
Curriculum Proof Repair with Learned Counterexamples

Anonymous Author(s)

Affiliation

Address

email

Abstract

1 Formal verification provides strong correctness guarantees, but proof development
2 remains difficult because developers must diagnose coarse verifier errors and repair
3 failed proof obligations. Recent approaches leverage language models (LMs) to
4 improve proof automation through verifier-in-the-loop repair, but they often rely
5 on hand-engineered prompts, retrieved examples, or sparse pass/fail feedback. We
6 present TOPOVERUS, an LM-based framework for Verus proof repair that internalizes
7 counterexample-guided diagnosis as a model capability. Our key insight
8 is to train counterexample generation not as a standalone validation task, but as a
9 data-driven diagnostic objective: a counterexample is useful when conditioning on
10 it helps the model produce a safe, verifier-passing proof. Given a failed Verus proof
11 and verifier error, TOPOVERUS generates a structured source-level counterexample,
12 repair rationale, and repaired proof in one trajectory, optimized with reinforcement
13 learning from Verus feedback and a proof-code safety gate. To address sparse
14 and order-sensitive credit assignment, TOPOVERUS introduces a topological graph
15 reward that derives invariant dependencies from inductiveness tests and encourages
16 counterexample generation following the same topological order as the invariant
17 dependencies. Across seven Verus benchmarks with 471 problems, TOPOVERUS
18 achieves 61.6%/72.2% weighted-average Safe-Pass@1/Safe-Pass@3, outperforming
19 the strongest baseline, Claude Sonnet 4.5, by 5.7/6.3 points. We also show
20 that TOPOVERUS’s learned counterexample can directly augment the reasoning of
21 existing LM-based proof repair approaches, improving their success rate by 64.3%
22 compared to using the chain-of-thought diagnostics.

23 1 Introduction

24 Formal verification provides machine-checked guarantees that software satisfies rich correctness
25 specifications, enabling the construction of safety-critical, large-scale software systems (Leroy et al.,
26 2016; Klein et al., 2009; Sun et al., 2024b; Zhou et al., 2024; Chen et al., 2023). Yet these guarantees
27 remain costly to obtain: developers must write specifications, discover inductive invariants, and repair
28 failed proof obligations. Recent work using language models (LMs) for verification automates parts
29 of this proof-construction loop (Sun et al., 2026; Zhong et al., 2025; Chen et al., 2025; Yang et al.,
30 2025, 2026a; Zhong & Si, 2025; Aggarwal et al., 2025; Mugnier et al., 2025; First et al., 2023),
31 lowering the expertise barrier for applying formal verification to real software.

32 As LM-generated proof candidates are not sound by construction, existing LM-based approaches
33 typically keep a verifier in the loop: the model proposes proof annotations, the verifier reports failures,
34 and the model revises the proof based on the error messages (Yang et al., 2025; Sun et al., 2026;
35 Zhong et al., 2025; Aggarwal et al., 2025). This loop resembles interactive verification, where
36 developers debug failures by constructing concrete failing states and tracing where an invariant or
37 postcondition breaks (Clarke et al., 2000, 2003). However, verifier messages often expose only

38 a failed proof obligation with limited diagnostic context, e.g., `postcondition not satisfied`,
39 rather than the source-level state, execution path, or missing fact responsible for the failure. The LM
40 must thus infer the root cause from ambiguous feedback, making repair brittle and token-inefficient.

41 Existing LM-based proof repair approaches can be viewed as attempts to compensate for coarse
42 verifier feedback. Some encode verifier-specific repair rules or syntax guidance in the prompt (Yang
43 et al., 2025; Sun et al., 2026), supply in-context proof examples (Yang et al., 2025; Mugnier et al.,
44 2025), or retrieve relevant proof fragments (Zhong et al., 2025). Others bypass explicit diagnosis and
45 use verification outcomes, i.e., pass/fail signals, to guide search, data synthesis, self-improvement,
46 or fine-tuning (Aggarwal et al., 2025; Chen et al., 2025). While effective, these approaches leave a
47 diagnostic gap: engineered prompt knowledge is costly to curate and maintain as verifier behavior
48 and proof idioms evolve (Yang et al., 2026a,b), whereas verifier-outcome objectives provide only
49 sparse, indirect supervision for repair.

50 A more actionable feedback signal is a source-level counterexample: a concrete program state,
51 optionally with a short execution trace, that falsifies a proof obligation. Unlike verifier-specific
52 repair rules, counterexamples describe the concrete failing program state, making them largely
53 independent of diagnostic formats and directly useful for localizing repairs. They are central to classic
54 counterexample-guided verification (Clarke et al., 2000, 2003; Bradley, 2011, 2012), where failed
55 states drive refinement or blocking of bad regions. Recent LM-based works also show that generated
56 counterexamples can improve proof repair (Yang et al., 2026b; Li et al., 2026; Huang et al., 2026).
57 Unfortunately, a useful counterexample requires nontrivial, execution-level code reasoning: it must
58 be feasible, expose the relevant corner case, and identify the missing proof fact, rather than merely
59 state a plausible failure (Copet et al., 2025). Moreover, proof failures often conflate multiple missing
60 or incorrect invariants, so which counterexample is most useful to address first is not well defined.

61 **Our approach.** We present TOPOVERUS, an LM-based verification framework for Verus proof
62 repair that internalizes counterexample-guided diagnosis as a model capability. Our key insight is to
63 formulate counterexample generation as a diagnostic learning problem: a counterexample is useful
64 not merely when it is plausible or independently valid (Yang et al., 2026b), but when conditioning
65 on it helps the model produce a safe, verifier-passing proof. Given a failed Verus proof attempt and
66 the corresponding verifier error, TOPOVERUS generates a structured source-level counterexample,
67 a concise repair rationale, and a candidate repaired proof in this order within a single trajectory.
68 We optimize the full trajectory using reinforcement learning with verifiable rewards (RLVR) from
69 Verus, together with a proof-code safety gate that rejects repairs modifying the executable program or
70 user-supplied specifications. As a result, counterexamples are not trained as standalone artifacts with
71 hand-written validators (Yang et al., 2026b) or predefined verifier-specific repair taxonomies (Yang
72 et al., 2025). Instead, they receive indirect credit through downstream repair success, biasing the
73 model toward diagnostics that are actionable for safe proof repair.

74 This formulation, however, creates a key technical challenge. The training signal for counterexample
75 utility is sparse, delayed, and order-sensitive. A failed proof often contains multiple missing or
76 incorrect invariants, so final verifier success alone does not reveal whether a generated counterexam-
77 ple was wrong, merely unhelpful, or aimed at a downstream proof obligation whose preconditions
78 were not yet established (Section 2.1). To expose this latent structure during learning, TOPOVERUS
79 approximates dependencies among invariants with a graph derived from inductiveness tests, and
80 rewards repaired proofs whose invariants respect a topological order over the resulting graph. The
81 resulting topological graph reward turns sparse outcome feedback into a dependency-aware *curricu-*
82 *lum*: establish state-constraining facts first, then generate counterexamples and repairs for dependent
83 obligations in the corresponding order, where the counterexample’s diagnostics become informative.

84 **Results.** Our evaluation shows that TOPOVERUS improves Verus proof repair and provides actionable
85 diagnostics for downstream repair tools. Across seven Verus benchmarks comprising 471 problems,
86 TOPOVERUS achieves 61.6% and 72.2% in Safe-Pass@1 and Safe-Pass@3, outperforming the
87 strongest baseline, i.e., Claude Sonnet 4.5, by **5.7** and **6.3** points, respectively. The gains are
88 largest on structurally difficult settings such as VeriCogen and ObfsBench, where proof templates
89 and syntactic hints are less reliable. Ablations show that counterexample-augmented reasoning in
90 TOPOVERUS improves weighted Safe-Pass@1 by 11.2 points over the repair with natural language
91 diagnostic, e.g., chain-of-thought, while the topological graph reward further improves over binary
92 verifier-only RL by **7.9** points. Importantly, TOPOVERUS’s generated counterexamples are also

93 directly usable by existing proof repair tools, improving their repair success by 64.3% compared to
 94 chain-of-thought diagnostics.

95 2 Overview

96 We focus on Verus, a Rust-native verifier for verifying Rust programs. A Verus
 97 proof consists of executable Rust code together with specifications (e.g., preconditions
 98 and postconditions), and proof annotations (e.g., invariants, assertions). The
 99 verifier discharges the proof into Verification Conditions (VCs) and checks them
 100 with off-the-shelf SMT solvers (de Moura & Bjørner, 2008; Barbosa et al., 2022).
 101 Among the proof annotations, inferring invariants has been a key challenge of automating
 102 formal verification (Flanagan & Leino, 2001; Garg et al., 2014; Kamath et al., 2023).
 103 Loop invariants are properties that always hold regardless of which iteration the loop
 104 is running at.

```

1 fn search_insert(nums: Vec<i32>, target: i32) -> (res: i32)
2 requires
3   1 <= nums.len() <= 10000,
4   forall|i, j| 0 <= i < j < nums.len() ==> nums[i] < nums[j],
5 ensures /* insertion-position postcondition */
6   0 <= res <= nums.len(),
7   nums[0] <= target <= nums[nums.len() - 1] ==>
8   (forall|i| 0 <= i < res ==> nums[i] < target) &&
9   (forall|i| res <= i < nums.len() ==> nums[i] >= target),
10 {
11   let mut low = 0i32;
12   let mut high = nums.len() as i32 - 1;
13   while low <= high
14     invariant
15       1 <= nums.len() <= 10000, //I1
16       0 <= low <= nums.len(), //I2
17       -1 <= high <= nums.len() - 1, //I3
18       forall|i| 0 <= i < low ==> nums[i] < target, //I4
19       forall|i| high < i < nums.len() ==> nums[i] > target //I5
20   {
21     let mid = ((low + high) as usize / 2) as i32;
22     if nums[mid as usize] == target { return mid; }
23     if target < nums[mid as usize] { high = mid - 1; }
24     else { low = mid + 1; }
25   }
26   low
27 }
28

```

Figure 1: Proof annotation for binary-search insertion.

113 2.1 Motivating Example

114 Figure 1 shows the Verus code and proof that motivate our approach. The function implements binary-
 115 search insertion: it returns the position of `target` in a sorted array, or the index where `target`
 116 should be inserted. The `requires` (preconditions) assume the input to be a sorted array, and the `ensures`
 117 (postconditions) specify that `target` is inserted correctly. The loop invariants are the properties that
 118 are derived from the preconditions and execution code. They hold before entering the loop (base
 119 case) and remain true in each loop iteration (inductiveness). Note that invariants often depend on
 120 other invariants or are mutually dependent. We use this example to illustrate such dependencies and
 121 how fixing invariants following such dependencies benefits counterexample reasoning, diagnosis and
 122 repair of buggy proofs.

