ALPHAVERUS: BOOTSTRAPPING FORMALLY VERIFIED CODE GENERATION THROUGH SELF-IMPROVING TRANSLATION AND TREEFINEMENT

Pranjal Aggarwal, Sean Welleck
Carnegie Mellon University
{pranjala,wellecks}@cmu.edu

ABSTRACT

Automated code generation with large language models has gained significant traction, but there remains no guarantee of the correctness of generated code. We aim to use formal verification to provide mathematical guarantees that the generated code is correct. However, generating formally verified code with LLMs is hindered by the scarcity of training data and the complexity of formal proofs. To tackle this challenge, we introduce `AlphaVerus`, a self-improving framework that bootstraps formally verified code generation by iteratively translating programs from a higher-resource language and leveraging feedback from a verifier. `AlphaVerus` operates in three phases: exploration of candidate translations, Treefinement—a novel tree search algorithm for program refinement using verifier feedback, and filtering misaligned specifications and programs to prevent reward hacking. Through this iterative process, `AlphaVerus` enables `LLaMA-3.1-70B` model to generate verified code without human intervention or model finetuning. `AlphaVerus` shows an ability to generate formally verified solutions for HumanEval and MBPP, laying the groundwork for truly trustworthy code-generation agents.

1 INTRODUCTION

There has been an enormous effort to train code-generating large language models (LLMs) Chen et al. (2021); Austin et al. (2021); Li et al. (2023); Rozière et al. (2024); Team (2024), leading to LLM-powered agents that can perform tasks ranging from fixing bugs in software repositories to solving Olympiad-level algorithmic problems Jimenez et al. (2023); Li et al. (2022a). Despite these successes, multiple studies have identified disturbing mistakes in LLM-produced code, including subtle bugs and serious security vulnerabilities Hendler (2023); Pearce et al. (2021); Jesse et al. (2023); Zhong & Wang (2023); Perry et al. (2023); Elgedawy et al. (2024). Ultimately these mistakes stem from a fundamental property of LLMs: language models can generate any string of code, without regard to correctness. As a result, automatically checking the correctness of LLM-generated code is one of the grand challenges facing the research community.

The generated code must be correct for all inputs it may receive. However, today's code generation methods select or filter generations with imperfect correctness proxies, such as runtime testing or human inspection. Achieving perfect test coverage is typically infeasible Li et al. (2022b); Liu et al. (2023), and incomplete coverage leads to unreliable signals that models can exploit Pan et al. (2022); Liu et al. (2023); Denison et al. (2024). Relying on human review is problematic as it scales poorly and humans often struggle to verify LLM-generated code's correctness Perry et al. (2023). Consequently, the difficulty of trusting generated code reduces potential productivity gains from LLMs and can introduce unexpected vulnerabilities or unreliable signals for model improvement.

In contrast, generating code in a *verification-aware programming language* such as Dafny Leino (2010), F* Swamy et al. (2016), or Verus Lattuada et al. (2023) offers a promising approach to addressing these challenges by providing mathematical guarantees that a program obeys a specification for all inputs. In this paradigm, code is paired with a specification and proof written in a specialized language, and a mechanical verifier checks if the code meets the specification. Doing so could improve the trustworthiness of the generated code: if the verifier passes, the LLM's generated program

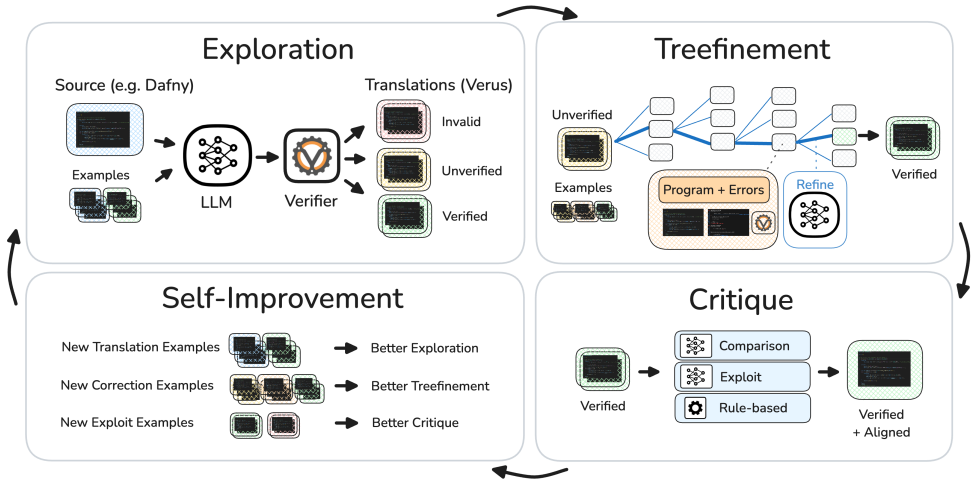

Figure 1: Overview of AlphaVerus, a self-improving framework for generating formally verified code. Each iteration consists of three key steps: (1) *Exploration* translates programs from a source language to Verus by sampling multiple trajectories and selecting partially correct ones using verifier feedback, (2) *Treefinement* iteratively fixes errors guided by verifier feedback and tree search, and (3) *Critique* validates and filters out underspecified or incorrect translations. The framework bootstraps new exemplars after each iteration to continuously improve performance without human intervention.

is mathematically guaranteed to meet the specification. However, writing formal specifications and proofs introduces layers of complexity. Furthermore, although LLMs have demonstrated success in theorem proving in mathematical domains Lu et al. (2023); Li et al. (2024), their capability to generate verified code for even basic algorithms is limited Sun et al. (2023); Lohn & Welleck (2024).

A significant barrier to automatically generating real-world, formally verified code is the scarcity of training data. In particular, verification-aware research languages have a rich history (e.g., Dafny Leino (2010), F* Swamy et al. (2016)), yet verifying real-world code in mainstream languages remains nascent. For example, Verus Lattuada et al. (2023)–a verification language for the very popular language Rust–has fewer than 10 public repositories, despite Rust itself having millions of code examples. Hence, enabling formally verified code generation in a mainstream language such as Rust faces a *bootstrapping problem*–how do we create an initial model that can generate even relatively simple verified programs, given the absence of training data?

We propose AlphaVerus, a framework for bootstrapping a formally verified code generation model by iteratively translating programs from a resource-rich domain and self-improving using feedback from the verifier. As illustrated in Figure 1, each iteration of AlphaVerus has three phases. First, the *exploration* phase generates candidate programs by translating from a source language (such as Dafny) to the target language (here, Verus) by generating multiple candidates and saving partially and completely verified attempts. Second, *Treefinement* refines the imperfect candidates through a novel tree search over the space of output programs using feedback from the verifier, saving the final verified program, along with its ancestors to serve as error correction examples. We show that Treefinement leads to substantial gains over vanilla refinement strategies that resemble those used in concurrent work Yang et al. (2024); Chen et al. (2024). Third, *critique models* detect misaligned translations and specifications–the one part of the pipeline that lacks formal guarantees. Crucially, this alleviates *reward hacking*, in which models learn to game the system by generating trivial or incomplete specifications, or even by identifying verifier limitations that cause trivial programs to pass the verifier. While previous work has investigated methods that rely on test cases Sun et al. (2023), our critique models address the challenging problems of automated specification generation and validation without relying on any unit test cases.

