1}$: filterlist length equals List.count
4
$L_{1.2}$: some element has count 1
4
$L_{1.3}$: scanner finds the unique element
5
$L_{2.1}$: count-1 element is unique
4
$S = v \cdot r = 1.0 \times 0.33$
Stage 2: Lemma Completion
$L_{1.1}$
$L_{1.2}$
$L_{1.3}$
$L_{2.1}$
leaf lemmas
LLM π
tactics
Lean Kernel
pass
Verified ✓
error feedback
Decomposition Score: $S = v(\{L_i\}; G) \cdot r(\{L_i\}; G)$ ⟶ training reward & inference-time ranking

Figure 2: **Overview of the hierarchical proof search framework.** *Stage 1 (Recursive Lemma Decomposition):* The verification goal is recursively decomposed into structurally simpler sub-lemmas. At each step, the LLM policy π proposes candidates verified via proof reconstruction and quickcheck; the decomposition score S ranks candidates by combining validity with structural reduction in operator footprint $d(\cdot)$. *Stage 2 (Iterative Lemma Completion):* Leaf lemmas are proved by the same policy through iterative tactic generation with Lean 4 compiler feedback. The same score serves as both the training reward and the inference-time ranking criterion.

whose properties must be discovered without prior library support, unlike mathematics where shared libraries such as Mathlib provide a closed vocabulary of reusable lemmas. On the other hand, discharging these subgoals requires program-specific automation (*tactic distribution shift*): tactics such as `grind`, SMT-backed lemmas, and computational reflection (`native_decide`) that differ substantially from the algebraic tactics prevalent in mathematics; models trained on mathematical corpora routinely fail to invoke these tools correctly.

In this work, we propose a *hierarchical proof search* framework for automated code verification in Lean 4, in which the verification goal is recursively decomposed into a series of lemmas, followed by iteratively proving each lemma (Figure 2). To achieve this, we train *Göedel-Code-Prover-8B*, a single unified policy that performs both decomposition and completion. Since the quality of decomposition is the primary determinant of success, we introduce a principled *decomposition score* as both the training reward and inference-time search criterion. This score evaluates candidates along two axes: constructive justification (whether proposed subgoals provably entail the parent theorem and survive automated counterexample search via *quickcheck* (Claessen & Hughes, 2000; Lean Prover Community, 2024)), and structural simplicity (measured by a goal complexity metric). An additional challenge arises from reward mismatch: decomposition benefits from this dense, continuous score, whereas completion yields only sparse binary signals (proof accepted or rejected), causing decomposition gradients to dominate and completion proficiency to stagnate. We resolve this through hybrid reinforcement learning after supervised initialization, optimizing decomposition with the continuous score while stabilizing completion through supervised replay of high-quality proof trajectories.

At inference time, the trained model operates in a planning-and-proving loop. (1) *Lemma decomposition* transforms a top-level verification theorem into a set of intermediate subgoals that collectively entail the original statement. (2) *Lemma completion* then discharges each subgoal individually through iterative tactic generation and refinement. The same decomposition score guides the inference-time search, selecting among candidate decompositions and ensuring strict alignment between the training objective and the deployment criterion.

We evaluate our framework on three Lean 4 code verification benchmarks: Verina (Ye et al., 2025), Clever (Thakur et al., 2025), and AlgoVeri (Zhao et al., 2026), comprising 427 tasks in total. Our 8B-parameter model achieves success rates of 68.8%, 54.0%, and 62.3% respectively, a 2.6× improvement over the strongest baseline while surpassing neural provers up to 84× larger. Verified proofs average 8–17 decomposed lemmas and over 130 lines of proof code, with the most complex exceeding 680 lines, demonstrating that our framework can sustain deep structured reasoning over non-trivial verification tasks. We further observe consistent inference-time scaling: success rates improve monotonically with search iterations and sampling budget, indicating unsaturated scaling potential.

In summary, this paper makes the following contributions:

- A hierarchical proof search framework for code verification in Lean 4 that decomposes complex verification goals into structurally simpler subgoals before tactic-level proving;
- A principled decomposition score combining constructive justification with structural simplicity, used consistently as both training reward and inference-time search criterion;
- A hybrid reinforcement learning pipeline that jointly trains decomposition and completion within a unified policy, balancing structural exploration with stable proof generation;
- Extensive evaluation on three benchmarks, showing that a purpose-trained 8B-parameter model substantially outperforms frontier models and neural provers up to 84× larger.

2 Lean-based Code Verification

We formalize the automated code verification task in Lean 4 and analyze the core difficulties that motivate our hierarchical proof search approach.

2.1 Problem Formulation

We model automated code verification in Lean 4 using a Hoare-style specification framework. Concretely, we are given:

- a *program* C (a Lean definition or function),
- a *precondition* P and a *postcondition* Q, both expressed as Lean predicates.

The verification objective is to establish a Hoare-style triple

$$\{P\}\ C\ \{Q\},$$

which asserts that for any input x satisfying $P(x)$, the evaluation of $C(x)$ yields a result v such that $Q(x, v)$ holds. In the functional setting of Lean, C is typically a pure function, and the triple is encoded as a Lean proposition relating inputs and outputs of C. In our running example (Figure 1), $C =$ `FindSingleNumber`, P requires every element appears once or twice with exactly one unique element, and Q asserts the output is that unique element.

Formally, the input to the verification task is a tuple (C, P, Q) expressed in Lean, and the goal is to construct a machine-checkable proof term τ such that the Lean kernel accepts τ as a valid derivation of the verification goal G:

$$G\ :\ \forall x, P(x) \to Q(x, C(x)).$$

We assume that the program C and its associated predicates P and Q are already formalized in Lean; translating informal specifications into Lean is out of scope. The core challenge is therefore the synthesis of the proof term τ, which often involves quantifier elimination, complex induction, invariant generation, and the formulation of auxiliary assertions.

When we employ an LLM to automate this synthesis, the task reduces to a conditioned generation problem via the interactive proving protocol of Lean 4. In Lean's interactive proving model, proof construction proceeds step by step: at each step, the prover observes a *goal state*—comprising a set of local hypotheses (previously established facts) and a target proposition to be proved—and issues a *tactic* command. The Lean kernel checks the tactic, updates the proof state, and generates new subgoals if necessary; this loop continues until all goals are discharged or the attempt is abandoned. The LLM thus acts as a tactic generator: it receives the current goal state and produces a tactic sequence intended to close the goal. A generation is deemed successful if and only if the Lean kernel accepts the resulting proof and all subgoals are discharged.

2.2 Verification Paradigms and Why Lean

Modern program verification predominantly relies on SMT-based "auto-active" verifiers such as Dafny (Leino, 2010) and Verus (Lattuada et al., 2023). These tools translate high-level code and annotations into verification conditions (VCs) discharged by automated solvers such as CVC5 (Barbosa et al., 2022) or Z3 (De Moura & Bjørner, 2008). While highly effective for systems engineering, the reasoning process is largely mediated by external SMT solvers, which act as black boxes from the LLM's perspective. When a VC fails, the solver typically returns a generic counterexample or timeout rather than a structured diagnostic, making it difficult for an LLM agent to identify the root cause and iteratively refine its annotations.

We adopt Lean 4 (Moura & Ullrich, 2021) as our verification platform for LLM-driven proof automation. From a foundational standpoint, Lean offers two structural advantages well-suited to LLM-driven workflows: (i) *Small trusted kernel.* Lean relies on a minimal logical kernel as its sole verification authority, so even when an LLM produces unpredictable or creative proof attempts, correctness is guaranteed by a small, well-audited checker rather than a complex solver stack. (ii) *Rich library ecosystem.* Lean's dependent type theory supports higher-order and richly structured specifications, and libraries such as `Mathlib` (mathlib Community, 2020) and `CSLib` (Barrett et al., 2026) provide an extensive knowledge base of verified lemmas and tactics that an LLM can directly invoke, facilitating the encoding of properties that extend beyond first-order SMT reasoning.

More importantly, Lean exposes the entire proof construction process to the LLM agent. Unlike opaque VC failures in SMT-based systems, Lean provides explicit goal states, local hypotheses, and fine-grained compiler diagnostics at each proof step. This transparency enables the LLM to plan proof strategies by inspecting open goals, execute tactics with immediate feedback, and debug failed attempts using structured error messages. Such an interactive loop is particularly suitable for iterative, model-driven verification.