123 **Semantic dependencies among invariants.** The five invariant clauses in Figure 1 play different
 124 semantic roles in verifying the program. **I1** bounds the length range of `nums`. **I2** and **I3** use that
 125 length fact to keep `low` and `high` within index ranges. **I4** and **I5** are the semantic partition invariants:
 126 all indices strictly left of `low` contain values below `target`, while all indices strictly right of `high`
 127 contain values above `target`. The semantic partition invariants become useful only when the
 128 boundary invariants (**I2** and **I3**) hold.

129 Our dependency analysis makes this semantic structure operational by testing which previously
 130 established invariants are needed for another invariant to become inductive. In this example, the
 131 analysis identifies **I1** as a prerequisite for proving the bounds invariants **I2** and **I3**, and identifies
 132 the established bounds facts as prerequisites for proving the partition invariants **I4** and **I5**. The
 133 resulting graph is therefore not just a syntactic relation among clauses: it reflects that high-level
 134 partition information is supported by lower-level index-range information. This motivates using the
 135 dependency graph as an operational proxy for proof progress, rather than as a complete semantic
 136 account of the proof.

137 **Under-constrained diagnostic.** Consider a repair attempt that contains only the right-partition
 138 invariant but writes it with an inclusive lower bound:

```
forall|i| high <= i < nums.len() ==> nums[i] > target.
```

139 Because **I1**–**I3** are absent, verification may fail on preliminary preservation or postcondition obliga-
 140 tions before isolating the semantic boundary error. The annotated diagnostic in Figure 2 illustrates
 141 the failure mode: the output has a plausible *State–Trace–Violation* format, but the three parts are not
 142 mutually consistent. With `high=2` and `len=3`, the range `high <= i < len` contains only `i=2`; a

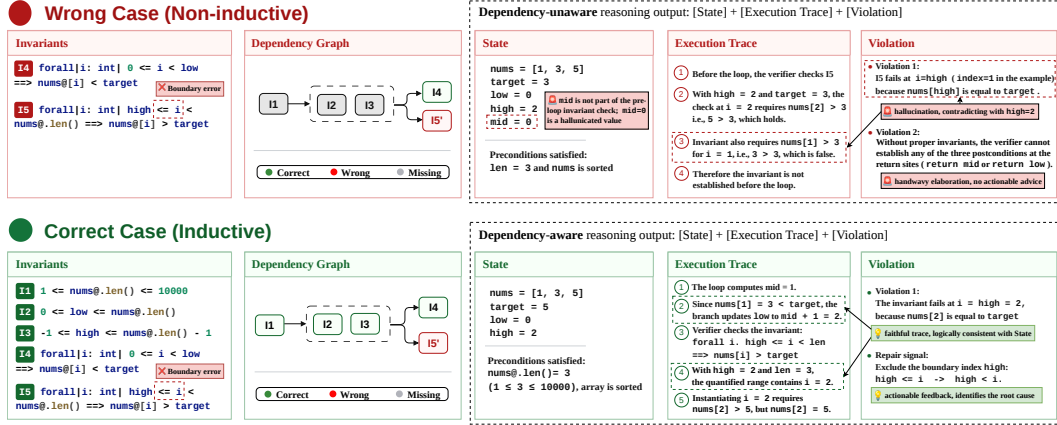


Figure 2: Comparison of dependency-unaware (top) and dependency-aware (bottom) diagnostics.

143 trace that instantiates $i=1$ is outside the quantified domain, even though $\text{nums}[1]$ is not greater than
 144 target . Its secondary violation is even less actionable: it broadly says that return-site postconditions
 145 cannot be proved, but does not identify a concrete failing return state or a local repair. The diagnostic
 146 therefore points at a partition invariant without providing a valid counterexample for the claimed
 147 quantified failure.

148 **Constrained diagnostic.** Now suppose I1–I3 have already been established and the only remaining
 149 bug is the boundary in I5. The same diagnostic format becomes informative because the state and trace
 150 are anchored by the prerequisite invariants. The bottom half of Figure 2 shows the intended contrast:
 151 after one iteration with $\text{nums}=[1, 3, 5]$, $\text{target}=5$, $\text{low}=2$, and $\text{high}=3$, the bounds invariants justify
 152 that the quantified range is the intended in-bounds suffix and I4 rules out unrelated left-partition
 153 failures. Now the incorrect I5 includes $i=\text{high}=2$, requiring $\text{nums}[2] > 5$, but $\text{nums}[2]=5$. The
 154 counterexample directly pinpoints the repair solution: change $\text{high} <= i$ to $\text{high} < i$.

155 **Implication for training.** The example shows why a good counterexample is critical to proof
 156 progress. If a model is rewarded only for final verifier success, it may spend attempts on diagnostics
 157 whose states and traces are not anchored by prerequisite invariants. The Topological Graph Reward
 158 in Section 3.3 turns the dependency graph above into process feedback: it encourages repairs to make
 159 prerequisite facts available, so that counterexamples for subsequent, semantic invariants are checked
 160 in a better-constrained proof state.

161 3 Methodology

162 3.1 Problem Formulation

163 We formally define the problem of curriculum-based proof repair as a dependency-aware generation
 164 process. Given a program \mathcal{P} with a specification $\Phi = (P_{pre}, Q_{post})$, i.e., pre-conditions and post-
 165 conditions, the task is to synthesize a proof Π consisting of proof annotations (invariants, assertions,
 166 etc.) that verifies the program.

167 **Counterexample-guided repair.** Let Π_t denote the proof annotations at repair step t . If the proof
 168 contains a verification error e_t corresponding to a failing condition, we decompose this failing
 169 constraint into an antecedent $A_t(\sigma)$ (pre-state) and a consequent $C_t(\sigma)$ (proof obligation). A valid
 170 counterexample is a concrete program state σ that witnesses this failure, satisfying:

$$\sigma \models A_t(\sigma) \wedge \neg C_t(\sigma), \quad (1)$$

171 Rather than relying on an external SMT-based tool to statically resolve this state, our approach
 172 instructs the model to generate a structured diagnostic tuple (σ, τ, ψ) as an intermediate reasoning
 173 step, i.e., counterexample, execution trace, and violation in Figure 2. τ represents the specific
 174 execution trace by executing from σ , and ψ is a natural-language description detailing why the
 175 obligation was falsified.

176 **Invariant dependency.** Let $V = \{v_1, \dots, v_n\}$ denote the canonical invariants required to verify
 177 \mathcal{P} . We model their prerequisite structure via a dependency graph $G = (V, E)$, where $(v_j, v_i) \in E$
 178 represents that prerequisite v_j must be established prior to inductively verifying v_i . At step t , the
 179 objective is to synthesize Π_{t+1} to resolve errors e_t . Rather than blindly blocking state σ , the model
 180 learns the curriculum where it tends to follow the topological ordering of G , via reinforcement
 181 learning, as detailed in Section 3.3.

182 3.2 Reward Architecture

183 We optimize a response-level policy $\pi_\theta(y | x)$ with GRPO, where x is the current repair context and
 184 y is the complete structured response. The learning signal is a shaped reward:

$$R(x, y) = S_{\text{safe}}(x, y) \left[G_{\text{format}}(y) (R_{\text{pass}}(x, y) + \lambda_{\text{dep}} R_{\text{dep}}(x, y)) + \lambda_{\text{format}} R_{\text{format}}(x, y) \right], \quad (2)$$

185 where S_{safe} is a safety gate, R_{pass} is the primary task reward from verifier feedback, R_{dep} is a
 186 dependency-aware shaping reward over proof structure, R_{format} is an auxiliary format reward, and
 187 $G_{\text{format}}(y)$ acts as a feasibility gate that downweights malformed outputs. The scalars λ_{dep} and λ_{format}
 188 are fixed reward weights; when the dependency reward is disabled, we set $\lambda_{\text{dep}} = 0$.

189 The verifier reward is binary: $R_{\text{pass}} = \mathbf{1}$ when Verus accepts the repaired program. The format reward
 190 $R_{\text{format}} \in [0, 0.4]$ is an auxiliary term that rewards the presence of required structural components
 191 (XML reasoning tags and the three counterexample fields: state, execution trace, violation). The
 192 format gate G_{format} scales the primary reward according to structural completeness, i.e., full score
 193 when both reasoning blocks are present, a reduced score when only one is present, and a smaller
 194 score when neither is present, preventing the mode collapse into standard Chain-of-Thought repair
 195 that bypasses the counterexample reasoning step.

196 The safety gate $S_{\text{safe}} \in \{0, 1\}$ uses Lynette, an AST-level checker developed by Yang et al. (2025),
 197 to check if the model only modified the proof annotations (invariants, assertions, etc.); unsafe
 198 modifications (e.g., execution code) receive zero reward.

199 3.3 Topological Graph Reward

200 A core component of TOPOVERUS is the *Topological Graph Reward* ($R_{\text{dep}} \in [0, 1]$), which captures
 201 structural dependencies among proof obligations and encourages the model to establish prerequisite
 202 invariants before dependent ones.

203 **Definition 1** (Inductive Subset). *Let $V = \{v_1, \dots, v_n\}$ be the canonical set of loop invariants
 204 for a program. A subset $S \subseteq V$ is inductive if the conjunction $\bigwedge_{v \in S} v$ is (i) established by the
 205 loop precondition, (ii) preserved by every loop body iteration, and (iii) sufficient to discharge all
 206 verification conditions in S . We write $\mathcal{I}(V)$ for the family of all inductive subsets of V .*

207 **Definition 2** (Invariant Dependency Graph). *For each invariant $v_i \in V$, define its dependency set as:*

$$\text{deps}(v_i) = \left(\bigcap_{S \in \mathcal{I}(V), v_i \in S} S \right) \setminus \{v_i\}. \quad (3)$$

208 *That is, $\text{deps}(v_i)$ consists of the invariants present in every inductive subset containing v_i , i.e., the
 209 invariants that v_i structurally depends on. The invariant dependency graph is $G = (V, E)$ with:*

$$E = \{(v_j, v_i) \mid v_j \in \text{deps}(v_i)\}. \quad (4)$$

210 *An edge $(v_j, v_i) \in E$ encodes that v_j must be established before v_i can be inductively verified.*

211 **Definition 3** (SCC Condensation). *When $G = (V, E)$ contains cycles (mutually inductive invariants),
 212 we compute its strongly connected components $\text{SCC}(G) = \{S_1, \dots, S_p\}$ via Tarjan’s algorithm and
 213 form the condensation:*

$$\hat{G} = (\hat{V}, \hat{E}), \quad \hat{V} = \text{SCC}(G), \quad \hat{E} = \{(S_a, S_b) \mid \exists v_i \in S_a, v_j \in S_b : (v_i, v_j) \in E, a \neq b\}. \quad (5)$$

214 *The condensation \hat{G} is a DAG that admits a topological ordering \prec over \hat{V} , capturing dependency
 215 order represented by the universal-prerequisite graph.*

216 **Invariant matching.** Given the model’s generated code, we extract candidate invariants and match
 217 them against the canonical set V via: (1) exact substring matching, and (2) fuzzy token-set matching
 218 with Jaccard similarity $J(\cdot, \cdot) \geq 0.6$. Let $V' \subseteq V$ denote the set of matched canonical invariants.

