ExVerus: Verus Proof Repair via Counterexample Reasoning

ANONYMOUS AUTHOR(S)

Large Language Models (LLMs) have shown promising results in automating formal verification. However, existing approaches treat proof generation as a static, end-to-end prediction over source code, relying on limited verifier feedback and lacking access to concrete program behaviors. We present ExVerus, a counterexample-guided framework that enables LLMs to reason about proofs using behavioral feedback via counterexamples. When a proof fails, ExVerus automatically generates and validates counterexamples, and then guides the LLM to generalize them into inductive invariants to block these failures. Our evaluation shows that ExVerus significantly improves proof accuracy, robustness, and token efficiency over the state-of-the-art LLM-based Verus proof generator.

ACM Reference Format:
Anonymous Author(s). 2025. ExVerus: Verus Proof Repair via Counterexample Reasoning. 1, 1 (October 2025), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in formal verification, a domain traditionally requiring significant human effort. It has been increasingly promising that LLMs can automate the proof development process by generating useful proof annotations and scripts for a broad spectrum of proof assistants and verifiers [2, 9, 12, 26, 34, 38, 41, 42, 51, 57], making formal methods more accessible to developers without specialized expertise.

Among these verifiers, Verus [35, 36] has been particularly amenable to developers to verify real-world systems [40, 52, 63]. Thanks to its Rust-native design, Verus brings verification closer to how developers already reason about safety and concurrency without leaving the Rust toolchain. As a result, it has already been applied to verify the correctness of large-scale, critical systems, including cluster management controllers [52], virtual machine security modules [63], and microkernels [13].

Recent efforts to automate Verus proofs have been primarily focused on employing LLMs to generate proof annotations and iteratively repair the failed ones [2, 12, 57, 60, 62]. To facilitate more effective reasoning, existing approaches attempt to distill expert knowledge in the proof-writing and repairing process [32] via prompting [57], or synthesizing proofs for training [12] by exploiting Verus's efficient validation of the synthetic proofs. These approaches all formulate the task as a *static, end-to-end* prediction, where LLMs are tasked with generating the proof directly from the existing source code and specifications, with sparse guidance from the error messages of the verifier. This guidance is often insufficient and lacks elaborations to meaningfully guide the refinement of the proofs. Therefore, these approaches either rely on substantial manual effort in crafting prompts to include explicit search strategies [57], or spend expensive resources on training and collecting the training samples. For example, SAFE [12] has synthesized over 45K samples that also require heavy cleanup to be trainable, while AutoVerus [57]'s prompts are extensively optimized for specific Verus versions (see Section 5.1).

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The key limitation of the end-to-end proof generation is that it constrains the reasoning of LLMs only to the *static code*. However, static code has often been shown to lack a concrete characterization of the program execution, especially when LLMs rely on the reasoning about program behaviors [7, 22, 23, 30, 43, 45, 46]. Specifically, generating and repairing proofs often requires LLMs to understand and reason about the precise semantics of both the program (i.e., the fine-grained intended behavior of the source code) and the proofs (i.e., the code properties that the proofs intend to state). Unfortunately, existing approaches do not offer such information that characterizes the code behavior during proof generation, nor do they explicitly elicit the models' reasoning along this perspective.

Our approach. We introduce ExVerus, a framework for automated Verus proof generation that enables LLMs to reason about the proof via concrete *counterexamples*, which explicitly characterizes the program behavior that the target proofs aim to specify. When presented with a proof error, ExVerus explicitly guides the LLM to generate concrete and verifiable counterexamples and generalize the proofs from counterexamples to block them rather than guessing from coarse verifier feedback. In this paper, our core methodology focuses specifically on loop invariants. Loop invariant is particularly challenging, as it requires finding a suitable invariant that 1) is stronger than (and can thus imply) the target specification (i.e., postconditions), 2) correctly describes the semantics of the loop, and 3) preserves itself between iterations (i.e., inductiveness). Counterexamples[1] have been shown to be especially helpful for loop invariant reasoning, as it serve as concrete hints on the shape of reachable regions that the target invariants aim to characterize, and can provide more actionable guidance to LLMs for proof refinement at next iteration.

To this end, ExVerus features a fully automated counterexample generation and generalization pipeline, where both the generated counterexamples and the repaired proofs (generalized from the counterexamples) can still be verified for correctness. In particular, when a proof fails, ExVerus first prompts an LLM to generate a Z3 query that can synthesize a set of counterexamples that can be further validated via concrete executions. The LLM will then analyze these validated counterexamples and attempt to generalize them into new invariants so they can be blocked. Specifically, it uses a specialized prompting mutator to search for candidate invariants that block the validated counterexamples, turning an open-ended search into a constrained guided search task. Figure 1 shows the high-level workflow of ExVerus.

Results. We evaluate ExVerus on a broad range of Verus proof benchmarks. ExVerus outperforms the state-of-the-art LLM-based Verus prover by 45%. We also evaluate ExVerus's robustness on the obfuscated benchmarks. 90% of the proofs verified by ExVerus remain verifiable after deliberate code obfuscation, while the state-of-the-art baseline has only 33.3%-50% success rate. ExVerus's success rate of its generalized proofs blocking the counterexample reaches 70.18%, significantly higher than the 24.84% success rate of the baseline. We also analyze the token cost, where ExVerus's average token cost and time cost per task is only 25% of that of SOTA methods.

We make the following contributions:

- We introduce ExVerus, a novel automated invariant reasoning framework that synthesizes, validates, and generalizes counterexamples for efficient proof repair.
- We implement ExVerus with a fully automated pipeline that can repair proof errors guided by counterexamples at scale.
- We create ObfsBench, a challenging benchmark derived from VerusBench, with 266 new verification tasks, for robustness evaluation — a critical task that assesses the semantic understanding of LLMs of both the source code and proofs.

[1]While Verus relies on Z3 that should naturally return counterexamples, it lacks public support for producing them at source-level due to the nontrivial gap between those at solver- and source-level and thus the concern of confusing users

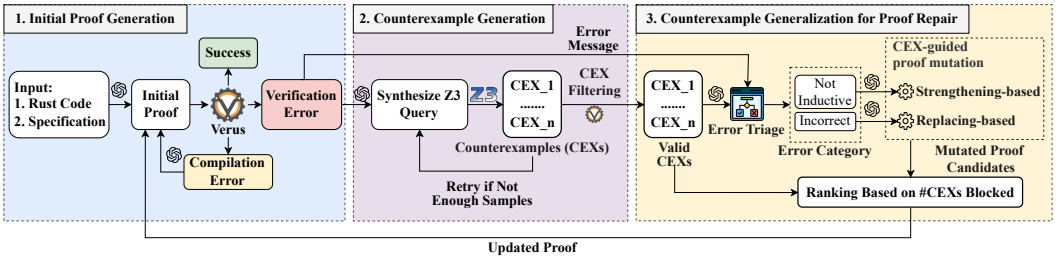


Fig. 1. Workflow of ExVerus

- Our extensive evaluation across a wide range of benchmarks, including the challenging robustness evaluation, demonstrates that ExVerus outperforms the state-of-the-art in terms of accuracy, robustness, and inference cost.

2 Overview

We begin by providing background on Rust verification using Verus, as well as the challenges in proof generation and repair in Verus. We then use a motivating example to demonstrate the advantage of ExVerus over the existing approaches.

2.1 Background: Automated Proof Generation and Repair in Verus

With improved reasoning capabilities, LLMs have increasingly automated proof generation and repair [26, 39, 57]. This capability has made formal methods more accessible to practitioners without specialized verification expertise, while accelerating the proof development process for experts. These tasks are particularly amenable to LLM-based automation as writing correct proofs is challenging, while verifying the generated proof is relatively simple.

In this work, we focus on Verus, a verification-ready programming language that builds upon Rust. Like other automated software verifiers, Verus requires users to provide suitable proof annotations to assist verification. These annotations include not only pre- and post-conditions that specify individual functions modularly, but also auxiliary definitions such as loop invariants and lemmas that convey insights about program runtime behavior. The proof annotations, together with the program, are processed by Verus to produce verification conditions that are discharged to off-the-shelf satisfiability modulo theory (SMT) solvers. While automated verification has made software verification more practical compared to manually constructing proofs in an interactive theorem prover, the black-box nature of verification condition checking provides limited feedback, which inhibits effective LLM-based reasoning. Because of this, despite the success demonstrated by recent work on leveraging LLMs for proof generation and iterative repair for Verus [12, 57], inductive properties such as loop invariants that are particularly difficult for verification engineers to formulate pose significant challenges for LLMs as well. We argue that the failure is largely due to LLMs' inherent limitation of discovering or even understanding the inductiveness required by the verifier, presenting valuable research opportunities.

On the other hand, the formal methods community has long recognized the central challenge of inductive invariants in proof automation, converging on a common recipe – learn from failed verification attempts and then generalize invariants until reaching inductiveness. Their key contributions usually address the question: how to generalize from failed attempts and rule out the entire class of failures? The early seminal work on hardware model checking [17] and software verification [27] adopts abstract interpretation and abductive reasoning. More recently, invariant generation

also benefits from data-driven machine learning [28, 29, 65]. While these symbolic techniques can greatly alleviate the burden on end-users of formal methods, domain-specific knowledge, like the language in which invariants can be specified, is still necessary to instantiate those learning frameworks, remaining an obstacle to embracing formal methods. The emerging capabilities of LLMs' few-shot generalization show potential in bridging this gap – not by replacing symbolic reasoning about inductiveness, but by providing the creative generalization that symbolic reasoning alone struggles to achieve.

2.2 Workflow of ExVerus

Figure 1 shows the high-level workflow of ExVerus. ExVerus takes as input a Rust program and its specifications, prompts the LLM to generate an initial proof (Module 1), iteratively fixes invariant errors via counterexample generation (Module 2) and counterexample generalization (Module 3), until the repaired proof passes Verus verification, or the max attempts are reached. For initial proof generation, we directly reuse the prompt of the initial proof generation phase of the state-of-the-art LLM-based proof generation tool AutoVerus [12]. Our framework focuses on the other two phases, repairing proofs via counterexample generation and generalization:

Counterexample generation with validation. Given a target invariant error, e.g., "invariant not satisfied at end of loop body", the goal of ExVerus is to generate concrete counterexamples to faithfully reveal the root cause of the invariant failure. To obtain a set of high-quality counterexamples, ExVerus iteratively synthesizes SMT queries via prompting the LLM and executing them, until the desired number of counterexamples are obtained or reaching the max attempts.