Additionally, recent cross-system evaluations corroborate this choice. The `AlgoVeri` benchmark (Zhao et al., 2026) translates the same set of algorithmic problems into Dafny, Verus, and Lean under aligned functional contracts, enabling a controlled comparison across verification paradigms. Proving performance turns out to be comparable across all three systems, and the primary bottleneck is shared: the absence of auxiliary lemmas and annotations needed to guide the automated proving. This suggests that the LLM-plus-Lean paradigm, while forgoing the raw automation of SMT solvers, can achieve competitive verification performance by leveraging the richer proof interaction that Lean affords.

2.3 Key Challenges

Despite recent advances in LLM-based theorem proving, automated code verification remains fundamentally harder than its mathematical counterpart. State-of-the-art planning-and-proving methods have achieved remarkable success in formal mathematics, reaching competition and IMO level performance (Hubert et al., 2025; Chen et al., 2025), yet their efficacy does not transfer to code verification. We identify three key challenges that explain this gap.

Ungrounded decomposition. Complex verification goals are seldom provable in a single pass; the prover must carefully plan its proof strategy and decompose the goal into auxiliary lemmas whose proofs collectively entail the top-level statement. In formal mathematics, LLMs acquire strong planning ability from extensive pretraining over textbooks, competition solutions, and online discussions, enabling them to propose meaningful decompositions even for challenging problems (Hubert et al., 2025; Chen et al., 2025). However, virtually no natural-language corpus describes how a program's specification is proved; reasoning about program correctness has no comparable textual footprint in LLM pretraining data. Without this grounding, frontier models frequently produce subgoals that are semantically invalid or no simpler than the original objective.

Concept proliferation. Even given a correct decomposition, proving each subgoal remains difficult because each program introduces a fresh conceptual universe. In formal mathematics, the relevant objects (real numbers, groups, combinatorial structures) form a relatively closed vocabulary supported by shared libraries such as Mathlib, and proofs can reuse a rich collection of existing lemmas. In contrast, every function definition in a program brings new computational behavior, control flow patterns, and data-structure invariants whose properties must be established from scratch without prior library support (Pei et al., 2023; Kamath et al., 2023). For example, the postcondition in Figure 1 relies on `filterlist`, a helper function defined within the program itself; no existing library contains lemmas about its behavior, so all necessary properties must be proved from scratch during verification.

Tactic distribution shift. Beyond the conceptual gap, the tactics needed to discharge code verification subgoals differ markedly from those prevalent in formal mathematics. Mathematical proofs rely heavily on algebraic normalization tactics such as `linarith`, `polyrith`, and `gcongr` over continuous domains. Code verification, by contrast, operates over discrete and recursive domains (e.g., machine integers, bit-vectors, lists, and graphs), requiring specialized automation such as `grind`, SMT-backed tactics, and computational reflection (`native_decide`). For instance, proving properties about `filterlist` in our running example requires reasoning about list induction and element membership, which is best handled by `grind` or `native_decide` rather than the algebraic tactics that dominate mathematical proofs. Models trained primarily on mathematical corpora often fail to invoke these program-specific tools correctly, struggling to close even well-formulated subgoals.

The compound search challenge. Together, these three challenges create a compound search problem with an exceptionally fragile search space: a well-crafted set of subgoals can collapse a complex verification condition into tractable obligations, while a slightly misspecified lemma renders the entire proof intractable. This sensitivity motivates our approach, which trains a domain-specific decomposer guided by a principled scoring mechanism to identify decompositions that are both semantically valid and tactically provable.

3 Hierarchical Verification Framework

Our method is organized around a single LLM that interacts with Lean in a planning-and-proving loop. At each interaction step, the policy either proposes a decomposition of the current goal (planning) or emits proof code/tactics to close a subgoal (proving). To make this loop trainable and searchable, we center the framework on a *decomposition score* that quantifies candidate lemma decompositions and serves consistently as both the training reward and the inference-time ranking criterion. We first define the score (section 3.1) and then describe the unified training and inference pipeline (section 3.2).

3.1 Score for Lemma Decomposition

Lemma decomposition is a critical bottleneck in Lean-based code verification: a good decomposition turns a global correctness theorem into a set of provable subgoals, whereas a poor one produces invalid, redundant, or equally intractable obligations. We therefore seek a *computable* score that correlates with downstream provability and can serve both as a training reward and as an inference-time ranking criterion. From the perspective of policy optimization, this score should be cheap to evaluate online, informative under partial progress, and consistent between training-time supervision and test-time search control. To this end, we define the score along two complementary axes: *(i) constructive justification* and *(ii) decomposition effectiveness*.

Constructive justification. A lemma decomposition is only meaningful if the proposed lemmas $(L_1, \ldots, L_k)$ logically entail the parent theorem (G). Specifically, a candidate decomposition must provide a proof reconstruction: a proof term π_{parent}, produced by the LLM as part of its decomposition output, such that Lean can verify the implication $(L_1 \wedge \cdots \wedge L_k) \Rightarrow G$. We require that π_{parent} explicitly invokes each proposed lemma, ensuring the decomposition is not merely a syntactic list but a constructively justified reduction of the original goal.

However, constructive justification alone does not guarantee the validity of the subgoals. A decomposition could be useless if a decomposed lemma is vacuous. For instance, an unprovable or false lemma (e.g., `l = [] ∧ l.length > 0`) can technically prove the parent theorem, but it shifts the burden to an impossible sub-task. To prevent the search from pursuing such unprovable branches, we integrate *quickcheck* as a semantic filter.

Quickcheck performs automated counterexample search: it randomly samples concrete inputs and checks whether each proposed lemma holds, discarding any lemma that fails. This validation serves as a binary gate: if a counterexample is found for any L_i, the entire decomposition is discarded before expensive proof search is attempted, ensuring that the framework only allocates search budget to subgoals that are both logically sufficient and empirically consistent.

Decomposition effectiveness. A decomposition is effective if it reduces the difficulty of the original verification task. In Hoare-triple terms, consider splitting $\{P\}\, C\, \{Q\}$ into $\{P\}\, C_1\, \{R\}$ and $\{R\}\, C_2\, \{Q\}$, where $C_1; C_2$ is a partition of the program C and R is an intermediate assertion characterizing the state between the two fragments. Such a decomposition is only productive if the resulting sub-goals are strictly simpler than the original, in both the logical assertions (R simpler than P, Q) and the programs (C_1, C_2 simpler than C).

To quantify this notion of simplicity, we observe that in Lean's abstract syntax tree (AST), both assertions and programs are uniformly expressed through operators: logical operators (connectives and quantifiers) and program operators (arithmetic, bitwise, or data-structure constructors). We define the structural difficulty $d(G)$ of a verification goal G as its *operator footprint*: the total number of logical and domain-specific operator occurrences in its AST. This metric captures reasoning cost because each operator corresponds to a specific class of proof obligations: reducing logical operators in R simplifies propositional reasoning, while reducing computational operators in C_1, C_2 simplifies symbolic execution.

We illustrate this with the running example (Figure 1). The goal `FindSingleNumber_spec` combines specification operators (∀, →, ∧, ∨, ∈, =) with program operators (`filterlist`, `List.count`, `List.map`, `List.length`), yielding $d(G) = 18$. A decomposition splits G into L_1: "`filterlist` on the unique element returns a singleton list" ($d(L_1) = 7$) and L_2: "every other element appears in `filterlist` exactly twice" ($d(L_2) = 8$). Each sub-lemma isolates a single property of `filterlist` with a smaller footprint, confirming a positive structural reduction.

3.2 Unified Policy Learning and Inference

The quality of a decomposition is quantified by a scalar score S that integrates both constructive justification and decomposition effectiveness. Crucially, the same score is used as the training reward and the inference-time ranking criterion, ensuring strict alignment between optimization and deployment. We implement two custom Lean tactics to compute it: `operatorcount`, computing the operator footprint of a goal, and `quickcheck`, performing automated counterexample search. Both tactics can be invoked at any proof state.

The score combines a binary validity gate with a continuous structural reduction ratio. A candidate decomposition is valid only if (i) the proposed lemmas collectively discharge G via proof reconstruction, and (ii) every lemma survives quickcheck without counterexamples. We encode this jointly as

$$v(L_1, \ldots, L_k; G) = \mathbb{1}_{\text{proof}}(L_1, \ldots, L_k; G) \cdot \prod_{i=1}^{k} \mathbb{1}_{\text{qc}}(L_i),$$

where $\mathbb{1}_{\text{proof}}(L_1, \ldots, L_k; G) = 1$ if proof reconstruction succeeds and 0 otherwise, and $\mathbb{1}_{\text{qc}}(L_i) = 1$ if no counterexample is found for L_i and 0 otherwise.