219 **Reward computation.** Let $E' = \{(v_j, v_i) \in E \mid v_j, v_i \in V'\}$ be the dependency edges between
 220 matched invariants, let $\hat{E}' = \{(v_j, v_i) \in E' \mid \hat{S}(v_j) \neq \hat{S}(v_i)\}$ be the inter-SCC edges, and let $\text{pos}(v)$
 221 denote the source position of invariant v in the model’s output. The topological graph reward is:

$$R_{\text{dep}} = \underbrace{\frac{|V'|}{|V|}}_{\alpha \text{ (coverage)}} \times \underbrace{\frac{|\{(v_j, v_i) \in \hat{E}' \mid \text{pos}(v_j) < \text{pos}(v_i)\}|}{|\hat{E}'|}}_{\beta \text{ (edge satisfaction)}}, \quad (6)$$

222 where $\hat{S}(v)$ denotes the SCC of v in the condensation \hat{G} , and $\text{pos}(v_j) < \text{pos}(v_i)$ means the model
 223 generated v_j before v_i . Intra-SCC edges (mutually inductive invariants) are excluded, since no
 224 meaningful ordering exists within an SCC. When $\hat{E}' = \emptyset$, we set $\beta = 1$. The coverage term
 225 α rewards producing more canonical invariants; the edge satisfaction term β rewards generating
 226 prerequisite invariants before dependent ones, inducing a micro-curriculum within each repair attempt.
 227 *Remark.* The dependency graph G and inductive subsets $\mathcal{I}(V)$ are pre-computed offline for each
 228 training instance. In training, the reward computation reduces to invariant matching and DAG edge
 229 checking, i.e., an $O(|V| + |E|)$ operation that adds negligible overhead to the GRPO training loop.

230 4 Evaluation

231 4.1 Experimental Setup

232 **Benchmarks.** We evaluate on 471 test problems from seven Verus benchmark suites: **VerusBench** (26
 233 general verification problems from AutoVerus (Yang et al., 2025)), **DafnyBench** (17 translated
 234 Dafny problems (Loughridge et al., 2024)), **HumanEval** (15 algorithmic problems (Chen et al.,
 235 2021)), **LeetCode** (5 algorithmic problems (Dai, 2025)), **VeriCogen** (89 proof-stripped HumanEval
 236 variants from AlphaVerus (Bursuc et al., 2025)), **ObfsBench** (266 problems with obfuscated variable
 237 names (Yang et al., 2026b)), and **VeruSAGE** (53 systems-level tasks from Anvil, Mimalloc, and
 238 NrOS (Yang et al., 2026a)).

239 **Models.** All TOPOVERUS variants start from QwQ-32B (Team, 2025). **CoT-Init** is supervised
 240 fine-tuning with chain-of-thought repair only. **CTX-Init** uses the counterexample output format but
 241 no RL. **CTX-Binary** applies GRPO with a binary verifier reward. **CTX-Dep**, the full TOPOVERUS
 242 model, applies GRPO with the Topological Graph Reward. We compare against four prompted cloud
 243 LM baselines: Sonnet-4.5 (Anthropic, 2025), GPT-4o (Hurst et al., 2024), Qwen3-Coder (Cao et al.,
 244 2026), and DeepSeek-V4 (using the DeepSeek-V4-Flash model).

245 **Protocol and metrics.** The standard repair setting uses iterative verifier feedback for up to 10 steps.
 246 We report Safe-Pass@1 and Safe-Pass@3; Safe-Pass@3 counts a problem as solved if any of three
 247 independent repair chains succeeds. A solution is *safe* if Lynette confirms that executable code and
 248 user specifications were not changed. For test-time scaling, we use 32 chains \times 10 iterations for
 249 Pass@ K curves and 1 chain \times 32 iterations for iteration-scaling curves.

250 **Training details.** SFT uses DeepSpeed ZeRO-3 on 4 NVIDIA L40S GPUs with learning rate
 251 2×10^{-5} , batch size 32, maximum sequence length 8,192, and 2 epochs. RL uses GRPO with FSDP2,
 252 4-way Ulysses sequence parallelism, vLLM rollouts at temperature 0.7 and top- $p = 0.9$, 8 rollouts
 253 per prompt, learning rate 2×10^{-6} , no KL penalty, PPO clip ratio 0.28, and 2 epochs over 249
 254 training samples.

255 4.2 Main Results

256 Table 1 shows that TOPOVERUS achieves the strongest average Safe-Pass@1 / Safe-Pass@3, 61.6% /
 257 72.2%. This is 5.7 / 6.3 points above Sonnet-4.5, the strongest cloud baseline by average, and 7.9 / 4.3
 258 points above CTX-Binary, the strongest non-dependency 32B variant. The clearest suite-level gain
 259 over cloud baselines is on ObfsBench, where TOPOVERUS reaches 69.9% Safe-Pass@1 and 80.1%
 260 Safe-Pass@3, improving over Sonnet-4.5 by 7.9 and 4.9 points. On VeruSAGE, closed cloud models

Table 1: Overall verification repair safe pass rates. Each cell reports Safe-Pass@ k (%) across seven benchmarks. Cloud LLMs use iterative repair with verifier feedback; our models use 3 chains \times 10 iterations. A solution is *safe* if Lynette confirms only ghost (proof) code changed.

Approach	VerusBench		DafnyBench		HumanEval		Leetcode		VeriCogen		ObfsBench		VeruSAGE	
	$k=1$	$k=3$	$k=1$	$k=3$	$k=1$	$k=3$	$k=1$	$k=3$	$k=1$	$k=3$	$k=1$	$k=3$	$k=1$	$k=3$
<i>Cloud LLMs</i>														
Sonnet-4.5	73.1	84.6	70.6	76.5	13.3	13.3	40.0	40.0	26.7	30.0	62.0	75.2	75.5	84.9
GPT-4o	42.3	42.3	76.5	76.5	6.7	13.3	0.0	0.0	40.4	41.4	21.8	22.9	17.0	24.5
Qwen3-Coder	34.6	34.6	52.9	52.9	6.7	6.7	0.0	0.0	20.0	25.6	14.3	18.8	45.3	64.2
DeepSeek-V4	23.1	38.5	47.1	82.4	13.3	20.0	0.0	0.0	28.9	36.7	12.8	29.3	58.5	66.0
<i>External proof-repair systems</i>														
AutoVerus	38.5	50.0	58.8	70.6	6.7	6.7	20.0	20.0	22.5	28.1	13.2	23.7	7.5	9.4
<i>Ours (QwQ-32B)</i>														
CoT-Init	46.2	50.0	82.4	100	20.0	26.7	0.0	0.0	38.2	48.3	42.9	60.9	15.1	24.5
CTX-Init	88.5	92.3	88.2	100	26.7	26.7	20.0	40.0	48.3	61.8	55.3	73.3	9.4	22.6
CTX-Binary	88.5	96.2	94.1	100	26.7	26.7	20.0	40.0	44.9	60.7	58.6	75.9	24.5	30.2
CTX-Dep	88.5	88.5	100	100	20.0	33.3	20.0	20.0	49.4	65.2	69.9	80.1	30.2	43.4

261 remain stronger, but TOPOVERUS still improves over the other QwQ-32B variants, suggesting that
 262 dependency-aware RL helps the open model without eliminating all model-capacity gaps.

263 4.3 Ablation Study

264 Table 1 also isolates the effect of counterexample formatting and dependency-aware RL. Moving
 265 from CoT-Init to CTX-Init raises weighted safe Pass@1 from 39.3% to 50.5%, a gain of 11.2 points.
 266 This supports the core premise that asking the model to expose a concrete failing state before repair is
 267 more useful than unstructured reasoning. The improvement is especially large on VerusBench (46.2%
 268 to 88.5%) and ObfsBench (42.9% to 55.3%), indicating that the format helps even before RL.

269 Binary RL improves CTX-Init from 50.5% to 53.7%, but its gains are uneven. It improves VeruSAGE
 270 from 9.4% to 24.5%, yet degrades VeriCogen from 48.3% to 44.9%. This pattern is consistent with
 271 the sparsity problem discussed in Section 3.3: verifier success is useful, but binary reward does not
 272 distinguish whether a repair trajectory is building the proof structure in a robust order.

273 The full dependency reward produces the largest gain, improving weighted safe Pass@1 to 61.6%
 274 and Pass@3 to 72.2%. It is the best variant on DafnyBench, VeriCogen, ObfsBench, VeruSAGE,
 275 and the weighted average. The largest absolute gain over CTX-Binary is on ObfsBench (+11.3
 276 Pass@1 points), where obfuscated identifiers reduce the usefulness of memorized templates and
 277 make structural proof ordering more important.

278 4.4 Experiment 3: Counterexample Utility

279 Table 2 tests whether generated counterexamples help repair models other than
 280 the generator itself. We freeze 118 hard failed states from unverified repair at-
 281 tempts. For each state, we keep the buggy program and verifier error fixed, replace
 282 only the counterexample source, and ask three independent repairers (DeepSeek-
 283 V4, Sonnet-4.5, and Qwen3-Coder) to produce a repair. CTX-Dep counterexamples yield the
 284 best Sonnet repair rate, 39.0%, a 24.3% relative improvement over CoT-Init diagnostics. DeepSeek-
 285 V4 and Qwen3-Coder also improve over CoT-Init when given CTX-Dep diagnostics. The result
 286 supports a conservative claim: the learned diagnostics need not be independently formal certificates,
 287 but they carry actionable information that transfers to separate repair policies.

Table 2: Counterexample usability on 118 hard failed states. Cells report frozen-repair success; colored deltas are relative changes over CoT-Init.