Each iteration of AlphaVerus collects new exemplars that improve the models in each phase, creating a cycle of improvement. Thus, unlike recent work that relies on human experts to write correction prompts Yang et al. (2024), our method requires no human intervention and automatically learns to generate better code. Moreover, the system operates using a single language model (e.g.,

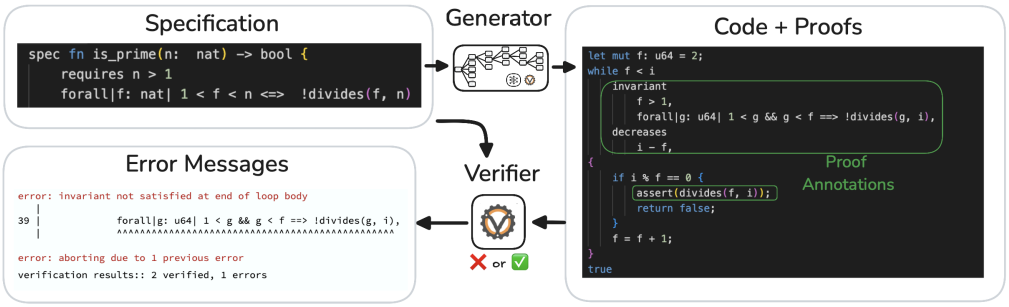

Figure 2: **Example of formally verified code generation.** Given a specification, AlphaVerus generates the corresponding code and proof. The verifier checks the proof and provides either verification success or detailed error messages.

Llama 70b), without the need for the expensive GPT-4 initialization used in concurrent work Chen et al. (2024). Finally, the collected exemplars can be used to improve the verified code generation performance of any model without any finetuning.

To demonstrate AlphaVerus, we consider Dafny Leino (2010) programs as the source domain, since the Dafny language has been around for over a decade and has accumulated a reasonable amount of code. We run AlphaVerus to automatically collect the DAFNY2VERUS-COLLECTION, a dataset of trajectories containing translated programs, error corrections, and critique examples based on the source dataset DafnyBench Loughridge et al. (2024)–a dataset of 562 programs of varying difficulty. Finally, we evaluate the AlphaVerus pipeline by using the resulting data as few-shot exemplars for the downstream task of *formally verified code generation*: generating complete, formally verified implementations—including both algorithmic code and proof annotations—given human-written specifications Formally verified code generation is a significant step over concurrent work that focuses solely on the simplified, artificial setting of generating proof annotations for correct pre-written code Yang et al. (2024); Chen et al. (2024). We show AlphaVerus enables Llama-70B to successfully generate verified solutions to 33% of HumanEval-Verified The HumanEval-Verus Contributors (2024), outperforming GPT-4o-based methods. Furthermore, through ablations, we establish the necessity of each component in AlphaVerus.

In summary, our contributions are five-fold: (1) We propose AlphaVerus, a self-improving framework for generating formally verified code; (2) We present a novel combination of tree search and refinement that improves over time; (3) We introduce a critique phase, the first neural method that improves specification quality without test cases; (4) We introduce a dataset with formally verified Verus programs and error pairs; and (5) We demonstrate our approach's effectiveness, evaluating its code generation abilities and component impact. Notably, AlphaVerus is the first to achieve non-zero performance on a verified version of HumanEval Chen et al. (2021), thus establishing a starting point for code generation models that generate increasingly complex—yet trustworthy—code.

2 FORMALLY VERIFIED CODE GENERATION

Our goal is to develop a model that generates formally verified code in a real-world programming language, which we refer to as *formally verified code generation*. Next, we provide background and then introduce AlphaVerus.

Formal verification of code. Formal verification ensures a program adheres to its specification. Formally verified code consists of: (1) specifications y_S; (2) implementation y_I; (3) proof y_P. Verifier $v(y_S, y_I, y_P) \to \{0, 1\}$ checks if implementation meets specifications. A verifier $v(y_S, y_I, y_P) \to \{0, 1\}$ uses the proof to statically check that the implementation meets the specification for all possible inputs, returning 1 if the program is correct with respect to y_s. Upon failure, the verifier additionally returns a set of messages $\{m_1, \ldots, m_M\}$ containing the number of verified statements, the number of errors, and localized error messages (e.g., see Figure 2).

Misaligned specs and implementations. Specifications aren't verified; they must reflect desired behavior. We use the term *misaligned* to refer to situations in which the specification does not reflect

the desired input-output behavior. This includes misalignments between the specification and the developer's intent or the implementation, which can occur due to reasons such as use of language features that cause programs to pass the verifier trivially (e.g., using "`assume(false)`").

Formally verified code generation. We aim to generate verified code from a specification. Specifically, $(y_I, y_P) \sim G(y_S; c, \theta)$, where $G(\cdot)$ is a generation algorithm such as sampling from a language model with parameters θ, and the model generates both an implementation y_I and proofs y_P given a specification y_S and any additional context c. The goal is for the resulting code to verify, i.e., $v(y_S, y_I, y_P) = 1$. We refer readers to Appendix B for a more detailed related work.

Bootstrapping formally verified code generation. A practical goal is to perform formally verified code generation in a mainstream language, such as Rust code verified with the Verus verifier Lattuada et al. (2023). However, doing so raises a technical challenge: it is infeasible to train a model on (y_S, y_I, y_P) examples since such examples do not exist. We refer to this as a *bootstrapping problem*, since we need to create an initial generation model (that we may subsequently improve) without any training data. Next, we describe `AlphaVerus`, a framework for bootstrapping a verified code generation model by translating from a more resource-rich language.

3 ALPHAVERUS

To enable verified code generation in the absence of training data in our target language (Verus), we propose to iteratively translate programs from a higher-resource domain into Verus. Each iteration collects data by exploring candidate translations, refining them with a novel tree search, and then filtering out misaligned programs. Finally, we use the data to enable a verified code generation model (via few-shot learning), and evaluate the model plus the tree search on the downstream task of verified code generation: generating verified code and proofs given a held-out test specification.

3.1 TRANSLATION

`AlphaVerus` translates programs using a three-stage pipeline consisting of *exploration*, *refinement*, and *critique*. The exploration stage translates source programs into candidate Verus programs. The refinement stage repairs the programs using a novel tree search over program refinements. The critique stage uses a suite of models to discard flawed specifications and implementations that could degrade future iterations. The pipeline iterates, creating a self-reinforcing cycle where verified programs and refinement trajectories improve the models' capabilities, enabling translation of increasingly complex programs. The result is a growing synthetic dataset of progressively more complex and reliable Verus programs. The complete algorithm is listed in Algorithm 1 and visualized in Figure 1.

Exploration. Given a source program x (e.g., a Dafny implementation, specification, and proofs), exploration uses a model to generate candidate target (i.e., Verus) programs:

$$\{y_1, \ldots, y_k\} \sim G_{explore}\left(x; D^{(i)}_{x \to y}\right), \tag{1}$$

where G is a generation algorithm (e.g., LLM sampling) that is given the source and a set of (source, target) examples $D^{(i)}_{x \to y}$. Initially, $D^{(0)}_{x \to y}$ has a few hand-written examples.

Any generated (source, verified program) pairs are placed in a candidate set, C, that will be passed to the filtering stage. If no candidates verify for source x, candidates that are syntactically correct proceed to refinement. Intuitively, this stage serves as initial "exploration", in that it generates a set of candidates that may eventually be refined and filtered into verified programs in the later stages. Unlike other methods of bootstrapping Zelikman et al. (2022); Lin et al. (2024) that discard anything but correct solutions, we use both syntactically correct programs and fully verified programs for further improvement, expanding the learning signal.