Algorithm 1 Hybrid RL with Online Lemma Collection

```
Require: Initial problem set $\mathcal{P}_0$, policy $\pi$, loss coefficient $\lambda$
1: for iterations $t = 1, 2, \ldots$ do
2:     Sample problem $G \sim \mathcal{P}_{t-1}$.
3:     Roll out $\pi$ to produce decomposed lemmas $\mathcal{L}_t := \{L_1, \ldots, L_k\}$.
4:     Compute reward $S = r(L_1, \ldots, L_k; G) \cdot v(L_1, \ldots, L_k; G)$ and GRPO objective $\mathcal{J}_{\text{GRPO}}$.
5:     Attempt completion of $\mathcal{L}_t$: first roll out $\pi$; if $\pi$ fails within $m$ attempts, fall back to an
       off-the-shelf model to produce $\mathcal{T}_t := \{\tau_1, \ldots, \tau_k\}$.
6:     Compute SFT objective $\mathcal{J}_{\text{SFT}}$ on completion trajectories $(\mathcal{L}_t, \mathcal{T}_t)$.
7:     Update $\pi$ with hybrid objective $\mathcal{J} = \mathcal{J}_{\text{GRPO}} + \lambda \cdot \mathcal{J}_{\text{SFT}}$.
8:     Augment problem set: $\mathcal{P}_t \leftarrow \mathcal{P}_{t-1} \cup \mathcal{L}_t$.
9: end for
```

Next, we quantify how much simpler the sub-goals are relative to the original. Let $d(G)$ denote the operator footprint of the original theorem and $d(L_i)$ the footprint of each decomposed lemma. A natural aggregate is $\max_i d(L_i)$, since the hardest subgoal governs overall difficulty. However, $\max(\cdot)$ is insensitive to partial progress: reducing one of two equally hard subgoals leaves the $\max(\cdot)$ unchanged, assigning no credit to a strictly improved decomposition. We instead adopt LogSumExp, which approximates the maximum while remaining sensitive to reductions in any individual subgoal:

$$\bar{d}(L_1, \ldots, L_k) = T \cdot \log \sum_{i=1}^{k} \exp\left(d(L_i)/T\right),$$

where T controls the smoothness. The structural reduction ratio is then

$$r(L_1, \ldots, L_k; G) = \max\left(1 - \frac{\bar{d}(L_1, \ldots, L_k)}{d(G)}, 0\right).$$

The final decomposition score is $S = r(L_1, \ldots, L_k; G) \cdot v(L_1, \ldots, L_k; G)$. Because this score is used identically during training (as the reward) and inference (as the ranking criterion), policy learning and tree-search control are optimized against exactly the same objective.

3.2.1 *Training Pipeline*

We train the same autoregressive policy for both planning and proving in two stages: (1) supervised initialization with scaffolded data; (2) hybrid reinforcement learning that optimizes decomposition while stabilizing completion.

Supervised fine-tuning (SFT). Using frontier models (GPT-5.2 and Gemini-3-Flash), we generate scaffolded trajectories for both lemma decomposition and lemma completion. This process yields 281K decomposition input–output pairs and 151K completion pairs. We merge the two types of trajectories and fine-tune a Qwen-3-8B backbone to obtain an initial policy π_0. At this stage, the model already supports both decomposition and completion behaviors under the same prompting scaffolding.

Hybrid reinforcement learning (RL). A central challenge in jointly optimizing decomposition and completion is reward mismatch: decomposition receives a dense, continuous score in $[0, 1]$, while completion yields only a sparse binary signal (proof accepted or rejected). Naively merging these objectives causes decomposition gradients to dominate and completion proficiency to stagnate.

We resolve this by decoupling the learning objectives while maintaining a shared parameter space. Decomposition is optimized via group relative policy optimization (GRPO), utilizing the structural score S to drive exploration and curriculum expansion. Completion is stabilized through supervised replay of high-quality proof trajectories (generated by either the policy itself or frontier models), avoiding reliance on brittle binary signals.

Algorithm 1 summarizes the full loop. For completion, we adopt a *policy-first* strategy: π attempts each lemma first, and a frontier model is invoked only when π fails within m

Algorithm 2 Hierarchical Inference (Single Run)

```
Require: Theorem $G$, policy $\pi$
 1: // Stage 1: Lemma Decomposition
 2: Initialize open goal set $\mathcal{O} \leftarrow \{G\}$.
 3: for decomposition iterations $i = 1, 2, \ldots$ do
 4:     Select the highest-scoring goal $g \in \mathcal{O}$ as the decomposition target.
 5:     Roll out $\pi$ to decompose $g$ into sub-lemmas $\{L_1, \ldots, L_k\}$.
 6:     Verify proof reconstruction and quickcheck for each $L_i$; if failed, discard and retry.
 7:     Update $\mathcal{O}$: remove $g$, add $\{L_1, \ldots, L_k\}$; compute scores for new goals.
 8: end for
 9: // Stage 2: Lemma Completion
10: Collect remaining leaf lemmas $\mathcal{L} = \{L_1, \ldots, L_m\}$ from $\mathcal{O}$.
11: for completion iterations $j = 1, 2, \ldots$ do
12:     Roll out $\pi$ to generate candidate proofs for each unclosed $L_i \in \mathcal{L}$.
13:     Refine proofs based on Lean compiler feedback.
14:     if all lemmas closed then return success.
15: end for
```

attempts, ensuring that π is exposed to its own successful completions for replay. Additionally, successfully decomposed lemmas are fed back into the problem set, providing an ever-expanding curriculum of progressively simpler subgoals.

3.2.2 *Inference Pipeline*

Inference proceeds in two sequential stages: the decomposition stage runs first, iteratively breaking down the original goal until a budget is exhausted, after which the completion stage takes over and attempts to prove each remaining leaf lemma. We use the term *iteration* to refer to a single decomposition or completion step within its respective stage.

Lemma Decomposition. At each decomposition iteration, the policy selects the open goal with the highest operator footprint as the decomposition target, concentrating search effort on the most complex remaining obligation. The policy then rolls out a decomposition for the selected goal; the result is verified via proof reconstruction and quickcheck, and discarded if either check fails. Upon acceptance, the original goal is replaced by its sub-lemmas in the goal set, and scores are computed for the new goals. This process repeats until all goals are discharged or a computational budget is exhausted.

Lemma Completion. After the decomposition stage concludes, the remaining open lemmas are passed to the completion stage. At each completion iteration, the policy generates a candidate proof for each unclosed lemma and submits it to the Lean compiler. If compilation fails, the error diagnostics are fed back to the policy, which revises the proof accordingly. This generate-and-refine loop repeats until all proofs are accepted or the budget is exhausted.

Algorithm 2 summarizes the full inference pipeline. Beyond increasing the per-run budget, performance can be further improved by launching k independent runs in parallel and accepting the first successful proof (pass@k).

4 Experiments

We evaluate our framework through four research questions:

- **RQ1** (Overall Effectiveness): How does our framework compare to frontier reasoning models and advanced neural provers on code verification benchmarks?
- **RQ2** (Inference-Time Scaling): How does proving performance scale with increased inference-time compute (number of parallel runs k and per-run search budget)?
- **RQ3** (Decomposition Analysis): How do the two decomposition criteria, constructive justification and decomposition effectiveness, contribute to downstream proving success?

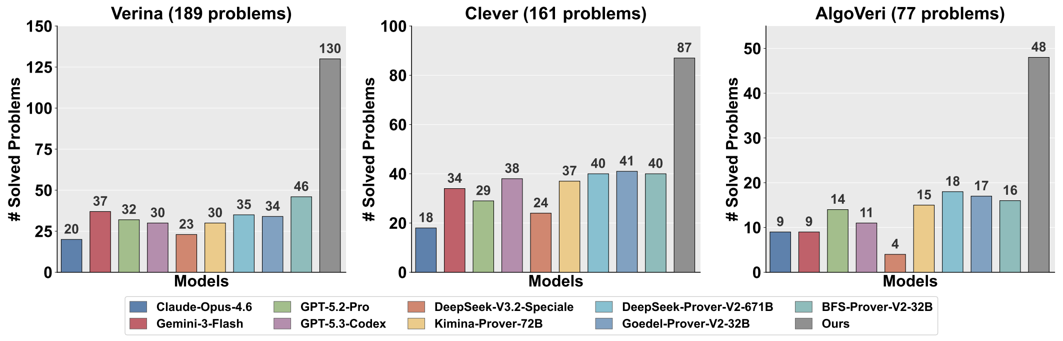

Figure 3: Number of solved problems by baselines and our framework across three benchmarks. Baselines are evaluated with parallel generation under a Pass@128 budget; our method operates under a search-based inference setting using Göedel-Code-Prover-8B. Our framework outperforms all baselines by a substantial margin across every benchmark.