CEX source	DeepSeek	Sonnet	Qwen
CoT-Init	11.9%	31.4%	13.6%
CTX-Init	15.3% (+28.6%)	32.2% (+2.7%)	12.7% (-6.2%)
CTX-Binary	19.5% (+64.3%)	33.1% (+5.4%)	15.3% (+12.5%)
CTX-Dep	19.5% (+64.3%)	39.0% (+24.3%)	15.3% (+12.5%)

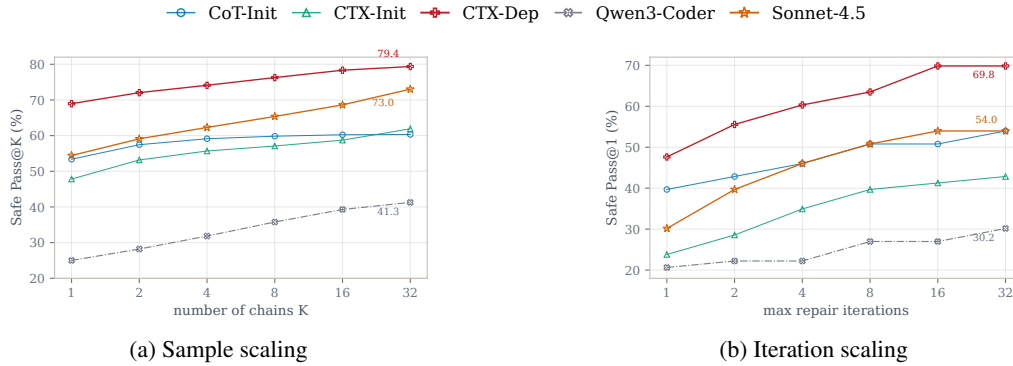


Figure 3: Safe test-time scaling. The sample-scaling panel reports Safe Pass@ K as the number of chains increases, while the iteration-scaling panel reports Safe Pass@1 as the repair budget increases.

292 4.5 Test-Time Scaling

293 Figure 3 reports test-time scaling on the 63-problem validation set drawn from VerusBench, Dafny-
 294 Bench, HumanEval, and LeetCode. CTX-Dep scales best with additional test-time compute, reaching
 295 79.4% Safe Pass@32 in sample scaling and 69.8% Safe Pass@1 with 32 repair iterations. The gains
 296 over CTX-Init, CoT-Init, and Sonnet-4.5 show that dependency-aware diagnostics improve scaling
 297 with both independent samples and iterative repair budgets.

298 5 Related Work

299 **LM-Based formal verification.** Recent efforts have significantly lowered the barrier to formal
 300 verification using LMs, across both mathematical theorem proving (Polu & Sutskever, 2020; Yang
 301 et al., 2023; Lample et al., 2022) and program verification (First et al., 2023; Sun et al., 2024a;
 302 He et al., 2026; Zhong et al., 2025). In Verus proof generation, prior works primarily focused on
 303 encoding heuristics and repair strategies via either engineering prompts Yang et al. (2025); Sun
 304 et al. (2026), or constructing large-scale synthetic training sets Chen et al. (2025). However, a
 305 common limitation across these systems is their reliance on the traditional iterative-refinement
 306 paradigm, heavily depending on coarse, abstract feedback from the underlying SMT solvers. Such
 307 feedback often fails to isolate the root cause of verification failures, frequently causing LM-based
 308 repair to degenerate into inefficient trial-and-error generation. Our work advances this paradigm by
 309 moving beyond verifier feedback; we train the model to internalize diagnostic reasoning, integrating
 310 actionable, concrete counterexamples directly into the repair policy.

311 **Counterexample-guided feedback for proof repair.** Traditional formal methods have long relied
 312 on counterexamples, i.e., concrete execution traces that witness proof failures, as the key actionable
 313 feedback to iteratively refine invariants and generate proofs (Clarke et al., 2000; Bradley, 2012).
 314 Recent works have begun to explore LM-driven counterexample generation to sidestep the challenge
 315 of reconstructing counterexamples from low-level SMT solver models (Yang et al., 2026b; Li et al.,
 316 2026). Nonetheless, these existing methods often demand strict formal validation of the generated
 317 counterexamples, which is often intractable, especially for complex proofs where there are multiple
 318 counterexamples to different missing/wrong invariants. Following the philosophy of reasoning over
 319 counterexamples, TOPOVERUS aims to generate high-quality counterexamples, but in a principled
 320 way, i.e., internalizing the capability of generating high-quality counterexamples following the
 321 topological dependency graph. Rather than searching for arbitrary counterexamples for a proof,
 322 TOPOVERUS trains the model to produce counterexamples whose diagnostic content is useful for
 323 successful downstream repair.

324 **Reinforcement learning and curriculum learning.** Reinforcement learning with verifiable rewards
 325 has shown strong results in theorem proving (AlphaProof & AlphaGeometry, 2024; Xin et al., 2024;
 326 Wang et al., 2025) and code generation (Le et al., 2022; Liu et al., 2023), but optimizing complex
 327 verification tasks remains challenging due to reward sparsity and long proof trajectories. Curriculum
 328 learning (Bengio et al., 2009; Portelas et al., 2021; Wang et al., 2021) mitigates such sparsity
 329 by structuring learning from easy to hard; traditionally this operates across episodes as a macro-

330 curriculum over training data. More recently, work on long-horizon reasoning has begun exploring
331 intra-episode micro-curricula that structure the generation process within a single rollout (Dou et al.,
332 2024; Zhang et al., 2026; Shen et al., 2025). These approaches define curricula through external
333 heuristics (e.g., sequence length, task decomposition), whereas our Topological Graph Reward
334 extracts an intrinsic micro-curriculum directly from the natural dependency DAG among Verus proof
335 obligations, rewarding the model for establishing prerequisite invariants before more complex ones
336 within each repair trajectory.

337 6 Discussion and Conclusion

338 **Limitations.** First, the dependency-aware micro-curriculum is fundamentally targeting loop in-
339 variants. While invariants are often a key bottleneck of formal verification, this focus restricts the
340 framework’s immediate generalization to other critical verification structures, e.g., assertions. Second,
341 the validity of the generated counterexamples is not formally guaranteed. Because exact reachability
342 checking is fundamentally challenging and often infeasible (Yang et al., 2026b), the model treats
343 the counterexample (σ, τ, ψ) strictly as an intermediate reasoning step optimized for downstream
344 repair success rather than absolute mathematical validity. Consequently, the model may occasionally
345 hallucinate imprecise execution traces that provide ambiguous guidance.

346 **Conclusion.** We presented TOPOVERUS, a 32B-parameter framework that transforms automated for-
347 mal verification from a chaotic, unordered trial-and-error process into a dependency-aware generation
348 pipeline. By training a response-level policy via GRPO, our model learns to synthesize structured,
349 intermediate counterexamples in a principled way and incrementally repair the proofs. The Topologi-
350 cal Graph Reward extracts a natural micro-curriculum directly from the DAG of proof obligations,
351 explicitly incentivizing the model to establish foundational constraints before writing dependent in-
352 variants. Our extensive evaluations demonstrate that synergistically coupling counterexample-guided
353 feedback with topological curriculum learning significantly mitigates the sparsity of verifier rewards
354 and outperforms the baselines.

355 Ethics Statement

356 This work develops AI methods for automated formal verification, which aims to improve the
357 reliability and correctness of software systems. We do not foresee direct negative societal impacts.
358 All benchmarks used are publicly available or synthesized from open-source codebases. Our models
359 are trained on code verification tasks and are not designed for generating harmful content.

360 References

- 361 Pranjali Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code
362 generation through self-improving translation and tree refinement. In *Forty-second International Con-*
363 *ference on Machine Learning*, 2025. URL <https://openreview.net/forum?id=tU8QKX4dMI>.
- 364 Team AlphaProof and Team AlphaGeometry. Ai achieves silver-medal standard solving international
365 178 mathematical olympiad problems. *DeepMind blog*, 179:45, 2024.
- 366 Anthropic. Claude sonnet 4.5 system card, 2025. URL <https://www.anthropic.com/claude-sonnet-4-5-system-card>. Accessed: 2026-05-07.
- 368 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Ab-
369 dalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias
370 Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and
371 industrial-strength smt solver. In Dana Fisman and Grigore Rosu (eds.), *Tools and Algorithms*
372 *for the Construction and Analysis of Systems*, pp. 415–442, Cham, 2022. Springer International
373 Publishing. ISBN 978-3-030-99524-9.
- 374 Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In
375 *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)*, pp.
376 41–48. ACM, 2009. doi: 10.1145/1553374.1553380.

- 377 Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on*
378 *Verification, Model Checking, and Abstract Interpretation*, pp. 70–87. Springer, 2011.
- 379 Aaron R Bradley. Understanding ic3. In *International Conference on Theory and Applications of*
380 *Satisfiability Testing*, pp. 1–14. Springer, 2012.
- 381 Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa,
382 Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, et al. A benchmark
383 for vericoding: formally verified program synthesis. *arXiv preprint arXiv:2509.22908*, 2025.
- 384 Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing,
385 Kaixin Li, Mingze Li, Junyang Lin, et al. Qwen3-coder-next technical report. *arXiv preprint*
386 *arXiv:2603.00729*, 2026.
- 387 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared
388 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri,
389 Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan,
390 Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian,
391 Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios
392 Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino,
393 Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders,
394 Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa,
395 Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob
396 McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating
397 large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- 398 Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain
399 Misu, Hao Yu, Nan Duan, Peng CHENG, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong
400 Zhou. Automated proof generation for rust code via self-evolution. In *The Thirteenth International*
401 *Conference on Learning Representations*, 2025. URL [https://openreview.net/forum?id=](https://openreview.net/forum?id=2NqssmiXLu)
402 [2NqssmiXLu](https://openreview.net/forum?id=2NqssmiXLu).
- 403 Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere:
404 Towards practical verified kernels in rust. In *Proceedings of the 1st Workshop on Kernel Isolation,*
405 *Safety and Verification, KISV '23*, pp. 9–17, New York, NY, USA, 2023. Association for Computing
406 Machinery. ISBN 9798400704116. doi: 10.1145/3625275.3625401. URL [https://doi.org/](https://doi.org/10.1145/3625275.3625401)
407 [10.1145/3625275.3625401](https://doi.org/10.1145/3625275.3625401).
- 408 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided
409 abstraction refinement. In *International Conference on Computer Aided Verification*, pp. 154–169.
410 Springer, 2000.
- 411 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided
412 abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794,
413 2003.
- 414 Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix
415 Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. Cwm: An open-weights llm for
416 research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.
- 417 Weituo Dai. verus-study-cases-leetcode. [https://github.com/WeituoDAI/](https://github.com/WeituoDAI/verus-study-cases-leetcode)
418 [verus-study-cases-leetcode](https://github.com/WeituoDAI/verus-study-cases-leetcode), 2025. Accessed: 2025-10-02.
- 419 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan
420 and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp.
421 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 422 Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao
423 Wang, Xiaoran Fan, Zhiheng Xi, et al. StepCoder: improving code generation with reinforcement
424 learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association*
425 *for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024.