Refinement with Treefinement. Having a verifier opens the possibility of refining candidate programs into verified ones by providing detailed feedback, including unverified functions and specific errors like overflows, unsatisfied conditions, and syntactic mistakes (e.g., Figure 2). While human programmers often use such feedback for iterative corrections, naively providing LLMs with incorrect solutions and feedback often fails to produce improvements. Our key insight is that verifier feedback induces an implicit ordering of solutions based on verified functions and error severity. This

ordering lets us extend common refinement techniques by framing refinement as a tree search over the space of refined programs, which we call *Treefinement*.

Specifically, the refinement stage takes syntactically correct but unverified candidate translations $\{y_1, \ldots, y_{k'}\}$ and performs a tree search to discover verified programs. Each node in the tree contains an imperfect program and its associated errors, $(y, e(y))$. Nodes are expanded by invoking a refinement model:

$$\{y'_1, \ldots, y'_k\} \sim G_{refine}\left(y, e(y); D^{(i)}_{y \to y'}\right), \tag{2}$$

where $D^{(i)}_{y \to y'}$ is a set of (program, error, correct program) examples, initially containing a few hand-written examples.

Given a node scoring function $v(y) \to \mathbb{R}$ that is used to prioritize nodes, we can search over the space of program refinements with a tree search algorithm that selects and expands nodes, such as breadth-first or depth-first search. We develop a symbolic scoring function based on the number of (un)verified functions, errors, and warnings:

$$s(y) = \frac{n_{\text{ver}}(y) - \alpha n_{\text{err}}(y) - \beta n_{\text{warn}}(y)}{n_{\text{ver}}(y) + n_{\text{unver}}(y)}$$

where $n_{\text{ver}}(y)$ is the number of verified functions in y, $n_{\text{err}}(y)$ and $n_{\text{warn}}(y)$ are the counts of errors and warnings from the verifier for the node's program y. α and β are hyperparameters controlling the penalties for errors and warnings, respectively. Intuitively, programs that are closer to a verified program have higher scores, with proximity determined by the proportion of verified functions, resolved errors, and resolved warnings. Upon generating a verified program, the program's search trajectory is added to a candidate set C_τ, and the new (source, program) pair to the candidate set C that is passed to the critique stage.

Treefinement extends two kinds of prior methods into a new search over program refinements. First, refining LLM outputs is a common technique Madaan et al. (2023); Kamoi et al. (2024), but not within a tree search. On the other hand, tree search developed in step-by-step mathematical problem solving involves appending solution steps rather than refining a full program Wu et al. (2024a). Our approach specifically addresses the non-local nature of error fixes. Although Treefinement can use any tree search algorithm, we use REBASE (REward BAlanced SEarch) Wu et al. (2024b). REBASE allocates an exploration budget by sampling nodes from a distribution determined by the node scores at the current depth, providing an effective balance of exploration and exploitation. The search continues until it finds a verified program or reaches a maximum depth.

Critique. Synthesized specifications are the one part of the translation pipeline that lacks formal guarantees, which can result in a mismatch between the intended and actual functionality of generated programs. Furthermore, in a few cases, there can be a mismatch between the specification's intent and the program's implementation, since Verus has features that can result in trivial programs passing the verifier (e.g., `assume(false)`). These can lead to reward hacking, causing a snowballing effect when used as exemplars in future iterations. Hence, we propose a three-part approach for filtering out such misaligned programs: a *rule-based model*, a *comparison model*, and a *exploit model*.

The rule-based model receives a generated program y, and detects if y uses a Verus feature which leads to a trivial program. Since there are a relatively small number of such features, and these features can be detected through string matching, it suffices to use a list of hand-coded filters. This includes checking for `assume(false)`, "`#[verifier::external]`", and trivial preconditions.

The comparison model $f(x, y)$ receives a source input x and a program candidate y, and evaluates whether the specifications and algorithms used in the candidate match those from the source in intent and structure. In practice, we prompt a model to generate multiple evaluation sequences and reject an output if at least r sequences indicate rejection.

The exploit model is an adversarial approach that leverages the feedback from Verus. We use a generator prompted to generate simple and often trivial solutions–such as returning an empty array–that satisfy the specifications, i.e., $(y_I, y_P) \sim G_{\text{exploit}}\left(y_S; D^{(i)}_{\text{exploit}}\right)$, where y_S is a generated specification, y_I, y_P is an implementation and proof, and $D^{(i)}_{\text{exploit}}$ contains (specification, implementation+proofs) examples. If such simple solutions pass verification, it indicates that the specification is flawed, and the corresponding translation is discarded. This often includes subtle forms of misspecification.

Self-improvement. Finally, the newly generated programs and a subset of the error trajectories are added to data that is used by the translator, refinement, and critique models in the next iteration. In this sense, the models "self-improve" given access to the Verus environment, so long as the generated examples are useful exemplars. Formally, for exploration, we create a new pool of examples, $D_{x\to y}^{(i+1)} = D_{x\to y}^{(i)} \cup \tilde{D}_{x\to y}^{(i+1)}$, where $\tilde{D}_{x\to y}^{(i+1)}$ consists of the (source, program) candidates C that were collected during exploration and refinement, and that additionally pass the critique stage.

For refinement, we create a new pool of examples using the successful trajectories C_τ collected during refinement. Namely, we keep those trajectories whose final program passes the critique stage, and pair each intermediate program y and its errors with the final program y', i.e.,

$$\tilde{D}_{y\to y'}^{(i+1)} = \{(y, e(y), y') \mid y \text{ is an ancestor of } y'\}, \tag{3}$$

and set $D_{y\to y'}^{(i+1)} = D_{y\to y'}^{(i)} \cup \tilde{D}_{y\to y'}^{(i+1)}$.

Similarly, for the exploit model, we add (specification, program) exploits that pass the verifier into $D_{\text{exploit}}^{(i+1)}$ to be used by the exploit model in the next iteration. Table 3 summarizes the components, feedback sources, models, and generated synthetic data at each stage.

We employ a stochastic few-shot sampling approach to use synthetic data as in-context exemplars. In each generator call, k examples are randomly from its respective data pool. This reduces computational costs associated with fine-tuning large models, and as our results demonstrate that this enables other models to leverage the data pool and improve performance without training. Nevertheless, fine-tuning models and developing learning objectives remain interesting future directions.

Source domain: Dafny. As our initial source domain, we consider Dafny–a language that follows a similar paradigm to Verus and has been in use for over a decade, resulting in a larger set of available data. Specifically, we use the `DafnyBench` dataset, consisting of 562 programs. Translating Dafny programs to Verus presents several challenges due to two major differences: 1. *Language Constructs:* Significant differences exist in supported features, data types, and the design of the underlying verifier, rendering direct translations infeasible. 2. *Proof Requirements:* Verus imposes more rigorous proof obligations, such as overflow checks, making proofs harder to verify.

3.2 Downstream Evaluation

After generating high-quality synthetic data in the form of formally verified Verus programs and error-feedback-correction triples, we use the data to enable a model that performs formally verified code generation. Unlike prior work that requires LLMs to fill proof annotations in existing code and specifications Loughridge et al. (2024); Yang et al. (2024); Chen et al. (2024), we evaluate our models on more challenging task of generating both the code and the proofs given only the specifications.