- **RQ4** (Ablation Study): How much do the trained decomposition and completion modules each contribute, and does joint training yield synergistic gains?

4.1 Experimental Setup

Benchmarks. We evaluate on three Lean 4 code verification benchmarks. Verina (Ye et al., 2025) contains 189 tasks from introductory programming exercises, covering code generation, specification synthesis, and proof generation. Clever (Thakur et al., 2025) provides 161 problems with formal specifications designed to avoid test-case leakage and trivial solutions. AlgoVeri (Zhao et al., 2026) comprises 77 classical algorithms with identical functional contracts across Dafny, Verus, and Lean; we adopt its Lean subset. Among these, Verina ships complete programs with preconditions, implementations, and postconditions, forming a self-contained evaluation suite. Clever and AlgoVeri only release specifications without reference implementations; we therefore prompt GPT-5.2 to generate the code and manually verify the results. Together, the three benchmarks yield 427 verification tasks spanning straightforward functional correctness to deep algorithmic reasoning.

Baselines. We compare against two categories of baselines. (1) *Frontier reasoning models*: Claude-Opus-4.6, Gemini-3-Flash, GPT-5.2-Pro, GPT-5.3-Codex, and Deepseek-V3.2-Speciale, each prompted to produce a complete proof in a single pass. (2) *Neural provers*: Kimina-Prover-72B (Wang et al., 2025), DeepSeek-Prover-V2-671B (Ren et al., 2025), Goedel-Prover-V2-32B (Lin et al., 2025), and BFS-Prover-V2-32B (Xin et al., 2025b), which are trained on large-scale formal proof corpora; among them, BFS-Prover employs best-first tree search over tactic sequences, while the others generate whole proofs. All baselines use default inference configurations and report pass@128.

Implementation details. We fine-tune a Qwen-3-8B backbone (Yang et al., 2025) following our training pipeline; we refer to the resulting policy as *Göedel-Code-Prover-8B*. SFT uses 281K decomposition and 151K completion trajectories, followed by hybrid RL. At inference, the decomposition stage runs up to 128 iterations or until the number of open lemmas exceeds 32; the completion stage refines each lemma for up to 128 iterations. Additional training hyperparameters, prompt templates, as well as inference configuration are provided in Appendix B.

Metrics. We report the *prove success rate*: the fraction of problems for which a complete, Lean-verified proof is produced within the inference budget. To ensure soundness, we only admit the standard axioms `propext`, `Classical.choice`, and `Quot.sound`; proofs invoking `Lean.ofReduceBool` or `Lean.trustCompiler` are rejected, as these bypass the kernel's deductive checking and could mask unsound reasoning.

Table 1: Statistics of proved problems: mean and standard deviation of lemma count and proof length (lines of code).

Statistics	Verina	Clever	AlgoVeri	Total
Lemma count (mean)	17.02	12.13	8.48	14.38
Lemma count (std)	11.79	10.57	10.97	11.97
Proof length (mean)	167.11	137.78	129.87	154.28
Proof length (std)	108.50	97.79	83.03	103.11

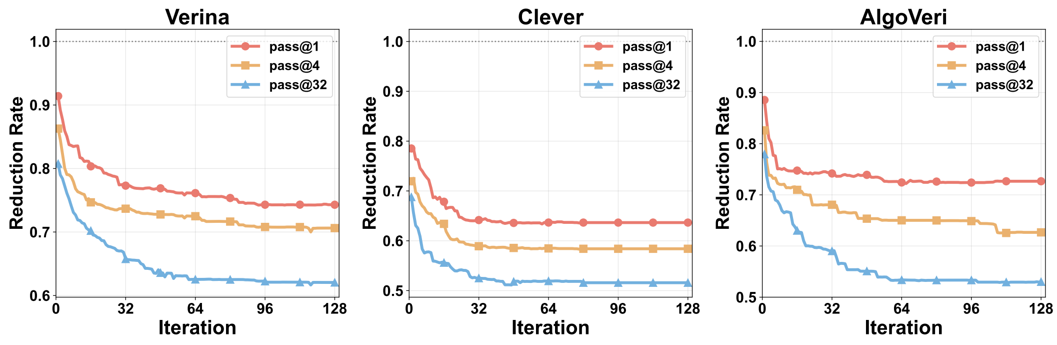

Figure 4: Decomposition reduction rate vs. iterations across three benchmarks. Lower values indicate more aggressive goal simplification.

4.2 RQ1: Overall Effectiveness

As shown in Figure 3, our framework achieves prove success rates of 68.8%, 54.0%, and 62.3% on Verina, Clever, and AlgoVeri, respectively, yielding an overall rate of 62.0% across all 427 tasks. In addition, quickcheck disproves 23, 10, and 14 problems on Verina, Clever, and AlgoVeri by producing concrete counterexamples.[1] Among frontier reasoning models, the strongest (GPT-5.3-Codex) reaches only 23.6% on Clever and below 20% on Verina and AlgoVeri, confirming that single-pass proof generation is insufficient for non-trivial verification. Among neural provers, BFS-Prover-V2-32B achieves the best overall score of 23.8%, yet our framework surpasses it by 38.2 percentage points.

Table 1 further characterizes the proofs our framework produces. On average, each verified problem requires 8 to 17 auxiliary lemmas and over 130 lines of proof code, with the longest proofs reaching 680, 559, and 534 lines on Verina, Clever, and AlgoVeri, reflecting the depth of reasoning our framework can sustain.

Response to RQ1: our framework achieves a **2.6**$\times$ improvement over the strongest baseline across all benchmarks. Notably, this is achieved with an 8B model, far smaller than the 32B–671B neural provers it surpasses, demonstrating that hierarchical decomposition enables a small, purpose-trained policy to tackle complex verification tasks.

4.3 RQ2: Inference-Time Scaling

Our framework exposes two orthogonal scaling axes: the *per-run search budget* (number of iterations within each decomposition or completion stage) and the *parallel sampling budget* (pass@k, where k independent runs are launched and the first successful proof is accepted). We examine how performance varies along each axis.

As shown in Figure 4, the decomposition reduction rate decreases steadily as iterations progress, indicating that the framework progressively identifies more effective goal sim-

[1]Quickcheck is also applied directly to the top-level goal. When a counterexample is found, it is provided to the LLM, which then uses it as evidence to disprove the problem.

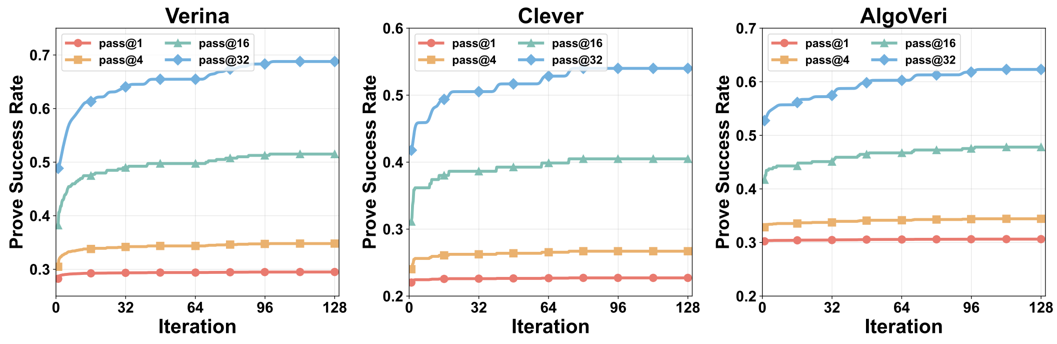

Figure 5: Prove success rate vs. completion iterations under different pass@k budgets.

