

# AUTOVERUS: Automated Proof Generation for Rust Code

**CHENYUAN YANG**, University of Illinois at Urbana-Champaign, USA

**XUHENG LI**, Columbia University, USA

**MD RAKIB HOSSAIN MISU**, University of California Irvine, USA

**JIANAN YAO**, University of Toronto, Canada

**WEIDONG CUI**, Microsoft Research, USA

**YEYUN GONG**, Microsoft Research Asia, China

**CHRIS HAWBLITZEL**, Microsoft Research, USA

**SHUVENDU LAHIRI**, Microsoft Research, USA

**JACOB R. LORCH**, Microsoft Research, USA

**SHUAI LU**, Microsoft Research Asia, China

**FAN YANG**, Microsoft Research Asia, China

**ZIQIAO ZHOU**, Microsoft Research, USA

**SHAN LU**, Microsoft Research, USA and University of Chicago, USA

Generative AI has shown its value for many software engineering tasks. Still in its infancy, large language model (LLM)-based proof generation lags behind LLM-based code generation. In this paper, we present AUTOVERUS. AUTOVERUS uses LLMs to automatically generate correctness proof for Rust code. AUTOVERUS is designed to match the unique features of Verus, a verification tool that can prove the correctness of Rust code using proofs and specifications also written in Rust. AUTOVERUS consists of a network of agents that are crafted and orchestrated to mimic human experts' three phases of proof construction: preliminary proof generation, proof refinement guided by generic tips, and proof debugging guided by verification errors. To thoroughly evaluate AUTOVERUS and help foster future research in this direction, we have built a benchmark suite of 150 non-trivial proof tasks, based on existing code-generation benchmarks and verification benchmarks. Our evaluation shows that AUTOVERUS can automatically generate correct proof for more than 90% of them, with more than half of them tackled in less than 30 seconds or 3 LLM calls.

CCS Concepts: • Software and its engineering → Formal software verification.

Additional Key Words and Phrases: Program Verification, Program Synthesis, Verus, Large Language Models

## ACM Reference Format:

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. 2025. AUTOVERUS: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 396 (October 2025), 29 pages. <https://doi.org/10.1145/3763174>

---

Authors' Contact Information: **Chenyuan Yang**, cy54@illinois.edu, University of Illinois at Urbana-Champaign, USA; **Xuheng Li**, xuheng@cs.columbia.edu, Columbia University, USA; **Md Rakib Hossain Misu**, mdrh@uci.edu, University of California Irvine, USA; **Jianan Yao**, jy3022@columbia.edu, University of Toronto, Canada; **Weidong Cui**, weidong.cui@microsoft.com, Microsoft Research, USA; **Yeyun Gong**, yegong@microsoft.com, Microsoft Research Asia, China; **Chris Hawblitzel**, Chris.Hawblitzel@microsoft.com, Microsoft Research, USA; **Shuvendu Lahiri**, shuvendu@microsoft.com, Microsoft Research, USA; **Jacob R. Lorch**, jaylorch@gmail.com, Microsoft Research, USA; **Shuai Lu**, shuailu@microsoft.com, Microsoft Research Asia, China; **Fan Yang**, fanyang@microsoft.com, Microsoft Research Asia, China; **Ziqiao Zhou**, ziqiaozhou@microsoft.com, Microsoft Research, USA; **Shan Lu**, shanlu@microsoft.com, Microsoft Research, Redmond, USA and University of Chicago, Chicago, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART396

<https://doi.org/10.1145/3763174>

## 1 Introduction

Generative AI (GenAI) techniques have been shown to be effective for many software engineering tasks. GenAI-based code-synthesis tools, like GitHub Copilot, have been widely used in practice. A recent survey among professional developers has shown that more than 60% of developers are currently using AI-assistants in their code development, and yet two thirds of them do not trust the code generated by AI [40]. It would be great if GenAI could synthesize not only code but also code-correctness proof. Indeed, recent research [5, 13, 30, 33, 34, 42, 54] has shown the potential of GenAI in synthesizing proofs in proof-oriented languages like LEAN [9], F\* [44], Dafny [27]. However, the state of the art of GenAI-for-proof is still far behind that of GenAI-for-coding both in terms of benchmark building and generation capability. Furthermore, there has been little work about how GenAI can help directly prove the correctness of programs written in a general-purpose popular programming language. In this paper, we explore using LLMs to automatically generate proof annotations that allow Verus [25] to formally verify the correctness of Rust code.

We choose to focus on Verus for two main reasons. First, the usage opportunity: Verus is a state-of-the-art SMT-based verifier for Rust with a particular focus on practical usage [24, 43, 55]. Verus directly works on Rust code; Verus specifications and proof annotations are also written with Rust syntax and are inlined in the Rust code under proof in the form of *ghost code*. This way, users do not need to learn a new language. Verus also leverages the Rust type system to avoid the complexity of reasoning memory-safety and aliasing, largely reducing the amount of needed proof annotation. Considering the growing popularity of Rust [35, 38, 41], an effective proof assistant for Verus has the potential to benefit millions of developers.

Second, the research opportunity: Verus has unique features that present intriguing research questions for GenAI. Unlike many verifiers, Verus does not require LLMs to learn a new language — LLMs should be familiar with Rust already. However, also unlike many established verifiers, Verus is very young, with its development started about 3 years ago, and hence has much fewer examples for LLMs to learn from. Furthermore, although Verus proof annotations (short as proof) are written in Rust syntax, their usage is very different from regular Rust executable code and require expertise in verification to master. Verus' emphasis on verification speed, a property that is critical for Verus to work for large system code, sometimes comes at the cost of extra proof annotations that may not be needed by other SMT-based verifiers (e.g., Dafny), which creates extra challenges for GenAI.

### 1.1 An Example

Figure 1 illustrates an example of Rust code annotated with Verus specification and proof, which we will refer to as *Verus program* in this paper. This particular example is based on Task 764 in the MBPP dataset [2], which says “Write a function to count the number of digits in a given string”. This is exactly what the Rust function `count_digits` in Figure 1b does it through a while loop.

This function’s correctness specification includes two parts: 1) a spec-function `cnt_dig` implementing the same functionality in a recursive manner in Figure 1a; 2) a post-condition of `count_digits` (Line 20) stating that its result should be the same as that of spec-function `cnt_dig`.

For Verus to successfully verify this post-condition, two types of proof annotations are needed: 1) a number of loop invariants that state what properties are true at the beginning and the end of each loop iteration (Line 26–28); 2) two assert statements that help the underlying SMT-solver to reason about the collection `text` (Line 33–36). Note that, these asserts are different from the Rust builtin `assert!` macro. The Rust `assert!` is used for run-time checking. Instead, the expression enclosed in Verus `assert` is to be verified statically and to provide hints to the SMT solver.

We use this example to highlight a few points mentioned earlier, and we will offer more explanation about this example in the next section.

```

1spec fn is_digit(c: u8) -> bool {
2    c >= 48 && c <= 57
3}
4
5spec fn cnt_dig(seq: Seq<u8>) -> int
6    decreases seq.len(),
7{
8    if seq.len() == 0 {
9        0
10   } else {
11       cnt_dig(seq.drop_last()) +
12       if is_digit(seq.last()) {
13           1 as int
14       } else {
15           0 as int
16       }
17   }
18}
19fn count_digits(text: &Vec<u8>) -> (ret: usize)
20    ensures ret == cnt_dig(text@),
21{
22    let mut count = 0;
23    let mut i = 0;
24    while i < text.len()
25        invariant
26            i <= text.len(),
27            count <= i,
28            count == cnt_dig(text@.subrange(0, i as int)),
29        {
30            if text[i] >= 48 && text[i] <= 57
31                {count += 1;}
32            i += 1;
33            assert(text@.subrange(0, i - 1 as int)
34                  == text@.subrange(0, i as int).drop_last());
35        }
36        assert(text@ == text@.subrange(0, i as int));
37    count
38}

```

(a) The spec functions.

(b) The implementation.

Fig. 1. A Rust function, in gray background, with Verus annotations. The dark-yellow background highlights the proof annotation needed by Verus to prove the specifications highlighted in the light-yellow.

First, the goal of this work is to automatically generate proof annotations when provided with Rust code and its specifications. For this example, we would aim to generate all the loop invariants and the `assert` statements (i.e., all the content in dark-yellow background in Figure 1b).

Second, luckily, we do not need to teach LLMs a new language. As shown in Figure 1, except for a few keywords like `ensures`, `invariant`, `spec`, everything else is written in Rust syntax.

Third, there are still many challenges to address. In this example, why those two `assert` statements are needed and why the '@' symbol is needed on Line 20, 28, 33–36, can be difficult to understand for inexperienced Verus users and LLMs. The loop invariants in this example are not as difficult, but still non-trivial. For example, it is not obvious why the invariant `count <= i` is needed on Line 27 — it is needed so that Verus can prove there will be no arithmetic overflow inside the loop. Things get more complicated when proof functions and quantifiers (i.e., `forall`, `exists`) are needed as we will see later.

## 1.2 Contributions

Facing these unique opportunities and challenges, we have designed a tool based on GenAI, called **AUTOVERUS** , with the following principles in mind:

*No reliance on large training data sets.* Unlike prior GenAI-for-proof work that trains a fine-tuned model [16, 44] or builds a Retrieval Augmented Generation system to dynamically pick different examples for different proof tasks [33, 54], we rely on the code understanding and code synthesis capability of existing LLMs (e.g., GPT-4o).

*Using human Expertise*  to compensate for the lack of training data. Specifically, we summarize common proof strategies used by Verus experts and put them into the instructions of a set of LLM agents. We also design AUTOVERUS to orchestrate these agents following the proof-development process of Verus experts. Human experts do *not* aim to write a perfect proof like the one in Figure 1 in one attempt, neither does AUTOVERUS.

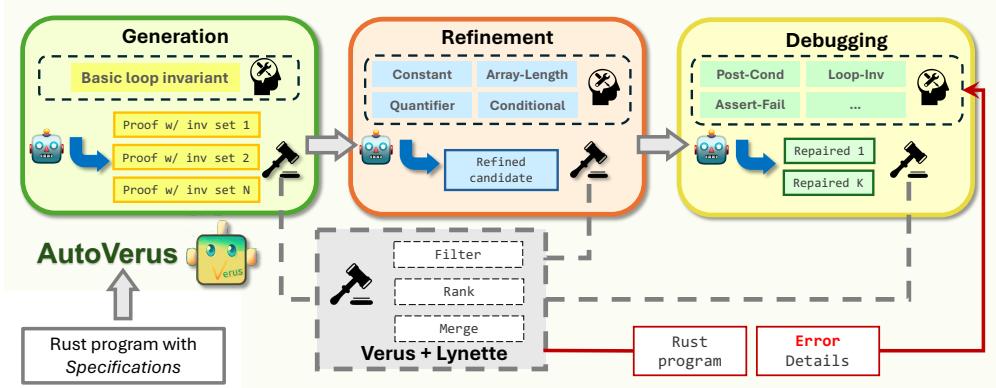


Fig. 2. The workflow of AUTOVERUS

*Unleashing the Creativity 🤖 of LLMs when expert knowledge does not help.* It is impossible to enumerate all proof-writing and debugging tricks, or to anticipate all mistakes that LLM may make. To handle inevitable and yet unexpected challenges in writing a proof, AUTOVERUS configures all its agents with a high-temperature, multi-output setting so that a wide range of diverse and creative proofs can be generated for AUTOVERUS to pick from.