We use a two-part approach consisting of *exploration* and *Treefinement*. During exploration, given a specification y_s, we generate k candidate programs $\{y^{(1)}, \ldots, y^{(k)}\}$. If any candidate passes verification, we consider the task solved. Otherwise, we initialize Treefinement with the candidates and run it until we obtain a verified solution or reach a maximum number of iterations. This can be seen as a generator that uses the collected data as a source of few-shot exemplars, $(y_I, y_P) \sim G(y_S; D_y, D_{y\to y'})$, which means generating an implementation and proofs using a language model prompted with a subset of the collected verified programs D_y and a test specification y_S, followed by Treefinement with the collected refinement examples $D_{y\to y'}$.

4 Experimental Setup

Generators. We use `LLaMA-3.1-70B` for translation experiments and additionally evaluate `LLaMA-3.1-8B`, `Qwen-32B`, and `GPT-4o` for downstream tasks. The exploration phase uses $k = 256$ samples, while tree search uses breadth 32 and maximum depth 8. **Translation.** We use `DafnyBench` consisting of 562 programs as our source domain D_{src} for our translation experiments. The exploration model $G_{explore}$ is initialized using a Verus syntax file and 5 examples from the Verus repository. **Downstream Evaluation.** We evaluate formally verified code generation, where models must generate both an implementation and proof annotations given a specification. We

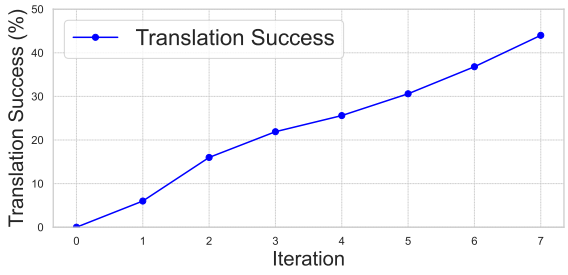

Figure 3: **Programs translated over iterations**. The translation success rate consistently improves over iterations.

Method	HumanEval	MBPP
Baselines		
GPT-4o	27.1%	35.9%
Llama 3.1 70B	11.8%	26.9%
Ablations (Treefinement Variants)		
Single-Turn Linear Self-Refine	29.4%	61.5%
Multi-Turn Linear Self-Refine	29.4%	62.8%
Best-First Search	28.2%	61.5%
AlphaVerus (Llama 3.1 70B)		
Exploration	27.1%	59.1%
+ Treefinement (Rebase)	**32.9%**	**65.7%**

Table 1: **Verified code generation performance** on the HumanEval and MBPP benchmarks (pass@256).

measure Pass@K, where success requires at least one correct solution of the K generated programs. **Datasets.** We evaluate on verified versions of the MBPP and HumanEval datasets. In particular, MBPP-verified is sourced from Yang et al. (2024); Misu et al. (2024) and contains 78 programs from the original MBPP dataset Austin et al. (2021). HumanEval-Verus is sourced from a concurrent open-source effort The HumanEval-Verus Contributors (2024) to translate existing HumanEval programs to Verus. For brevity, we refer to HumanEval and MBPP as their respective verified versions throughout this paper. **Baselines.** Our primary evaluation is performed on verified code generation. Since no existing baselines exist for the task, we use few-shot variants (Listing C) of base models. We tried our best to adapt AutoVerus Yang et al. (2024) to verified code generation, but due to the complexity of its hand-written prompts, we were not able to achieve non-trivial performance. Hence, we compare AlphaVerus on the MBPP *proof annotation task* against SAFE++ Chen et al. (2024) and AutoVerus Yang et al. (2024). We refer readers to Appendix A.2 for more details.

5 Results and Analysis

AlphaVerus translation success monotonically increases. Figure 3 shows the number of successful translations over each iteration. We see a steady increase in the number of translations as the iterations increase. The results indicate that AlphaVerus learns to translate and generate more complex programs over iterations. Altogether, AlphaVerus translates around 45% of DafnyBench into Verus programs that are verified by Verus and aligned according to the critique models. Listings 2, 3, and 4 in the Appendix show example translations. The generated exemplars during the translation process are collected into Dafny2Verus-Collection, totaling 247 translated programs, 102 error trajectories, and 579 exploit pairs. These exemplars are used for downstream tasks.

AlphaVerus enables verified code generation. Table 1 shows the verified code generation performance for the AlphaVerus model obtained from the final translation iteration. AlphaVerus leads to a substantial increase over its underlying Llama 3.1 70B model and a prompted GPT-4o model. Moreover, Treefinement leads to an additional increase in performance over the exploration stage. Listings 1, 5, and 6 show example generations. Next, we analyze the impact of the various components in AlphaVerus.

Treefinement leads to a jump in performance. We evaluate the effectiveness of tree search compared to further scaling the parallel sampling (exploration) budget without refinement. Figure 4 shows the percentage of solved problems versus the generation budget for both approaches. Treefinement leads to a substantial jump in performance over exploration. Notably, exploration plateaus while tree search continues improving as the generation budget is increased.

Critique is crucial for preventing reward hacking. Without the critique phase, our analysis of 100 DafnyBench examples reveals the model learns to game the verification system by using `assume(false)` statements, leading to trivially verified but incorrect implementations. We observe a snowballing effect where this behavior spreads across all programs (see Figure 5). While such cases can be disallowed as done by our rule-based critic model, we find more complicated reward hacking instances, such as incomplete specifications and degenerate translations (detailed in Figure 6). The results shows the need for our 3-model critique phase for preventing reward hacking.

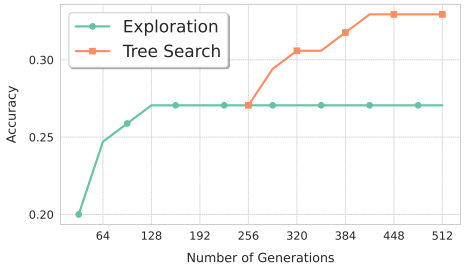

Figure 4: Treefinement vs. exploration (HumanEval). Treefinement leads to a jump in performance that cannot be obtained by additional parallel sampling (exploration).

	HumanEval
Llama 8B - Few Shot	11.8%
+ DAFNY2VERUS-COLLECTION	**18.8%**
Qwen-32B - Few Shot	14.1%
+ DAFNY2VERUS-COLLECTION	**27.1%**
GPT-4o - Few Shot	27.1%
+ DAFNY2VERUS-COLLECTION	**37.7%**

Table 2: Transfer of DAFNY2VERUS-COLLECTION to other language models without finetuning. All models show significant improvements over their few-shot variants.

Treefinement outperforms linear refinement. We compare Treefinement against standard refinement that refines linearly, either by performing one step of refinement across multiple parallel branches or several steps across branches. Using equivalent generation budgets, we adjust the breadth and depth parameters accordingly. We also evaluate the best-first search as a baseline. As seen in Figure 1, all methods improve upon initial exploration, demonstrating Treefinement's compatibility with various search algorithms, and tree-search based refinement outperforms linear refinement. For the tree search, using REBASE outperforms the best-first search. Also note that the linear refinement variants are special cases of REBASE ($depth = 1$ with large breadth, and $temperature = \infty$).