Note that there is no guarantee that the counterexamples generated by the LLM are truly counterexamples. Therefore, ExVerus relies on a validation module to replay the invariant failure with a concrete counterexample. Specifically, ExVerus isolates the loop containing the invariant error into a standalone function, denoted as `loop_func`, to simulate one loop transition. Then ExVerus (1) inserts the value assignments of the generated counterexample into `loop_func`, (2) translates the loop invariants into both assertions before and after the loop body (see Figure 2 for details). Our insight is that the isolated and extracted loop body can effectively simulate one loop transition, and the execution of `loop_func` can be "hijacked" and manipulated by injecting assignments of counterexample values, which is impossible for loop execution. Each generated counterexample will be checked by the validation module, and the counterexamples that fail to replay the invariant failure will be filtered.

Counterexample generalization with blocking validation. The validated counterexamples can serve as good guidance for invariant repair, since a repaired proof should block the counterexamples. ExVerus leverages the strong inductive reasoning capability to reason about and extract patterns of a handful of counterexamples via generalization, aiming to not only block the generated concrete counterexamples, but also find the root cause of the invariant failure and block all similar, unseen counterexamples. Given the counterexamples, ExVerus first queries the LLM-based Error Triage (in the yellow block) to reason whether the invariant is fundamentally incorrect (Incorrect) or it is correct but not inductive (Not Inductive). Then ExVerus performs CEX-guided proof mutation with the respective specialized mutator. For each mutant (Mutated Proof Candidate), ExVerus invokes the validation module to check whether each counterexample can be blocked by the mutant, e.g., when replaying the counterexample on the mutant, counterexamples can no longer be detected. The mutants are ranked based on the number of blocked counterexamples (Blocking Score Ranking), and the top-ranked mutant will be passed to the next iteration as it is considered to have the best generalizability.

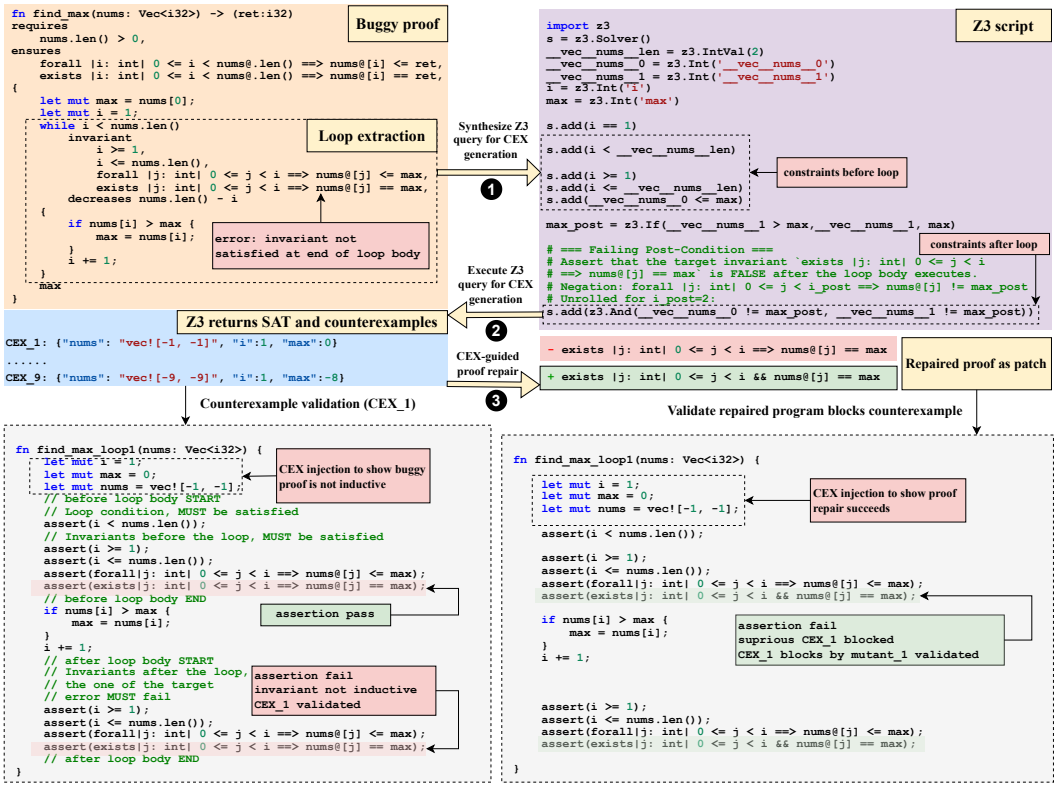


Fig. 2. Motivating example from VerusBench (Misc benchmark, task findmax) showing how ExVerus generates, validates and blocks counterexamples.

2.3 Motivating Example

Figure 2 shows a repair attempt in ExVerus's trajectory on the task Misc_findmax in VerusBench [57]. The goal of this task is to prove the return value of the function is the max value of the array, and the value exists in the array. The buggy proof (orange block) reports an error "invariant not satisfied at end of loop body". This error represents that the loop invariant `exists |j: int| 0 <= j < i ==> nums@[j] == max` is *not inductive*, i.e., it is correct in one iteration, but no longer holds after one transition (loop body execution).

ExVerus targets at the error, prompting the LLM to generate an SMT query, e.g., a `Z3Py` [20] program (purple block), aiming to reveal this invariant error by translating the invariants into SMT constraints. The Z3 query generated by the LLM first assumes some facts, such as `nums.len == 2`, to minimize the search space. Then it encodes both the loop condition and the invariants into Z3 constraints. It further simulates the loop transition via mimicking the execution of loop body statements. In the end, it encodes the negated error invariant into a Z3 constraint. The Z3 constraints are conjuncted and solved by the backend solver, returning "SAT" and a set of distinct counterexamples, shown in the green block.

ExVerus then synthesizes a validation program to validate whether the Z3-generated counterexample (say CEX_1) is a real counterexample or not, as shown in the blue block on the left. In the validation program, the loop that contains the buggy invariant is isolated and extracted as a function, e.g., `find_max_loop_1`. The invariants are translated into assertions right before and

after the loop body. The value assignments of a counterexample, e.g., CEX_1, are inserted at the beginning of the loop. A good counterexample for "invariant not satisfied at end of loop body" errors should pass the assertions before the loop body but fail the assertions of the buggy invariants after the loop body. The counterexamples that are validated are kept and used to evaluate the quality of candidate invariant patches. ExVerus then adopts a mutation-and-validation paradigm to generate invariant candidates. ExVerus first queries an LLM-based diagnosis, determines whether the buggy invariants need to be strengthened (correct but too weak) or removed/replaced (incorrect). In this example, the buggy invariants are classified as "too weak" and need strengthening; the verdict and rationale from the diagnosis are shown in Listing 1.

Then ExVerus invokes a specialized prompting-based strengthening mutator with a set of few-shot examples of different patterns of invariant strengthening. It summarizes the pattern of the counterexamples, generalizes it, and effectively generates multiple mutants aiming to block all counterexamples, e.g., mutant_1 patch on the right. ExVerus then invokes the validation program derived from the mutant to check whether the mutant can block the counterexamples. For the "invariant not satisfied at end of loop body" error, a mutant that blocks a counterexample might fail the assertions before the loop body, as the counterexamples are considered spurious and should be filtered out by stronger invariants. If a mutant successfully blocked the counterexamples, it will be prioritized and passed to the next iteration (or exit if it passes the verification).

Our key observations are 1) LLMs are good at translating invariants to SMT constraints to reproduce concrete and readable counterexamples, 2) concrete and validated counterexamples can accurately pinpoint the fundamental root cause of invariant failures, thus can serve as straightforward guidance and an appropriate interface to release the burden of reasoning over abstract invariants and effectively repair the invariants.

```
{
    "error_category": "too_weak",
    "rationale": "The failing invariant is `exists |j: int| 0 <= j < i ==> nums@[j] == max`. This invariant
      is a correct property of the intended algorithm; the variable `max` should indeed always hold a value
      that has been seen in the prefix of the array `nums[0..i)`. Thus, it is not a `wrong_fact`. The error
      type is `InvFailEnd`, which means the verifier cannot prove that this invariant is maintained across a
      loop iteration. The provided counterexamples are spurious, meaning they represent states that are not
      reachable in an actual execution. For instance, the counterexample `{'__vec__nums__len': 2,
      '__vec__nums__0': -1, '__vec__nums__1': -1, 'i': 1, 'max': 0}` shows `max` as `0`, a value that does
      not exist in the array `[-1, -1]`. The verifier generates these spurious states because the other
      invariants are not strong enough to fully constrain the relationship between `max` and the array
      elements. The verifier needs to be able to prove that the `exists` invariant holds, but the current set
      of invariants is insufficient to eliminate these impossible scenarios. This indicates the overall set
      of invariants is too weak."
}
```

Listing 1. Error category given by the LLM-based error triage.

2.4 Problem Formulation

We formally define the problem of generating and refining invariants. Given a property to verify (or post-condition) P, the goal of invariant generation is to find an invariant I, Each refinement step of ExVerus presented in Figure 1 concerns a single loop. Without loss of generality, we center the discussion of this section on single-loop verification and elaborate on the respective components of one refinement step that aims to produce better invariants than the last step.

We formulate the single-loop verification problem in terms of Hoare Logic [31]. Assume a set of state variables X that programs operate over. Program states s denotes assignments of values to the variables in X. Logical assertions P are formulas over X with $s \models P$ denoting that s satisfies P. Programs S denote the relation between input and output states. We write a Hoare triple $\{P\}$ `while` B `do` S $\{Q\}$ for the verification task of proving that if the while loop terminates when started in any state satisfying pre-condition P, then the resulting state satisfies post-condition Q. The correctness of the triple relies on an *inductive invariant* I such that:

(1) **Initialization** $P \Rightarrow I$: all initial states satisfy I;
(2) **Induction** $\{I \wedge B\}\ S\ \{I\}$: the loop execution, along with loop condition, preserves I;
(3) **Finalization** $I \wedge \neg B \Rightarrow Q$: I, along with the exit condition, permits only correct resulting states.