*Enforcing Discipline 🔨 upon creative LLMs through formal methods and static checking.* A network of creative LLM agents can produce a huge number of proof candidates<sup>1</sup>. Many, if not most, of them are too flawed to further explore and may contain “cheating behaviors” like modifying specifications, modifying the Rust code under proof, adding `assume(...)`<sup>2</sup>, etc. To effectively navigate through this huge search space, we combine formal methods and static analysis to quickly filter out invalid candidates, to rank proof candidates based on Verus’ feedback, and to stitch proof snippets together in the hope of creating a perfect proof out of a collection of imperfect ones.

Putting these together, AUTOVERUS conducts its proof generation for input Rust code<sup>3</sup> in three phases, as shown in Figure 2:

- (1) *Generation phase*: AUTOVERUS uses an LLM agent to generate preliminary loop invariants;
- (2) *Refinement phase*: AUTOVERUS runs a series of agents, each looking for a common mistake or a common omission in loop invariants and conducting refinement accordingly;
- (3) *Debugging phase*: Guided by Verus, AUTOVERUS goes through rounds of debugging to produce the correct proof, not only fixing loop invariants but also adding proof annotations like assert statements and lemma functions. In each round, AUTOVERUS examines all the verification errors reported by Verus, decides which error to focus on, and invokes a corresponding agent to fix the error—around 10 agents are designed, each handling one major type of verification errors following the common strategies used by Verus experts (e.g., “post-condition not satisfied”, “loop invariant not satisfied before the loop”, etc.).

Throughout these three phases, proof filtering, ranking, and merging are conducted using Verus and Lynette, an AST-level code analysis tool that we build for AUTOVERUS upon Verus parser.

To thoroughly evaluate AUTOVERUS and also to facilitate future research in AI-for-Verus, we have curated a benchmark suite that consists of 150 non-trivial proof tasks, with each task being

<sup>1</sup>Imagine a set of 10 LLM agents each producing 5 Verus programs based on an input Verus program. Depending on which agent’s output to feed into which other agent, there could be more than  $10! \times 6^{10}$  proof candidates to choose from!

<sup>2</sup>`assume(P)` creates an assumption that *P* is true. It is used for proof debugging and should not appear in the final proof.

<sup>3</sup>In our current prototype, a single Rust function is the unit of proof generation.

a small Rust program associated with Verus specification. This is the first benchmark suite designed for Verus proof generation, and we refer to it as **Verus-Bench**. These 150 proof tasks are mainly translated from other benchmark suites (Diffy [6], MBPP [33], and CloverBench [42]) into Rust/Verus. More details of Verus-Bench are in § 6.1.

Our evaluation on Verus-Bench shows encouraging results:

- AUTOVERUS has a good proof-synthesis capability. It automatically proves more than 90% of the tasks in Verus-Bench (137 out of 150). In contrast, a baseline design of directly prompting GPT-4o can only prove 45% of these tasks (67 out of 150), although this baseline has a much longer time budget, a larger LLM-call budget, a carefully designed prompt, and an unfair advantage of including the answer proof to four complicated Verus-Bench tasks as examples.
- AUTOVERUS is efficient in its proof synthesis. Even with a tight budget of 30 seconds or 3 LLM calls for each task, AUTOVERUS can figure out the correct proof for more than half of the Verus-Bench tasks, while the baseline can manage fewer than 40 tasks.
- AUTOVERUS is robust and consistent across proof tasks and models. It not only successfully finishes 100% of the Verus-Bench tasks coming from Diffy and CloverBench, but also 87% of the more complicated tasks coming from MBPP, which often require proof annotations that go beyond loop invariants. The AUTOVERUS framework, including its three-phase proof generation methodology and proof filtering, ranking, merging support, works well with different LLM models: GPT-4o, GPT-4-turbo, and Deepseek-R1 all achieve similar results, with DeepSeek-R1 proving slightly fewer tasks than GPT-4o/turbo, and GPT-4-turbo spending more time in proving the same number of tasks than GPT-4o.

Of course, AUTOVERUS is still just a starting point in our pursuit of GenAI-for-proof. We hope our Verus-Bench will help evaluate and guide future research in this domain. We also hope our experience of combining human expertise, LLMs, and compiler and formal methods in AUTOVERUS will be useful for future work in GenAI-for-proof.

## 2 Background: Verus

Verus verification incurs *no* run-time checks. Instead, Verus statically analyzes the Rust executable code and the Verus annotations, which are referred to as *ghost code*, and constructs SMT formulas to query the underlying SMT solver. Based on the query result, Verus knows if the Rust code can satisfy some specifications for all possible executions. After the verification, Verus can erase all the ghost code to reproduce the original Rust implementation, allowing easy integration with other Rust code that developers may choose not to verify. This workflow is unlike many other verification tools that require the verification to be discharged in a different project as the implementation.

Verus has made small extensions on Rust syntax through macros, including new abstract data types. Since fixed-bit-width integer types in Rust (e.g., u64, 64-bit unsigned integer) are not easy for SMT solvers to reason about, Verus additionally supports nat for any natural numbers and int for any integer numbers. By default, int data type is used in all Verus specifications. Similarly, Verus provides some collection types (Seq, Map, Set) that can be used to abstract Rust collections.

Generally speaking, Verus specification annotations include function pre-condition (`requires`), function post-condition (`ensures` as in Figure 1b), and spec functions. The spec function in Figure 1a looks very similar to native Rust functions. However, being ghost code, it is required to be purely functional without mutations of any variables. This feature allows it to be easily transformed into a function in the SMT solver and can be called from other ghost code, such as in the function post-condition on Line 20, and in the loop invariant on Line 28. Note that, executable functions *cannot* be called in ghost code – a mistake that LLM makes a lot.

Verus proof annotations generally include loop invariants, assert statements, and proof functions. Loop invariants are needed for the verifier to reason about a loop. asserts are generally used to help SMT solvers reason about complicated properties involving quantifiers, collections, etc. For example, a lot of commonly used Verus data-structures (e.g., Seq, Set) and their specification APIs (e.g., subrange, take, filter, push) require accompanying axioms for SMT solvers to reason about. To avoid complexity explosion in SMT solver, many of these axioms are not automatically triggered in Verus and need to be explicitly added through asserts, like that in Figure 1b.

Just like spec functions, proof functions are ghost code and need to be purely functional. Proof functions can come with its own premises, specified using requires, and conclusions, specified using ensures. The body of the proof function contains ghost code used to assist Verus in constructing SMT formulas to prove the conclusions. A proof function can be called by other ghost code. As long as the premise of the function is satisfied beforehand, the conclusion of the proof function can help prove properties after its call site. We will see an example of proof function later in § 4.3.

### 3 Unique Challenges and Opportunities for AUTOVERUS

**Training data and benchmarks.** At the time of writing, there are fewer than 10 projects on GitHub developed under Verus. In comparison, there are hundreds of GitHub projects with thousands to hundreds of thousands of files developed under *each* of those more established verification tools like LEAN [9], Coq [4], Dafny [27], Isabelle [36], and F\* [44]. Making things worse, Verus has been designed to verify large system projects (e.g., storage system [32], VM hypervisor component [55], operating system modules [8]). Even with our best efforts, extracting single-function/file benchmark programs from these projects failed due to complicated code dependency and specifications.

**Language syntax.** Unlike most verification tools, Verus allows LLMs to work with a widely used language, Rust. However, this also produces challenges, as LLMs easily stumble at those subtle syntax extensions made by Verus upon Rust.

One such challenge is about the integer data type. Verus uses its abstract int type by default in specification functions and hence often requires type casting, such as the ‘i as int’ on Line 28, 33–36 in Figure 1. This type of casting is difficult for LLMs like GPT that are used to Rust syntax.

As another example, since Rust executable functions cannot be called in ghost code, misuse of Rust APIs in Verus annotation is very common for inexperienced users and LLMs. In lucky cases, like Line 26 of Figure 1, Verus provides a spec-function len() for the Rust collection type Vec. Therefore, text.len() can be used as both a valid Rust API call and a valid spec call in ghost code. Unfortunately, many other Rust APIs do not have corresponding spec. For example, there is no subrange spec function for Rust Vec type. In Figure 1b, if we use text.subrange inside the loop invariants or the assert statement, compilation errors will rise. Instead, text@.subrange is used, where text@ turns text from Rust Vec into Verus Seq, which has a subrange spec function. There are also cases where even @ does not help. For instance, GPT-4o often tries to create a Verus sequence using !seq[...] or access the first x elements with seq[..x], both of which follow Rust syntax but are invalid for Verus Seq data type.

**What constitutes a proof.** For different types of verification tools, what users need to write to accomplish a proof can be vastly different, which creates different proof-automation challenges.

Using an interactive theorem prover, such as Coq [4] and LEAN [9], the user writes proof tactics to move the verification forward step by step. In contrast, the SMT solver does the heavy-lifting in Verus and the user only provides hints to help the SMT query. In most cases, this means less effort from Verus users. However, this also implies challenges in measuring progress. When using an interactive theorem prover, the verification goal is transformed based on the tactics used and it is easier to evaluate whether each tactic moves the verification forward, backward, or stagnant by comparing the verification goal before and after the tactic is applied; in contrast, although Verus

```

1  while i < N {
2      if i == 0 {
3          sum.set(0, 0);
4      } else {
5          sum.set(0, sum[0] + a[i]);
6      }
7      i = i + 1;
8  }

```

Fig. 3. A code snippet from Difffy benchmark

```

1  sum[0] = 0;
2
3  while i < N {
4      sum.set(0, sum[0] + a[i]);
5      i = i + 1;
6  }
7

```

Fig. 4. GPT-4 output when fed w/ the code in Fig. 3

provides the information regarding which part of the code has caused the constructed SMT formula to fail to hold, it is difficult to precisely tell whether progress has been made towards the end goal of proving function post conditions, a challenge shared by other non-interactive verifier [5].

Using a proof-oriented language like F\*, code implementation and proof are intertwined; in contrast, in Verus, code implementation and proof are separated. On one hand, this means an easier task for AUTOVERUS, as it only needs to generate the proof but not the implementation unlike that in GenAI-for-F\* [5]. On the other hand, this raises the challenge of disciplining LLMs into not modifying the given implementation. This turns out to be difficult. For example, when given the code in Figure 3, GPT-4 tends to change it to be that in Figure 4.

**Miscellaneous features of Verus.** The design of Verus has paid particular attention to the verification speed. For example, Verus often takes only a second to verify a function. Although this allows AUTOVERUS to feed many proof candidates to Verus to see which one works, it sometimes comes at the cost of extra annotations of loop invariants and asserts.

## 4 Design

**Terminology.** The input and output of AUTOVERUS, as well as each of its agents, are *Verus programs* that contain Rust code and Verus annotations. The input program to AUTOVERUS contains no proof annotations; the output of AUTOVERUS contains not only Rust code, specification, but also proof annotations added by it. We refer to a Verus program as *unsafe*, if it modifies the Rust code and/or specifications under proof; as *invalid*, if it leads to Verus compilation errors; as *correct*, if it is safe, valid, and can be completely verified by Verus.

To design AUTOVERUS, we have interviewed multiple co-authors of the Verus paper [24]. Their suggestion about how to handle common Verus errors guided the design of AUTOVERUS agents. Furthermore, their description of the common proof-development process laid the foundation of the three-phase workflow. Specifically, we learned that the proof development of human experts is an iterative process of repeatedly running Verus, checking and prioritizing Verus errors, developing and editing proof to fix them, and so on—a process that is also confirmed through user studies [20].

In the following, we present each of the three iterative phases of AutoVerus, discussing how the LLM agents of each phase are designed, how they are coordinated, and how their output is processed leveraging Verus.

### 4.1 Phase 1: Preliminary Proof Generation

Verus proof typically includes loop invariants, asserts, and sometimes proof functions/blocks. Since asserts and proof functions/blocks are typically added during the proof debugging to provide extra hints to SMT solver, at this first phase of proof generation, AUTOVERUS focuses on loop invariants. An agent is designed to generate loop invariants for every loop in the target Rust code.