`AlphaVerus` exemplars transfer to other models. A key advantage of `AlphaVerus` is its ability to transfer learned exemplars without model weight updates. Concretely, we use the exemplars collected during `AlphaVerus`'s translation phase, which used Llama 3.1 70B (i.e., the DAFNY2VERUS-COLLECTION), to enable verified code generation on various models using the same few-shot prompting strategy outlined in §3.2. Table 2 shows successful transfer to both smaller and larger models, yielding significant improvements in verified code generation. Notably, we set a new state-of-the-art on both `HumanEval`, using `GPT-4o` but without finetuning.

`AlphaVerus` enables strong proof annotation. Unlike prior works focused solely on proof annotation (generating proofs for correct code), our method addresses the more challenging task of generating both code and proofs. Despite this, We outperform SAFE Chen et al. (2024) and AutoVerus Yang et al. (2024) by 17% and 10% respectively on proof annotation (see Table 4). This is remarkable as AutoVerus was specifically engineered for this task, yet we achieved superior results using only 562 Dafny programs and an open 70B model, compared to SAFE's extensive GPT-4 invocations and training on thousands of programs. These results highlight the effectiveness, flexibility, and data efficiency of `AlphaVerus`. We refer readers to Appendix D.1 for more details.

Other experiments. Our analysis of model learning progression (Figure 7) shows that `AlphaVerus` systematically advances from basic syntax to complex array-related concepts across iterations. Example translations (Listing 1–2–4) demonstrate `AlphaVerus`'s capability to handle complex programs with multiple specifications and proof annotations. Finally, our scaling analysis (Figure 8) reveals that `LLaMA-3.1-8B` is more cost-effective at lower budgets while `LLaMA-3.1-70B` achieves better asymptotic performance.

6 CONCLUSION

We introduced `AlphaVerus`, a novel self-improving framework for generating formally verified code in mainstream programming languages. By leveraging iterative translation from a higher-resource language (Dafny) to Verus and utilizing verifier feedback through our Exploration, Treefinement, and Critique stages, `AlphaVerus` overcomes the challenges of scarce training data, reward hacking and the complexity of formal proofs. Our approach operates without human intervention, hand-engineered prompts, or extensive computational resources, yet achieves significant performance improvements on verified code generation. We also contribute a new dataset of formally verified Verus programs, providing valuable resources for future research. `AlphaVerus` opens up new avenues for grounding code generation and developing trustworthy AI-assisted programming tools.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Yiming Yang, and Mausam. Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with llms, 2023. URL https://arxiv.org/abs/2305.11860.

Afra Feyza Akyürek, Ekin Akyürek, Aman Madaan, Ashwin Kalyan, Peter Clark, Derry Wijaya, and Niket Tandon. Rl4f: Generating natural language feedback with reinforcement learning for repairing model outputs. arXiv preprint arXiv:2305.08844, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. https://arxiv.org/abs/2108.07732, 2021.

Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for smt-assisted proof-oriented programming. https://arxiv.org/abs/2405.01787, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. https://arxiv.org/abs/2107.03374, 2021.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong Zhou. Automated proof generation for Rust code via self-evolution, 2024. URL https://arxiv.org/abs/2410.15756.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128, 2023.

I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios. arXiv preprint arXiv:2307.13528, 2023.

Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr/, 2020.

Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, Buck Shlegeris, Samuel R. Bowman, Ethan Perez, and Evan Hubinger. Sycophancy to subterfuge: Investigating reward-tampering in large language models. https://arxiv.org/abs/2406.10162, 2024.

Ran Elgedawy, John Sadik, Senjuti Dutta, Anuj Gautam, Konstantinos Georgiou, Farzin Gholamrezae, Fujiao Ji, Kyungchan Lim, Qian Liu, and Scott Ruoti. Occasionally secure: A comparative analysis of code generation assistants. https://arxiv.org/abs/2402.00689, 2024.

Emily First, Yuriy Brun, and Arjun Guha. Tactok: semantics-aware proof synthesis. Proc. ACM Program. Lang., 4(OOPSLA), Nov 2020. doi: 10.1145/3428299. URL https://doi.org/10.1145/3428299.

Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, pp. 1229–1241, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616243. URL https://doi.org/10.1145/3611643.3616243.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. Critic: Large language models can self-correct with tool-interactive critiquing, 2024. URL https://arxiv.org/abs/2305.11738.

James Hendler. Understanding the limits of AI coding. Science, 379(6632):548–548, 2023.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners, 2024. URL https://arxiv.org/abs/2402.06457.

Isabelle. Isabelle. https://isabelle.in.tum.de/.

Kevin Jesse, Toufique Ahmed, Prem Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 563–575, 2023. URL https://api.semanticscholar.org/CorpusID:257636802.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world GitHub issues? ArXiv, abs/2310.06770, 2023. URL https://api.semanticscholar.org/CorpusID:263829697.

Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olšák. Reinforcement learning of theorem proving. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, pp. 8836–8847, Red Hook, NY, USA, 2018. Curran Associates Inc.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. Transactions of the Association for Computational Linguistics, 12:1417–1440, 2024. doi: 10.1162/tacl_a_00713. URL https://aclanthology.org/2024.tacl-1.78.

P. Langley. Crafting papers on machine learning. In Pat Langley (ed.), Proceedings of the 17th International Conference on Machine Learning (ICML 2000), pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. Proc. ACM Program. Lang., 7(OOPSLA1), April 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

Lean FRO. Lean theorem prover. https://leanprover.github.io/.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 2010.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! https://arxiv.org/abs/2305.06161, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven

Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. Science, 378:1092 – 1097, 2022a. URL https://api.semanticscholar.org/CorpusID:246527904.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. Science, 378(6624):1092–1097, 2022b. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. In First Conference on Language Modeling, 2024. URL https://openreview.net/forum?id=zlw6AHwukB.

Haohan Lin, Zhiqing Sun, Yiming Yang, and Sean Welleck. Lean-star: Learning to interleave thinking and proving. https://arxiv.org/abs/2407.10040, 2024.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In Thirty-seventh Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Evan Lohn and Sean Welleck. minicodeprops: a minimal benchmark for proving code properties. https://arxiv.org/abs/2406.11915, 2024.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification, 2024. URL https://arxiv.org/abs/2406.08467.

Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. A survey of deep learning for mathematical reasoning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 14605–14631, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.817. URL https://aclanthology.org/2023.acl-long.817.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In Thirty-seventh Conference on Neural Information Processing Systems, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. Proceedings of the ACM on Software Engineering, 1(FSE):812–835, July 2024. ISSN 2994-970X. doi: 10.1145/3643763. URL http://dx.doi.org/10.1145/3643763.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In International Conference on Learning Representations, 2022. URL https://openreview.net/forum?id=JYtwGwIL7ye.

Hammond A. Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. 2022 IEEE Symposium on Security and Privacy (SP), pp. 754–768, 2021. URL https://api.semanticscholar.org/CorpusID:245220588.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. arXiv preprint arXiv:2302.12813, 2023.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23. ACM, November 2023. doi: 10.1145/3576915.3623157. URL http://dx.doi.org/10.1145/3576915.3623157.

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

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. arXiv preprint arXiv:2202.01344, 2022.

Joseph Redmon and Alex Sanchez-Stern. Proverbot 9000 : Neural networks for proof assistance, 2016. URL https://api.semanticscholar.org/CorpusID:11622595.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. https://arxiv.org/abs/2308.12950, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. https://arxiv.org/abs/2310.17807, 2023.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. Easy-to-hard generalization: Scalable alignment beyond human supervision. In The Thirty-eighth Annual Conference on Neural Information Processing Systems, 2024. URL https://openreview.net/forum?id=qwgfh2fTtN.