2.5 A Taxonomy of Incorrect Invariant

Loop verification fails if and only if any of these criteria is not met. The key to meet all these criteria is to identify a correct invariant. Now we classify the invariants that cause verification failures:

Incorrect Invariants. A candidate invariant $\hat{I}$ is considered incorrect if there exists a bounded unrolling of loops whose execution from an initial state satisfying P violates $\hat{I}$. That is, there exists $n \in \mathbb{N}$ such that the execution of $(\neg B; S)^n$ from a state $s_0 \models P$ results in a state $s \not\models \hat{I}$. When such n is zero, the invariant $\hat{I}$ fails to establish during initialization, i.e., formally $P \not\Rightarrow \hat{I}$. The failure suggests the existence of an initial state s_0 such that $s_0 \models P$ but $s_0 \not\models \hat{I}$. This state s_0 then becomes a counterexample to the initialization rule.

Not-Inductive Invariants. A candidate invariant $\hat{I}$ fails to be preserved during the loop execution when it is not strong enough to be inductive. The failure suggests the existence of a state s such that $s \models \hat{I} \wedge B$ but the execution of S from s results in a state $s' \not\models \hat{I}$. Such a state s is a counter-example to the inductiveness of $\hat{I}$.

Insufficient Invariants. A candidate invariant $\hat{I}$ fails to establish the post-condition Q when $\hat{I}$, along with the exit condition, is not sufficient to prove Q. The failure suggests the existence of a post-state s such that $s \models \hat{I} \wedge \neg B$ and $s \not\models Q$. Note that such a state s may not be reachable because $\hat{I}$ over-approximates reachable states of the program, but s does witness the insufficiency of $\hat{I}$. Although this error originates from missing invariants, the error message usually refers to "postcondition not satisfied" instead of invariant errors. This category is not the focus of this paper.

3 ExVerus Framework

We introduce ExVerus, a counterexample-guided proof invariant repair framework for Verus proofs. As shown in Figure 1 and Algorithm 1, the framework is primarily composed of three modules: initial proof generation, counterexample generation, and counterexample generalization.

3.1 Module 1: Initial Proof Generation

We initiate our pipeline with a preliminary proof generation step, as shown from line 2 to line 4 in Algorithm 1. This initial proof synthesis is conducted using a straightforward LLM generation strategy. We employ the same prompt as the one used in the *preliminary proof generation* phase of AutoVerus [12] for easier and fair comparison. If the initial generation does not pass the verification, it proceeds into the iterative repair process, i.e., Module 2 and 3, until the proof is repaired or the maximum attempts are reached (10 in our paper). If a proof in the iterations falls into compilation errors, e.g., syntax errors or type mismatch, the prompting-based compilation fixer will be invoked in the next iteration, to deliberately fix the compilation error, since Modules 2 and 3 are designed to fix verification errors. Otherwise, when encountering verification errors, such as "invariant not satisfied before loop" (denoted as InvFailFront) and "invariant not satisfied at end of loop body" (denoted as InvFailEnd), ExVerus will step to counterexample generation and generalization.

3.2 Module 2: Counterexample Generation with Validation

As discussed in Section 2.5, an invariant failure could be either **incorrect** or **not inductive**. For either invariant failure, there are some concrete examples that can reveal the failure. For instance, if the invariant is **incorrect**, there exists a reachable state s_0 such that $s_0 \models P$ but $s_0 \not\models \hat{I}$. If the invariant is correct but **not inductive**, there exists a state s such that $s \models \hat{I} \wedge B$ but the one-loop transition

Algorithm 1 ExVerus Pipeline

```
1: procedure ExVerus(unverified_proof, model, max_attempts, k)
2:     p ← Generation(unverified_proof, model)
3:     if Verify(p) = Pass then
4:         return {p, status=Pass, phase=init_gen}
5:     end if
6:     for t ← 1 to max_attempts do
7:         st ← Verify(p)
8:         if st = Pass then
9:             return {p, status=Pass, phase=cex_repair}
10:        end if
11:        if st = CompileError then
12:            p ← CompilationFixer(p, model)
13:            continue
14:        end if
15:        err ← Extract_And_Prioritize_Err(p)
16:        slice ← Symbolic_Transform(p, err)
17:        msg ← ConsoleMsg(p, err)
18:        C ← Z3Cex(p, err, model, k, msg, slice)
19:        if slice ≠ ∅ ∧ C ≠ ∅ then
20:            C′ ← Validation(p, slice, C)
21:        else
22:            C′ ← C
23:        end if
24:        p′ ← MutVal(p, err, C′, model)
25:        if p′ = ∅ then
26:            continue
27:        else
28:            p ← p′
29:        end if
30:    end for
31:    if Verify(p) = Pass then
32:        return {p, status=Pass, phase=cex_repair}
33:    else
34:        return {p, status=Fail, phase=cex_repair}
35:    end if
36: end procedure
```

Algorithm 2 Z3 Cex Generation

```
1: procedure Z3Cex(p, err, model, k, msg, slice)
2:     for i ← 1 to MaxZ3 do
3:         q ← MakeZ3Prompt(p, err, k, msg, slice)
4:         z3_script ← LLM_SynthesizeZ3(q, model)
5:         (status, raw) ← RunZ3(z3_script)
6:         if status ≠ sat then
7:             q ← Feedback(q, status)
8:             continue
9:         end if
10:        norm ← Normalize(raw)
11:        if ¬IdsValid(norm, p, slice) or |norm| < k/2 then
12:            q ← Feedback(q, gatefail)
13:            continue
14:        end if
15:        return MakeCEX(norm, err)
16:    end for
17:    return ∅
18: end procedure
```

Algorithm 3 Mutation-and-Validation-based Generalization

```
1: procedure MutVal(p, err, C, model)
2:     (err_cat, rationale) ← Classify(err, C, model)
3:     S ← Mutate(p, err, C, err_cat, rationale, model)
4:     if S = ∅ then
5:         return ∅
6:     end if
7:     S ← FilterCompilable(S)
8:     if AnyPass(S) then
9:         return FirstPass(S)
10:    end if
11:    return Ranked_By_Num_Cex_Blocked(S, C)
12: end procedure
```

Fig. 3. Pseudo-code of ExVerus. Algorithm 1 illustrates the overall pipeline of ExVerus, Algorithm 2 illustrates the core component of Counterexample Generation, and Algorithm 3 illustrates the core part of Counterexample Generalization for Proof Repair.

of S from s results in a state $s' \not\models \hat{I}$. Despite being aware of the existence of counterexamples, it is non-trivial to obtain the concrete examples. Verifiers (such as Verus compiler) compile high-level constraints all the way to the very low-level SMT queries, making the recovery from a solver's "SAT" to a concrete, human-readable source-level counterexample almost impossible. ExVerus bypasses this problem by leveraging the LLM's code translation ability to simulate a local verification process. The algorithm for ExVerus pipeline is shown in Algorithm 1.

Counterexample generation. Instead of performing traditional verification, i.e., trying to prove there are no counterexamples, ExVerus works on an easier task, i.e., synthesizing SMT queries to

search for concrete counterexamples that can reveal and replay the invariant failures. Specifically, ExVerus prompts the LLM with the buggy proof and the invariant error, instructing the LLM to translate the Verus invariants and execution code into an SMT query (`Z3Py` script in our implementation). To obtain a set of high-quality counterexamples, ExVerus iteratively synthesizes SMT queries and executes them, until enough distinct counterexamples are generated or reach the max attempts, for example, the purple block in Figure 2.

After obtaining a set of SMT-generated counterexamples, e.g., the blue block in Figure 2, ExVerus invokes the validation module to check whether they are truly counterexamples that reveal the invariant failure.

Counterexample validation. Due to the non-determinism of LLMs and potential threat of hallucination, the generated counterexamples are not guaranteed to be real counterexamples w.r.t. the invariant failures. ExVerus leverages a non-LLM verifier-based validation module to validate counterexamples. The validation module consists of three steps:

(1) Loop extraction: it isolates and extracts the loop body of the loop containing invariant into a standalone function, denoted as `loop_func`.
(2) Invariant translation: it then translates the loop invariants into assertions both before the loop body and after the loop body, mimicking one loop execution with invariant checking. We denote the assertions as loop-start assertions and loop-end assertions, respectively.
(3) Counterexample injection: it instruments `loop_func` and injects the value assignments of a counterexample at the beginning of the function, e.g., the left side white block in Figure 2.

Then the counterexample-injected `loop_func` (denoted as `loop_func_injected`) is checked by Verus verifier, and any assertion error would be captured. Specifically, ExVerus expects different symptoms for different invariant failures:

(1) InvFailFront. The invariant does not hold even before the loop, indicating the invariant is likely incorrect. For this error, ExVerus expects a (reachable) counterexample that fails the corresponding loop-start assertion.
(2) InvFailEnd. The invariant is satisfied at the loop start, but it is not satisfied at the end of the loop body, indicating the invariant is not inductive. It can be either incorrect or correct, but not strong enough. For this error, ExVerus expects a counterexample that passes the corresponding loop-start assertion, satisfies the loop condition, but fails the loop-end assertion.

ExVerus automatically captures any assertion errors and checks whether the corresponding symptoms are triggered. If so, the counterexample is considered validated. The validated counterexamples are passed to the counterexample generalization module.

3.3 Module 3: Counterexample Generalization

Given the set of validated counterexamples, ExVerus aims to reason over them, generalize from them, and summarize a pattern so that the root cause of counterexamples can be blocked. ExVerus first reasons about and classifies the error via an LLM-based error triage. Then it launches a mutation-and-validation searching process using a specialized prompting-based mutator, guided by the feedback from the validation module.

Counterexample-based error triage. As discussed in Section 2.5, different types of invariant failures require different repair strategies, or directions, even with the same verifier feedback. For instance, for an InvFailEnd failure, it is possible that the invariant is correct but not strong enough to preserve itself, and it is also possible that this invariant is incorrect and inconsistent with code semantics. To determine the categorization of the invariant error, ExVerus queries an

LLM-based error triage with the buggy proof, the counterexamples, and verifier feedback. The error triage is prompted to analyze whether the counterexample is reachable from a valid initial state or if it represents a spurious state, thereby classifying the invariant error. For example, a reachable counterexample indicates that the invariant is incorrect. The error triage returns the error categorization and provides the rationale. e.g., Listing 1 shows the output from the triage.