**4.1.1 Agent design.** In the prompt to LLMs, we ask it to “add loop invariants to the given Rust code, so that Verus can verify the given function behaves exactly what is described in the specifications.”

---

**Algorithm 1:** Preliminary proof generation with post-processing
 

---

```

1 Function GenPrelimProof(program):
2   pCandidates  $\leftarrow$  LoopInvAgent(program)
3   safeCandidates  $\leftarrow$  {p  $\in$  pCandidates | is_safe(program, p)}
4   Sort safeCandidates by score
5   mergeP  $\leftarrow$  program
6   foreach p  $\in$  safeCandidates do
7     mergeP  $\leftarrow$  merge (mergeP, p)
8     if is_correct (mergeP) then
9       return mergeP
10  return first element of safeCandidates
11 Function is_correct(p):
12   return verus_correct (p) or verus_correct (houdini (p))

```

---

Furthermore, we teach LLMs three things that we believe are the most important in writing loop invariants in Verus: 1) to describe the initial value of every variable read in the loop; 2) to describe the assigned value of every variable written in the loop; and 3) to leverage spec and proof functions when needed. We have intentionally kept this agent simple and leave it to later refinement/repair agents to correct any mistakes or oversights of this first agent.

We also include three toy Verus examples written by us in the prompt to teach LLMs about basic Verus features like invariant, quantifiers (i.e., forall and exists), and abstract data structures (e.g., vstd::Seq). Each example contains less than 30 lines of Rust code and 10–20 lines of proof annotations. We stick to these same examples across all proof tasks.

**4.1.2 Post-Processing of Agent Output.** Phase-1 of AUTOVERUS is much more than just invoking an LLM agent, as much post-processing is needed to filter, rank, and merge the LLM agent’s output.

As shown in Algorithm 1, after invoking the LLM agent described above to produce 5 candidates, AUTOVERUS first filters out the candidates that are unsafe, like the one in Figure 4, through static analysis. AUTOVERUS also filters out invalid candidates that contain syntax errors, unless AUTOVERUS can fix the errors through simple code editing, such as changing *v[i]* to *v[i as int]* when Verus complains that “type int is expected for the expression i”.

AUTOVERUS then ranks remaining LLM-generated Verus programs based on a score tuple  $\{V, E\}$  (Line 4 in Algorithm 1), with *V* being the number of functions<sup>4</sup> successfully verified by Verus, and *E* being the number of verification errors reported by Verus. For any two programs that contain the same executable Rust code, AUTOVERUS considers the one with higher *V* score as the better one (i.e., more functions proved), with the tie-breaker being having a lower *E* score (i.e., fewer verification errors). If the top-ranked Verus program allows verification to succeed, the whole proof task is done. Otherwise, AUTOVERUS moves on.

Finally, given a list of *K* valid and safe programs,  $P_1, P_2, \dots, P_k$ , AUTOVERUS checks if merging some of them might produce a better proof (Line 6 – 9 in Algorithm 1). Since merging takes time, AUTOVERUS only explores linear, instead of exponential, number of merging schemes: AUTOVERUS starts with the highest ranked program, considered as best-so-far  $P_B$ , and goes down the ranked list. When merging the best-so-far program  $P_B$  with a program  $P_i$  produces a higher-scored program  $P'$ , AUTOVERUS updates  $P_B$  to be  $P'$  and continues to check the next program on the list  $P_{i+1}$  until all

---

<sup>4</sup>Verus counts the loop as a separate function. For example, when a function containing two loops is verified, *V* is 3.

```

1invariant
2 N > 0,
3 i <= N as usize,
4 sum.len() == 1,
5 sum[0] <= N + i,
6 forall|k: int|0<=k<N ==> b[k]==1,
7 b.len() == N,
8// (V, E) = (4, 2)

(a) Loop invariants of best-so-far  $P_B$ .
```

```

1invariant
2 N > 0,
3 i <= N as usize,
4 sum.len() == 1,
5 sum[0] <= N + i,
6 forall|k: int|0<=k<N ==> b[k]==1,
7 N < 1000,
8// (V, E) = (3, 2)

(b) Loop invariants of a lower-ranked  $P_i$ .
```

```

1i = 0;
2
3while (i < N as usize)
4 invariant
5 N > 0,
6 i <= N as usize,
7 sum.len() == 1,
8 sum[0] <= N + i,
9 forall|k: int|0<=k<N ==> b[k]==1,
10 b.len() == N,
11 N < 1000,
12{
13 sum.set(0, sum[0] + b[i]);
14 i = i + 1;
15}
16// (V, E) = (5, 0)
17// Verified
```

(c) The merged program  $P'$  which is *verified*.

Fig. 5. An example of a merged program for a problem from Difffy benchmark.

the  $k$  programs are checked. If AUTOVERUS fails to identify a perfect proof throughout this process, the final best-so-far program  $P_B$  will become the input to the next phase of AUTOVERUS (Line 10).

Figure 5 shows an example of how AUTOVERUS handles a problem from Difffy benchmark that contains 4 loops. For this task, none of the five programs output by the LLM is correct. In fact, two of them cannot even be compiled, because LLM used quantifiers incorrectly – it adds “`forall |k:int| ==> sum[0] == k + i`” as an invariant for the loop in Figure 5c (there is no way `sum[0]` can equal all integers!). Fortunately, the other three programs are safe and valid. In the highest-ranked program, three loops and the function post-condition are verified (i.e.,  $V$  score is 4), but two verification errors are reported for the last loop with the loop invariants shown in Figure 5a. Another program encountered index out-of-bound errors in the last two loops, and hence is ranked lower (i.e.,  $V$  is 3). Its invariants for the last loop are shown in Figure 5b. As we can see, these two sets of invariants only differ by the last line: the lack of  $N < 1000$  in Figure 5a caused arithmetic overflow errors; the lack of  $b.len() == N$  in Figure 5b caused its index out-of-bound error. When we merge the second program (Figure 5b) into the first, we get the perfect proof in Figure 5c.

Some implementation details are skipped here and will be presented in § 5: we have implemented a static analysis and code transformation tool, Lynette, to check whether a Verus program is safe and to merge two Verus programs together if they are based on the same Rust code. Furthermore, for every valid and safe Verus program  $P$  (no matter generated directly by LLM or through merging), we apply Houdini algorithm [17] to efficiently check if a subset of the Verus proof annotations in  $P$  is a correct proof. This Houdini treatment applies to the next two phases as well.

## 4.2 Phase 2: Generic Proof Refinement

This phase aims to refine the loop invariants in the best-so-far Verus program  $P_B$  generated by phase-1, correcting common verification mistakes and omissions.

**4.2.1 Refinement-Agent Design.** The refinement phase consists of a series of LLM agents, each aiming one common mistake in the writing of loop invariants in Verus:

Constant-Propagation agent checks if the function pre-condition includes properties about a read-only parameter, and adds these properties as invariants to every loop in the function. For

example, loop invariants  $N > 0$  and  $N < 1000$  in Figure 5c are part of the function pre-condition in that example. If they are missing, this agent will add them.

Array-Length agent checks if the length information of every array/container used in a loop is specified as a loop invariant, and adds such information if not. For example, if the invariant `sum.len() == 1` is missing from Figure 5c, it is be added here.

Quantifier agent checks if quantifier-related invariants are used correctly. For example, if Line 9 of Figure 5c mistakenly has `forall |k:int| 0 <= k < i ==> b[k] == 1`, instead of `forall |k:int| 0 <= k < N ==> b[k] == 1`, it would be corrected here.

Conditional-Loop-Invariant agent urges LLM to check if any loop invariant may only apply to some, instead of all, of the loop iterations and make adjustment accordingly. For instance, this would help LLMs handle the example in Figure 3.

Note that, this set of refinement agents can be easily extended, and the post-processing presented below makes sure that different agents can easily be composed. In the current prototype of AUTOVERUS, we keep this set relatively small, as these agents are invoked unconditionally regardless of how many and what verification errors are reported by Verus. These four agents are designed based on common mistakes that we ourselves typically commit when we write loop invariants.

**4.2.2 Agent Composition and Output Post-Processing.** Algorithm 2 shows the refinement workflow. AUTOVERUS invokes every refinement agent sequentially (Line 3 of Algorithm 2), starting with the simplest, Constant-Propagation, and progressing to the more complex ones. Each agent takes the best-so-far Verus program  $P_B$  as its input and generates a new refined program  $P$ . If  $P$  turns out to contain the correct proof, the whole task is done; otherwise, AUTOVERUS replaces the original best-so-far program  $P_B$  with  $P$  when two conditions are met: 1)  $P$  is a valid and safe Verus program after AUTOVERUS’s type-casting edits, if needed; and, 2)  $P$  does not lower the number of functions successfully verified by Verus in  $P_B$ .

---

#### Algorithm 2: Proof Refinement

---

```

1 Function RefineProof(program):
2   best  $\leftarrow$  program
3   foreach refineAgent  $\in$  refineAgentList do
4     p  $\leftarrow$  refineAgent(best)
5     if is_correct (p) then
6       return p
7     if accept_refine (p) then
8       best  $\leftarrow$  p
9   return best

```

---

The post-processing here is simpler than that of Phase-1: each agent only generates one, instead of five, proof candidates, and hence there is no merging or ranking needed. The rationale is that each agent in this phase conducts a relatively straightforward action, and, though every agent is useful in general, it may or may not be necessary for every Verus program given to it.

### 4.3 Phase 3: Error-Driven Proof Debugging

Proof construction is similar to code writing in that debugging is inevitable for realistic and complicated tasks. Human experts expect to see verification errors and are experienced at repairing proofs to fix those errors one by one. Therefore, we have designed this debugging phase for AUTOVERUS to mimic human experts when its first two phases fail to accomplish the proof.

This debugging phase is arguably the most important phase of AUTOVERUS: any mistake or limitation of the first two phases can be rescued here. This is also the phase that allows AUTOVERUS to generate proof annotations beyond just loop invariants, as we see below.

**4.3.1 Repair-Agent Design.** The agent design in this phase is guided by a list of common Verus verification errors, with each agent  $A_t$  designed to fix one type  $t$  of Verus verification errors.

**Algorithm 3:** Debug Proof Generation

```

1 Function DebugProof(program):
2   for i  $\leftarrow$  1 to MaxIter do
3     if is_correct (program) then
4       return program
5     errors  $\leftarrow$  verus_compile (program)
6     e  $\leftarrow$  select_error (errors)
7     repair_agent  $\leftarrow$  select_repair_agent (e)
8     repairedCandidates  $\leftarrow$  repair_agent (program, e)
9     foreach p  $\in$  repairedCandidates do
10      if is_correct (p) then
11        return p
12      if accept_repair (p) then
13        program  $\leftarrow$  p
14        break
15   return program

```

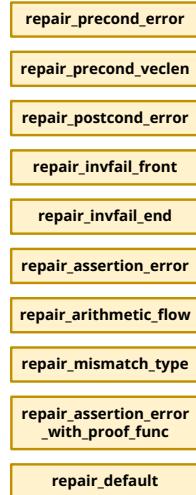


Fig. 6. AUTOVERUS repair agent list

Specifically, AUTOVERUS targets the eight most common types of verification errors, including function pre-condition errors, vector-index out-of-bound errors (a common sub-type of function pre-condition errors), function post-condition errors, loop-invariant unsatisfied at the beginning of a loop errors, loop-invariant unsatisfied at the end of a loop errors, assertion errors, arithmetic under/overflow errors, and type mismatch (syntax) errors. As listed in Figure 6, one repair agent is designed for each type of error. In addition, an extra repair agent is designed to handle assertion errors, the most diverse type of verification errors. Finally, a generic repair agent, “repair\_default” in Figure 6, is designed to handle any verification error that does not fall into the eight types listed above. In practice, it is very rare to encounter verification errors outside these eight types, as we will see in our evaluation.