Nikhil Swamy, Cătălin Hriţcu, 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. Dependent types and multi-monadic effects in F*. 2016. ISBN 978-1-4503-3549-2.

Qwen Team. Qwen2.5: A party of foundation models. https://qwenlm.github.io/blog/qwen2.5/, September 2024.

The HumanEval-Verus Contributors. Humaneval-verus: Hand-written examples of verified verus code derived from humaneval, 2024. URL https://github.com/secure-foundations/human-eval-verus.git.

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators, 2024. URL https://arxiv.org/abs/2408.02666.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171, 2022.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In The Eleventh International Conference on Learning Representations, 2023. URL https://openreview.net/forum?id=hH36JeQZDaO.

Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. From decoding to meta-generation: Inference-time algorithms for large language models. Transactions on Machine Learning Research, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=eskQMcIbMS. Survey Certification.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models. https://arxiv.org/abs/2408.00724, 2024a.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. arXiv preprint arXiv:2408.00724, 2024b.

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. AutoVerus: Automated proof generation for Rust code, 2024. URL https://arxiv.org/abs/2409.13082.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models, 2023. URL https://arxiv.org/abs/2308.01825.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. arXiv preprint arXiv:2305.04087, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Li Zhong and Zilong Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In AAAI Conference on Artificial Intelligence, 2023. URL https://api.semanticscholar.org/CorpusID:261048682.

A EXPERIMENTAL DETAILS

A.1 TRANSLATION

We use `DafnyBench` as our source domain D_{src} for our translation experiments. Starting with 782 programs, we filter to 562 by excluding those that verify without proof annotations. The exploration model $G_{explore}$ is initialized using a Verus syntax file and 5 examples from the Verus repository.

A.2 DATASETS

We evaluate on verified versions of the `MBPP` and `HumanEval` datasets. In particular, `MBPP`-verified is sourced from Yang et al. (2024); Misu et al. (2024) and contains 78 programs from the original MBPP dataset Austin et al. (2021). `HumanEval`-Verus is sourced from a concurrent open-source effort The HumanEval-Verus Contributors (2024) to translate existing `HumanEval` programs to Verus. Since each task in `HumanEval`-Verus is typically implemented and verified using multiple functions, we split each program into individual provable functions, ensuring that all dependent functions needed are present. Specifically, we split 49 programs into 85 functions and evaluate methods on these 85 functions. We use a snapshot from November 4th, 2024 with commit hash `ddb9ba3`. For brevity, we refer to `HumanEval` and `MBPP` as their respective verified versions throughout this paper.

A.3 HYPERPARAMETERS

We use consistent decoding parameters, with temperature set to 0.7, top-p set to 1.0 and max_tokens set to 2048. For the translation step, we generate 256 examples per program in the translation phase. We set breadth and depth to 32 and 8 in the treefinement stage. α is set to 0.1 and β is set to 0.03 as defined in Equation 3. We set the rebase node sampling temperature to 0.1. We generate 32 samples for the comparison model and exploit model. We use the same setting in both inference and translation. For stochastic sampling as described in Equation 3.1, we randomly choose $k/2$ examples from the pool of k exemplars. All sampling is done with a batch size of 32. We do not tune hyperparameters and use the conventional settings throughout. We use the 'gpt-4o-2024-08-06' version for `GPT-4o` modal.

A.4 CONTAMINATION ANALYSIS

Despite independent development of `HumanEval` and `MBPP`, we observe significant overlap between these datasets and `DafnyBench` programs. To mitigate contamination in downstream evaluations, we employ GPT-4 for systematic filtering of collected exemplars. Specifically, we prompt GPT-4 with each collected exemplar paired against individual programs from `HumanEval` and `MBPP`, requesting the identification of similar programs. We generate 4 independent evaluations per pair and flag contamination when similarity is detected in more than two evaluations. Flagged examples are excluded from the in-context examples during evaluation of the corresponding program. The prompt used is listed in Listing C.

Manual analysis confirms this approach significantly outperforms traditional n-gram analysis and aligns well with human assessment of contamination. We recommend future work adopt similar contamination detection methods rather than relying solely on n-gram analysis for program similarity. Notably, existing baseline methods for proof annotations in Verus Yang et al. (2024); Chen et al. (2024) lack such contamination analysis.

A.5 HARDWARE AND SOFTWARE

We use L40S GPUs for inference. We use SgLang for inference Zheng et al. (2024). We design a scalable and parallel version of the translation and inference stage, where each program is run on a separate node. We release the complete codebase and our DAFNY2VERUS-COLLECTION for reproducibility.

B Related Work

Automated Formal Verification. Automated formal verification has a long-standing history in interactive theorem provers Redmon & Sanchez-Stern (2016); Kaliszyk et al. (2018); Polu & Sutskever (2020); First et al. (2020); Lu et al. (2023); Li et al. (2024), such as Coq Coq Development Team (2020), Lean Lean FRO, and Isabelle Isabelle. These approaches typically generate step-by-step proof statements for a given problem, with the theorem prover providing feedback on intermediate steps. While these methods have achieved significant success in proving complex mathematical theorems, their application to formal verification of code is typically limited to theorems from existing projects (e.g., First et al. (2023)) or simple program properties Lohn & Welleck (2024) rather than end-to-end verified code generation. An alternative paradigm integrates language models with languages that offload proving to automated reasoning tools (e.g., SMT), including Dafny Leino (2010); Sun et al. (2023); Loughridge et al. (2024) and F* Swamy et al. (2016); Chakraborty et al. (2024). However, enabling verified code generation in these research languages may have limited applicability to real-world software and workflows.

Automated Formal Verification in Rust. In contrast, Verus Lattuada et al. (2023) offers a verification framework for Rust, a widely adopted programming language. However, unlike in formal theorem proving or long-standing verification languages, there is a substantial lack of data for Verus. Two existing works, released during the development of `AlphaVerus`, attempt to overcome data scarcity. First, AutoVerus Yang et al. (2024) prompts GPT-4 with a pipeline of hand-engineered prompts tailored to specific errors and programs. This allows for refining some errors but requires human expertise to support new strategies through additional prompts. In contrast, our Treefinement method learns new refinement strategies automatically. Second, the concurrent work SAFE++ Chen et al. (2024) proposes translating an existing Rust dataset to Verus and training generation and refinement models on the collected data. However, the translation process in Chen et al. (2024) was initialized with over a month of continuous generation from GPT-4. In contrast, `AlphaVerus` relies only on a single openly available model, without an expensive GPT-4 initialization. `AlphaVerus` also incorporates a new tree-search refinement strategy that outperforms the linear strategy used in SAFE++, and a critique phase to ensure the generated specifications are high quality. These innovations contribute to better results, despite our method using open models and 100 times less data. Finally, these two existing works study the simplified task of proof generation, while we study the more general setting of verified code generation: generating the implementation and its proofs.

Inference-Time Strategies. Recent studies have shown that increasing inference-time compute can improve performance in reasoning, mathematics, and code generation via meta-generation strategies Welleck et al. (2024) such as parallel sampling Wang et al. (2022); Aggarwal et al. (2023); Sun et al. (2024), tree search Yao et al. (2024); Wu et al. (2024a), and refinement Welleck et al. (2023); Madaan et al. (2023); Snell et al. (2024). Our Treefinement algorithm can be viewed as a hybrid meta-generator that combines tree search and refinement, following initial parallel sampling (exploration). A variety of tree search methods generate one step of a mathematical solution at a time, with a verifier guiding the search process by assigning a score to the current state Wu et al. (2024a). In contrast, Treefinement uses verifier feedback on the complete solution, modeling tree nodes as full programs and edges as refinement steps. Our strategy addresses the non-local nature of error fixes, and does not need an additional trained scoring model.