Customized mutation. Given the error categorization and rationale, ExVerus selects a customized prompting-based mutator. A "Not Inductive" invariant would need the strengthening-based mutator where invariant fixing examples are strengthening. The goal is to strengthen the invariant; the mutator proposes logically stronger predicates designed to make the invariant inductive while still holding true for all reachable states. An incorrect" invariant would need the replacing-based mutator, which includes different patterns of invariant fixing (replacing, removing, etc.) as few-shot examples. In this case, a simple strengthening is insufficient. This normally requires a more aggressive modification, such as retracting parts of the invariant or introducing new clauses to correctly capture the program semantics. For other errors (which are not the focus of our work), we provide the LLM with few-shot examples of common repair patterns, e.g., solving issues that may arise from the limitations of the Verus verifier. Note that the rationale from the error triage and the counterexamples are also included in the mutation prompt, to help the LLM better reason about and generalize from the given counterexamples.

Counterexample blocking-based mutant ranking. After obtaining a set of mutant candidates, we prioritize candidates based on verifier feedback. Prior work often relies on the number of verified sub-goals, a metric that can be susceptible to reward hacking and often provides a sparse reward signal [2, 56]. Instead, we propose a more fine-grained scoring function: the number of validated counterexamples that a candidate invariant successfully blocks. The intuition is that the more counterexamples a mutant can block, the more generalizable it is. "Block" here means that a counterexample that used to trigger an invariant failure can no longer trigger it after invariant repair [21]. Specifically, ExVerus invokes the **blocking** validation module, i.e., the validation module with same steps (1) and (3), but replaces the invariants in (2) with the new invariants in the mutant. The blocking validation module captures two signals of blocking:

(1) InvFailFront. The counterexample used to fail the loop-start assertion; now it does not contradict the updated loop-start assertion.
(2) InvFailEnd. The counterexample used to pass the loop start assertion but fail the loop-end assertion, now it either fails the updated loop-start assertion or passes both the updated loop-start and loop-end assertions.

ExVerus records the number of counterexamples blocked for each mutant, and the top candidate will either pass the verification or be advanced to the next iteration.

4 Setup

4.1 Implementation

ExVerus is implemented in Verus version `0.2025.07.12.0b6f3cb`. All experiments are conducted on a server running Ubuntu 22.04 LTS with AMD EPYC 9554 CPU that has 64-core/128-threads 3.10 GHz CPU and 1.1 TB RAM. Our implementation is based on Python (~13K LoC) and Rust (~2K LoC). For SMT solving, we use the Python Z3 API [4] (version 4.15.1.0). For Rust/Verus parsing, we develop tools (mainly for counterexample validation) using Rust Syn (version v2.0.106) and Verus Syn (version v0.0.0-2025-08-12-1837).

Models and parameters. To comprehensively evaluate our approach, we utilize several state-of-the-art Large Language Models (LLMs), including GPT-4o (`gpt-4o-2024-11-20`), O4-mini, Qwen3

Coder (`Qwen3-480B-A35B`), and DeepSeek-V3.1. For all LLM inference tasks, we set the temperature to 1.0 to encourage a broad exploration of the solution space (same with the AutoVerus paper for a fair comparison). The maximum number of repair iterations for our compilation and translation repair modules is set to 10. We set the max attempts for the iterative `Z3Py` script generation to 5. The number of LLM responses in mutants generation (counterexample generalization phase, Section 3.3) is set to 5.

Metrics. We evaluate the performance based on a set of standard metrics, including: success percentage, number of verified tasks, end-to-end execution time (in seconds), number of input and output tokens (in 1k tokens), and cost per task (in USD).

4.2 Baselines

We evaluate our approach against two baselines that closely align with our task.

AutoVerus. [57]: The first baseline is AutoVerus, a multi-agent system designed to simulate expert-driven proof refinement. We use its official public implementation, but note that it is originally evaluated on a now-deprecated version of Verus and thus suffers from performance degradation on the current toolchain. Despite this limitation, we include AutoVerus as it represents the state-of-the-art in prompt-engineered, agent-based systems for Verus proof generation.

Iterative Refinement. [48]: The second baseline is an iterative refinement method inspired by the approach of Shefer et al. [48]. This method operates in a direct loop, providing the LLM with the unverified code, the corresponding error message from the verifier, and a dedicated repair prompt. To ensure this method serves as a strong baseline, we meticulously engineer the repair prompt for maximum effectiveness.

We select these two baselines as they best match our task objectives and allow for reproducible comparison. Other recent works on automated Verus proof generation are not suitable for direct comparison due to fundamental differences in their objectives and experimental setups. For example, while SAFE [12] proposes a self-evolution framework using compute-intensive fine-tuning, the absence of publicly available models, code, and data prevents reproducibility. Other works tackle fundamentally different problems than ours. AlphaVerus [2] generates both code and proofs from specifications, while we focus on generating proofs for cases with existing code and specifications. Similarly, RAGVerus [62] employs Retrieval-Augmented Generation with repository-wide indexing, while our approach operates within a more constrained and intra-procedural setting.

4.3 Dataset

To facilitate a comprehensive evaluation, we construct a multi-source benchmark consisting of a diverse set of Verus proof tasks. Our primary objective is to build a high-quality benchmark suite where all problems are non-trivial, verifiable by the current Verus toolchain, and mutually exclusive across the different sources. To accomplish such a goal, we perform a systematic curation of the following related datasets:

VerusBench. [57]: This dataset contains a collection of proof tasks curated and Verus translated from different formal verification benchmarks such as MBPP-DFY-153, CloverBench, Diffy, and examples from the Verus documentation. Furthermore, we identify that the rapid evolution of the Versus framework introduces breaking changes that make many of the tasks unverifiable with the current tool-chain. To address this issue, we patch the source code to restore verifiability, primarily by resolving two prevalent categories of errors: enforcing termination via inserting decreases clauses to all loops, and reconciling stricter borrow-checking rules in array and vector manipulations. This process results in a total of 146 verified tasks from the original 150 tasks.

Table 1. Evaluation results across different methods, models, and benchmarks. Here EX, IR, and AV are abbreviations for ExVerus, Iterative, and AutoVerus, and all results are measured as the percentage of successes.

Benchmark	DeepSeek-V3.1			GPT-4o			Qwen3-Coder			O4-mini		
	EX	IR	AV	EX	IR	AV	EX	IR	AV	EX	IR	AV
VerusBench	**71.9**	60.3	24.7	**51.4**	43.2	39.0	**71.9**	69.2	51.4	**74.7**	69.2	32.2
DafnyBench	**88.1**	73.1	76.1	**88.1**	82.1	79.1	**95.5**	89.6	86.6	**95.5**	82.1	77.6
LCBench	**10.7**	**10.7**	**10.7**	**10.7**	**10.7**	7.1	**10.7**	7.1	**10.7**	**25.0**	14.3	10.7
HumanEval	**17.6**	11.8	14.7	**14.7**	8.8	**14.7**	**22.1**	19.1	16.2	**30.9**	20.6	20.6

Dafny2Verus. [2]: This dataset consists of part of the tasks from the DafnyBench dataset [38], which are translated to Verus via the AlphaVerus methodology. To address our quality concerns with this synthetically generated benchmark, we perform a systematic curation of the suite. First, we collect all programs and discard those that do not verify under the current Verus toolchain. Then, we select the subset of problems whose proofs contain invariants. At this stage, we find that many of the selected problems show signs of reward hacking via the inclusion of tautological preconditions and postconditions that make the programs trivial to verify—a known problem on synthetic data generation for formal verification problems [2, 56]. To overcome this, we follow a similar approach to that of the rule-based model proposed by AlphaVerus. Given a program x, we prompt an LLM to evaluate whether it contains specifications that lead to a trivial program, and to decide whether the program should be accepted or rejected. We repeat this process five times, each with a slight prompt variation, and take a majority vote. As a result, we obtain a total of 67 high-quality proof tasks.

Leetcode-Verus. [19]: This dataset is composed of challenging proof tasks derived from the LeetCode platform. The collection is curated by human experts who manually translate a selection of early-numbered LeetCode problems into Verus and author their complete, formal proofs. These complex and difficult tasks require extensive reasoning, with their complete proof blocks averaging approximately 200 lines. This process results in a curated set of 28 high-difficulty proof tasks.

HumanEval-Verus. [3]: This collection is part of an open-source effort to translate tasks from the HumanEval benchmark [11] to Verus. We curate the tasks following a similar approach to the one described in AlphaVerus [2], as well as applying our previously described patch-based curation approach. The process results in 68 tasks.

5 Experiment

Our evaluation aims to answer the following research questions:

- RQ1: How does ExVerus perform compared with state-of-the-art?
- RQ2: How robust is ExVerus against the semantics-preserving code transformations?
- RQ3: How do key components like counterexample error triage, customized mutation strategies, and counterexample generation contribute to the performance of ExVerus?
- RQ4: How does ExVerus perform on tasks of different difficulty, and how is ExVerus complementary to the state-of-the-art?
- RQ5: What is the cost of running ExVerus?

5.1 RQ1: Overall Performance

Our evaluation demonstrates that ExVerus consistently outperforms the established baselines in our comprehensive benchmark suite. As shown in Table 1, ExVerus achieves leading scores across different benchmark suits and base models.

Note that AutoVerus's performance on VerusBench performance in our experiment setting compared with the performance reported in their paper. Specifically, our reproduction of AutoVerus with the same described setting, i.e., GPT-4o and Verus version of 2024/8/13 on VerusBench. obtains a result of 75.33%, close to the reported numbers in the original paper. However, using the 2025/7/12 version, the performance degrades to 52.7%. After taking a closer look, we found that AutoVerus's prompts appear to be coupled to the older Verus version and do not work well with the newer ones, e.g., AutoVerus's prompts describing error fixing strategies are tailored for error messages specific to older Verus versions.

Furthermore, the generality of our approach is further confirmed on additional benchmarks like DafnyBench, where ExVerus consistently achieves the highest success rates, peaking at 95.5% with the O4-mini model. On more challenging coding benchmarks like HumanEval and LCBench, ExVerus remains competitive. Notably, ExVerus solves three and four more LCBench tasks and seven more HumanEval tasks using O4-mini. These results indicate that incorporating counterexamples into the repair process yields measurable improvements on this task.

5.2 RQ2: Robustness

In this section, we demonstrate the robustness of our approach against code obfuscation i.e., program transformations that intentionally obscure the logic and structure of source code without altering its semantic behavior. We find that our method is able to successfully generate proofs for a large number of tasks despite them being obfuscated with a wide variety of strategies.