The prompt to agent  $A_t$  that handles a verification error of type  $t$  includes not only the Verus program under repair, but also 1) the code snippet and detailed information about an error  $E_t$  to be fixed, which has the verification-error type  $t$ , and 2) the instruction about how to fix errors of type  $t$ . The instruction is crafted by experienced Verus users.

It could be boring to go through every repair agent one by one, as each agent basically summarizes how human experts handle a type of verification error. Thus, we skip that discussion here. Later in this section, we will go through a case study of how AUTOVERUS works on the proof task fibonacci, where we will see what some of the main agents are and how they actually perform in practice.

**4.3.2 Repair-Agent Composition and Output Post-Processing.** It is challenging to coordinate all the agents because a Verus program often contains more than one verification error and each error may take multiple attempts to get fixed. Furthermore, fixing one error may introduce another error(s), and it could be difficult to tell whether the repair has made the proof better or worse.

With these challenges in mind, AUTOVERUS conducts its debugging in iterations, as shown in Algorithm 3, starting with the best-so-far Verus program  $P_B$  generated by the refinement phase. In each iteration, AUTOVERUS collects all the verification errors  $E$  in  $P_B$  from Verus (Line 5), picks one error  $E$  from  $E$  to fix next (Line 6), and prompts a repair agent that suits  $E$  (Line 7). AUTOVERUS then examines the agent’s output one by one to decide whether to accept a repair to replace the best-so-far proof  $P_B$  (Line 13). Finally, AUTOVERUS prepares for its next iteration based on whether

the target error  $E$  has been fixed or not. AUTOVERUS keeps running these iterations until a correctly proved program is found or a pre-configured threshold is reached (10 iterations by default). We explain these three steps in more detail below.

**How to pick the error to fix next?** We have designed an ordered list of Verus error types and AUTOVERUS ensures that a verification error is picked only when higher-priority errors no longer exist for the program under proof. The top one is *type error*, because they have to be fixed before Verus can compile the program and conduct verification attempt. The next two are bound-related errors, including the *index-bound-error* and *arithmetic-overflow/underflow* errors. The reason is that once Verus identifies a bound error for any component of an expression  $E$ , it will make no further attempt to prove any property that involves  $E$ . For example, Verus will not attempt to verify *any* of the loop invariants if a bound error is found in the loop body. After that, *invariant-not-satisfied-before-loop* errors get selected, as they are often easier to fix than other types of errors. Beyond that, all remaining errors are repaired one by one based on where they show up in the report of Verus.

**When to accept a repair?** Naturally, any repair that introduces unsafe changes or compilation errors is not accepted; and, any repair that produces a better verification score than  $P_B$  can be accepted like that in Phase 1. Beyond that, AUTOVERUS would also accept a program  $P$  as the new best-so-far proof, as long as  $P$  has resolved the error-under-repair  $E$  without compromising previously verified functions (i.e., the  $V$  score should not drop). Since it is difficult to precisely judge whether  $E$  is fixed, AUTOVERUS counts the number of verification errors of the type of  $E$  in  $P$ , and considers  $E$  to be fixed if that number has dropped from that in  $P_B$ . Once AUTOVERUS finds an acceptable repair generated by the agent, it does not check the remaining output of the agent, because it is very difficult, even for human experts, to rank multiple imperfect repairs.

**How to adjust for the next repair iteration?** AUTOVERUS would review the progress (or lack of) made by the past iteration and adjust accordingly. If the target error  $E$  has not been fixed in this iteration, AUTOVERUS configures the next repair agent to produce 5 programs (3 otherwise). For assertion-failure errors, the most common type in later phase of repair, if the target assert error is not fixed after a threshold number of attempts (3 by default), the more complex assert-repair agent is employed. Finally, if the target error fails to be repaired in even more iterations (5 by default), AUTOVERUS would delete the corresponding proof entity (e.g., a loop invariant or an assert).

**4.3.3 Case Study: AUTOVERUS on Fibonacci.** Figure 7a contains a Rust function that returns the first  $n$  numbers in the fibonacci sequence. It sets the first two numbers in this sequence to be 0 and 1, and then computes the remaining ones through a while loop (Lines 16–36). The spec function fibo in Figure 7b provides a strict mathematical specification of the Fibonacci sequence. What we need to prove for the Rust function `fibonacci(n)` is that every number in its result vector matches that produced by the spec `fibo` under the premise that the  $n$ th fibonacci number fits in the `i32` data type. As we will see below, it is quite challenging to generate a complete and correct proof. AUTOVERUS will accomplish this through several iterations of debugging.

**Iteration 1:** All the gray-background code in Figure 7a, including the red line `fib.len() == n`, comes from the best-so-far proof  $P_B$  from AUTOVERUS’s refinement phase. Verus only reports one error for this version: `invariant fib.len() == n` on Line 24 is not satisfied before the loop.

Our agent dedicated to *invariant-not-satisfied-before-loop* is invoked. Its prompt includes these repair strategies: 1) to add an assert about this invariant right before the loop; 2) in case of nested loops, to add this invariant to its outer loop when applicable; 3) to modify or delete this loop invariant when it is incorrect or unnecessary. One of this agent’s outputs took both the first and the third strategy, as highlighted by -1 and +1 in Figure 7a: it deleted this invariant from Line 23, and added an `assert(fib.len() == 2)` right before the loop on Line 15, accompanied by a comment explaining its action on Line 14. In fact, deleting this invariant is the right repair action: the length

```

1fn fibonacci(n: usize) -> (ret: Vec<i32>)
2    requires
3        fibo_fits_i32(n as int),
4        n >= 2,
5    ensures
6        forall|i: int| 2 <= i < n ==>
7            #[trigger] ret[i] == fibo(i),
8            ret.len() == n,
9    {
10    let mut fib = Vec::new();
11    fib.push(0);
12    fib.push(1);
13    let mut i = 2;
14    // Assert the invariant right before the loop
15    +1 assert(fib.len() == 2);
16    while i < n
17        invariant
18            0 <= 2 <= i <= n,
19            fib.len() == i,
20            fibo_fits_i32(n as int),
21            forall|j: int| 0 <= j < i ==>
22                # [trigger] fib[j] == fibo(j),
23    -1 fib.len() == n;
24    }
25    // Assert the addition won't overflow an i32
26    +2 assert(fib[i-1] as int + fib[i-2] as int
27              < 0x8000_0000) by {
28    +2 assert(fib[i-1] == fibo(i-1));
29    +2 assert(fib[i-2] == fibo(i-2));
30    +2 assert(fibo(i-1)+fibo(i-2)==fibo(i));
31    +2 lemma_fibo_monotonic(i, n);
32    +2 };
33    let next_fib = fib[i-1] + fib[i-2];
34    fib.push(next_fib);
35    i += 1;
36}
37fib
38}
39+2 proof fn lemma_fibo_monotonic(..) {...}

```

(a) Repair the “the loop invariant not satisfied before loop” and “arithmetic overflow” errors.

```

1spec fn fibo(n: int) -> nat
2    decreases n,
3{
4    if n <= 0 { 0 }
5    else if n == 1 { 1 }
6    else { fibo(n - 2) + fibo(n - 1) }
7}
8spec fn fibo_fits_i32(n: int) -> bool {
9    fibo(n) < 0x8000_0000
10}

```

(b) The spec functions.

```

1proof fn lemma_fibo_monotonic(n:int, m:int)
2    requires n <= m,
3    ensures fibo(n) <= fibo(m),
4    decreases m - n
5{
6    if n < m {
7        lemma_fibo_monotonic(n, m - 1);
8    + assert(fibo(n) <= fibo(m - 1));
9    + assert(fibo(m - 1) <= fibo(m));
10   }
11}

```

(c) Repair the post-condition not satisfied error.

```

1proof fn lemma_fibo_monotonic(n:int, m:int)
2    ...
3{
4    if n < m {
5        lemma_fibo_monotonic(n, m - 1);
6        assert(fibo(n) <= fibo(m - 1));
7    - assert(fibo(m - 1) <= fibo(m));
8    + if m > 1 { // Avoiding the case m == 1 to
9        prevent negative indexing in fibo(m-2)
10       assert(fibo(m-2)+fibo(m-1)==fibo(m));
11       assert(fibo(m - 1) <= fibo(m));
12    }
13}

```

(d) Repair the assertion failure error.

Fig. 7. How `fibonacci` is proved by repair agents (all the `comments` are output of GPT-4o agents as well).

of `fib` array matches the loop index  $i$  in the loop, not always  $n$ . Of course, `fib.len() == 2` is correct right before the loop. So, this repair succeeds.

**Iteration 2:** After the first repair, Verus now reports two new errors: 1) an *arithmetic-overflow* for `fib[i-1] + fib[i-2]` on Line 33; 2) the invariant on Line 21 that states `fib[j] == fibo(j)` for all  $j$  between 0 and  $i$  does not hold at the end of the loop.

AUTOVERUS decides to fix the *arithmetic-overflow* first. The corresponding agent is instructed to add loop invariants or asserts to state the upper/lower bound of the arithmetic operations that Verus complains about. The prompt also includes tips about using parametric bound or approximated bound when needed, and reasoning the monotonicity of a data series when needed.

The output of this repair agent is highlighted by the `+2` heading in Figure 7a. As we can see, this agent’s repair includes several parts: 1) it made the right decision that an `assert`, instead of a

loop invariant, is needed here to state the bound on Line 26 (again accompanied by a comment explaining its action); 2) it figured out the right bound 0x8000\_0000 for the expression `fib[i-1] + fib[i-2]`, probably based on the bound expressed in the `fibo_fits_i32` spec-function in Figure 7b; 3) it realized that proving the monotonicity of the fibonacci sequence is needed and hence synthesized a new proof function `lemma_fibo_monotonic` and also used this function to support the bound assertion very nicely on Lines 28–31.

**Iteration 3:** With all this effort, a simpler problem like the one in Figure 1 would have been proved at this point. However, for this example, Verus now reports that the while-loop in Figure 7a is completely verified, but there is a new error—the post-condition `fib(n) <= fib(m)` of the new function `lemma_fibo_monotonic` shown in Figure 7c is not satisfied at the end of the function.

The agent dedicated to *function-post-condition-unsatisfied* error is now called. This agent’s instruction suggests adding asserts (or loop invariants) that correspond to the failed post-condition to the function exit, where the post-condition does not hold (or a relevant loop). Therefore, the agent added the two green-highlighted lines in Figure 7c. Interestingly, the LLM does not simply assert `fib(n) <= fib(m)`, the failed post condition. Instead, it added two asserts that describe the transitive relationship of `fib(n) <= fib(m-1)`, which matches the post-condition of the proof function right before it on Line 7, and `fib(m-1) <= fib(m)`, which is quite nice.

**Iteration 4 & 5:** Unfortunately, this seemingly perfect repair is still incorrect. Verus reports an *assertion-failure* on the newly added Line 9 of Figure 7c — `fib(m-1) <= fib(m)` cannot be verified to always hold. Therefore, AUTOVERUS calls upon its assertion-failure repair agent. This agent is given some general options: 1) if the `assert` expression is related to Verus data structure APIs like `Seq::subrange`, `Seq::filter`, and so on, a set of off-the-shelf lemma functions can be used; 2) change or delete any loop invariants that might be related to the `assert`; 3) add more `assert` statements; 4) add more proof functions if needed. This fix is not easy. In fact, the first time this repair agent is called upon, no good repair is produced. At the second time, the repair agent comes up with the repair shown in Figure 7d: it added two `assert` statements to help the SMT solver reason why `fib(m-1)` cannot be larger than `fib(m)`; and it also added the `if m > 1` condition to avoid negative indexing, as explained by its own comment.