- 426 Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and
427 related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995. doi: 10.1137/
428 S0097539793250299.
- 429 Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair
430 with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering
431 Conference and Symposium on the Foundations of Software Engineering*, pp. 1229–1241, 2023.
- 432 Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In
433 *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for
434 Increasing Software Productivity*, FME ’01, pp. 500–517, Berlin, Heidelberg, 2001. Springer-
435 Verlag. ISBN 3540417915.
- 436 Michael L. Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive
437 normal forms. *Journal of Algorithms*, 21(3):618–628, 1996. doi: 10.1006/jagm.1996.0059.
- 438 Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. Ice: A robust
439 framework for learning invariants. In *International Conference on Computer Aided Verification*,
440 pp. 69–87. Springer, 2014.
- 441 Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive
442 validity cores for safety properties. In *Proceedings of the 2016 ACM SIGSOFT International
443 Symposium on Foundations of Software Engineering (FSE)*, pp. 314–325. ACM, 2016. doi:
444 10.1145/2950290.2950349.
- 445 Baoding He, Zenan Li, Wei Sun, Yuan Yao, Taolue Chen, Xiaoxing Ma, and Zhendong Su. Stepwise:
446 Neuro-symbolic proof search for automated systems verification, 2026. URL [https://arxiv.
447 org/abs/2603.19715](https://arxiv.org/abs/2603.19715).
- 448 Zhechong Huang, Zhao Zhang, Zeyu Sun, Huifeng Sun, and Yingfei Xiong. Reinforcement learning
449 with negative tests as completeness signal for formal specification synthesis. *arXiv preprint
450 arXiv:2604.05820*, 2026.
- 451 Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Os-
452 trow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint
453 arXiv:2410.21276*, 2024.
- 454 Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri,
455 Akash Lal, Aseem Rastogi, Subhjit Roy, and Rahul Sharma. Finding inductive loop invariants
456 using large language models. corr abs/2311.07948 (2023). *arXiv preprint arXiv:2311.07948*, 2023.
- 457 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin,
458 Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal
459 verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating
460 systems principles*, pp. 207–220, 2009.
- 461 Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat,
462 Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem
463 proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.
- 464 Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl:
465 Mastering code generation through pretrained models and deep reinforcement learning. *Advances
466 in Neural Information Processing Systems*, 35:21314–21328, 2022.
- 467 Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian
468 Ferdinand. Compert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real
469 Time Software and Systems, 8th European Congress*, 2016.
- 470 Zenan Li, Zhaoyu Li, Kaiyu Yang, Xiaoxing Ma, and Zhendong Su. Learning to disprove: Formal
471 counterexample generation with large language models. *arXiv preprint arXiv:2603.19514*, 2026.
- 472 Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rlrf: Reinforce-
473 ment learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.

- 474 Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish
475 Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for
476 formal software verification, 2024. URL <https://arxiv.org/abs/2406.08467>.
- 477 Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan.
478 Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM*
479 *on Programming Languages*, 9(OOPSLA1):1519–1545, 2025.
- 480 Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving.
481 *arXiv preprint arXiv:2009.03393*, 2020.
- 482 Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic
483 curriculum learning for deep rl: a short survey. In *Proceedings of the Twenty-Ninth International*
484 *Joint Conference on Artificial Intelligence, IJCAI’20*, 2021. ISBN 9780999241165.
- 485 Junhong Shen, Hao Bai, Lunjun Zhang, Yifei Zhou, Amrith Setlur, Shengbang Tong, Diego Caples,
486 Nan Jiang, Tong Zhang, Ameet Talwalkar, et al. Thinking vs. doing: Agents that reason by scaling
487 test-time interaction. *arXiv preprint arXiv:2506.07976*, 2025.
- 488 Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code
489 generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024a.
- 490 Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill,
491 and Clark Barrett. Veristruct: Ai-assisted automated verification of data-structure modules in verus,
492 2026. URL <https://arxiv.org/abs/2510.25015>.
- 493 Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada,
494 Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: verifying liveness of cluster
495 management controllers. In *Proceedings of the 18th USENIX Conference on Operating Systems*
496 *Design and Implementation, OSDI’24, USA, 2024b*. USENIX Association. ISBN 978-1-939133-
497 40-3.
- 498 Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, 2025.
- 499 Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung,
500 Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal
501 reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.
- 502 Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE transactions on*
503 *pattern analysis and machine intelligence*, 44(9):4555–4576, 2021.
- 504 Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li,
505 and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale
506 synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.
- 507 Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong,
508 Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. Autoverus: Automated proof
509 generation for rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):
510 3454–3482, 2025.
- 511 Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R. Lorch, and Shan Lu. Verusage: A
512 study of agent-based verification for rust systems, 2026a. URL [https://arxiv.org/abs/2512.](https://arxiv.org/abs/2512.18436)
513 [18436](https://arxiv.org/abs/2512.18436).
- 514 Jun Yang, Yuechun Sun, Yi Wu, Rodrigo Caridad, Yongwei Yuan, Jianan Yao, Shan Lu, and
515 Kexin Pei. Exverus: Verus proof repair via counterexample reasoning, 2026b. URL <https://arxiv.org/abs/2603.25810>.
- 517 Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J
518 Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented
519 language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.
- 520 Yiqi Zhang, Huiqiang Jiang, Xufang Luo, Zhihe Yang, Chengruidong Zhang, Yifei Shen, Dongsheng
521 Li, Yuqing Yang, Lili Qiu, and Yang You. Sortedrl: Accelerating rl training for llms through online
522 length-aware scheduling. *arXiv preprint arXiv:2603.23414*, 2026.

- 523 Si Cheng Zhong and Xujie Si. Towards repository-level program verification with large language
 524 models. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models
 525 and Programming Languages*, pp. 27–39, 2025.
- 526 Sicheng Zhong, Jiading Zhu, Yifang Tian, and Xujie Si. Rag-verus: Repository-level program
 527 verification with llms using retrieval augmented generation. *arXiv preprint arXiv:2502.05344*,
 528 2025.
- 529 Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. Verismo:
 530 a verified security module for confidential vms. In *Proceedings of the 18th USENIX Conference
 531 on Operating Systems Design and Implementation, OSDI’24, USA, 2024*. USENIX Association.
 532 ISBN 978-1-939133-40-3.

533 A Inductive Subset Enumeration Algorithm

534 A.1 Problem Formulation and Computational Complexity

535 Let $\mathcal{V} = \{v_1, \dots, v_n\}$ be the set of loop invariant clauses in a verified Verus program. A subset
 536 $S \subseteq \mathcal{V}$ is *inductive* if the program annotated with only the clauses in S passes the Verus verifier. The
 537 family of all inductive subsets forms an *antichain-closed* (monotone) collection: if S is inductive and
 538 $S \subseteq S'$, then S' is also inductive. The dependency graph $G_{\mathcal{V}}$ is derived from the *minimal inductive*
 539 *subsets* \mathcal{I}_{\min} (Definitions 3.3–3.6 in the main text).

540 **Hardness of complete enumeration.** Enumerating *all* minimal inductive subsets of a mono-
 541 tone Boolean oracle is equivalent to the *Hypergraph Transversal Enumeration* problem (Trans-
 542 Enum) (Eiter & Gottlob, 1995). Formally, given a hypergraph $H = ([n], \mathcal{E})$, a *minimal transversal* is
 543 a minimal set $T \subseteq [n]$ that intersects every hyperedge; enumerating all minimal transversals is pre-
 544 cisely our problem when each hyperedge encodes a “necessary support” constraint for inductiveness.

545 This problem occupies a rare position in the complexity landscape (Eiter & Gottlob, 1995). Despite
 546 30+ years of effort, it is not known to be solvable in polynomial time for general hypergraphs. At
 547 the same time, it is not known to be NP-hard: were it NP-hard, the existence of a quasi-polynomial
 548 algorithm would imply $\text{NP} \neq \text{co-NP}$, a separation stronger than $\text{P} \neq \text{NP}$. The best known algorithm
 549 runs in quasi-polynomial time $N^{O(\log N / \log \log N)}$ (Fredman & Khachiyan, 1996), where N is the
 550 combined size of the input and output. This problem is the enumeration-complexity analogue of
 551 Graph Isomorphism: believed hard, but provably outside standard NP-hardness. The connection to
 552 formal verification is not coincidental: the same structure arises as *Minimal Inductive Validity Core*
 553 (MIVC) enumeration (Ghassabani et al., 2016), where one seeks all minimal subsets of a transition
 554 system sufficient to prove a safety property.

555 The intractability of exact enumeration motivates a practical approximation: collect a sufficient
 556 sample of inductive subsets within a fixed verifier call budget B , yielding an approximate \mathcal{I}_{\min} that
 557 is exact for the vast majority of programs and near-exact for the rest.

558 A.2 Level-wise BFS Enumeration Algorithm

559 Given that exact enumeration is intractable in general, we design an algorithm that (i) is *correct*
 560 when the budget suffices to exhaust the search space, and (ii) degrades gracefully to a well-structured
 561 approximation when the budget is exceeded. The algorithm exploits the antichain-closed structure
 562 of $\mathcal{I}(\mathcal{V})$ via a level-wise breadth-first search: subsets are enumerated in increasing cardinality order
 563 $k = 1, 2, \dots, n$, and any superset of an already-found inductive set is pruned without a verifier call.

564 *Remark* (Correctness under mutual inductiveness). Some invariant groups are *mutually inductive*:
 565 no individual clause $\{v_i\}$ verifies on its own, but the group $S = \{v_i, v_j, \dots\}$ does (these form SCCs
 566 in $G_{\mathcal{V}}$). A seed-based approach that initializes from singletons would produce an empty queue for
 567 such groups and miss them entirely. The level-wise algorithm below handles this correctly by starting
 568 from $k = 1$ and increasing k until the first inductive subsets are found.