ObfsBench dataset construction. Our obfuscated benchmark, **ObfsBench**, was systematically generated from a set of verifiable proofs in VerusBench by applying widely recognized obfuscation strategies. We designed specific prompts for LLM to process the code according to a variety of these strategies. To ensure the correctness of the generated output, we then employed an iterative repair process, guided by error messages. This process yielded a novel and challenging set of 266 verifiable, yet obfuscated, Verus proofs. We detail the specific obfuscation strategies used below.

- **Layout.** This strategy modifies the code's visual appearance and non-functional elements. E.g., *Identifier renaming* replaces descriptive variable and function names with generic or obscure identifiers to mask their intended purpose (e.g., changing `quotient` to `x`).
- **Data.** This category focuses on complicating the program's data storage and manipulation. Techniques include *Dead Variable Insertion*, which introduces variables and operations that have no effect on the final output (e.g., inserting `let mut junk = x * 3; junk = junk + 1;` where `junk` is unused). Furthermore, *Instruction Substitution* replaces simple operations with functionally equivalent, yet more complex, sequences of instructions (e.g., transforming `y = 191 - 7 * x;` into `let s = 7 * x; y = 191 - s;`).
- **Control flow.** This category alters the program's execution path, making the sequence of operations difficult to follow. Examples include *Dead Code Insertion*, which embeds blocks of code that are guaranteed never to be executed (e.g., `if (1 == 0) { y = 0; }`). Another technique is the use of *Opaque Predicates*, which are conditional expressions whose outcome is constant but is difficult for static analysis to determine (e.g., `if x * x >= 0 { ... }`) . Finally, *Control Flow Flattening* disrupts structured control flow by creating redundant branches with identical operations (e.g., a redundant `if-else` structure), making the execution trace much harder to reconstruct.

Table 2. Accuracy Comparison Across Different Verification Subsets on DeepSeek-V3.1. All results are success rates in percentages.

		ExVerus Verified		**AutoVerus Verified**		**All**	
Category	**Sub-strategy**	ExVerus	AutoVerus	ExVerus	AutoVerus	ExVerus	AutoVerus
Layout	Identifier Renaming	89.6	27.1	94.4	33.3	81.5	24.1
Data	Dead Variables	91.8	29.8	96.8	37.2	81.7	27.5
	Instruction Substitution	92.3	26.4	96.2	37.7	79.9	24.7
Control Flow	Dead Code Insertion	80.0	30.0	83.3	50.0	73.9	30.4
	Opaque Predicates	90.5	28.6	88.9	44.4	86.4	27.3
	Control Flow Flattening	91.7	39.6	96.0	40.0	86.5	36.5

Table 3. Ablation study on generalization strategies. Here IR, EX_{NO_MUT}, and EX refer to Iterative Refinement, $ExVerus_{NO\_MUT\_VAL}$, and ExVerus, respectively. The results are average success percentages.

Benchmark	DeepSeek-V3.1			GPT-4o			O4-mini			Qwen3-Coder		
	IR	EX_{NO_MUT}	EX	IR	EX_{NO_MUT}	EX	IR	EX_{NO_MUT}	EX	IR	EX_{NO_MUT}	EX
VerusBench	60.3	64.4	**71.9**	43.2	46.6	**51.4**	69.2	68.5	**74.7**	69.2	65.8	**71.9**
DafnyBench	73.1	**88.1**	**88.1**	82.1	**89.6**	88.1	89.6	85.1	**95.5**	82.1	92.5	**95.5**
LCBench	**10.7**	7.1	**10.7**	**10.7**	**10.7**	**10.7**	7.1	17.9	**25.0**	**14.3**	10.7	10.7
HumanEval	11.8	**17.6**	**17.6**	8.8	8.8	**14.7**	19.1	22.1	**30.9**	20.6	19.1	**22.1**
ObfsBench	61.3	65.4	**81.6**	28.6	35.3	**41.0**	69.9	72.9	**79.7**	71.4	71.8	**76.7**

Performance analysis on ObfsBench.

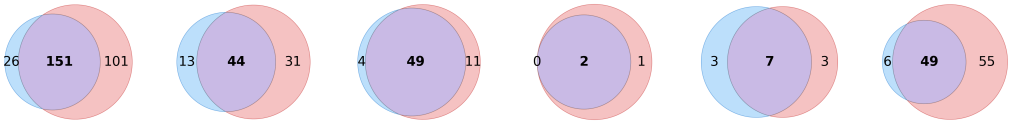
We separately analyze the sets of tasks successfully verified by ExVerus (ExVerus Verified) and AutoVerus (AutoVerus Verified) with DeepSeek-V3.1. Notably, the set of proofs solved by AutoVerus is a subset of those solved by ExVerus.

As shown in Table 2, ExVerus demonstrates a significant and consistent performance advantage over AutoVerus across the entire **ObfsBench** dataset. ExVerus maintains strong resilience against all tested obfuscation strategies with success rates consistently above 73%, whereas AutoVerus's success rates remain below 50%. This stark performance gap suggests that AutoVerus's heavy-weight, error-specialized prompting suffers from limited robustness when confronted with code that deviates from expected syntactic features. This highlights ExVerus's ability to reason about underlying program semantics, rendering it resilient to superficial syntactic disturbances introduced by obfuscation.

5.3 RQ3: Ablations

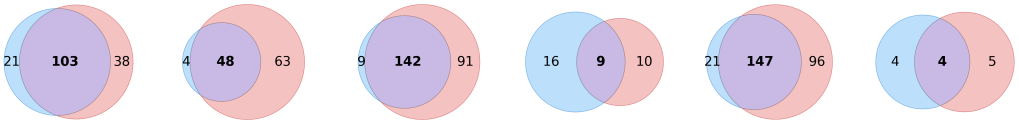
To investigate the effect of the counterexample-guided invariant mutation and counterexample-driven error triage in the generalization phase, we designed a baseline that alternatively instructs the LLM to directly fix the invariants based on the counterexamples, denoted as $ExVerus_{NO\_MUT\_VAL}$. We also include the numbers of Iterative Refinement as a reference.

As shown in Table 3, the counterexample-guided invariant mutation and validation component is crucial to ExVerus. Across nearly all scenarios, the full ExVerus framework significantly outperforms its ablated version, $ExVerus_{NO\_MUT\_VAL}$. On VerusBench, for example, the full system



(a) AllBench (b) VerusBench (c) DafnyBench (d) LCBench (e) HumanEval (f) ObfsBench

Fig. 4. Venn charts per benchmark. AutoVerus ( ); ExVerus ( ).



(a) Invariants <5 (b) Invariants >=5 (c) Assertions w/o (d) Assertions w/ (e) Proofs w/o (f) Proofs w/

Fig. 5. All-benchmarks Venn charts by difficulty. AutoVerus ( ); ExVerus ( ).

boosts the pass rate from 64.4% to 71.9% with the DeepSeek model. This performance gap is even more significant on the challenging ObfsBench, which is designed to test robustness. Here, our structured mutation and validation process increases the pass rate from 65.4% to a much higher 81.6% (16.1 percentage points).

Interestingly, even the simplified ExVerus$_{\text{NO_MUT_VAL}}$ variant still consistently outperforms the Iterative Refinement baseline, confirming that providing concrete counterexamples to the LLM is a fundamentally effective strategy. However, the superior performance of the full ExVerus system proves that our structured approach to generalizing from these counterexamples is the key factor in achieving state-of-the-art results.

5.4 RQ4: Sensitivity Analysis

Controlled single-step counterexample generation, verification, and generalization. To understand the effect of multi-counterexample guided repair and the verification module (that checks a counterexample is validated and blocked), we design a controlled experiment that only attempts to repair the buggy proof once, focusing on invariant errors. Specifically, we curate a dataset that consists of a number of buggy proofs that are almost correct, and can be fixed by modifying exactly one invariant (no assertions/proofs need to be fixed), denoted as InvariantInjectBench [2]. We select 142 tasks that require invariants from VerusBench, instruct the LLM to inject a high-quality and challenging one-line invariant bug using each of the three following prompts: *invariant strengthening*, *invariant weakening*, and *invariant removal*. Then we apply the following filters to get the high-quality dataset:

(1) The injected proof is buggy, leading to verification error(s) (instead of compilation error)
(2) The injected proof should contain at least one error of the expected error type, w.r.t. the prompt. For example, "invariant not satisfied at end of loop body" for *invariant weakening* and *invariant removal* injection, and "invariant not satisfied before loop" for *invariant strengthening* injection.
(3) The injected proof should only be one-invariant-different from the ground-truth proof.

[2]We also tried to extract intermediate proofs from AutoVerus's trajectories, but unfortunately we did not find many, since many AutoVerus proof files have more assertion statements than ground truth.

After applying the above filters, we obtain 187 (out of 426) slightly buggy proofs. To investigate the effect of the number of counterexamples, we run both ExVerus (using 10 counterexamples by default) and a variant of ExVerus that uses one counterexample, denoted as ExVerus$_{\text{ONE_CEX}}$, with one repair attempt. Out of 187 tasks, ExVerus proves 106 tasks while ExVerus$_{\text{ONE_CEX}}$proves 100, showing that more counterexamples are contributing positively to counterexample-guided invariant repair.

To understand how the verifier helps distinguish good and bad mutants based on counterexample-blocking, we count the number of blocked counterexamples per mutant and check whether the mutant passes the verification or not. The statistics show that among mutants produced by ExVerus that block no counterexamples, only 32 out of 83 (38.55%) can pass the verification. The mutants come from 21 tasks, while only nine are repaired (42.86%). In contrast, 158 out of 245 (64.49%) mutants that block at least one counterexample pass the verification. The mutants come from 51 tasks, while 41 of them are repaired (64.49%). Similarly, among the mutants produced by ExVerus$_{\text{ONE_CEX}}$, only 38 out of the 153 mutants (24.84%) that block no counterexamples pass the verification. The number of repaired tasks where the mutants come from is 12 out of 36 (33.33%). However, 172 out of 243 mutants (70.78%) that block the one counterexample pass the verification. The mutants come from 53 tasks, while 45 of them are repaired (84.91%). The results show that mutants that block at least one counterexample are significantly more likely to fix the invariant bug compared with ones that block no counterexamples, demonstrating the discriminative power of ExVerus's verification module.