Finally, Verus reports that the whole proof task was successfully done!

What we describe above is the output from a particular run of AUTOVERUS. Since this is a difficult task, we have noticed that AUTOVERUS runs into different situations every time, and the final proof produced by it is not always the same. Fortunately, with the basic proof-repair knowledge embedded in AUTOVERUS agents and the robust post-processing, AUTOVERUS can reliably figure out a correct proof after a number of attempts.

## 5 Implementation

We implemented AUTOVERUS as a command line Python tool using GPT-4o invoked by Azure OpenAI APIs. In the following, we discuss some implementation details.

**Houdini alg.** Given a set of proof annotations A that fail the verification, if a subset of A is correct and sufficient to prove, Houdini algorithm guarantees to identify this subset in a linear number of verifier invocations. More details of this long-established algorithm can be found in previous work [17]. This algorithm has been used for GenAI-for-C [23] and we apply it here for Verus.

**Lynette.** To add discipline into AUTOVERUS, we have implemented Lynette using the Verus front-end parser in Rust to post-process the LLM-generated code at the AST level. To check whether a LLM output  $P$  is safe regarding the input Rust program and its specification  $P_o$ , Lynette checks if  $P$  and  $P_o$  can be compiled to the same executable by comparing the pure Rust AST of the files after erasing the ghost code; Lynette compares the pre- and post-condition of  $P$  and  $P_o$  so the LLM

Table 1. Summary of Verus-Bench

Benchmark Sources	CloverBench	Diffy	MBPP	Misc	Total
# of Proof Tasks	11	38	78	23	150
Executable LOC	175	951	1,333	390	2,849
Specification LOC	80	265	700	207	1,252

cannot tweak them to change the goal of the verification; Lynette also searches for debugging ghost function, such as `admit()` or `assume()`, in  $P$ .

Another use of Lynette is in program merging. Naively using a text-merging tool does not work for proof candidates,  $P_1$  and  $P_2$  of a complicated proof task, as  $P_1$  and  $P_2$  may differ quite a lot. Lynette first erases all of the ghost code in the program to obtain the pure Rust AST, which is the same for  $P_1$  and  $P_2$ , and uses it as the anchor to merge the ghost code in  $P_1$  and  $P_2$ . The minimum merging unit is a ghost expression, *i.e.*, a Verus assertion, a proof block, or an invariant.

The third use of Lynette is to support the Houdini algorithm in deleting the unprovable proof annotations from the proof candidate. Since Lynette is AST-aware, it makes sure that the deletion honors the boundary of ghost expression, and guarantees to produce a syntactically correct result.

**Verus configuration.** Verus has a loop-isolation configuration that can affect the difficulty of proof. When it is set to be false, Verus will consider facts (e.g., function pre-conditions, asserts) from outside a loop body in its proof inside the loop body. By default, this configuration is set to be true for performance reasons. If AUTOVERUS fails to find a correct proof after the first two phases, it would add a `#[verifier::loop_isolation(false)]` statement to the program under proof to make proof debugging a little bit easier.

## 6 Methodology

### 6.1 Verus-Bench

To evaluate AUTOVERUS, we need a suite of proof tasks, with each task being a Rust program annotated with Verus specifications. Since such a suite does not yet exist, we crafted Verus-Bench.

We tried our best to make Verus-Bench a non-trivial, non-biased, and meaningful suite through the following methodology. First, instead of writing proof tasks ourselves, we mainly looked into three verification-related benchmark suites in other languages and translated those tasks into Verus. Second, we make sure **not** to include any trivial proof tasks (*i.e.*, tasks that can be proved by Verus without any proof annotations). In fact, 88 proof tasks from our source benchmark suites are rejected for this reason! Finally, we have divided our team early so that the team who designed the AUTOVERUS does not overlap with the team who set up the majority of the Verus-Bench proof tasks (*i.e.*, all those in MBPP and CloverBench).

In total, Verus-Bench contains 150 proof tasks as shown in Table 1.

78 proof tasks are translated from the **MBPP**-DFY-153 [33] dataset. That dataset contains 153 problems with specifications and verified implementation in/for Dafny, based on the MBPP dataset [2]. Among these 153, 67 of them require *no* proof annotations to verify; 8 are too difficult to translate due to code features not well supported by Rust/Verus like floating points and string; the remaining 78 are all translated and included in Verus-Bench.

11 are from **CloverBench** [42]. This suite includes 60 example programs, as might be found in CS textbooks, written and verified in Dafny. The authors of CloverBench also provide the Verus translation for 39 of them. Our manual inspection found that 3 of them cannot be verified by the

latest version of Verus, 4 have weaker specifications than their variants already included in the dataset, and 21 can be verified without proof annotations. We add all remaining 11 into Verus-Bench.

38 come from a SV-COMP-2021 [12] benchmark suite, **Diffy** [6]. Diffy contains 69 programs written in C that were designed to evaluate array and loop related verification. Most tasks in this suite contain multiple loops and hence require good loop-invariant synthesis. 31 tasks here require reasoning about non-linear equations (e.g., knowing that  $(i + 1)^3 == i^3 + 3 * i^2 + 3 * i + 1$ ), which is not the focus of Verus system verification. We translated all the remaining 38 tasks from C to Rust.

The final 23 is referred to as **Misc**. This is a collection of 23 miscellaneous Verus programs that appeared in Verus tutorials or libraries. They include some algorithmic programs, like bubble sort, binary search, fibonacci sequence, and tasks that contain challenging features like nested loops, expressing function post conditions as asserts instead of ensures, etc.

## 6.2 Experimental Setup

**6.2.1 Hardware and software.** All our experiments are run on a machine with Ubuntu 22.04.1 LTS, 24-core CPUs, and 64 GB RAM. By default, AUTOVERUS uses GPT-4o (model version 2024-05-13). For comparison, we will also run AUTOVERUS on three other models, GPT-4-turbo, GPT-3.5-turbo, and Deepseek-R1-32B.

**6.2.2 Alternative designs in comparison.** Given the huge design space of LLM-for-Verus-proof, we will refer to the following design as a *baseline*. In this design, we repeatedly invoke GPT-4o to generate Verus proof for the input proof task. To make this baseline competitive, we designed its prompt to essentially be a summary of all the prompts used in AUTOVERUS agents. It uses the same LLM-temperature setting as AUTOVERUS (1.0), and is configured to generate 5 outputs in each invocation, same as the Phase-1 agent in AUTOVERUS. Finally, we give this baseline an unfair advantage of “cheating”: we put the answers to four complicated tasks from Verus-Bench as examples in this baseline’s prompt. Notably, there is no overlap between the examples used by AUTOVERUS agents and Verus-Bench. In addition to this baseline, we will also evaluate schemes that differ from AUTOVERUS at various design points (i.e., different designs in or lack of phase 1, phase 2, and phase 3) or use different LLM models (i.e., GPT-4-turbo, GPT-3.5-turbo, Deepseek-R1) in our extensive ablation study. These schemes are mostly more competitive than the baseline above, and can offer insights about how different components of AUTOVERUS work together.

**6.2.3 Evaluation metrics.** We will mainly evaluate three metrics: *number of correctly verified tasks*, *time* and *number of LLM calls* required to come up with the correct proof. Due to the randomness of LLMs, the number of correctly verified tasks may change over runs. By default, we report the number of correctly verified tasks after running AUTOVERUS (the whole three phases) for three times; the same applies for various variants of AUTOVERUS. When comparing with the baseline, we will show the number of verified tasks under the same time and LLM-call budget to be fair. Keep in mind that since Verus can automatically tell whether a proof is correct, there is no manual effort involved even if we invoke LLM or execute the AUTOVERUS workflow for multiple times. The difference between verifying a task in one attempt and in multiple attempts is the time cost and the number of LLM-calls, which we will report.

## 7 Experimental Results

### 7.1 Overall Results

Table 2 shows the overall results of AUTOVERUS on Verus-Bench. AUTOVERUS successfully proves 137 out of 150 (91.3%) benchmark problems. That is, provided with the Rust implementation and specification, AUTOVERUS is able to generate proof annotations that allow Verus to prove the

Table 2. Number of tasks proved by AUTOVERUS (max: 3 tries) and baseline (max: 10 min) in Verus-Bench.

Source	#Tasks	Proved by AUTOVERUS				Proved by Baseline
		Total	Phase-1	Phase-2	Phase-3	
CloverBench	11	11	7	2	2	8
Diffy	38	38	26	12	0	5
MBPP	78	68	36	4	28	43
Misc	23	20	9	4	7	11
Total	150	137	78	22	37	67

specification is guaranteed to hold for 137 out of 150 tasks. Notably, AUTOVERUS verifies all of the problems from the CloverBench and Diffy benchmarks. Even though the default setting allows up to three attempts for AUTOVERUS, AUTOVERUS was able to prove 122 tasks in its first attempt, with 10 more proved in its second attempt and 5 more on its third attempt.

As also shown in Table 2, Phase-1 is able to finish slightly over half of the 150 proof tasks, while Phase-2 refinement and Phase-3 debugging also contribute to 22 and 37 additional proven tasks respectively. This phase-breakdown also shows differences among different benchmark sources. Since the Diffy benchmark focuses on loop invariants, all the tasks from Diffy are handled by AUTOVERUS in the first two phases. In comparison, MBPP and Misc both heavily rely on Phase-3 debugging to generate proof annotations that go beyond loop invariants.

Processing these 137 tasks across three rounds required an average of 8.8 LLM calls per task. While approximately half of the tasks were completed with two calls or fewer, a small number required more than 40. This entire process consumed 3 million input and 1.5 million output tokens. Based on the Azure GPT-4o pricing at the time of our evaluation (\$5 per 1M input tokens and \$15 per 1M output tokens), the total cost for the benchmark suite was approximately \$37, including tasks that failed after all three rounds.

## 7.2 Comparison with Baseline

Table 2 also shows that the baseline scheme of repeatedly invoking LLM (*i.e.*, GPT-4o) was only able to prove 67 out of the 150 tasks (44.7%), even though it was given 10 minutes for each task. Note that, this baseline scheme uses the same LLM-temperature and 5-output per invocation setting as AUTOVERUS, and has more sophisticated prompt and examples than any AUTOVERUS agent.

The baseline does fine in generating loop invariants for simple loops, succeeding in a similar number of tasks from Clover, MBPP, Misc as Phase 1, 2 of AUTOVERUS. Of course, the baseline took much longer and many more LLM calls to achieve this than AUTOVERUS, which we discuss below.

However, the baseline cannot handle more complicated loop invariants, and hence succeeded for only 5 out of 38 Diffy tasks. In comparison, AUTOVERUS’s Phase-1 alone generated correct proof for 26 Diffy tasks. Furthermore, the baseline scheme is poor in generating more complicated proof annotations, and failed many more tasks in MBPP and Misc than AUTOVERUS.

To understand how efficient AUTOVERUS and the baseline are, we plotted how many tasks can be correctly proved by each of them under a fixed time budget (up to 10 minutes per task) and the number of LLM-call budget (up to 60 calls per task), as shown in Figure 8a and Figure 8b. As we can see, AUTOVERUS consistently outperforms the baseline, finishing about twice as many tasks, under the same time and LLM-call budget constraints. In order for the baseline to succeed in 67 tasks, it needs a budget of 458 seconds per task or 59 LLM-call per task; in contrast, 22-second per-task or 2 LLM-call per task is sufficient for AUTOVERUS to generate correct proof for at least 67 tasks.