Algorithm 1 Level-wise BFS Enumeration of Minimal Inductive Subsets

Require: Invariant set $\mathcal{V} = \{v_1, \dots, v_n\}$, verifier oracle $\text{VERIFY}(S) \rightarrow \{\text{true}, \text{false}\}$, budget B **Ensure:** Family \mathcal{I}_{\min} of minimal inductive subsets

```
1:  $\mathcal{I}_{\min} \leftarrow \emptyset$ ; Visited  $\leftarrow \emptyset$ ; calls  $\leftarrow 0$ 
2: for  $k = 1, 2, \dots, n$  do ▷ level = subset size
3:   for each  $S \in \binom{\mathcal{V}}{k}$  in lexicographic order do
4:     if  $S \in \text{Visited}$  then continue
5:     end if
6:     Visited  $\leftarrow \text{Visited} \cup \{S\}$ 
7:     if  $\exists T \in \mathcal{I}_{\min} : T \subseteq S$  then continue ▷ antichain pruning
8:     end if
9:     if calls  $\geq B$  then goto DONE
10:    end if
11:    calls  $\leftarrow$  calls + 1
12:    if  $\text{VERIFY}(S)$  then
13:       $\mathcal{I}_{\min} \leftarrow \mathcal{I}_{\min} \cup \{S\}$ 
14:    end if
15:  end for
16:  if calls  $\geq B$  then break
17:  end if
18: end for
19: DONE: return  $\mathcal{I}_{\min}$ 
```

569 **A.3 Complexity Analysis**

570 **Verifier call budget.** Each subset S is tested at most once (tracked in Visited). The *antichain*
571 *pruning* step (line 5 of Algorithm 1) skips any superset of an already-found inductive set: once a
572 minimal inductive subset T is found, all $2^{n-|T|}$ supersets are guaranteed inductive by the antichain-
573 closed property and are pruned without further verifier calls. In the best case (self-inductive singletons
574 exist), the algorithm terminates at level $k = 1$ with n calls and prunes the remaining $2^n - n - 1$
575 supersets. In the general case, the worst-case call count is $\sum_{k=1}^{k^*} \binom{n}{k}$ where k^* is the size of the
576 smallest inductive subset, compared to $2^n - 1$ for naive enumeration. For programs with $n > 8$ (2%
577 of our dataset), we apply a budget cap of $B = 256$ calls.

578 **Empirical preprocessing statistics.** We ran the level-wise BFS enumeration offline on all 531
579 verified programs across five benchmark datasets (Table 3). The median program has $n = 3$ invariants;
580 98% have $n \leq 8$. During actual training, we only use dependency graphs from 249 training samples.

Table 3: Preprocessing statistics per dataset. “Calls” = actual verifier invocations (total). “Naive”
= calls without the budget cap ($2^n - 1$ per file, summed). “Reduction” = Naive / Calls. “Est. time”
assumes 0.42 s per Verus call (measured mean on our hardware; range 0.25–0.71 s).

Dataset	Files	n (mean)	Calls (total)	Naive (total)	Reduction	Est. time
Dafnybench	55	1.9	205	205	1.0×	1.4 min
HumanEval-AlphaVerus	47	3.4	1,135	2,669	2.4×	7.9 min
LeetCode	23	6.6	2,817	72,955	25.9×	19.7 min
Obfuscated-VerusBench	266	3.0	3,498	3,498	1.0×	24.5 min
cleaned-VerusBench	140	2.7	1,561	1,816	1.2×	10.9 min
Total	531	3.0	9,216	81,143	8.8×	≈64 min

581 This concretizes the approximation motivated in §A.1: for the 98% of programs with $n \leq 8$,
582 the full search space ($2^n - 1 \leq 255$ subsets) is exhausted within budget $B = 256$, so \mathcal{I}_{\min} is
583 *exact*—equivalent to the intractable complete enumeration, achieved cheaply because our benchmark
584 programs are small. For the remaining 2% ($n > 8$, all from LeetCode, $n \leq 16$), the budget cap
585 triggers the approximation: the returned \mathcal{I}_{\min} contains the minimal inductive subsets found within B
586 calls, which are guaranteed to be *locally* minimal (no proper subset was found inductive) but may
587 not be globally minimal. The level-wise traversal order ensures that small subsets are tested first,

588 so the approximation is biased toward finding small inductive sets—the most informative ones for
 589 dependency graph construction. These 9 files constitute a negligible fraction of the training data,
 590 and the overall reduction in verifier calls is $8.8\times$ (9,216 actual vs. 81,143 naive), with LeetCode
 591 benefiting most ($25.9\times$).

592 **Dependency graph construction and amortization.** Given \mathcal{I}_{\min} , the dependency edges E are
 593 computed in $O(|\mathcal{I}_{\min}| \cdot n^2)$ time (Definition 3.7), and SCC condensation via Tarjan’s algorithm runs
 594 in $O(n + |E|)$ —both polynomial in n . The entire offline preprocessing—9,216 verifier calls across
 595 531 programs—completes in approximately 64 minutes on a single machine (measured mean Verus
 596 call time: 0.42 s; range 0.25–0.71 s across benchmarks), and is performed once before training begins.
 597 At training time, the reward function performs only a dictionary lookup ($O(1)$) plus topological order
 598 evaluation ($O(n + |E|)$), adding negligible overhead to the RL rollout.

599 B Qualitative Counterexample Case Studies

600 We include three representative examples where the generated counterexample context directly
 601 identifies the proof obligation that should be repaired. In each example, the display keeps the
 602 source-level artifact fixed: the left panel contains the task and buggy proof fragment, while the
 603 right panel contains the raw verifier message and the generated [State], [Execution Trace],
 604 and [Violation] fields. The accompanying text below analyzes why the raw error message is
 605 insufficient and how the counterexample makes the repair target explicit.

606 **Assertion placement.** The first case illustrates a failure mode where the relevant counterexample
 607 is an intermediate program state rather than a loop-entry or loop-exit state. The task is to move
 608 elements from `shrink` into `grow` while preserving the sum of their lengths. The loop invariant is
 609 semantically correct at the loop boundary, but the assertion is placed after `shrink.pop()` and before
 610 `grow.push(i)`, where the length sum is temporarily one smaller. The verifier message reports only
 611 an assertion failure at the source line, so it does not distinguish a wrong invariant from an assertion
 612 placed at the wrong program point. The generated counterexample makes this distinction explicit by
 613 tracing the execution to the precise state where the assertion is evaluated.

Case Study A: Assertion Checked at the Wrong Program Point

Task. Dafnybench/program_422_count_less_than
 Move elements from `shrink` to `grow` while preserving the total length.

Buggy code.

```
fn count_less_than(numbers: Vec<u32>, threshold: u32)
  -> (count: u32)
  ensures count == numbers.len() as u32,
{
  let mut shrink = numbers.clone();
  let mut grow: Vec<u32> = Vec::new();

  while shrink.len() > 0
    invariant
      grow.len() + shrink.len() == numbers.len(),
      decreases shrink.len(),
    {
      let i = shrink.pop().unwrap();
      // BUG: checked before grow.push(i).
      assert(grow.len() + shrink.len()
        == numbers.len());
      grow.push(i);
    }
  assert(numbers.len() == grow.len());
  grow.len() as u32
}
```

The assertion is checked after `pop` and before the compensating `push`; the loop invariant only holds at loop boundaries.

Verus error.

```
error: assertion failed
--> program_422_count_less_than.rs:19:16
|
19 | assert(grow.len() + shrink.len() == numbers.len());
|
```

Counterexample.

[State] `numbers = [1,2,3]`, `threshold = 5`. At loop entry: `grow = []`, `shrink = [1,2,3]`; hence `grow.len() + shrink.len() = 0 + 3 = 3`.

[Execution Trace] Execute the first loop iteration. The statement `shrink.pop()` removes one element, so `shrink = [1,2]` while `grow` is still empty. The assertion is checked immediately at this intermediate point, before `grow.push(i)`.

[Violation] At the assertion site, `grow.len() + shrink.len() = 0 + 2 = 2`, but `numbers.len() = 3`. The invariant is re-established only after the subsequent `push`; it cannot justify an assertion placed between the two updates.

614

615 **Prefix-scoped semantic invariant.** The second case demonstrates how counterexample analysis
 616 identifies the scope of a semantic loop invariant. The function computes the maximum profit over a
 617 price sequence, but during the loop `max_pro` only summarizes the prefix processed so far. The buggy
 618 proof copies the final full-array property into the loop invariant, requiring `max_pro` to bound profits
 619 involving future prices even before the first iteration. The verifier reports that the invariant is not

620 satisfied before the loop, but the message does not reveal which quantifier range is too strong. The
 621 generated counterexample instantiates the full-array claim with `prices = [5,3,7]`, `j = 0`, and `k`
 622 `= 2`, yielding the false obligation $7 - 5 \leq 0$. This turns a coarse invariant failure into a localized
 623 repair target: replace the full-array range with the processed-prefix range `k < i`.

Case Study B: Full-Array Claim Used as a Prefix Invariant

Task. Leetcode/121-BestTimeSell

Compute the maximum profit from one buy-sell pair in a price sequence.

Buggy code.

```
pub fn max_profit(prices: Vec<i32>) -> (res: i32)
  requires
    1 <= prices.len() <= 100000,
    forall|i: int| 0 <= i < prices.len()
      ==> 0 <= prices[i] <= 100000,
  ensures
    forall|j: int, k: int|
      0 <= j < k < prices.len()
      ==> prices[k] - prices[j] <= res,
{
  let mut max_pro = 0;
  let mut min_price = i32::MAX;

  for i in 0..prices.len()
    invariant
      forall|j: int| 0 <= j < i
        ==> prices[j] >= min_price,
      i > 0 ==> exists|j: int| 0 <= j < i
        && prices[j] == min_price,
      // BUG: full-array range; should be k < i.
      forall|j: int, k: int|
        0 <= j < k < prices.len()
        ==> prices[k] - prices[j] <= max_pro,
  {
    min_price = min_i32(min_price, prices[i]);
    max_pro = max_i32(max_pro, prices[i] - min_price)
  }
  max_pro
}
```

The last invariant talks about all future days. It should only summarize the processed prefix: `0 <= j < k < i`.

Verus error.

```
error: invariant not satisfied before loop
--> best_time_sell_v2.rs:37:13
|
37 | forall|j: int, k: int|
| .....
```

Counterexample.

[State] `prices = [5,3,7]`. Before the first loop iteration, `i = 0`, `max_pro = 0`, and `min_price = i32::MAX`. All input bounds hold.

[Execution Trace] Verus checks the loop invariants at entry. The buggy invariant requires every pair in the whole array to have profit at most `max_pro`. Instantiate it with `j = 0`, `k = 2`. Then `prices[2] - prices[0] = 7 - 5 = 2`.

[Violation] The invariant demands `2 <= max_pro = 0`, which is false. The counterexample identifies the quantifier range as the root cause: before processing any day, the proof can only make claims about the empty prefix `0..i`, not about the full input sequence.

624

625 **Arithmetic-safety context.** The third case shows that a counterexample can diagnose proof-context
 626 loss rather than a concrete runtime overflow. The function preconditions already state that each square
 627 and cube of an input element is within the `i32` range, but the loop invariant only records the output
 628 length and the values already pushed. Consequently, the verifier reports a possible overflow at the
 629 multiplication site, without indicating that the missing ingredient is the arithmetic-bound precondition
 630 inside the loop context. The generated counterexample keeps the execution state concrete and benign,
 631 then explains why the verifier still rejects it: at the current index `i`, the loop context does not contain
 632 the bound needed to prove `nums[i] * nums[i] * nums[i]` safe.