Performance on tasks of different difficulty. To compare the performance of ExVerus and AutoVerus on tasks of different difficulty, we divide the tasks based on the number of invariants (low ≤ 5 and high > 5), assertions (w/o and w/), and proof functions/blocks (w/o and w/) based on the ground-truth verified proofs.

To ensure a fair comparison, we normalize the ground-truth proofs before difficulty classification by pruning redundant or semantically unnecessary invariants. To eliminate redundant invariants, we adopt a strategy inspired by the Houdini algorithm [27]. Specifically, we iteratively remove each invariant and check whether its absence causes any verification errors. An invariant is deemed redundant if its removal does not affect the verification outcome. For each proof case, we enumerate invariants such as loop invariants, intermediate assertions, proof-function attributes, and proof blocks. We then comment out one component at a time, rerun Verus, and retain only those components whose absence alters the verification result. A greedy pass accumulates all redundant components, and we finally record the simplified proof corresponding to the smallest invariant set that successfully passes the verifier.

As shown in table 4, both with GPT-4o, ExVerus's performance surpasses AutoVerus across both difficulty levels on all three difficulty dimensions on 3 out of 5 benchmarks: ObfsBench, VerusBench, and DafnyBench. This demonstrates ExVerus generalizes across proofs with diverse categories. Though ExVerus does not beat AutoVerus in some minor conditions, those marginal disadvantages do not undermine its overall superiority across the broader spectrum of tasks. On VerusBench, AutoVerus proves more successful on the challenging tasks that require the synthesis of assert statements (39.4% vs. 18.2%) and proof blocks (36.4% vs. 9.1%). This aligns with AutoVerus's design, which features a sophisticated, multi-agent debugging phase specifically engineered to generate and repair these complex proof annotations.

In fact, AutoVerus involves ten dedicated repair agents for different verification errors, e.g., `PreCondFail`, `InvFailFront`, `AssertFail`, etc.. The `AssertFail` agent will select a customized prompt based on the fine-grained error type, e.g., if the assertion error contains the keyword `.filter(`, it will use the prompt *"Please add 'reveal(Seq::filter);" at the beginning of the function*

Table 4. Success rate categorized by different bisections (number of invariants, w and wo assertions, w/ and wo proof functionsblocks) across different benchmarks. We use GPT-4o in this experiment.

Benchmark	Technique	Invariants		Assertions		Proofs	
		low	high	w/o	w/	w/o	w/
VerusBench	AutoVerus	**70.0**	17.4	38.9	**39.4**	39.3	**36.4**
	ExVerus	58.3	**46.5**	**61.1**	18.2	**54.8**	9.1
DafnyBench	AutoVerus	87.9	25.0	80.3	**0.0**	80.0	**100.0**
	ExVerus	**93.1**	**75.0**	**90.9**	**0.0**	**90.8**	**100.0**
HumanEval	AutoVerus	21.4	**3.8**	**29.4**	**9.8**	**36.4**	4.3
	ExVerus	**23.8**	0.0	**29.4**	**9.8**	27.3	**8.7**
LCBench	AutoVerus	**28.6**	0.0	**50.0**	0.0	33.3	**0.0**
	ExVerus	**28.6**	**4.8**	**50.0**	**4.2**	**50.0**	**0.0**
ObfsBench	AutoVerus	35.7	16.1	19.7	**25.0**	20.8	8.3
	ExVerus	**71.4**	**30.3**	**40.6**	**25.0**	**39.6**	**25.0**

where the failed assert line is located. This will help Verus understand the filter and hence prove anything related to the filter.". Such heuristics and customized prompting can help solve more tasks that require assertions/proofs, thus complementing ExVerus whose focus is on refining invariants instead of assertions/proofs.

Additionally, on HumanEval, ExVerus does not always outperform AutoVerus on more difficult tasks (>5 number of invariants) (0.0% vs. 3.8% on HumanEval) and tasks that do not require proof synthesis (27.3% vs. 36.4%). But it is noticeable that AutoVerus's success rate is very close to ExVerus, which means AutoVerus only gains very little advantage over ExVerus.

Distribution of repaired proofs. We present Venn charts on the number of fixed proofs to show how overlapped or complementary ExVerus and AutoVerus are in terms of solving the tasks, shown in Figure 4 and Figure 5. While ExVerus is broadly more capable, the two methods are also highly complementary, with each tool demonstrating unique strengths. Overall, ExVerus uniquely solves 101 tasks that AutoVerus cannot, while AutoVerus uniquely solves 26 tasks.

Figure 5 reveals the source of these distinct capabilities. ExVerus's unique strength is concentrated in more complex problems: It uniquely solves 63 tasks whose solutions require a high number of invariants, compared to only four for AutoVerus. In contrast, AutoVerus's unique contribution is most apparent on tasks whose solutions require the synthesis of assertions, where it uniquely solves 15 problems compared to ExVerus's 10. But on tasks that require no assertions, it only uniquely solved 10 tasks, compared with 91 tasks solved uniquely by ExVerus. This aligns with its design of a heuristics-based customized `AssertFail` repair agent, as discussed earlier. These findings again confirm ExVerus's advantage on tasks where invariants are the bottleneck, while it is complimentary to AutoVerus whose heuristics and heavy-weight prompting are good at repairing assertion errors.

5.5 RQ5: Cost Analysis

In addition to superior verification accuracy, ExVerus exhibits a substantial advantage in computational efficiency. As detailed in Table 5, the average token cost of ExVerus is 0.04, making it four times more economical than AutoVerus, which has a cost of 0.17. This efficiency gain is also

Table 5. Token consumption (input/output in 1k tokens), cost ($), and execution time (s) for ExVerus (EX) and AutoVerus (AV), measured per task across DeepSeek-V3.1 and GPT-4o.

		Hard-Inv			Easy-Inv			Total		
Model	Method	#Tokens	Cost	Time	#Tokens	Cost	Time	#Tokens	Cost	Time
DeepSeek-V3.1	EX	111.2/14.6	0.05	702.2	68.7/15.0	0.04	746.5	93.8/14.8	0.04	720.3
	AV	431.1/62.0	0.18	3463.0	411.4/35.3	0.15	2352.3	422.7/50.6	0.17	2989.1
GPT-4o	EX	101.4/25.6	0.51	299.5	57.3/14.5	0.29	179.8	83.1/21.0	0.42	250.0
	AV	118.2/62.8	0.92	305.5	57.5/23.9	0.38	137.6	92.3/46.2	0.69	233.9

reflected in the end-to-end execution time. ExVerus required an average of 720.34 seconds per task, whereas AutoVerus took 2989.07 seconds to complete the same tasks.

The cost-effectiveness of our framework is a direct consequence of its methodology. By leveraging concrete counterexamples, ExVerus conducts a focused search for a correct invariant, minimizing exploration cost. This approach avoids the brute-force "generate-and-validate" loops that represent the philosophy of the iterative refinement baseline, which often explore a vast and unproductive portion of the potential solution space. The targeted nature of our repair process ensures that each LLM call is directed by specific, actionable feedback derived from a validated counterexample.

Ultimately, ExVerus achieves a dual improvement in both verification performance and economic viability. This combination of higher accuracy and lower computational overhead makes it a more scalable and practical solution for automated proof generation. The efficiency of our approach suggests its potential for broader application in real-world software development workflows, where both correctness and resource constraints are critical considerations.

6 Case Study

In this section, we use two interesting cases where the buggy invariant is incorrect to illustrate how ExVerus repairs an invariant step by step and show the broad applicability of ExVerus.

Invariant weakening via state pruning. Figure 6 shows an almost-correct proof produced in solving task Diffy_brs1 in VerusBench [57]. This goal of the proof to prove the sum of array `sum[0] <= N`. The Verus verifier provides feedback "error: invariant not satisfied before loop", pointing to the buggy invariant `sum[0] <= i`. This failure occurs because the LLM overlooked an edge case, i.e., in the first iteration, `sum` hasn't been initialized yet, so we cannot make any claims about its value. In every iterations after that, `sum[0] <= i` holds. The LLM realized that something like `sum[0] <= i` is necessary to prove the post-condition. It also realized that `sum[0] <= i` looks probably correct. Although it appeared to be easy to solve, the state-of-the-art LLM-based proof generation tool, AutoVerus, failed to prove this task after 15 preliminary proof generation attempts (Phase 1), 4 generic proof refinement attempts (Phase 2), and 21 error-driven proof debugging attempts (Phase 3). After inspecting the trajectory of AutoVerus, we observed that AutoVerus spent 16 attempts (Phase 3) to fix "error: invariant not satisfied before loop", but none of them worked. This "off-by-one" incorrect invariant error (i.e. fails only in the first iteration: i = 0) and the struggling repairing process boil down to the fundamental limitation of the LLM's weakness in abductive reasoning [15, 24, 49] – what necessary invariants for the verification task should look like. LLMs alone, while excel in generating invariant candidates, often struggles to reach a perfect invariant that not only succinctly characterize safe program behaviors and but also entails the correctness, i.e., `sum[0] <= N` in this case.