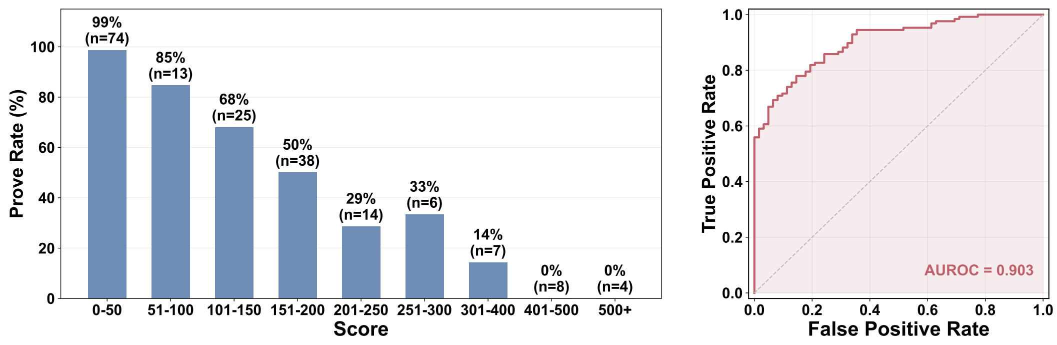

Figure 6: Decomposition score vs. prove rate on Verina. The score reliably separates provable from unprovable instances (AUROC = 0.903).

plifications with additional compute. Increasing the pass@k budget further lowers the reduction rate across all three benchmarks, confirming that broader parallel search at the decomposition stage produces structurally simpler sub-goals.

Figure 5 shows that the prove success rate grows steadily with both more completion iterations and larger pass@k, demonstrating clear inference-time scaling across all benchmarks. The persistent gap between pass@1 and pass@32 suggests that the framework benefits substantially from parallel search and has not yet saturated its scaling potential within the evaluated budget. By contrast, the pass@k curves of baseline verifiers plateau rapidly (see Appendix C.1), indicating that brute-force sampling without hierarchical decomposition yields diminishing returns.

Response to RQ2: performance scales consistently along both axes: more iterations improve decomposition quality, and larger k broadens search coverage, with no sign of saturation.

4.4 RQ3: Decomposition Analysis

Table 2 quantifies how each criterion filters out unproductive decompositions. Among 32 parallel runs per problem, 31.8%–46.4% are discarded entirely because at least one proposed lemma fails quickcheck (QC Failed). Within the surviving runs, proof reconstruction still rejects 44.9%–59.4% of individual decomposition iterations (Proof Failed), confirming that constructing a logically sound reduction remains the dominant bottleneck. The two criteria thus act as successive filters: quickcheck eliminates semantically invalid runs early, and proof reconstruction catches logically unsound iterations within the remaining runs.

Figure 6 further confirms that the combined decomposition score is a strong predictor of downstream provability: on Verina, higher scores correlate with higher prove rates, achieving an AUROC of 0.903. This means that ranking candidate decompositions by S at inference time steers search toward decompositions that are more likely to yield complete

Table 2: Decomposition failure rates. Proof Failed: % of iterations where proof reconstruction rejects the decomposition. QC Failed: % of runs where at least one lemma fails quickcheck.

Dataset	Proof Failed	QC Failed
Verina	59.4	46.4
Clever	44.9	31.8
AlgoVeri	52.8	32.6

Table 3: Ablation study on each component. We progressively replace each component with our trained model to isolate its contribution.

Decomposition	**Completion**	Verina
—	Gemini-3-Flash	26.4
GPT-5.2-Pro	Gemini-3-Flash	54.4
Ours	Gemini-3-Flash	58.2
GPT-5.2-Pro	Ours	59.2
Ours	Ours	**68.7**

proofs, validating its use as both a training reward and an inference-time selection. We also verify this correlation on our training data (see Appendix C.2): the same positive trend holds, confirming that the score captures a robust, transferable signal of proof tractability.

Response to RQ3: both decomposition criteria are effective at filtering unproductive decompositions, and the combined score S is highly predictive of proving success (AUROC = 0.903), enabling it to steer inference-time search toward provable decompositions.

4.5 RQ4: Ablation Study

Table 3 isolates the contributions of our trained decomposition and completion modules by swapping each with baseline alternatives on Verina. Without decomposition, directly prompting Gemini-3-Flash for whole-proof generation achieves only 26.4%, confirming that hierarchical decomposition is essential. Adding GPT-5.2-Pro decomposition raises the rate to 54.4%; replacing it with our trained decomposer yields a further gain to 58.2%, showing that our policy learns strategies beyond what frontier models achieve via prompting. Similarly, replacing Gemini-3-Flash completion with our trained completer (GPT-5.2-Pro + Ours: 59.2%) shows that domain-specific completion training is at least as valuable as improved decomposition. Combining both trained components achieves 68.7%, surpassing every mixed configuration and confirming synergistic gains from joint training: the decomposer and completer co-adapt to each other's strengths.

We also visualize how training shifts the decomposition score distribution in Appendix C.3: after training, scores shift markedly toward higher values across all benchmarks, confirming that our pipeline teaches the model to produce more sound and effective decompositions.

Response to RQ4: both modules contribute substantially, and their combination yields gains that exceed the sum of individual improvements. Hierarchical decomposition is the single most impactful design choice.

5 Related work

LLM-based theorem proving and the math-code gap. Language models have achieved remarkable progress in formal theorem proving, predominantly in mathematics. Starting from tactic-level generation (Polu & Sutskever, 2020) and retrieval-augmented proving (Yang et al., 2024b), the field has advanced through tree-search methods such as HyperTree Proof Search (Lample et al., 2022) and, more recently, large-scale neural provers including DeepSeek-Prover (Xin et al., 2024), Kimina-Prover (Wang et al., 2025), BFS-Prover (Xin et al., 2025a), and Goedel-Prover (Lin et al., 2025). Among these, Goedel-Prover is particularly noteworthy for its curriculum-driven synthetic data pipeline, which iteratively generates and verifies proof data to bootstrap prover capability.

However, the success of these systems does not directly transfer to code verification. In mathematics, a well-curated library such as `Mathlib` (mathlib Community, 2020) provides a dense, reusable inventory of lemmas, and the space of core concepts is comparatively concentrated. Code verification, by contrast, lacks a comparable lemma ecosystem: each program introduces its own function definitions, data-type invariants, and control-flow structures, effectively defining a fresh set of domain-specific concepts per task. This *concept proliferation* renders the proof distribution far more heterogeneous and precludes reliance on a fixed library of reusable lemmas, creating a substantial generalization gap for methods trained primarily on mathematical corpora.

Automated code verification with LLMs. Traditional program verification relies on SMT-based auto-active tools such as Dafny (Leino, 2010) and Verus (Lattuada et al., 2023), where correctness is established by discharging verification conditions through automated solvers. Recent efforts have explored using LLMs to assist these workflows, for example by generating loop invariants or annotations (Kamath et al., 2023; Pei et al., 2023). In the Lean ecosystem, Baldur (First et al., 2023) demonstrated whole-proof generation for software verification, and Rango (Thompson et al., 2024) introduced adaptive retrieval-augmented proving. However, these approaches either operate in a flat, single-pass generation mode or rely on tactic-level tree search without explicit goal decomposition. Our framework differs by introducing a planning layer that decomposes verification goals into structurally simpler sub-lemmas before invoking tactic-level proving, enabling the system to tackle problems whose complexity exceeds what flat generation can handle.

Hierarchical proof search and reinforcement learning. Decomposing complex goals into simpler sub-problems has a long history in automated reasoning (Li et al., 2024). In neural theorem proving, Draft-Sketch-Prove (Jiang et al., 2022) generates informal proof sketches before formalizing them, and subgoal-based approaches (Achim et al., 2025; Ren et al., 2025) have explored decomposition for mathematical theorems. AlphaProof (Hubert et al., 2025) achieved IMO-gold-level performance by combining reinforcement learning with a planning-and-proving paradigm. Our work adapts this paradigm to code verification, where useful decompositions differ fundamentally: intermediate assertions must capture semantic program properties (such as inductive invariants and termination measures) rather than syntactic refinements of the goal statement. On the reinforcement learning side, prior work has applied expert iteration (Polu & Sutskever, 2020), off-policy RL with tree search (Xin et al., 2025b), and process reward models (Lightman et al., 2024) to improve proof search. A persistent challenge is reward sparsity: binary proof success/failure signals provide limited gradient information. Our hybrid RL approach addresses this by introducing a continuous decomposition score as a dense reward for the planning stage, while stabilizing the proving stage through supervised replay.

6 Conclusion

We presented a hierarchical proof search framework for automated code verification in Lean 4 that decomposes complex verification goals into structurally simpler sub-lemmas before applying tactic-level proving. At its core is a decomposition score combining constructive justification with structural effectiveness, serving as both the training reward and the inference-time selection criterion. We train a single unified policy for both decomposition and completion via supervised initialization followed by hybrid reinforcement learning.