Table 4: Ordinary Pass, Safe Pass, and safe rate on the 63-problem TTS validation set from Verus-Bench, DafnyBench, HumanEval, and LeetCode. Safe rate is computed as Safe Pass divided by ordinary Pass at the endpoint budget.

Method	Sample scaling, $K = 32$			Iteration scaling, 32 iters		
	Pass	Safe Pass	Safe Rate	Pass	Safe Pass	Safe Rate
CoT-Init	87.3	60.3	69.1%	76.2	54.0	70.8%
CTX-Init	85.7	61.9	72.2%	73.0	42.9	58.7%
CTX-Dep	90.5	79.4	87.7%	79.4	69.8	88.0%
Qwen3-Coder	57.1	41.3	72.2%	46.0	30.2	65.5%
Sonnet-4.5	90.5	73.0	80.7%	85.7	54.0	63.0%

Case Study C: Missing Arithmetic-Safety Fact in the Loop Context

Task. cleaned-verusbench/task_id.447

Return a vector containing the cube of each input element.

Buggy code.

```
fn cube_element(nums: &Vec<i32>) -> (cubed: Vec<i32>)
  requires
    forall|k: int| 0 <= k < nums.len()
      ==> i32::MIN <= nums[k] * nums[k]
        <= i32::MAX,
    forall|k: int| 0 <= k < nums.len()
      ==> i32::MIN <= nums[k] * nums[k] * nums[k]
        <= i32::MAX,
  ensures
    forall|i: int| 0 <= i < nums.len()
      ==> cubed[i] == nums[i] * nums[i] * nums[i],
{
  let mut cubed_array: Vec<i32> = Vec::new();
  let mut i = 0;
  while i < nums.len()
    invariant
      i <= nums.len(),
      cubed_array.len() == i,
      forall|k: int| 0 <= k < i
        ==> cubed_array[k] == nums[k] * nums[k] *
        nums[k],
      decreases nums.len() - i,
    {
      // BUG: loop invariants omit cube bounds.
      cubed_array.push(nums[i] * nums[i] * nums[i]);
      i += 1;
    }
  cubed_array
}
```

The function precondition has the needed arithmetic bounds, but the loop invariant does not carry them into the loop body.

Verus error.

```
error: possible arithmetic underflow/overflow
--> task_id.447.rs:40:26
|
40 | cubed_array.push(nums[i] * nums[i] * nums[i]);
|
```

Counterexample.

[State] $nums = [10, 20, 30]$. At the first iteration, $i = 0$, $cubed_array.len() = 0$, and all function preconditions are satisfied.

[Execution Trace] The loop guard gives $i < nums.len()$. The program then evaluates $nums[i] * nums[i] * nums[i]$. Although the function precondition bounds every square and cube, the loop invariant only states the index bound, result length, and previously computed elements.

[Violation] Verus cannot prove that the current multiplication is within the $i32$ range from the loop context alone. The repair is to carry the arithmetic-bound facts as loop invariants, so the verifier can instantiate them at the current index i .

633

634 C Additional Results

635 C.1 Safe Rate under Test-Time Scaling

636 Table 4 compares ordinary verifier Pass with Safe Pass at the endpoint budgets used in Figure 3. We
 637 define the safe rate as $SafePass/Pass$, i.e., the fraction of verifier-passing successes retained after
 638 filtering repairs that modify executable code. CTX-Dep has the highest safe rate in both regimes:
 639 87.7% for sample scaling and 88.0% for iteration scaling. This indicates that its test-time scaling
 640 gains are not primarily due to unsafe verifier-satisfying edits; most of its additional verifier successes
 641 survive the safety filter.

642 D Prompt Templates

643 We document the prompt templates used during RL training and iterative test-time evaluation. Both
 644 templates elicit the same structured output format: a chain-of-thought reasoning block (`think`), a
 645 semi-formal counterexample block (`counterexample`) with three mandatory subsections (`[State]`,
 646 `[Execution Trace]`, `[Violation]`), a repair rationale block (`reasoning`), and a single rust
 647 code block containing the repaired program.

648 **D.1 RL Training Prompt**

649 During GRPO training, each rollout is conditioned on a fixed system message that establishes the
650 model’s role and a per-problem user message that supplies the buggy program and the Verus verifier
651 output.

System Message

You are an expert in Verus, a formal verification framework for Rust. You fix buggy Verus programs by analyzing verification errors, constructing concrete counterexamples, and writing repaired code that passes the verifier.

Common Verus fix patterns:

- Missing decreases clause: add ‘decreases <expr>’ to while/for loops
- Postcondition not satisfied: add loop invariants that mirror the postcondition
- Precondition not satisfied: add invariants proving bounds (e.g., `index < len`)
- Arithmetic overflow: add bounds invariants or use checked arithmetic
- Recommendation not met: add ‘assert’ to guide the prover

652

User Message

Fix the following buggy Verus Rust program so it passes the verifier.

Respond with these sections in order:

1. <counterexample> with [State], [Execution Trace], [Violation]
2. <reasoning> with a concise repair plan
3. A single ‘‘rust ... ‘‘ block with the complete fixed program

Benchmark: {benchmark}
Problem: {problem}

[Buggy Code]
‘‘rust
{buggy_code}
‘‘

[Verifier Error]
{verifier_error}

653

654 **D.2 Iterative Repair Prompt (Refinement Turn)**

655 During test-time scaling, the model is queried for up to T iterations. For iteration $t \geq 2$, a refinement
656 turn is appended after the original prompt, supplying the previous repair attempt and the verifier’s
657 updated error log. This enables the model to construct a fresh counterexample and revise its repair
658 strategy based on the remaining verification failures.

Refinement Turn (appended after the original prompt, iteration $t \geq 2$)

{original_prompt}

Previous Attempt
The code generated in the previous turn was:
‘‘rust
{previous_code}
‘‘

659

Verifier Output

The Verus verifier reported the following errors:

{verifier_output}

Task

Analyze the remaining verification errors. Respond with:

1. <counterexample> with [State], [Execution Trace], [Violation]
2. <reasoning> with a concise repair plan
3. A single `'''rust ... '''` block with the complete fixed program.

660

661 **NeurIPS Paper Checklist**

662 The checklist is designed to encourage best practices for responsible machine learning research,
663 addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove
664 the checklist: **The papers not including the checklist will be desk rejected.** The checklist should
665 follow the references and follow the (optional) supplemental material. The checklist does NOT count
666 towards the page limit.

667 Please read the checklist guidelines carefully for information on how to answer these questions. For
668 each question in the checklist:

- 669 • You should answer [Yes], [No], or [N/A].
- 670 • [N/A] means either that the question is Not Applicable for that particular paper or the
671 relevant information is Not Available.
- 672 • Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

673 **The checklist answers are an integral part of your paper submission.** They are visible to the
674 reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it
675 (after eventual revisions) with the final version of your paper, and its final version will be published
676 with the paper.

677 The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation.
678 While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a
679 proper justification is given (e.g., error bars are not reported because it would be too computationally
680 expensive” or “we were unable to find the license for the dataset we used”). In general, answering
681 [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we
682 acknowledge that the true answer is often more nuanced, so please just use your best judgment and
683 write a justification to elaborate. All supporting evidence can appear either in the main paper or the
684 supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification
685 please point to the section(s) where related material for the question can be found.

686 **1. Claims**

687 Question: Do the main claims made in the abstract and introduction accurately reflect the
688 paper’s contributions and scope?

689 Answer: [Yes]

690 Justification: The abstract and introduction accurately reflect the paper’s contributions and
691 scope.

692 Guidelines:

- 693 • The answer [N/A] means that the abstract and introduction do not include the claims
694 made in the paper.
- 695 • The abstract and/or introduction should clearly state the claims made, including the
696 contributions made in the paper and important assumptions and limitations. A [No] or
697 [N/A] answer to this question will not be perceived well by the reviewers.
- 698 • The claims made should match theoretical and experimental results, and reflect how
699 much the results can be expected to generalize to other settings.
- 700 • It is fine to include aspirational goals as motivation as long as it is clear that these goals
701 are not attained by the paper.

702 **2. Limitations**

703 Question: Does the paper discuss the limitations of the work performed by the authors?

704 Answer: [Yes]

705 Justification: The limitation discussion can be found in the Discussion and Conclusion
706 section.

707 Guidelines:

- 708 • The answer [N/A] means that the paper has no limitation while the answer [No] means
709 that the paper has limitations, but those are not discussed in the paper.

- 710 • The authors are encouraged to create a separate “Limitations” section in their paper.
- 711 • The paper should point out any strong assumptions and how robust the results are to
- 712 violations of these assumptions (e.g., independence assumptions, noiseless settings,
- 713 model well-specification, asymptotic approximations only holding locally). The authors
- 714 should reflect on how these assumptions might be violated in practice and what the
- 715 implications would be.
- 716 • The authors should reflect on the scope of the claims made, e.g., if the approach was
- 717 only tested on a few datasets or with a few runs. In general, empirical results often
- 718 depend on implicit assumptions, which should be articulated.
- 719 • The authors should reflect on the factors that influence the performance of the approach.
- 720 For example, a facial recognition algorithm may perform poorly when image resolution
- 721 is low or images are taken in low lighting. Or a speech-to-text system might not be
- 722 used reliably to provide closed captions for online lectures because it fails to handle
- 723 technical jargon.
- 724 • The authors should discuss the computational efficiency of the proposed algorithms
- 725 and how they scale with dataset size.
- 726 • If applicable, the authors should discuss possible limitations of their approach to
- 727 address problems of privacy and fairness.
- 728 • While the authors might fear that complete honesty about limitations might be used by
- 729 reviewers as grounds for rejection, a worse outcome might be that reviewers discover
- 730 limitations that aren’t acknowledged in the paper. The authors should use their best
- 731 judgment and recognize that individual actions in favor of transparency play an impor-
- 732 tant role in developing norms that preserve the integrity of the community. Reviewers
- 733 will be specifically instructed to not penalize honesty concerning limitations.

734 3. Theory assumptions and proofs

735 Question: For each theoretical result, does the paper provide the full set of assumptions and
 736 a complete (and correct) proof?

737 Answer: [Yes]

738 Justification: The theoretical formulations are defined in Section 3.

739 Guidelines:

- 740 • The answer [N/A] means that the paper does not include theoretical results.
- 741 • All the theorems, formulas, and proofs in the paper should be numbered and cross-
- 742 referenced.
- 743 • All assumptions should be clearly stated or referenced in the statement of any theorems.
- 744 • The proofs can either appear in the main paper or the supplemental material, but if
- 745 they appear in the supplemental material, the authors are encouraged to provide a short
- 746 proof sketch to provide intuition.
- 747 • Inversely, any informal proof provided in the core of the paper should be complemented
- 748 by formal proofs provided in appendix or supplemental material.
- 749 • Theorems and Lemmas that the proof relies upon should be properly referenced.