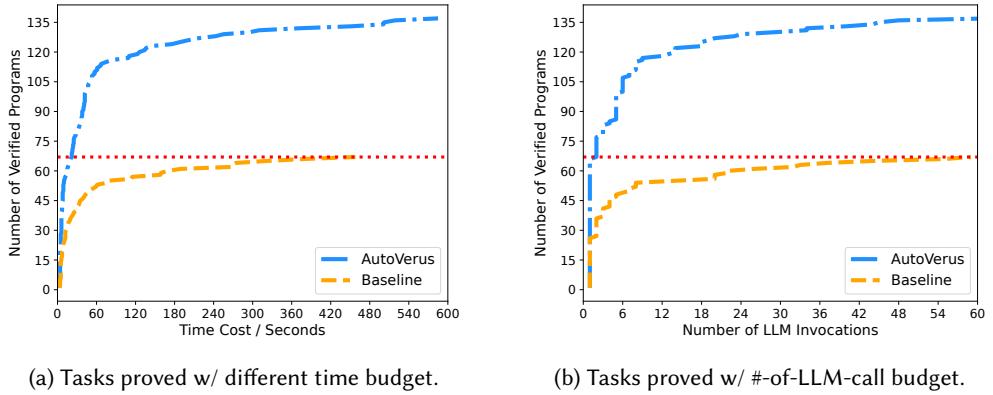


Fig. 8. Comparison with baseline in terms of time cost and LLM invocations.

Table 3. Statistics of AUTOVERUS repair agents.

Type of repair agents	# of uses	# of uses w/ successful repair	Repair success rate
AssertFail	310	145	46.8%
InvFailFront	131	84	64.1%
InvFailEnd	85	78	91.8%
ArithmetickFlow	73	31	42.5%
PostCondFail	26	18	69.2%
AssertFailProofFunc	16	11	68.8%
Default	8	6	75.0%
PreCondVecLen	8	4	50.0%
Total	657	377	57.4%

Furthermore, if we give each task only 30 seconds, AUTOVERUS can generate correct proof for 81 tasks, while the the baseline cannot even finish 40 tasks. Again, keep in mind that the baseline has the same temperature setting and 5-output per LLM-call as AUTOVERUS.

### 7.3 Detailed Results of AUTOVERUS Components

**7.3.1 The Effectiveness of Merging, Ranking, Filtering, and Houdini.** For 14 tasks, although none of the direct output of the Phase-1 agent is correct, merging some of them immediately provides correct proof just like the example in Figure 5.

During Phase-1, ranking LLM’s output based on Verus output has made a difference, as more than half of the top-ranked proof candidates, according to Verus verification results, are *not* the first output produced by the LLM, as shown in Figure 9. Interestingly, the fifth output is almost never the top-ranked candidate based on Verus, which happens to about 3% of the model invocation.

During the run of producing the results in Table 2, LLMs generated 2,637 proof candidates, of which as many as 326 (12.4%) are unsafe. Fortunately, they are filtered out by our Lynette tool.

Finally, the Houdini algorithm contributed to 16 successfully proved tasks across all phases, with 5 in Phase-1, 6 in Phase-2, and 5 in Phase-3. For each of these cases, the Houdini algorithm helped identify a correct proof from an incorrect one.

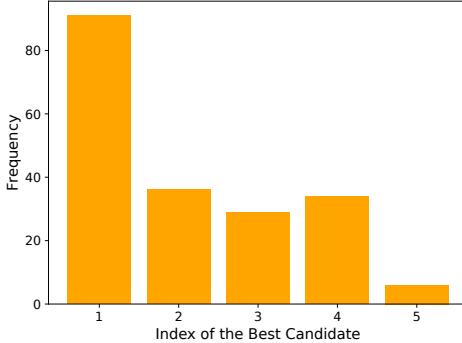


Fig. 9. Frequency of the  $k$ -th output of the LLM being the best-ranked candidate according to Verus in Phase-1 ( $k = 1 \dots 5$ ).

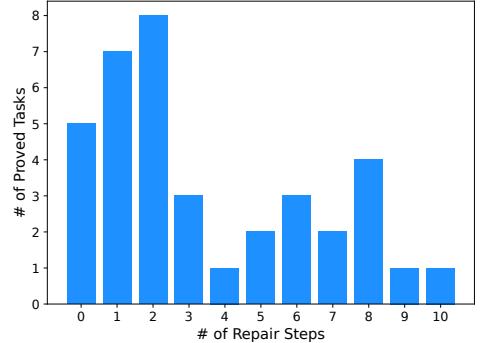


Fig. 10. Frequency of the number of debugging iterations needed for proved tasks during Phase-3.

**7.3.2 The Effectiveness of Agents.** For the 22 tasks that are not proved after Phase-1 but are successfully proved after Phase-2 (refinement) of AUTOVERUS, 6, 3, 3, and 10 tasks are proved immediately after the use of Constant-Propagation agent, Array-Length agent, Quantifier agent, and Conditional-Loop-Invariant agent, respectively.

For the 37 tasks that require Phase-3 of AUTOVERUS to prove, their average number of debugging iterations is 3.5 (median is 2). The exact distribution of debugging iterations is shown in Figure 10. Only 2 tasks require more than 8 iterations to be verified. Notably, 5 tasks are verified by adding `#[verifier::loop_isolation(false)]` to the best proof generated by Phase-2, and hence requires 0 iteration of LLM-based repair (as discussed in § 5).

To understand the contribution of individual repair agents, we analyzed their usage frequency and success rates, with a detailed breakdown shown in Table 3. Note, two repair agents listed in Figure 6 do not show up here: the “Mismatched-Type” repair agent operates deterministically by adding type casting without an LLM call and hence is not listed in this table; the pre-condition errors never occurred in our experiments and hence the “PreCondFail” repair agent was never used. As we can see, among all repair agents, the ones that repair Assertion Failure are used the most, for 310 times in total across 150 proof tasks, followed by the agent that repairs Loop Invariant Failed Before Loop (131 times) and at the End of the Loop (85 times). Due to the wide variety of assertion failures, the assertion failure agent only has a success rate of 46.8% (i.e., in 46.8% of times, the agent generates a repair that reduces the number of assertion failures). In comparison, all other agents, except for the Arithmetic Overflow repair agent, have a success rate of more than 50%.

Note that, only 8 out of 657 (1.2%) repair iterations required the default repair agent. This shows that our 9 error-specific repair agents offer good coverage of the most common verification errors.

## 7.4 Ablation Study

**7.4.1 Phase-1 and Phase-2 of AUTOVERUS.** As shown in Table 4, we conducted a number of ablation studies to understand different design choices for Phase-1 and Phase-2 of AUTOVERUS. For this study, we used two subsets of proof tasks from Verus-Bench. The *Inference Dataset* includes all the 78 tasks that can be correctly proved by AUTOVERUS using Phase-1 alone; the *Refinement Dataset* includes all the 22 tasks that require both Phase-1 and Phase-2 to be proved by AUTOVERUS.

Table 4. Ablation study: # of tasks proved under different variations of Phase-1 and Phase-2

	AUTOVERUS	Few-shot examples			Fusion of phases 1 and 2	Disable ranking
		default	W/o 1st	W/o 2nd	W/o 3rd	
Inference Dataset	78	67	69	48	75	71
Refinement Dataset	22	N/A	N/A	N/A	18	17

Table 5. Ablation study: AUTOVERUS with different LLMs and LLM temperatures.

	Default	AUTOVERUS with other LLMs			Temperature		
		GPT-4o, temp=1.0	GPT-4-turbo	GPT-3.5-turbo	Deepseek-R1	0.1	0.4
Sampled-dataset	26	26	18	23	21	24	25

**Few-shot examples.** As shown in Table 4, the number of tasks that can be proved drops if we remove any one of the three examples used by AUTOVERUS’s Phase-1 agent. The third example stands out as particularly influential, resulting in the most reduction in proved tasks when removed.

**Fusion of Phase-1 and Phase-2.** One may wonder the necessity of multiple separate agents in Phase-1 and Phase-2. To answer this question, we explored an alternative design in which all five agents in Phase-1 and Phase-2 are combined into one agent. This combined agent uses a prompt that includes the instructions from every individual agent in the form of “1. <Phase-1 Instructions>, 2. <Refinement 1 Instructions>, …”, and uses the same three few-shot examples used by the original Phase-1 agent. Under this alternative design, 7 tasks that originally can be proved now cannot. This includes 3 tasks that originally can be proved with just Phase-1. Notably, we maintain the same post-processing procedures, including filtering, ranking, merging, and the application of the Houdini algorithm, for the fused version.

**Ranking.** As also shown in the table, when we discard our ranking scheme, which is based on Verus results, and instead relies on LLMs to tell us the best output, AUTOVERUS verifies 12 fewer tasks. This demonstrates that the ranking mechanism plays a crucial role in prioritizing LLM-generated candidates effectively, leading to improved overall performance.

**7.4.2 Phase-3 of AUTOVERUS.** We have designed an alternative generic debugging agent. This agent is equipped with three few-shot examples that contain examples of asserts and proof functions, the same as what is used in our repair agents. This agent is programmed to take in the whole Verus error report, and can decide which error to fix and how to fix by itself. To compare this design with the current design of AUTOVERUS, we focus on those 37 tasks that can be correctly proved by AUTOVERUS through debugging. For each of these 37 tasks, we take the imperfect proof produced by the refinement phase of AUTOVERUS, which was used by AUTOVERUS’s debugging phase to get the correct proof. And, we change Verus’ loop-isolation configuration to the easy-to-prove mode, just like that in Phase-3, and feed these 37 imperfect proofs to this new debugging agent. Even with the same debugging budget (three attempts, each with up to 10 iterations), the new design only manages to produce a correct proof for 17 out of 37 tasks.

**7.4.3 LLM Choice in AUTOVERUS.** Due to resource constraints, instead of using the whole Verus-Bench, we conduct this part of the ablation study using a smaller dataset of 30 problems randomly sampled from Verus-Bench, preserving the original benchmark-source distribution (i.e., 2 from CloverBench, 8 Diffy, 15 MBPP, and 5 Misc).

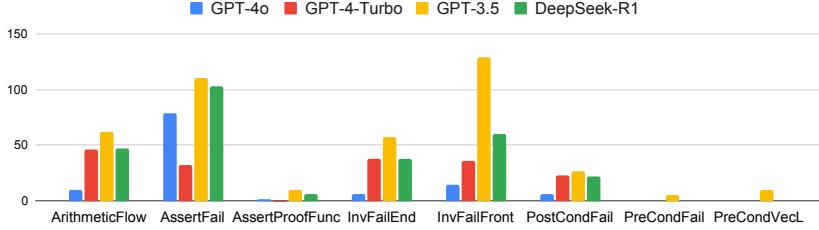


Fig. 11. The number of repair-agent invocations by AUTOVERUS with different LLMs on the 30 sampled tasks.

Our default setting (with GPT-4o, temperature 1.0) proves 26 tasks, the most among all settings (shown in Table 5). Note, the baseline scheme of repeatedly invoking GPT-4o can only prove 14 tasks. While trying three different GPT models, we observe that GPT-4-turbo delivers competitive performance compared to our default setting with GPT-4o, with both models power AUTOVERUS to prove 26 out of the 30 tasks. In contrast, GPT-3.5-turbo only allows AUTOVERUS to prove 18 tasks. However, upon further analysis, we find that GPT-4-turbo requires significantly more time in AUTOVERUS to figure out the correct proofx—an average of 229.8 seconds per task, compared to just 70.0 seconds for AUTOVERUS with GPT-4o. Furthermore, benefiting from the AUTOVERUS design, GPT-3.5-turbo outperforms the baseline of repeatedly invoking GPT-4o, proving 18 tasks.

We have also evaluated AUTOVERUS with the open-source model Deepseek-R1-32B, and have observed a slightly worse performance than AUTOVERUS with GPT-4 models (23 vs. 26). It shows that AUTOVERUS can work with different types of models. It also shows that code-proof remains a challenging task even for the new generation of reasoning models.