Experiments on three code verification benchmarks (Verina, Clever, and AlgoVeri) demonstrate that our framework achieves a 62.0% overall prove success rate, a 2.6$\times$ improvement over the strongest baseline, using an 8B-parameter model that outperforms neural provers up to 84$\times$ larger. Ablation studies confirm that both trained decomposition and completion contribute synergistically, and that the decomposition score is highly predictive of downstream provability.

Limitations and future work. Our framework currently assumes that programs and specifications are already formalized in Lean; extending the pipeline to handle informal specifications or automatic translation from other languages remains an important direction. The decomposition score relies on syntactic operator counting as a proxy for proof difficulty; incorporating semantic complexity measures or learned difficulty predictors could yield more informative signals. Finally, scaling to larger codebases with inter-procedural dependencies and richer data structures presents both an engineering and a research challenge that we leave to future work.

References

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, et al. Aristotle: Imo-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*, 2025.

Ali Al-Kaswan, Claudio Spiess, Prem Devanbu, Arie van Deursen, and Maliheh Izadi. Model see, model do? exposure-aware evaluation of bug-vs-fix preference in code llms. *arXiv preprint arXiv:2601.10496*, 2026.

Maha Alharbi and Mohammad Alshayeb. Automatic code generation techniques: A systematic literature review. *Automated Software Engineering*, 33(1):4, 2026.

Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442. Springer, 2022.

Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. CSLib: The lean computer science library. *arXiv preprint arXiv:2602.04846*, 2026.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience. *arXiv preprint arXiv:2512.17260*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 268–279, 2000.

Edmund M Clarke. Model checking. In *International conference on foundations of software technology and theoretical computer science*, pp. 54–56. Springer, 1997.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. 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*, pp. 1229–1241, 2023.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pp. 1–3, 2025.

Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. *arXiv preprint arXiv:2303.11455*, 2023.

Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

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

Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Aitor Roziere, Thibaut Lavril, Thomas Scialom, and Gabriel Synnaeve. HyperTree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems*, 2022.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1): 286–315, 2023.

Lean Prover Community. plausible. https://github.com/leanprover-community/plausible, 2024. GitHub repository.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pp. 348–370. Springer, 2010.

Caihua Li, Lianghong Guo, Yanlin Wang, Daya Guo, Wei Tao, Zhenyu Shan, Mingwei Liu, Jiachi Chen, Haoyu Song, Duyu Tang, et al. Advances and frontiers of llm-based issue resolution in software engineering: A comprehensive survey. *arXiv preprint arXiv:2601.11655*, 2026.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pp. 367–381, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pp. 27496–27520. PMLR, 2023.

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

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally verified code generation. *arXiv preprint arXiv:2505.13938*, 2025.

Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. *arXiv preprint arXiv:2412.14063*, 2024.

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

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *arXiv preprint arXiv:2504.14655*, 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

Ran Xin, Chenguang Xi, Jie Yang, Feng Chen, Hang Wu, Xia Xiao, Yifan Sun, Shen Zheng, and Ming Ding. 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)*, pp. 32588–32599, 2025a.

Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. Scaling up multi-turn off-policy rl and multi-agent tree search for llm step-provers. *arXiv preprint arXiv:2509.06493*, 2025b.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024a.

Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalapathy, Peiyang Song, Shixing Yu, Maruan Al-Shedivat, Jing Lei, and Eric P Xing. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems*, 2024b.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms. *arXiv preprint arXiv:2602.09464*, 2026.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A Complete Proof for the Running Example

Below is the complete machine-checked proof for `FindSingleNumber_spec` (the running example from Figure 1), as synthesized by our framework. The proof decomposes the top-level theorem into six auxiliary lemmas covering list filtering, element counting, uniqueness, and the recursive scanner, totaling over 130 lines of proof code. Each block below corresponds to one lemma or the final theorem.

```
-- Precondition and postcondition (specification)
def FindSingleNumber_precond (nums : List Int) : Prop :=
  let numsCount := nums.map (fun x => nums.count x)
  numsCount.all (fun c => c = 1 ∨ c = 2)
    ∧ numsCount.count 1 = 1

def FindSingleNumber_postcond (nums : List Int)
    (result : Int)
    (h : FindSingleNumber_precond nums) : Prop :=
  nums.length > 0
    ∧ (filterlist result nums).length = 1
    ∧ ∀ (x : Int), x ∈ nums →
        x = result ∨ (filterlist x nums).length = 2
```

```
-- Helper function used in the specification
def filterlist (x : Int) (nums : List Int) : List Int :=
  let rec aux (lst : List Int) : List Int :=
    match lst with
    | []      => []
    | y :: ys => if y = x then y :: aux ys else aux ys
  aux nums
```

```
-- Lemma 1: filterlist length equals List.count
lemma length_filterlist_eq_count (x : Int) (nums : List Int) :
    (filterlist x nums).length = nums.count x := by
  classical
  have aux_eq : ∀ lst : List Int,
      filterlist.aux x lst = lst.filter (fun y => y = x) := by
    intro lst; induction lst with
    | nil => simp [filterlist.aux]
    | cons y ys ih =>
        by_cases h : y = x <;> simp [filterlist.aux, ih, h]
  have eq_filter :
      filterlist x nums = nums.filter (fun y => y = x) :=
    by simp [filterlist, aux_eq]
  simpa [eq_filter] using
    (List.count_eq_length_filter (l := nums) (a := x)).symm
```

```
-- Lemma 2: the precondition guarantees a unique element exists
lemma exists_count_eq_one (nums : List Int)
    (h_precond : FindSingleNumber_precond nums) :
    ∃ x ∈ nums, nums.count x = 1 := by
  classical
  unfold FindSingleNumber_precond at h_precond
  rcases h_precond with ⟨_hall, huniq⟩
  have hmem : 1 ∈ (nums.map (fun x => nums.count x)) := by
    by_contra hnot
    have : (nums.map (fun x => nums.count x)).count 1 = 0 :=
      by simp [List.count_eq_zero, hnot]
    exact Nat.one_ne_zero (by simpa [this] using huniq)
  rcases List.mem_map.mp hmem with ⟨x, hx, hxcount⟩
  exact ⟨x, hx, by simpa using hxcount⟩
```

```
-- Lemma 3: the recursive scanner returns a unique element
lemma findUnique_returns_length_one (nums : List Int) :
    ∀ remaining : List Int,
      (∃ u ∈ remaining, (filterlist u nums).length = 1) →
        (filterlist (FindSingleNumber.findUnique
          (nums := nums) remaining) nums).length = 1 := by
  classical
  intro remaining; induction remaining with
  | nil => intro hex; rcases hex with ⟨u, hu, _⟩; cases hu
  | cons x xs ih =>
      intro hex; simp [FindSingleNumber.findUnique]
      by_cases hx1 : (filterlist x nums).length = 1
      · simp [hx1]
      · have hex' :
            ∃ u ∈ xs, (filterlist u nums).length = 1 := by
          rcases hex with ⟨u, hu_mem, hu_len⟩; simp at hu_mem
          cases hu_mem with
          | inl hu_eq =>
              subst hu_eq; exact False.elim (hx1 hu_len)
          | inr hu_mem_xs =>
              exact ⟨u, hu_mem_xs, hu_len⟩
        simpa [hx1] using ih hex'
```

```
-- Lemma 4: the algorithm returns a value with count = 1
lemma FindSingleNumber_result_characterization
    (nums : List Int)
    (h_precond : FindSingleNumber_precond nums) :
    nums.count (FindSingleNumber nums h_precond) = 1 := by
  classical
  rcases exists_count_eq_one nums h_precond
    with ⟨u, hu_mem, hu_count⟩
  have hu_len : (filterlist u nums).length = 1 :=
    by simp [length_filterlist_eq_count, hu_count]
  have hlen := by simpa [FindSingleNumber] using
    findUnique_returns_length_one nums nums
      ⟨u, hu_mem, hu_len⟩
  simpa [length_filterlist_eq_count] using hlen
```

```
-- Lemma 5: uniqueness of the element with count = 1
lemma precond_unique_count_one (nums : List Int)
    (h_precond : FindSingleNumber_precond nums)
    {a b : Int} (ha : a ∈ nums) (hb : b ∈ nums)
    (hca : nums.count a = 1) (hcb : nums.count b = 1) :
    a = b := by
  classical
  unfold FindSingleNumber_precond at h_precond
  rcases h_precond with ⟨_hall, huniq⟩
  by_contra hne
  let p : Int → Prop := fun x => nums.count x = 1
  have haF : a ∈ nums.filter p :=
    List.mem_filter.2 ⟨ha, by simp [p, hca]⟩
  have hbF : b ∈ nums.filter p :=
    List.mem_filter.2 ⟨hb, by simp [p, hcb]⟩
  have two_le {ff : Type} {l : List ff} {a b : ff}
      (ha : a ∈ l) (hb : b ∈ l) (hne : a ≠ b) :
      2 ≤ l.length := by
    induction l with
    | nil => cases ha
    | cons c tl ih =>
        have ha' : a = c ∨ a ∈ tl := by simpa using ha
        have hb' : b = c ∨ b ∈ tl := by simpa using hb
        cases ha' with
```

```
          | inl hac => cases hb' with
            | inl hbc =>
                exact (hne (by simpa [hac] using hbc.symm)).elim
            | inr hb_tl =>
                exact Nat.succ_le_succ
                  (Nat.succ_le_of_lt (List.length_pos_of_mem hb_tl))
          | inr ha_tl => cases hb' with
            | inl hbc =>
                exact Nat.succ_le_succ
                  (Nat.succ_le_of_lt (List.length_pos_of_mem ha_tl))
            | inr hb_tl =>
                exact le_trans (ih ha_tl hb_tl) (Nat.le_succ tl.length)
  have hlen_ge : 2 ≤ (nums.filter p).length :=
    two_le haF hbF hne
  have count_map_eq_aux : ∀ l : List Int,
      (l.map (fun x => nums.count x)).count 1 =
        (l.filter (fun a => nums.count a = 1)).length := by
    intro l; induction l with
    | nil => simp
    | cons a l ih =>
        by_cases h : nums.count a = 1 <;> simp [p, h, ih]
  have hlen_eq_one : (nums.filter p).length = 1 := by
    simp only [p]
    exact (count_map_eq_aux nums).symm.trans huniq
  exact Nat.not_succ_le_self 1
    (by simpa [hlen_eq_one] using hlen_ge)
```

```
-- Lemma 6: non-unique elements have count = 2
lemma count_eq_two_of_ne_of_count_eq_one
    (nums : List Int)
    (h_precond : FindSingleNumber_precond nums)
    (r x : Int) (hx : x ∈ nums)
    (hcr : nums.count r = 1) (hxr : x ≠ r) :
    nums.count x = 2 := by
  classical
  unfold FindSingleNumber_precond at h_precond
  rcases h_precond with ⟨hall, huniq⟩
  have precond_counts (x' : Int) (hx' : x' ∈ nums) :
      nums.count x' = 1 ∨ nums.count x' = 2 := by
    have hall' : ∀ n ∈ (nums.map fun y => nums.count y),
        (n = 1 ∨ n = 2) := by
      simpa [List.all_eq_true] using hall
    exact hall' (nums.count x')
      (List.mem_map.mpr ⟨x', hx', rfl⟩)
  have h_precond' : FindSingleNumber_precond nums := by
    unfold FindSingleNumber_precond; exact ⟨hall, huniq⟩
  have mem_of_one (a : Int) (h : nums.count a = 1) :
      a ∈ nums := by
    by_contra hmem
    have : nums.count a = 0 := by
      simp [List.count_eq_zero, hmem]
    exact Nat.zero_ne_one (by simpa [this] using h)
  cases precond_counts x hx with
  | inl h1 =>
      exact (hxr (precond_unique_count_one nums h_precond'
        hx (mem_of_one r hcr) h1 hcr)).elim
  | inr h2 => exact h2
```

```
-- Main theorem: combining all lemmas
theorem FindSingleNumber_spec_satisfied
    (nums : List Int)
    (h_precond : FindSingleNumber_precond nums) :
```

```
    FindSingleNumber_postcond nums
      (FindSingleNumber nums h_precond) h_precond := by
  classical
  constructor
  · by_contra h0
    have hnil : nums = [] := by
      simpa using List.length_eq_zero_iff.mp
        (Nat.eq_zero_of_not_pos h0)
    subst hnil
    simp [FindSingleNumber_precond] at h_precond
  constructor
  · simpa [length_filterlist_eq_count] using
      FindSingleNumber_result_characterization nums h_precond
  · intro x hx
    by_cases h : x = FindSingleNumber nums h_precond
    · exact Or.inl h
    · have hcount :=
        count_eq_two_of_ne_of_count_eq_one nums h_precond
          (FindSingleNumber nums h_precond) x hx
          (FindSingleNumber_result_characterization
            nums h_precond) h
      exact Or.inr
        (by simp [length_filterlist_eq_count, hcount])
```