Various refinement strategies use external feedback from knowledge bases Peng et al. (2023); Chern et al. (2023), code interpreters Chen et al. (2023); Zhang et al. (2023), tool outputs Gou et al. (2024); Schick et al. (2023), or separately trained reward models Akyürek et al. (2023). Our Treefinement algorithm uses a diverse set of feedback sources, including scalar and binary values, language feedback, and an exploit model. Moreover, whereas prior methods typically operate in a linear fashion–i.e., starting with an output and repeatedly refining it–our approach structures refinement as a tree search. This allows for prioritizing certain branches of refinement, which we find perform better.

Self-Improvement in LLMs. Various algorithms aim to improve a language model using data generated by the model along with an external feedback source Zelikman et al. (2022); Wang et al. (2024); Hosseini et al. (2024), which is colloquially termed *self-improvement*. Common approaches rely on variants of expert iteration or rejection finetuning Polu et al. (2022); Zelikman et al. (2022); Yuan et al. (2023); Lin et al. (2024), where multiple solutions are sampled, and an external signal

selects the positive ones for model fine-tuning. Our approach, AlphaVerus, builds upon these concepts but moves beyond the simple sample-and-filter strategy. Our method additionally uses refinement and tree search to collect data, and the data is collected using multiple modules (e.g., outputs from Treefinement may be used to improve exploration). Additionally, AlphaVerus uses various forms of feedback–such as trinary, scalar, language, and verifier outputs–rather than just binary filtering. Conceptually, we can view AlphaVerus as a meta-generation algorithm (i.e., a combination of parallel sampling, refinement, and tree search) that improves over time, rather than a model trained on filtered outputs.

C METHODOLOGY

Components Table 3 summarizes the components of our method at different stages, the feedback sources used, the models employed, and the data collected for bootstrapping.

Stage	Feedback	Model	Data Collected
Exploration	Verifier (errors)	LLM + Parallel Sampling	Verified Translations
Treefinement	Verifier (value), Verifier (errors)	LLM + Tree Search + Refinement	Error Fix Triplets, Verified Translations
Critique Module	Rules, Trivial Programs, Verifier (binary), Comparison LLM	Regex, String Manipulation, Prompted LLM, Exploit LLM	Exploit Pairs

Table 3: Different components used in iterative translation in AlphaVerus

Alorithm and Prompts We detail the complete algorithm for AlphaVerus in Algorithm 1. We list the prompt used for Exploration stage in Listing C, prompt used for Treefinement stage in Listing C, prompt used for exploit and comparison model in Listing C and Listing C, and for inference in Listing C. Unless specified in the prompt, we use user, assistant pairs to simulate few-shot examples.

Verus Code Completion

Consider the following incomplete Verus code:

```
{program}
```

The code contains the relevant spec functions and the preconditions (`requires`) and postconditions (`ensures`) for the main function. Your goal is to complete the function by adding the necessary procedure, along with proof statements (such as `invariants`, `asserts`, `proof` blocks, etc.) to prove the program.
Only output the new program and not the entire code. You are not allowed to create new functions; however, you can use any functions already defined if they are within the scope.

Translation: Exploration Prompt

Consider the following dafny code:

```
{program}
```

Your goal is to convert the code to Verus code. Based on the syntax I gave you, convert the code to Verus. Note that you may need to make some datatype-related changes for it to work in Verus. Specifically, use the most appropriate ones from the syntax and code examples provided earlier. However, do not change invariants or specifications (ensures and requires clauses). Make sure to include the use statements, proper start of code using verus!, and empty fn main() as done in the examples.

Translation Treefinement Prompt

SYSTEM: Here are some examples of fixing verus code based on compiler error message:

```
# Verus Error Fixing Example {i+1}:
## Incorrect Code:
```rust
{incorrect_code}
```
Error Message:
```
{error_message}
```
Corrected Code after fixing the errors:
```rust
{corrected_code}
```

<Other Examples>
```

USER:
Given a Verus program with function signature, preconditions, postconditions, and code, fix the errors present in the code. Effectively return the complete verys program by fixing all proof statements or adjusting the code, such that the code compiles correctly. Do no modify function signatures requires, ensures or specs. Repeat: Do not ever modify those lines in ensures clause, requires clause, function signatures. Just edit the proof. **Only in case of overflow errors**, you can make reasonable relaxations on the size of the input variables. For instance, considering the input length of array to be any value less than 10 is not reasonable. Similarly for integer inputs, considering them to be small numbers is not reasonable. Choose bigger bounds for relaxation. You can also use spec functions, to estimate the max value, and impose a condition accordingly. For instance, if error is integer overflow while doing multiplication, you can add requires statement such as:

```
forall|k: int| 0 <= k < nums.len() ==> (0 <= #[trigger] nums[k
    ] * #[trigger] nums[k] < i32::MAX)
```

However, absolutely no other changes to precondition and postcondition are permitted! Below is the program::

```
```
{program}
```
```

The program has following error message:

```
```
{error_messsage}
```
```

Solution Format:

```
[Thoughts on Error Message]
[Thoughts on Error Resolution]
[Thoughts on Corner Cases, such as Overflow etc.]
```rust
[Complete Code]
```
```

Translation: Exploit Model Prompt

You are a Verus exploit agent that finds trivial solutions for incomplete and inaccurate preconditions and postconditions. Your goal is to complete the code by proposing trivial solutions that pass all verification conditions. Here are some examples:
Input Problem:

```
use vstd::prelude::*;

verus! {

// Define a function to calculate the nth power of 2
fn power(n: u32) -> (result: u32)
    [Code Truncated]
}

// Define the function ComputePower to calculate 2^n for a given n
fn compute_power(n: u32) -> (result: u32)
    requires
        n >= 0,
        n <= 10000, // arbitrary bound, verus can't handle infinite recursion
    ensures
        result == result,
{
```

Trivial Solution:

```
    let mut result: u32 = 1;
    let mut x: u32 = 0;
    // invariant: 0 <= x <= n, and result == Power(x)
    while x!= n
        invariant
            0 <= x && x <= n,
            result == result, // result == Power(x),
    {
        x += 1;
        result = result.wrapping_add(result);
    }
    result
}

// Main function, empty for now
fn main() {}

} // verus!
```

<Other Examples>

Charactersitics of a trivial solution:
1. Usually 1-5 lines of code
2. Does not use any complex data structures
3. Usually returns constant values, that passes all test cases.
Your task is to provide only the trivially completed code, given a new program. Only output the new program and not the entire code.

Translation: Comparison Model Prompt

Consider the following function:

```
```rust
{rust_code}
```
```

and

```
```dafny
{dafny_code}
```
```

Consider the preconditions and postconditions of the various functions in the two programs along with the spec functions if present, that need to be proven.
Are the preconditions and postconditions from both the programs same? Note, since they are from different programming languages, minor changes are to be ignored. Minor changes include, adding extra preconditions to limit size of input in rust code, so as to ensure overflows are not encountered, or reformulating implication statements. Such changes are not to be considered, and the answer should be yes, if they are same. Further, preconditions on size of input is reasonable, if there is a possibility of overflow. For instance, for computing fibonacci numbers, using something like n¡=47 is reasonable, and answer should be yes. However, using n¡=5 would be incorrect, and answer should be no. Remember, you have to focus on ensures and requires clause of the main function as postconditions and preconditions respectively.
Follow the following format:
[What Preconditions and Postconditions of Program 1]
[What Preconditions and Postconditions of Program 2]
[Step by Step Thoughts on comparison between conditions. Minor differences can be ignored]
Finally, answer in format to the question if they are same: T̈herefore, both programs are same /not_same, the final answer is Yes/No.