Figure 11 presents the number of repair-agent invocations by AUTOVERUS with different underlying models for these 30 sampled tasks. The figure omits the non-LLM “Mismatched-Type” agent and the generic repair agent (“repair\_default”), which was never triggered in this ablation experiment. Generally speaking, AUTOVERUS invokes a similar set of repair agents with different LLMs: the repair agent for assertion failures is most frequently used for all models except for GPT-4-Turbo; the popularity of repair agents for arithmetic under/overflow errors, two types of invariant failures, and post-condition failures are similar; the two types of pre-condition failures are rarely encountered during proof debugging across all models. Of course, the exact number of invocations are very different across different LLMs: AUTOVERUS has to make the most repair-agent invocations with GPT-3.5, probably because the raw output of GPT-3.5 has the worst quality among all models; on the other hand, AUTOVERUS needs the fewest repair-agent invocations with GPT-4o. Overall, this result shows that the design of AUTOVERUS is generally applicable for different LLM models, and can effectively help boost the performance of LLMs.

Finally, while setting the temperatures of GPT-4o to be 0.1, 0.4, 0.7, and 1.0, AUTOVERUS proves 21, 24, 25, and 26 tasks, respectively. This aligns with our design intuition that a higher temperature setting is beneficial in fostering LLM’s creativity and exploring a wider range of potential solutions, as long as human experts and discipline are also applied.

## 7.5 Looking into AUTOVERUS Proofs

**Failed cases.** AUTOVERUS failed to prove 13 tasks. We categorize AUTOVERUS’s struggle into three types. First, AUTOVERUS sometimes struggles at proving an expression will never encounter arithmetic overflow/underflow. For example, task MBPP-170 contains a loop that iterates through every element of an array to compute their sum. Verus requires a proof that the statement “sum

= sum + arr[index] as i128" inside the loop body would never encounter arithmetic overflow, where arr has the type of `Vec<i64>`, index has the type of `u64`, and sum has the type of `i128`. In the human-provided proof, this is achieved through a bound approximation: "`sum <= i64::MAX * index`" is added as a loop invariant, which nicely provides a bound for sum. In AUTOVERUS, unfortunately, LLM tried to directly prove "`sum + arr[index] <= i128::MAX`", and failed. In total, this type of challenge caused AUTOVERUS to fail 6 tasks.

Second, Verus has a vstd library about proof-specific data structures (e.g., Set, Seq, Map), where axioms and helper lemma functions are provided. Without full knowledge of the vstd, AUTOVERUS struggles at proof tasks that involve complicated usage of Verus data structures. For example, the post-conditions in the Misc-deduplicate task involve two vstd APIs, `Seq::no_duplicates()` and `Seq::to_set()`. Without knowing the exact definitions and axioms related to these two APIs, proof writing is difficult and AUTOVERUS failed. In total, this led to the failure in 5 tasks.

Finally, AUTOVERUS seemed lost and failed on 2 tasks with complicated logic. For example, the proof task of MBPP-755 involves a recursive spec-function and two complicated function post-conditions as shown below. The proof attempt of AUTOVERUS was quite far from success.

```
ensures
2  forall |k: int| 0 <= k < numbers.len() && k != indices.0 && numbers[indices.0] == min_spec(
    numbers@) ==> (#[trigger] numbers[k] >= numbers[indices.1]),
3  exists |k: int| 0 <= k < numbers.len() && k != indices.0
4    && (#[trigger] numbers[k] == numbers[indices.1])
```

**Successful cases.** For the 137 tasks that AUTOVERUS successfully proved, at a high level, the quality of AUTOVERUS proof is good and in line with human-written proof — after all, passing the Verus verification is a high bar for quality.

To quantitatively compare AUTOVERUS proof and human proof, we calculated the average number of loop invariants<sup>5</sup> across these tasks that AUTOVERUS succeeded on and also identified which of these invariants are necessary. While every loop invariant in a valid proof is correct, it may not be needed for the ultimate goal of a proof task (i.e., function post-conditions). To distinguish necessary invariants from unnecessary ones, we have designed a tool that automatically removes one loop invariant at a time. If Verus can still verify the program after the removal, this loop invariant is considered unnecessary and discarded. Otherwise, it is useful and restored. This process repeats until no loop invariant can be removed without causing the proof to fail. We performed the same analysis on human-written proofs for comparison, as shown in Figure 12.

As we can see in the blue bars of Figure 12, proofs generated by AUTOVERUS and those by humans contain a similar number of useful loop invariants. AUTOVERUS used slightly fewer useful loop invariants on Misc and MBPP tasks, while the trend is reversed on Diffy and Clover tasks.

As we can also see in the orange bars of Figure 12, a more significant difference between AUTOVERUS and human proofs lies in the number of unnecessary loop invariants, which are far

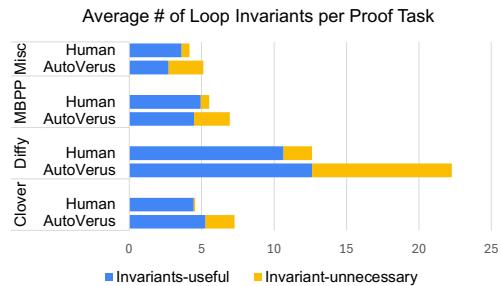


Fig. 12. Average number of necessary vs. unnecessary loop invariants from AUTOVERUS and human authors, evaluated on tasks successfully proven by AUTOVERUS.

<sup>5</sup>The proof annotations include not only loop invariants, but also `assert` statements and lemma functions. However, since `assert` and lemma functions are not needed by some benchmarks (e.g., all the Diffy benchmarks), we focus our study here on loop invariants.

more prevalent in AUTOVERUS proofs. For example, on the Diffy benchmark, AUTOVERUS generated an average of 22.2 loop invariants per task, 9.6 of which were unnecessary. In contrast, human proofs for the same tasks contained 12.6 total invariants, with only 2.0 being unnecessary. These unnecessary loop invariants exist even in human-written proofs, as it is challenging to identify the minimal set of evidence required by the verifier to finish a proof. For AUTOVERUS, this issue is exacerbated because LLM agents sometimes add identical loop invariants across multiple proof attempts. While this verbosity can harm code and proof readability, the negative impact is limited, as an automated process can effectively remove these unnecessary annotations.

## 8 Discussion & Threats to Validity

**General threats to validity.** There is an inherent randomness in AUTOVERUS and our evaluation given the probabilistic nature of LLM. We have made our best effort to make Verus-Bench an unbiased and meaningful benchmark suite. However, it is still just a collection of 150 tasks. It cannot represent many real-world proof tasks in large system code; it also cannot guarantee to represent all small-scale proof tasks. We have tried our best not to tune AUTOVERUS based on features of specific proof tasks in our Verus-Bench, and have our team that sets up the majority of the Verus-Bench not participating in the design of AUTOVERUS. However, the authors who participated in the design of AUTOVERUS are already familiar with some of the tasks in the Diffy benchmark and in the Misc set. Finally, our evaluation does not yet cover some of the newest Rust features supported by Verus, such as iterators.

**Data leakage.** Given the huge amount of training data used by modern LLMs, one may wonder whether LLMs already saw some of the proof tasks in its training data. We believe the impact of potential data leakage is minimal for two reasons:

First, the (un-)availability of online Verus resources. Among the four sources of Verus-Bench, Verus proofs for all 38 Diffy tasks and all 78 MBPP tasks were not publicly available throughout this research project. The 11 examples from CloverBench were released in the late spring of 2024, while we utilized earlier GPT-4o models (with 2024-02-15 and 2024-05-13 versions), predating the widespread public availability of CloverBench. The only concerns are three tasks in the Misc part of Verus-Bench, “fibonacci”, “tail\_triangle”, and “lemma\_len\_intersect”: these three tasks are created based on examples in the online Verus Tutorial [47]. We will elaborate on these three cases next.

Second, the poor performance of baseline LLM on “leaked” tasks. Without the support of AUTOVERUS, the baseline LLM (GPT-4o) is unable to generate proof for the three tasks that are already explained in the online Verus Tutorial. AUTOVERUS was able to conquer these three tasks. However, “fibonacci” remains one of the most difficult tasks even for AUTOVERUS as discussed in 4.3.3. Plus, the final proof generated by AUTOVERUS is *different* from the proof offered in Verus Tutorial. Comparing Figure 7d generated by AUTOVERUS and the listing below from the Verus Tutorial, we can see that although both functions prove the monotonicity of the Fibonacci sequence, they divide the proof into different sets of cases and also conduct the inductive proof differently. Based on these results, we suspect that existing Verus online resources are too few for LLMs to remember or understand.

```

1 proof fn lemma_fib_is_monotonic(i: nat, j: nat)
2   requires i <= j, ensures fib(i) <= fib(j), decreases j - i
3{
4   if j < 2 { } else if i == j { } else if i == j - 1 { } else {
5     lemma_fib_is_monotonic(i, (j - 1) as nat);
6     lemma_fib_is_monotonic(i, (j - 2) as nat);
7   }
8}

```

**How to scale up?** Moving forward, we expect three key challenges in applying AUTOVERUS to real-world project-level system verification. The first challenge comes from much more complicated

code dependencies in large systems. Verus allows code proof to be conducted at the function level, which helps address part of the scalability concern of AUTOVERUS. However, a function in a large system often calls other functions, both executable functions and specification functions in the world of Verus. Tackling such code dependency may require a hybrid approach of GenAI and traditional program analysis. The second challenge is related to specification. AUTOVERUS assumes that the specification (i.e., function pre-conditions and post-conditions) is already given. In a large system, deciding the specification of one function could be part of the proof task for another function. Furthermore, the quality of specification can greatly affect the proof task. The final challenge, and maybe the most difficult one, is to handle more complicated and diverse proof problems. In real-world verified systems [24, 26, 55], each function is often associated with a long list of pre-conditions and a long list of post-conditions, whose complexity is much higher than that in Verus-Bench. The proof skills used in those systems are also much more diverse – indeed, many proof annotations in those systems use advanced features of Verus [47] that are not covered by AUTOVERUS yet. We believe all three challenges will be interesting for future research to explore.

## 9 Related Work

**Verification for Rust.** As the popularity of Rust continues to grow, a variety of tools have been developed to verify Rust programs [1, 11, 18, 19, 22, 25, 31, 45]. Among these, Verus and Creusot stand out by allowing programmers to write specifications and proofs directly in Rust. Verus distinguishes itself further by integrating Rust’s borrow checker into its verification process, enforcing linearity and borrowing in specifications and proofs. This design simplifies reasoning about pointers and concurrency, making Verus particularly suited for systems verification. [43, 55].

**LLMs for proof synthesis.** LLMs have been revolutionizing various software engineering tasks [10, 46, 50–52], especially for code generation [2, 7], but the reliability of the generated code remains uncertain [29]. Formal proofs offer mathematical guarantees of correctness against program specifications, yet writing these proofs remains a labor-intensive, expertise-driven process. This challenge has sparked interest in using LLMs to automate formal proof synthesis [28].

Before LLMs, researchers explored using neural networks to synthesize proof for interactive theorem provers such as Coq [4], Isabelle/HOL [36], and LEAN [9], where proof to a theorem is completed by a sequence of proof steps, each being a proof tactic with its arguments [3, 14, 15, 21, 37, 39, 53]. Recent efforts like LeanDojo [54], LEGO-Prover [48], and Baldur [16] have used LLMs, and they all leverage the vast existing data of interactive theorem provers to create RAG databases or fine-tune models. However, they only work for interactive theorem provers and are primarily designed for maths rather than programs. Although it is theoretically possible to transpile programs in Rust to interactive theorem provers and then apply those tools, the cost is impractically high.