B Implementation Details

Custom Lean Tactics. We implement two custom Lean tactics used by the decomposition scoring mechanism. `operatorcount` traverses the goal's abstract syntax tree and returns the total number of logical operators (connectives, quantifiers) and domain-specific program operators (arithmetic, bitwise, data-structure constructors), excluding variable references and type annotations. `quickcheck` extends Lean's built-in `Plausible` library (Lean Prover Community, 2024): given a universally quantified lemma, it randomly samples concrete inputs (up to 1000 trials) and evaluates the lemma via native execution, reporting a counterexample if one is found. Both tactics are registered as Lean meta-programs and can be invoked at any proof state during decomposition verification.

Data Curation. We collect 5K unique problems from LeetCode (Xia et al., 2025) and OpenCodeInstruct (Ahmad et al., 2025). Each problem is automatically formalized into Lean using GPT-5.2. The formalizations undergo iterative refinement until they pass both syntactic validation and quickcheck filtering, ensuring empirical consistency. To prevent data contamination, we remove all problems whose title or description has an exact match with any problem in the Verina, Clever, or AlgoVeri evaluation benchmarks.

Trajectory Filtering. We apply different filtering criteria for the two trajectory types. For decomposition trajectories, we retain only those that are constructively justified (the proposed lemmas provably entail the parent theorem) and achieve a positive structural reduction (the decomposition score strictly decreases). For completion trajectories, we retain only those whose final proof is accepted by the Lean kernel.

Supervised Fine-Tuning. We fine-tune Qwen3-8B (Yang et al., 2025) using LLaMA-Factory (Zheng et al., 2024) for three epochs on a mixture of decomposition and proof completion trajectories. Training uses a learning rate of 1×10^{-5} with a cosine schedule and a warmup ratio of 0.1. The maximum context length is 10,240 tokens with sequence packing enabled. We use a batch size of 128 with sequence packing.

Reinforcement Learning. We apply GRPO (Shao et al., 2024) using the `verl` framework (Sheng et al., 2024) for 100 training steps across 4×4 GPUs. The learning rate is 5×10^{-6} with cosine decay, with a clip ratio of $[0.2, 0.28]$ and a sampling temperature of 0.9. We use a batch size of 64 prompts with n=8 parallel generations per prompt, with and a

Decomposition Prompt Template

System: You are an expert in Lean 4 theorem proving. Your task is to decompose a theorem into smaller, provable lemmas.
Think step by step. Before writing code, first analyze the theorem structure and explain your decomposition strategy in a `<reasoning>` block.
Decomposition Strategies:

- **Structural Decomposition** — Match goal structure: split $P \wedge Q$ into separate lemmas; prove one side of $P \vee Q$; find witness for $\exists x, P(x)$.
- **Induction Decomposition** — For recursive structures: base case + inductive step as separate lemmas.
- **Rewrite Chain** — For equational goals $A = D$: break into $A = B$, $B = C$, $C = D$.
- **Case Analysis** — Split by `Decidable`, `match`, or `if-then-else`; each branch becomes a separate lemma.

Output Requirements: (1) Wrap analysis in `<reasoning>...</reasoning>` tags before any code; (2) Helper lemmas use `lemma` keyword with `sorry` body; (3) Main theorem uses `theorem` keyword and proves using the lemmas (no `sorry`).

User: Decompose the theorem {`theorem_name`}.
{`formal_problem`}

Figure 7: Prompt template for the decomposition stage. The model receives the target theorem name and the full Lean formalization, and produces a reasoning trace followed by helper lemmas and a proof of the main theorem.

Completion Prompt Template

System: Lean 4 proof assistant. Complete the proof by filling in the `sorry` placeholder with valid tactics.
Important: Output reasoning in a `reasoning` block before any diff blocks.
Recommended tactics (try in order): `grind` (powerful automation), `aesop` (general automation), `simp_all` (aggressive simplification), `omega` (linear arithmetic), `by_cases` (case split), `induction` (induction with auto finish).
Output format: SEARCH/REPLACE diff blocks.