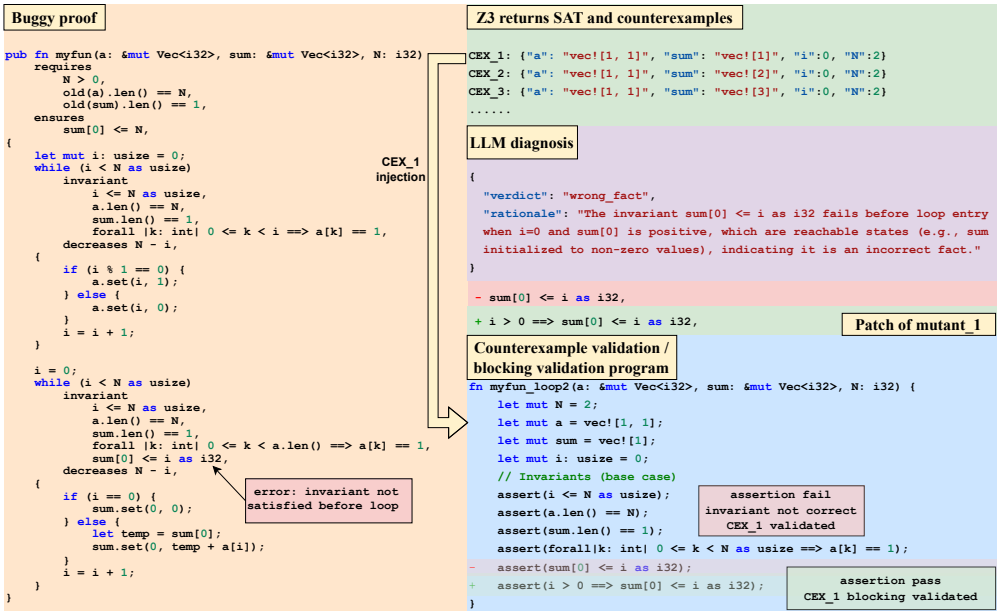


Fig. 6. Repairing a wrong invariant that involves an invalid state by pinpointing and pruning it

To mitigate this gap, ExVerus compensates the limited abductive reasoning capability of LLMs with the principled inductive reasoning capability of formal methods. Given a wrong/imperfect invariant, ExVerus attempts to synthesize a set of counterexamples that reveal the reason of the verification failure (dark green block). Then ExVerus starts to search for a set of candidate invariants that can block the counterexamples. Specifically, ExVerus inductively reasons about the pattern of the given counterexamples to the invariant by generalizing from the counterexamples. For instance, the light green block shows given the counterexamples to the buggy invariant, the LLM summarizes that "The invariant `sum[0] <= i` fails before loop entry when `i=0` and `sum[0]` is positive, which are reachable states (e.g., sum initialized to non-zero values), indicating it is an incorrect fact." ExVerus then selects the replacing-based mutator to search for the fixed invariant. For each of the mutants produced by the mutator, ExVerus invokes the validation module to check whether the cexs are successfully blocked. The mutant that blocks the most counterexamples will be prioritized and passed to the next iteration, e.g., mutant-1 that adds a precondition `i > 0` successfully blocked all 10 counterexamples. The mutant-1 successfully passed Verus verification, thus solving this task.

Wrong invariant detection and removal. Figure 7 shows another almost-correct proof in the trajectory of solving CloverBench_two_sum_3 in ObfsBench. The goal of this proof is to prove that the result tuple `(a, b)` is the first tuple that satisfies `a + b == target`. AutoVerus, failed to prove this task after 15 preliminary proof generation attempts (Phase 1), 1 generic proof refinement attempt (Phase 2), and 24 error-driven proof debugging attempts (Phase 3). AutoVerus spent 14 attempts (Phase 3) to fix assertion failures, but none of them worked.

The buggy invariant `forall|ii: int, jj: int|((0 <= ii && ii < r.0 && ii < jj && jj < nums.len()) || (ii == r.0 && ii < jj && jj < r.1)) ==> nums[ii] + nums[jj] != target` reports "error: invariant not satisfied before loop", indicating the invariant is incorrect.

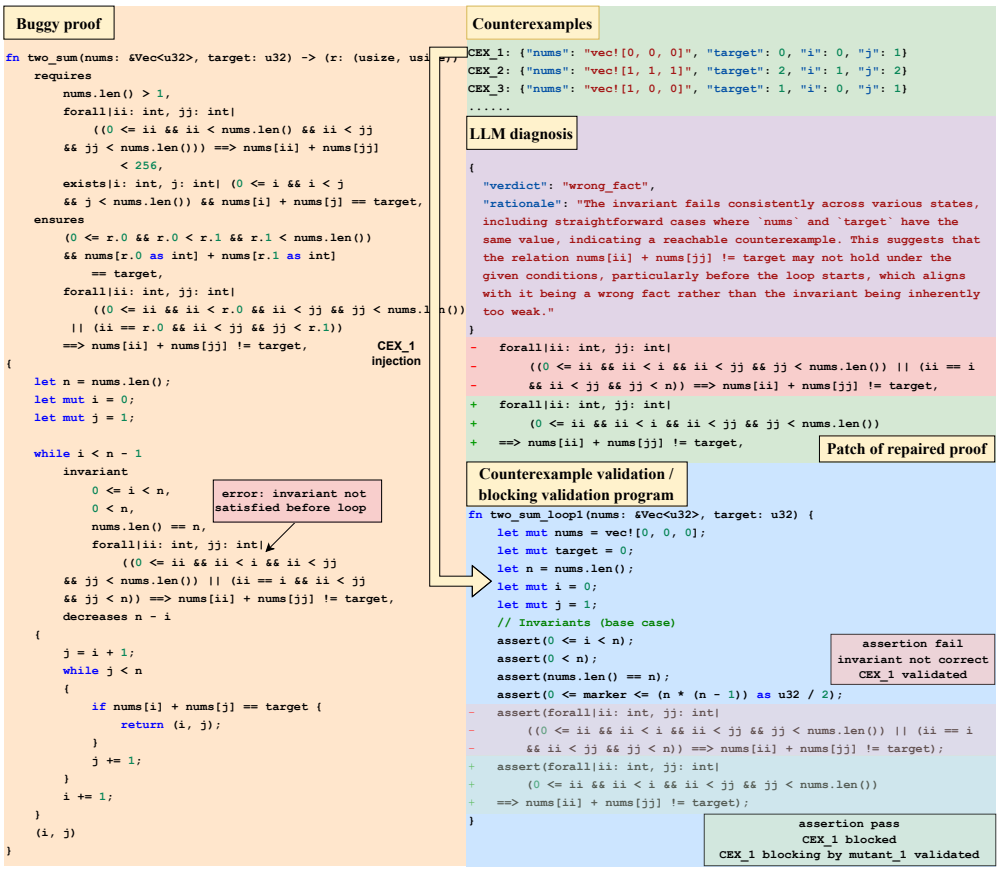


Fig. 7. Identifying and removing a wrong invariant guided by counterexamples

Similar to the state pruning case, the SMT query returns ten counterexamples (shown in the dark green block). All counterexamples trigger the red assertion (translated from the buggy invariant) in the blue block and are validated. The LLM-based diagnosis then reasons about the validated counterexamples, summarizing that the counterexamples are reachable. Therefore, the clause `nums[ii] + nums[jj] != target` may not hold under the given conditions. The LLM diagnosis labels it as an incorrect invariant. Consequently, it invokes the "replacing-based" prompting-based mutator and produce a set of mutants. The mutant_1 can successfully block all counterexamples, i.e., it passes the green assertion, and is prioritized for later operations. It turns out that mutant_1 can pass Verus verification, thus solving the task.

ExVerus avoids counting on LLM's abductive reasoning capability to fix invariants merely based on the limited feedback from the verifier. Instead, it alleviates the reasoning burden by synthesizing (and validating) a set of counterexamples, and using them as the guidance to search for candidate invariants. The invariants that can block more counterexamples, i.e., potentially characterizing a generalized pattern of counterexamples, are rewarded and prioritized. Through iteratively detecting and blocking the counterexamples, ExVerus effectively paves it way to gradually approach the correct invariants where no counterexamples exist.

7 Discussion and Threats to Validity

LLM-based SMT query generation and proof mutation. We employ LLMs to generate Z3 queries for counterexample generation and to mutate proofs to generalize them from counterexamples. While we always employ symbolic checkers to ensure their correctness, the capabilities of LLMs can be fundamentally limited in these tasks (i.e., Z3 query generation and counterexample-guided invariant mutation), potentially leading to significant token costs without termination guarantees. Future work could tackle this limitation by training LLMs to learn Z3 query translation or by adopting the same spirit of more advanced counterexample-guided invariant learning algorithms [17, 18].

Proof beyond loop invariants. Verification of real-world software often involves invariants that go beyond loops. Individual classes usually have implicit well-formedness conditions that govern the underlying data being represented and are critical for verification tasks [8]. While our counterexample reasoning primarily focuses on the loop invariants, our tool is a full-fledged proof repair framework that integrates prompts to handle errors beyond loop invariants. That being said, we currently focus on benchmarks where loops are prevalent. We aim to study ExVerus on repository-, system-level verification [6, 14, 52, 62, 64] in our future work.

Initial proof generation. This preliminary stage is deliberately kept straightforward, yet it plays a critical role in the entire pipeline: if the initial proof generation is poor, the subsequent repair stages may be completely derailed. To ensure a reliable starting point, we directly adopt the initial-generation prompt from AutoVerus, which provides a stable and well-tested baseline. Since our primary contribution lies in the counterexample-driven repair and generalization framework which is orthogonal to prior work that focuses on improving intrinsic proof generation through training or prompt optimization, we intentionally avoid engineering the initial generation prompt to achieve the best possible success rate. This allows us to better highlight the effectiveness of our downstream components. While we only use a single-round generation with an existing prompt, numerous techniques in prompt engineering [61] or finetuning [12] could further improve the performance of this stage. These enhancements are orthogonal to our focus and can be readily integrated into our framework in future work.

8 Related Work

Inductive invariant synthesis. The discovery of inductive invariants has been the central challenge in automated verification. The seminal work on counter-example guided abstraction refinement [17] effectively derive strong enough facts about the system in order to prove its correctness. An incremental and property-directed approach to invariant strengthening is later proposed by Bradley [5]. Since then, iterative invariant strengthening using counter-examples have seens success in different domains, from verification of numeric programs [21, 28], to distributed systems [44, 59] On the other hand, machine learning enables invariant synthesis that generalizes from program structures and execution traces. Code2Inv [50] reduces numerical loop invariant synthesis to deep reinforcement learning by mapping programs to graph neural networks, while CLN2INV [47] introduces continuous logic networks to learn precise SMT formulas directly from execution traces. Recently, large language models have emerged as a new paradigm for invariant synthesis [33, 45] that even benefit from natural language specifications [54]. Our approach synergizes the inductive reasoning from both traditional symbolic approaches and the generalization capability from LLMs to synthesize loop invariants that are strong enough for verification to succeed.

LLMs for theorem proving. Automated theorem proving and proof synthesis have been active areas of development in recent years. In particular, as LLMs have improved their reasoning and

coding abilities, there has been a growing interest in combining them with interactive theorem provers such as Lean and Coq [34, 37, 58]. Notably, recent work [1, 10] has demonstrated that these methods can exhibit exceptional performance on challenging math competitions such as the International Math Olympiad (IMO). It is worth noting that these methods follow similar principles to those presented in this work. For example, Seed-Prover [10] generates Lean proofs of a given formal statement and iteratively refines it based on compiler feedback. Complementary to these general-purpose approaches, domain-specific methods have demonstrated that incorporating domain knowledge can yield substantial improvements. Works like AlphaGeometry [16, 53], Newclid [1], and Seed-Geometry [10] combine LLMs with symbolic deduction for geometry theorem proving, achieving near-human performance on IMO geometry problems. Our work applies similar principles: combining symbolic reasoning with LLMs to the domain of software verification, where we expand the symbolic reasoning of LLMs at both abstract (traditional proofs) and concrete (counterexamples) levels.