Recent work also tried using LLMs to generate proof/code in proof-oriented languages like Dafny [30, 33, 34, 42] and F\* [5]. As discussed in § 2, implementation and proof are intertwined in F\*, and hence the proof generation faces different challenges from AUTOVERUS. Recent work [5] leverages the large quantity of existing F\* code to create fine-tuned small models and a RAG database that is used to augment its proof synthesis prompt.

Dafny is also an SMT-based verifier like Verus. It proves programs written in the Dafny language. None of the LLM-for-Dafny work focuses on synthesizing proof for a given code implementation. Thus, their designs and evaluation results are very different from AUTOVERUS. Laurel [34] aims to synthesize asserts to verify an existing proof function. It uses a lemma-similarity score to help LLMs learn from similar assert in the same code base, with a success rate of over 50%. It could help the Assertion Failure repair agent in AUTOVERUS in the context of a mostly verified code base. Two recent projects [33, 42] aim to create a workflow from natural language description to verified Dafny programs. Since their focus was not on generating proof annotations for a given implementation,

they simply feed the LLM with one generic prompt. Given the difficulty of generating the whole program based on natural language, most of the resulting Dafny programs contain easy or no proof annotations, as discussed in § 6.1. DafnyBench [30] creates a dataset of more than 750 Dafny programs and compares the proof-synthesis capability across different LLMs. It feeds the LLM with the program under proof and one generic prompt. When the LLM’s initial answer is wrong, DafnyBench adds the whole error message into the prompt, similar to the alternative design of AUTOVERUS’s debugging agent studied in § 7.4. Note, 26.9% of the proof tasks in DafnyBench do not require any proof annotations. This type of tasks is not included in Verus-Bench.

Lemur [49] transforms program verification tasks into a series of deductive steps suggested by LLMs (in the form of program invariants) and subsequently validated by automated reasoners (a model checker in Lemur). The current Lemur tool cannot be directly applied to Verus (e.g., the type of program invariants and benchmark problems discussed there do not involve quantifiers or proof functions). However, the high-level idea could be beneficial for future GenAI-for-Verus.

Recently, Loopy uses LLM to generate loop invariants in integer C programs (*i.e.*, no arrays or pointers) [23]. It uses a long prompt to generate loop invariants, and uses Houdini algorithm to identify the correct subset of loop invariants. If Houdini fails, it feeds the whole error message to a long fixed repair prompt. AUTOVERUS shares the similarity with Loopy as both leveraging Houdini algorithm. Loopy only focuses on loop invariants that do *not* involve quantifiers, arrays, etc., while AUTOVERUS does not have these constraints for loop-invariant synthesis. More importantly, as discussed earlier, AUTOVERUS is much more than just synthesizing loop invariants. In fact, among the 150 Verus-Bench proof tasks, only the 38 Diffy tasks focus on loop invariants. The majority of the proof tasks in the other three sources (Clover, MBPP, Misc) cannot be proved by adding loop invariants alone—for example, 43 out of 78 MBPP tasks and 11 out of 23 Misc tasks *cannot* be proved using loop invariants alone. Due to the wide variety of proof annotations that AUTOVERUS needs to deal with (`assert`, proof functions, loop invariants, quantifiers, collection types, etc.), its design is completely different from Loopy.

## 10 Conclusions

In this paper, we explored using GenAI techniques to generate Verus proof for Rust code. By embracing the unique features of Verus, we have designed AUTOVERUS that uses human expert knowledge and Verus-based formal methods to assist LLMs in writing proof. AUTOVERUS has achieved good results on Verus-Bench, and provides a starting point for future research.

The LLM is quickly evolving and Verus is also evolving. It is possible that some of the designs in AUTOVERUS will need to change when the next big LLM is released, just like almost every AI-based technique these days. We have designed AUTOVERUS to contain composable parts to ease future extension and improvement. We also believe some of the key insights in AUTOVERUS will last for a long time, such as the power of the combined capability of human expertise, LLM creativity, and formal methods’ rigor. We also hope our exploration to accommodate various features of Verus will be an interesting case study for other AI-for-software efforts.

## Acknowledgment

We are grateful to the anonymous reviewers for their valuable comments and feedback.

## Data-Availability Statement

The code and artifact of AUTOVERUS are available at [microsoft/verus-proof-synthesis](https://microsoft/verus-proof-synthesis).

## References

- [1] Vytautas Arausaskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. <https://doi.org/10.1145/3360573>
- [2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). <https://arxiv.org/abs/2108.07732>
- [3] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOLiST: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 454–463. <http://proceedings.mlr.press/v97/bansal19a.html>
- [4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. 1999. The Coq proof assistant reference manual. *INRIA, version 6*, 11 (1999). <https://hal.science/inria-00069919>
- [5] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu K. Lahiri, and Nikhil Swamy. 2024. Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming. *CoRR* abs/2405.01787 (2024). <https://doi.org/10.48550/ARXIV.2405.01787>
- [6] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. 2021. Diffy: Inductive Reasoning of Array Programs Using Difference Invariants. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 911–935. [https://doi.org/10.1007/978-3-030-81688-9\\_42](https://doi.org/10.1007/978-3-030-81688-9_42)
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *CoRR* abs/2107.03374 (2021). <https://arxiv.org/abs/2107.03374>
- [8] 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, KISV 2023, Koblenz, Germany, 23 October 2023*. ACM, 9–17. <https://doi.org/10.1145/3625275.3625401>
- [9] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. [https://doi.org/10.1007/978-3-030-79876-5\\_37](https://doi.org/10.1007/978-3-030-79876-5_37)
- [10] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 423–435. <https://doi.org/10.1145/3597926.3598067>
- [11] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*. Springer, 90–105. [https://doi.org/10.1007/978-3-031-17244-1\\_6](https://doi.org/10.1007/978-3-031-17244-1_6)
- [12] ETAPS. 2024. 10th Competition on Software Verification (SV-COMP 2021). <https://sv-comp.sosy-lab.org/2021/>. [Online], [Accessed: 2024-09-01].
- [13] Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramananandro, and Nikhil Swamy. 2024. 3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers. *CoRR* abs/2404.10362 (2024). <https://doi.org/10.48550/ARXIV.2404.10362>
- [14] Emily First and Yury Brun. 2022. Diversity-Driven Automated Formal Verification. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1–13. <https://doi.org/10.1145/3510003.3510138>
- [15] Emily First, Yury Brun, and Arjun Guha. 2020. TacTok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 231:1–231:31. <https://doi.org/10.1145/3428299>
- [16] Emily First, Markus N. Rabe, Talia Ringer, and Yury 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, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 1229–1241. <https://doi.org/10.1145/3611643.3616243>
- [17] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2021)*, José Nuno Oliveira and Pamela Zave (Eds.). Springer, 500–517. [https://doi.org/10.1007/3-540-45251-6\\_29](https://doi.org/10.1007/3-540-45251-6_29)

- [18] Lennard Gähler, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. <https://doi.org/10.1145/3656422>
- [19] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. <https://doi.org/10.1145/3547647>
- [20] Rijul Jain, Shraddha Barke, Gabriel Ebner, Md Rakib Hossain Misu, Shan Lu, and Sarah Fakhouri. 2025. What’s in a Proof? Analyzing Expert Proof-Writing Processes in F\* and Verus. *arXiv preprint arXiv:2508.02733* (2025).
- [21] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. [http://papers.nips.cc/paper\\_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html)
- [22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. <https://doi.org/10.1145/3158154>
- [23] 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). <https://doi.org/10.48550/ARXIV.2311.07948>
- [24] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th Symposium on Operating Systems Principles, SOSP 2024, Hilton Austin, Texas, USA, November 4–6, 2024*. ACM. <https://sigops.org/s/conferences/sosp/2024/accepted.html>
- [25] 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. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. <https://doi.org/10.1145/3586037>
- [26] Hayley LeBlanc, Jacob R Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, and Vijay Chidambaram. 2025. PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection. *OSDI (to appear)* (2025).
- [27] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. [https://doi.org/10.1007/978-3-642-17511-4\\_20](https://doi.org/10.1007/978-3-642-17511-4_20)
- [28] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A Survey on Deep Learning for Theorem Proving. *CoRR* abs/2404.09939 (2024). <https://doi.org/10.48550/ARXIV.2404.09939>
- [29] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. [http://papers.nips.cc/paper\\_files/paper/2023/hash/43e9d647cccd3e4b7b5baab53f0368686-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647cccd3e4b7b5baab53f0368686-Abstract-Conference.html)
- [30] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. DafnyBench: A Benchmark for Formal Software Verification. *CoRR* abs/2406.08467 (2024). <https://doi.org/10.48550/ARXIV.2406.08467>
- [31] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 841–856. <https://doi.org/10.1145/3519939.3523704>
- [32] Microsoft. 2024. Storage systems with verified correctness properties. <https://github.com/microsoft/verified-storage>. [Online], [Accessed: 2024-09-01].
- [33] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE (2024), 812–835. <https://doi.org/10.1145/3643763>
- [34] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2024. Laurel: Generating Dafny Assertions Using Large Language Models. *CoRR* abs/2405.16792 (2024). <https://doi.org/10.48550/ARXIV.2405.16792>
- [35] Office of the National Cyber Director. 2024. BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE. <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>
- [36] Lawrence C. Paulson. 1994. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*. Lecture Notes in Computer Science, Vol. 828. Springer. <https://doi.org/10.1007/BF0030541>

- [37] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *CoRR* abs/2009.03393 (2020). <https://arxiv.org/abs/2009.03393>
- [38] MIT Technology Review. 2023. How Rust went from a side project to the world's most-loved programming language. <https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/>. [Online], [Accessed: 2024-09-01].
- [39] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*. ACM, 1–10. <https://doi.org/10.1145/3394450.3397466>
- [40] Stackoverflow. 2024. "2024 Developer Survey: AI tools at development process; Challenges with AI at work". <https://survey.stackoverflow.co/2024/ai/>. [Online], [Accessed: 2025-03-01].
- [41] Stackoverflow. 2024. "Rust continues to be the most-admired programming language". <https://survey.stackoverflow.co/2024/technology>. [Online], [Accessed: 2024-09-01].
- [42] Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification - First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14846)*. Springer, 134–155. [https://doi.org/10.1007/978-3-031-65112-0\\_7](https://doi.org/10.1007/978-3-031-65112-0_7)
- [43] 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 *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 649–666. <https://www.usenix.org/conference/osdi24/presentation/sun-xudong>
- [44] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoué, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. <https://doi.org/10.1145/2837614.2837655>
- [45] The Kani Team. 2024. How Open Source Projects are Using Kani to Write Better Software in Rust | AWS Open Source Blog. <https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/>. [Online], [Accessed: 2024-09-01].
- [46] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. (2024), 4173–4198. <https://aclanthology.org/2024.findings-acl.247>
- [47] Verus. [n. d.]. Verus Tutorial and Reference. <https://verus-lang.github.io/verus/guide/>.
- [48] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. <https://openreview.net/forum?id=3f5PAlef5B>
- [49] Haoze Wu, Clark W. Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. <https://openreview.net/forum?id=Q3YaCghZNt>
- [50] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023). <https://doi.org/10.48550/ARXIV.2304.00385>
- [51] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.
- [52] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 560–573.
- [53] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 6984–6994. <http://proceedings.mlr.press/v97/yang19a.html>
- [54] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animeshree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. [http://papers.nips.cc/paper\\_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets\\_and\\_Benchmarks.html](http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html)
- [55] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A Verified Security Module for Confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 599–614. <https://www.usenix.org/conference/osdi24/presentation/zhou>

[osdi24/presentation/zhou](#)

Received 2025-03-26; accepted 2025-08-12