User: Complete the current proof goal.
{`code`}
Goal (Line {`line`}): {`goal`}
Output diff blocks only.

Figure 8: Prompt template for the completion stage. The model receives the current code, the target goal state, and responds with SEARCH/REPLACE diffs that fill in proof bodies.

mini-batch size of 256 for policy update. Rewards are computed by verifying proof attempts and decomposition quality as described. To ensure informative gradient signals, we filter out rollout groups whose mean reward is 0 or 1. As described, We additionally apply an auxiliary supervised fine-tuning loss on proof completion trajectories with coefficient λ=0.08, and refresh the online proof state buffer at a sampling ratio of $\frac{1}{4}$ per step.

Prompting Format. For decomposition, the prompt presents the current Lean file with the target theorem highlighted, and instructs the model to produce a `<reasoning>` block followed by helper lemmas (each with `sorry` placeholders) and a main theorem whose proof invokes these lemmas without `sorry`. For completion, the prompt shows the current goal state and compilation status, and the model responds with a SEARCH/REPLACE diff that modifies only the proof body while preserving theorem signatures and imports. We use the same prompting format for both training and inference. The full prompt templates are shown in Figure 7 and Figure 8.

Inference Configuration. Each problem is allocated a wall-clock budget of 30 minutes. The decomposition stage runs up to 128 iterations with a maximum of 32 lemmas per problem. The completion stage runs up to 128 epochs per lemma, with a two-phase strategy: (1) attempt tactic-based proving when the code compiles, (2) fix compilation errors if present. We deploy a distributed Lean verification service built on Ray (Moritz et al., 2018), which manages a pool of Lean worker processes across multiple nodes and supports up to 512 concurrent verification requests with a 300-second timeout per check.

C Additional Results

C.1 Baseline Inference Scaling Curves

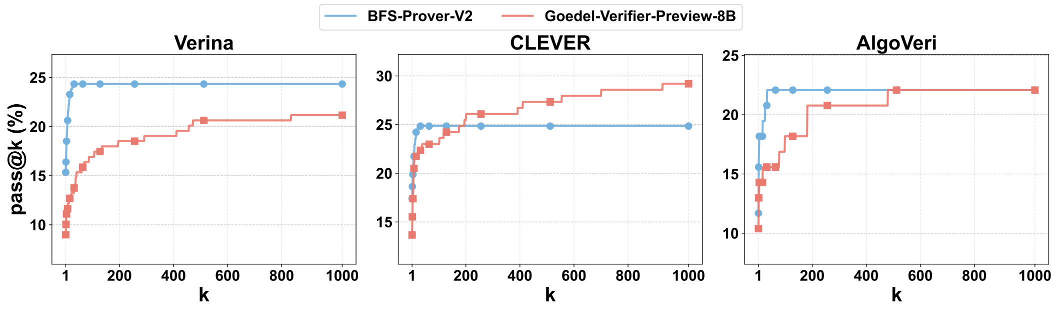

Figure 9: Inference scaling curves (pass@k) of two baseline verifiers, BFS-Prover-V2 and Göedel-Code-Prover-8B, on the three evaluation benchmarks. The x-axis denotes the number of sampled proof attempts k, and the y-axis shows the percentage of problems solved.

To contextualize the inference-time scaling behavior of our framework, we examine the pass@k curves of the two strongest baseline verifiers. As shown in Figure 9, both BFS-Prover-V2 and Göedel-Code-Prover-8Bexhibit diminishing returns as k increases: performance plateaus well before k=1024 on all three benchmarks, suggesting that simply scaling the number of independent proof attempts provides limited gains for whole-proof generation approaches. In contrast, our hierarchical framework continues to improve with additional compute (Section 4.3), highlighting the advantage of structured search over brute-force sampling.

C.2 Decomposition Score Correlation on Training Data

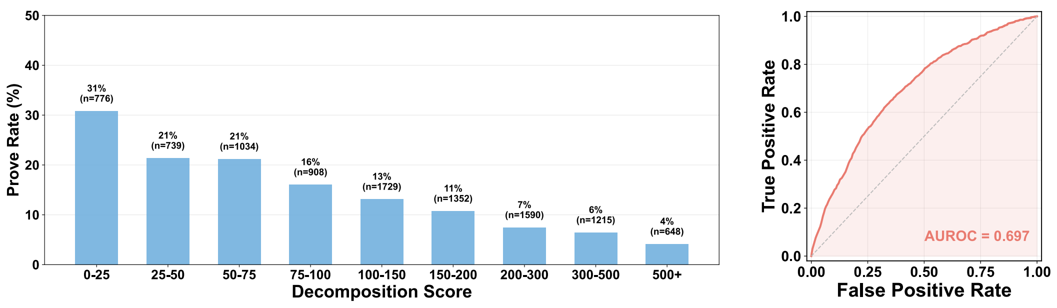

Figure 10: Decomposition score vs. prove rate on training data, showing that the positive correlation between score and provability generalizes beyond the Verina benchmark.

In Section 4.4, we demonstrate that the decomposition score S is a strong predictor of downstream provability on the Verina benchmark (AUROC = 0.903). To verify that this correlation is not an artifact of a single evaluation set, we repeat the analysis on our training data. As shown in Figure 10, the positive correlation between decomposition score and prove rate persists on the training distribution: problems whose best decomposition achieves a higher score are consistently more likely to be proved. This confirms that the decomposition score captures a generalizable signal of proof tractability, rather than overfitting to benchmark-specific patterns, and validates its use as a reward signal during reinforcement learning.

C.3 Effect of Training on Decomposition Quality

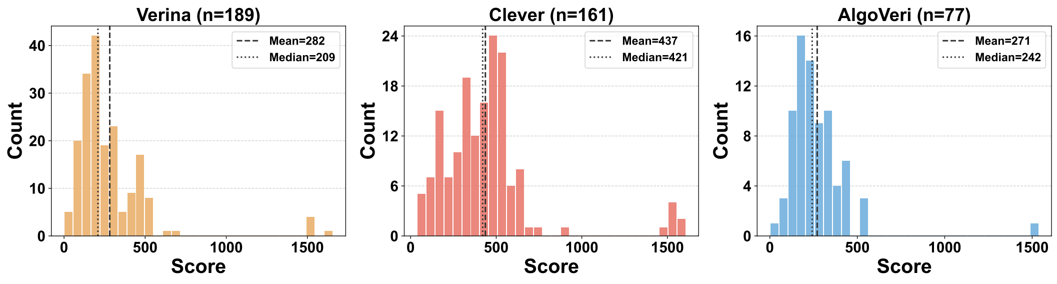

Figure 11: Decomposition score distribution before training across benchmarks.

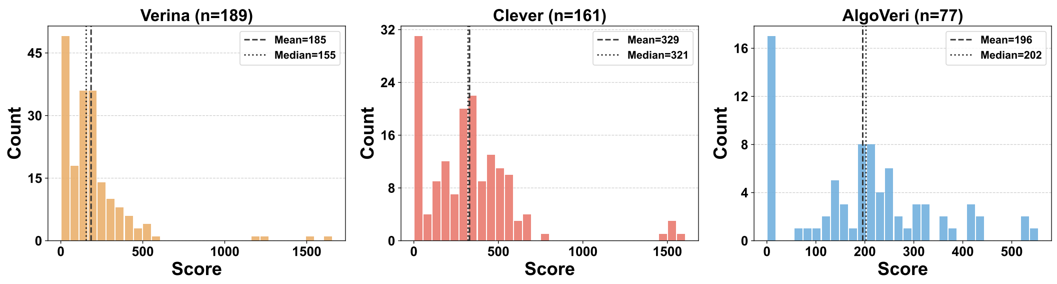

Figure 12: Decomposition score distribution after training across benchmarks.

To understand how training affects decomposition quality, we compare the distribution of decomposition scores before and after training across all three benchmarks. As shown in Figure 11, the pre-training distribution is heavily concentrated at low scores, indicating that the base model frequently produces decompositions that fail constructive justification or yield minimal structural reduction. After training (Figure 12), the distribution shifts markedly toward higher scores across all benchmarks, with a substantially larger fraction of decompositions achieving scores above 0.5. This shift confirms that our training pipeline, combining supervised fine-tuning with reinforcement learning, teaches the model to produce decompositions that are both logically sound and structurally effective.