LLMs for software verification. The application of large language models to software verification has emerged as a promising research direction, with various approaches targeting different aspects of the verification pipeline. Several works focus on synthesizing specifications and invariants: Wu et al. [55] propose LaM4Inv, which combines LLMs with bounded model checking through a "query-filter-reassemble" strategy where LLMs generate candidate invariants that are filtered by bounded model checking and reassembled into new candidates for SMT solvers. Similarly, Yao et al. [60] combine GPT-4 with static analysis to generate invariants and postconditions for Verus programs, though their approach lacks systematic counterexample-guided refinement. nl2postcond [25] takes a different perspective by investigating whether LLMs can transform informal natural language specifications into formal method postconditions, demonstrating that generated postconditions can catch real-world bugs. Moving beyond invariant synthesis, Chakraborty et al. [9] curates a dataset of F* programs for neural synthesis in SMT-assisted proof-oriented programming, targeting type-directed program and proof synthesis. In the interactive theorem proving setting, Baldur First et al. [26] introduces whole-proof generation and repair in Isabelle/HOL, generating complete proofs at once and employing a repair model that leverages error messages to fix failed proofs. Our work differs from these approaches by introducing counterexample-guided repair specifically for Verus, where concrete execution traces guide the refinement of inductive invariants, enabling more targeted and effective proof synthesis.

9 Conclusion

We presented ExVerus, an automated LLM-based Verus proof repair framework guided by counterexamples. Unlike prior LLM-based systems that rely on static code and coarse verifier feedback, ExVerus actively synthesizes, validates, and generalizes counterexamples to guide proof refinement. By grounding LLM reasoning in concrete program behaviors, ExVerus transforms open-ended proof search into more structured and guided process. Extensive experiments across multiple Verus benchmarks, including our newly introduced ObfsBench for robustness evaluation, demonstrate that ExVerus substantially outperforms the state-of-the-art automated Verus provers, achieving higher success rates, more robust prediction, and lower token costs. The framework also demonstrates strong resilience to semantic-preserving code transformations, underscoring the effectiveness of using counterexamples to reason about program and proof semantics.

Data-Availability Statement

An anonymized artifact accompanying this paper is available at https://anonymous.4open.science/r/verusinv-34CD/. The repository contains all datasets and the complete implementation of the

ExVerus pipeline used in our experiments, including scripts for counterexample generation, validation, generalization, and evaluation. The datasets cover VerusBench, DafnyBench, LCBench, HumanEval, and our robustness benchmark ObfsBench.

This artifact will be submitted for *Artifact Evaluation*. While the pipeline code and datasets are fixed, reproducing end-to-end results requires running large language model (LLM) inference. Consequently, re-runs may incur token costs and exhibit small variations in quantitative metrics (e.g., success rate, token usage) due to the stochasticity of LLM generation and provider-side updates. We provide scripts and configuration files to replicate our evaluation protocol. However, exact numerical values may not match the paper's numbers bit for bit. Qualitative findings and comparative trends are expected to remain consistent.

References

[1] Tudor Achim, Alex Best, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leister, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. 2025. Aristotle: IMO-level Automated Theorem Proving. arXiv:2510.01346 [cs.AI] https://arxiv.org/abs/2510.01346

[2] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2025. AlphaVerus: Bootstrapping Formally Verified Code Generation Through Self-Improving Translation and Treefinement. *ICML 2025* (2025).

[3] Alex Bai, Jay Bosamiya, Edwin Fernando, Md Rakib Hossain, Jay Lorch, Shan Lu, Natalie Neamtu, Bryan Parno, Amar Shah, and Elanor Tang. 2025. HumanEval-Verus: Hand-Written Examples of Verified Verus Code Derived From HumanEval. https://github.com/secure-foundations/human-eval-verus. Benchmark and contributors: Alex Bai, Jay Bosamiya, Edwin Fernando, Md Rakib Hossain, Jay Lorch, Shan Lu, Natalie Neamtu, Bryan Parno, Amar Shah, Elanor Tang.

[4] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M Wintersteiger. 2018. Programming Z3. In *International Summer School on Engineering Trustworthy Software Systems*. Springer, 148–201.

[5] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Springer, Berlin, Heidelberg, 70–87. doi:10.1007/978-3-642-18275-4_7

[6] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. 2023. Beyond isolation: OS verification as a foundation for correct applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 158–165.

[7] Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. 2025. CWM: An Open-Weights LLM for Research on Code Generation with World Models. *arXiv preprint arXiv:2510.02387* (2025).

[8] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K Rustan M Leino, Mikael Mayer, Sean McLaughlin, et al. 2025. Formally verified cloud-scale authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 703–703.

[9] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. 2024. Towards neural synthesis for smt-assisted proof-oriented programming. *arXiv preprint arXiv:2405.01787* (2024).

[10] Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. 2025. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving. arXiv:2507.23726 [cs.AI] https://arxiv.org/abs/2507.23726

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech

Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[12] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng CHENG, Fan Yang, Shuvendu Lahiri, Tao Xie, and Lidong Zhou. 2025. Automated Proof Generation for Rust Code Via Self-Evolution. *Computing Research Repository* (2025).

[13] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. 2023. Atmosphere: Towards Practical Verified Kernels in Rust. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification* (Koblenz, Germany) *(KISV '23)*. Association for Computing Machinery, New York, NY, USA, 9–17. doi:10.1145/3625275.3625401

[14] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. 2023. Atmosphere: Towards practical verified kernels in Rust. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*. 9–17.

[15] Kewei Cheng, Jingfeng Yang, Haoming Jiang, Zhengyang Wang, Binxuan Huang, Ruirui Li, Shiyang Li, Zheng Li, Yifan Gao, Xian Li, et al. 2024. Inductive or deductive? rethinking the fundamental reasoning abilities of llms. *arXiv preprint arXiv:2408.00114* (2024).

[16] Yuri Chervonyi, Trieu H. Trinh, Miroslav Olšák, Xiaomeng Yang, Hoang Nguyen, Marcelo Menegali, Junehyuk Jung, Vikas Verma, Quoc V. Le, and Thang Luong. 2025. Gold-medalist Performance in Solving Olympiad Geometry with AlphaGeometry2. arXiv:2502.03544 [cs.AI] https://arxiv.org/abs/2502.03544

[17] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

[18] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

[19] Weituo Dai. 2025. verus-study-cases-leetcode. https://github.com/WeituoDAI/verus-study-cases-leetcode. Accessed: 2025-10-02.

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

[21] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 443–456. doi:10.1145/2509136.2509511

[22] Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems* 37 (2024), 60275–60308.

[23] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[24] John Dougrez-Lewis, Mahmud Elahi Akhter, Federico Ruggeri, Sebastian Löbbers, Yulan He, and Maria Liakata. 2024. Assessing the Reasoning Capabilities of LLMs in the context of Evidence-based Claim Verification. *arXiv preprint arXiv:2402.10735* (2024).

[25] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912.

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

[27] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*. Springer, 500–517.

[28] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.

[29] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

[30] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065* (2024).

[31] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[32] Rijul Jain, Shraddha Barke, Gabriel Ebner, Md Rakib Hossain Misu, Shan Lu, and Sarah Fakhoury. 2025. What's in a Proof? Analyzing Expert Proof-Writing Processes in F* and Verus. *arXiv preprint arXiv:2508.02733* (2025).

[33] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding inductive loop invariants using large language models.

CoRR abs/2311.07948 (2023). *arXiv preprint arXiv:2311.07948* (2023).

[34] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2382–2385. doi:10.1145/3691620.3695357

[35] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. 2024. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 438–454.

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

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

[38] Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2025. DafnyBench: A Benchmark for Formal Software Verification. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=yBgTVWccIx

[39] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1509–1520.

[40] Microsoft. 2024. Verus Copilot for VS Code. GitHub repository. https://github.com/microsoft/verus-copilot-vscode Accessed: 2025-09-23.

[41] Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. 2024. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 812–835.

[42] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 134 (April 2025), 27 pages. doi:10.1145/3720499

[43] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662* (2024).

[44] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 614–630. doi:10.1145/2908080.2908118

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

[46] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[47] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2019. CLN2INV: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542* (2019).

[48] Aleksandr Shefer, Igor Engel, Stanislav Alekseev, Daniil Berezun, Ekaterina Verbitskaia, and Anton Podkopaev. 2025. Can LLMs Enable Verification in Mainstream Programming? *alphaxiv* (2025).

[49] Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941* (2025).

[50] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems* 31 (2018).

[51] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Lean copilot: Large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534* (2024).

[52] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: verifying liveness of cluster management controllers. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 35, 18 pages.

[53] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature* 625, 7941 (2024), 476–482. doi:10.1038/s41586-023-06747-5

[54] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2217–2235.

[55] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417.

[56] Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. 2025. Local Success Does Not Compose: Benchmarking Large Language Models for Compositional Formal Verification. *arXiv preprint arXiv:2509.23061* (2025).

[57] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2024. AutoVerus: Automated Proof Generation for Rust Code. *Computing Research Repository* abs/2409.13082 (2024).

[58] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. arXiv:2306.15626 [cs.LG] https://arxiv.org/abs/2306.15626

[59] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 485–501.

[60] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging large language models for automated proof synthesis in rust. *arXiv preprint arXiv:2311.03739* (2023).

[61] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

[62] Sicheng Zhong, Jiading Zhu, Yifang Tian, and Xujie Si. 2025. RAG-Verus: Repository-Level Program Verification with LLMs Using Retrieval Augmented Generation. *Computing Research Repository* abs/2502.05344 (2025).

[63] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VERISMO: a verified security module for confidential VMs. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 32, 16 pages.

[64] Ziqiao Zhou, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, Weidong Cui, et al. 2024. {VeriSMo}: A verified security module for confidential {VMs}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 599–614.

[65] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *ACM SIGPLAN Notices* 53, 4 (2018), 707–721.