750 4. Experimental result reproducibility

751 Question: Does the paper fully disclose all the information needed to reproduce the main ex-
 752 perimental results of the paper to the extent that it affects the main claims and/or conclusions
 753 of the paper (regardless of whether the code and data are provided or not)?

754 Answer: [Yes]

755 Justification: The experiment details are in Section 4.

756 Guidelines:

- 757 • The answer [N/A] means that the paper does not include experiments.
- 758 • If the paper includes experiments, a [No] answer to this question will not be perceived
- 759 well by the reviewers: Making the paper reproducible is important, regardless of
- 760 whether the code and data are provided or not.
- 761 • If the contribution is a dataset and/or model, the authors should describe the steps taken
- 762 to make their results reproducible or verifiable.

- 763 • Depending on the contribution, reproducibility can be accomplished in various ways.
764 For example, if the contribution is a novel architecture, describing the architecture fully
765 might suffice, or if the contribution is a specific model and empirical evaluation, it may
766 be necessary to either make it possible for others to replicate the model with the same
767 dataset, or provide access to the model. In general, releasing code and data is often
768 one good way to accomplish this, but reproducibility can also be provided via detailed
769 instructions for how to replicate the results, access to a hosted model (e.g., in the case
770 of a large language model), releasing of a model checkpoint, or other means that are
771 appropriate to the research performed.
- 772 • While NeurIPS does not require releasing code, the conference does require all submis-
773 sions to provide some reasonable avenue for reproducibility, which may depend on the
774 nature of the contribution. For example
 - 775 (a) If the contribution is primarily a new algorithm, the paper should make it clear how
776 to reproduce that algorithm.
 - 777 (b) If the contribution is primarily a new model architecture, the paper should describe
778 the architecture clearly and fully.
 - 779 (c) If the contribution is a new model (e.g., a large language model), then there should
780 either be a way to access this model for reproducing the results or a way to reproduce
781 the model (e.g., with an open-source dataset or instructions for how to construct
782 the dataset).
 - 783 (d) We recognize that reproducibility may be tricky in some cases, in which case
784 authors are welcome to describe the particular way they provide for reproducibility.
785 In the case of closed-source models, it may be that access to the model is limited in
786 some way (e.g., to registered users), but it should be possible for other researchers
787 to have some path to reproducing or verifying the results.

788 5. Open access to data and code

789 Question: Does the paper provide open access to the data and code, with sufficient instruc-
790 tions to faithfully reproduce the main experimental results, as described in supplemental
791 material?

792 Answer: [Yes]

793 Justification: The data and code are in supplemental materials.

794 Guidelines:

- 795 • The answer [N/A] means that paper does not include experiments requiring code.
- 796 • Please see the NeurIPS code and data submission guidelines ([https://neurips.cc/
797 public/guides/CodeSubmissionPolicy](https://neurips.cc/public/guides/CodeSubmissionPolicy)) for more details.
- 798 • While we encourage the release of code and data, we understand that this might not
799 be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not
800 including code, unless this is central to the contribution (e.g., for a new open-source
801 benchmark).
- 802 • The instructions should contain the exact command and environment needed to run to
803 reproduce the results. See the NeurIPS code and data submission guidelines ([https://
804 neurips.cc/public/guides/CodeSubmissionPolicy](https://neurips.cc/public/guides/CodeSubmissionPolicy)) for more details.
- 805 • The authors should provide instructions on data access and preparation, including how
806 to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- 807 • The authors should provide scripts to reproduce all experimental results for the new
808 proposed method and baselines. If only a subset of experiments are reproducible, they
809 should state which ones are omitted from the script and why.
- 810 • At submission time, to preserve anonymity, the authors should release anonymized
811 versions (if applicable).
- 812 • Providing as much information as possible in supplemental material (appended to the
813 paper) is recommended, but including URLs to data and code is permitted.

814 6. Experimental setting/details

815 Question: Does the paper specify all the training and test details (e.g., data splits, hyperpa-
816 rameters, how they were chosen, type of optimizer) necessary to understand the results?

817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866

Answer: [Yes]

Justification: The experiment details are in Section 4.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: The experiment details are in Section 4.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: The experiment details are in Section 4.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. Code of ethics

867 Question: Does the research conducted in the paper conform, in every respect, with the
868 NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

869 Answer: [Yes]

870 Justification: The research conforms with the Code of Ethics.

871 Guidelines:

- 872 • The answer [N/A] means that the authors have not reviewed the NeurIPS Code of
873 Ethics.
- 874 • If the authors answer [No], they should explain the special circumstances that require a
875 deviation from the Code of Ethics.
- 876 • The authors should make sure to preserve anonymity (e.g., if there is a special consid-
877 eration due to laws or regulations in their jurisdiction).

878 10. Broader impacts

879 Question: Does the paper discuss both potential positive societal impacts and negative
880 societal impacts of the work performed?

881 Answer: [Yes]

882 Justification: The broader impacts are discussed in the supplemental materials.

883 Guidelines:

- 884 • The answer [N/A] means that there is no societal impact of the work performed.
- 885 • If the authors answer [N/A] or [No], they should explain why their work has no societal
886 impact or why the paper does not address societal impact.
- 887 • Examples of negative societal impacts include potential malicious or unintended uses
888 (e.g., disinformation, generating fake profiles, surveillance), fairness considerations
889 (e.g., deployment of technologies that could make decisions that unfairly impact specific
890 groups), privacy considerations, and security considerations.
- 891 • The conference expects that many papers will be foundational research and not tied
892 to particular applications, let alone deployments. However, if there is a direct path to
893 any negative applications, the authors should point it out. For example, it is legitimate
894 to point out that an improvement in the quality of generative models could be used to
895 generate Deepfakes for disinformation. On the other hand, it is not needed to point out
896 that a generic algorithm for optimizing neural networks could enable people to train
897 models that generate Deepfakes faster.
- 898 • The authors should consider possible harms that could arise when the technology is
899 being used as intended and functioning correctly, harms that could arise when the
900 technology is being used as intended but gives incorrect results, and harms following
901 from (intentional or unintentional) misuse of the technology.
- 902 • If there are negative societal impacts, the authors could also discuss possible mitigation
903 strategies (e.g., gated release of models, providing defenses in addition to attacks,
904 mechanisms for monitoring misuse, mechanisms to monitor how a system learns from
905 feedback over time, improving the efficiency and accessibility of ML).

906 11. Safeguards

907 Question: Does the paper describe safeguards that have been put in place for responsible
908 release of data or models that have a high risk for misuse (e.g., pre-trained language models,
909 image generators, or scraped datasets)?

910 Answer: [Yes]

911 Justification: The experiment details are in Section 4. Verus verifier execution is safe and
912 has no risk.

913 Guidelines:

- 914 • The answer [N/A] means that the paper poses no such risks.
- 915 • Released models that have a high risk for misuse or dual-use should be released with
916 necessary safeguards to allow for controlled use of the model, for example by requiring
917 that users adhere to usage guidelines or restrictions to access the model or implementing
918 safety filters.

- 919
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- 920
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.
- 921
- 922
- 923

924 12. Licenses for existing assets

925 Question: Are the creators or original owners of assets (e.g., code, data, models), used in
926 the paper, properly credited and are the license and terms of use explicitly mentioned and
927 properly respected?

928 Answer: [Yes]

929 Justification: The experiment details are in Section 4. All used datasets are public.

930 Guidelines:

- 931 • The answer [N/A] means that the paper does not use existing assets.
- 932 • The authors should cite the original paper that produced the code package or dataset.
- 933 • The authors should state which version of the asset is used and, if possible, include a
934 URL.
- 935 • The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- 936 • For scraped data from a particular source (e.g., website), the copyright and terms of
937 service of that source should be provided.
- 938 • If assets are released, the license, copyright information, and terms of use in the
939 package should be provided. For popular datasets, paperswithcode.com/datasets
940 has curated licenses for some datasets. Their licensing guide can help determine the
941 license of a dataset.
- 942 • For existing datasets that are re-packaged, both the original license and the license of
943 the derived asset (if it has changed) should be provided.
- 944 • If this information is not available online, the authors are encouraged to reach out to
945 the asset's creators.

946 13. New assets

947 Question: Are new assets introduced in the paper well documented and is the documentation
948 provided alongside the assets?

949 Answer: [Yes]

950 Justification: The trained model and data are in supplemental materials.

951 Guidelines:

- 952 • The answer [N/A] means that the paper does not release new assets.
- 953 • Researchers should communicate the details of the dataset/code/model as part of their
954 submissions via structured templates. This includes details about training, license,
955 limitations, etc.
- 956 • The paper should discuss whether and how consent was obtained from people whose
957 asset is used.
- 958 • At submission time, remember to anonymize your assets (if applicable). You can either
959 create an anonymized URL or include an anonymized zip file.

960 14. Crowdsourcing and research with human subjects

961 Question: For crowdsourcing experiments and research with human subjects, does the paper
962 include the full text of instructions given to participants and screenshots, if applicable, as
963 well as details about compensation (if any)?

964 Answer: [N/A]

965 Justification: We did not do crowdsourcing experiments.

966 Guidelines:

- 967 • The answer [N/A] means that the paper does not involve crowdsourcing nor research
968 with human subjects.

- 969
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- 970
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.
- 971
- 972
- 973
- 974

975 **15. Institutional review board (IRB) approvals or equivalent for research with human**
976 **subjects**

977 Question: Does the paper describe potential risks incurred by study participants, whether
978 such risks were disclosed to the subjects, and whether Institutional Review Board (IRB)
979 approvals (or an equivalent approval/review based on the requirements of your country or
980 institution) were obtained?

981 Answer: [N/A]

982 Justification: We did not do crowdsourcing experiments.

983 Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
 - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
 - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
 - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.
- 984
- 985
- 986
- 987
- 988
- 989
- 990
- 991
- 992
- 993

994 **16. Declaration of LM usage**

995 Question: Does the paper describe the usage of LMs if it is an important, original, or
996 non-standard component of the core methods in this research? Note that if the LM is used
997 only for writing, editing, or formatting purposes and does *not* impact the core methodology,
998 scientific rigor, or originality of the research, declaration is not required.

999 Answer: [N/A]

1000 Justification: LM is used only for writing, editing, or formatting purposes and does not
1001 impact the core methodology, scientific rigor, or originality of the research, declaration is
1002 not required.

1003 Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LMs as any important, original, or non-standard components.
 - Please refer to our LM policy in the NeurIPS handbook for what should or should not be described.
- 1004
- 1005
- 1006
- 1007