Contamination Check Prompt

Consider the following set of program database:

```
```json
<in context exemplars>
```
```

Task: Your task is to find the program that is same or very similar ($\geq$50%) to this program:

```
```
{Program from dataset}
```
```

You should start the solution, by first thinking which programs would be closest and why. Then, you should output the json, containing the same keys as above: prog_num, program_text. It is possible that none of the programs is closest, or even similar. In that case return empty json object.

D RESULTS

D.1 REWARD HACKING

Next, we analyze the quality of translations without the critique phase. Figure 5 shows the effect of removing the critique models and continuing the self-improvement process on 100 examples from `DafnyBench`. Without the critique phase, the model is able to translate a large fraction of programs, but it is primarily because of learning to use `assume(false)` which renders any implementation trivially verified. This is primarily used by human developers to debug their proofs. However, here `AlphaVerus` figures out how to game the system by generating trivial proofs.

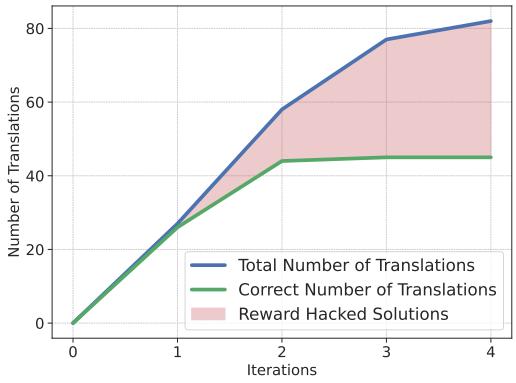

Figure 5: Impact of removing the critique models. Without filtering mechanisms, the model learns to exploit verification by increasingly using `assume(false)` statements. This snowballing effect shows the importance of critique models in preventing reward-hacked solutions.

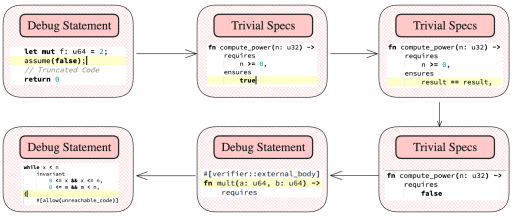

Figure 6: Illustration of reward hacking without the critique models. In particular, the agent first learns to use debug statements and uses them continuously. After fixing, it learns other hacks such as generating trivial specifications or exploring rare debug statements such as allowing infinite loops.

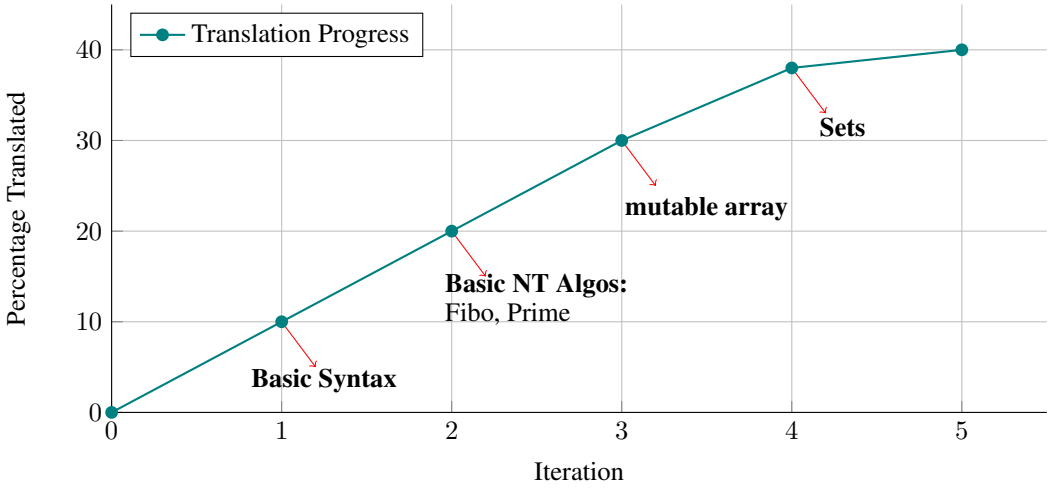

Figure 7: **Translation Progress by Concept**. The graph shows the incremental progress in translation capabilities as different programming concepts are mastered.

There is also a snowballing effect, where initially the model generates a single program with `assume(false)`, then learns to use it in all programs. This is evident from the leveling off of correct translations in the figure. While an obvious way is to disallow such statements (as done by our rule-based verifier), we see even more complicated cases of reward hacking, such as leaving small gaps in translated specifications or even generating degenerate translations, as illustrated in Figure 6. We conclude that the critique phase is critical for filtering out misaligned programs and preventing reward hacking.

`AlphaVerus` enables strong proof annotation. Unlike our work which evaluates methods on the difficult task of formally-verified-code generation that requires generating both code and proof, concurrent work on Verus evaluates on the task of proof annotation: generating proofs given correct code. This is a simpler task since the code is already known to be correct. We compare against SAFE Chen et al. (2024) using their reported results with DSCoder-33B at Pass@110, as their implementation is not publicly available. We also evaluate against AutoVerus Yang et al. (2024) using their default configuration with a 70B model.

Listing 1: AlphaVerus implementation of the unique_sorted function, which removes duplicates from a sorted vector.

```
// ----- Function signature and specification (given to model) -----
fn unique_sorted(s: Vec<i32>) -> (result: Vec<i32>)
    requires
        forall|i: int, j: int| 0 <= i < j < s.len() ==> s[i] <= s[j],
    ensures
        forall|i: int, j: int| 0 <= i < j < result.len() ==> result[i] < result[j],
        forall|i: int| #![auto] 0 <= i < result.len() ==> s@.contains(result[i]),
        forall|i: int| #![trigger s[i]] 0 <= i < s.len() ==> result@.contains(s[i]),
{// ----- Model generation starts here -----
    let mut result = Vec::new();
    let mut i: usize = 0;
    while i < s.len()
        invariant
            result.len() <= i,
            i <= s.len(),
            forall|j: int| 0 <= j < result.len() ==> s@.contains(result[j as int]),
            forall|j: int| 0 <= j < i ==> s@[j as int] == s[j as int],
    {
        if i == 0 || s[i] != s[i - 1] {
            result.push(s[i]);
        }
        i += 1;
    }
    proof {
        assert(forall|i: int, j: int| 0 <= i < j < result.len() ==> result[i] < result[j]) by
            {
            assert(forall|i: int, j: int| 0 <= i < j < s.len() ==> s@[i] <= s@[j]);
            assert(result.len() == s.len());
            assert(result@.len() as int == s@.len() as int);
            assert(result@.to_multiset() =~= s@.to_multiset());
        }
    }
    result
}
```

As shown in Table 4, AlphaVerus outperforms SAFE by 17% and AutoVerus by 10%. This is notable since AlphaVerus was not designed for the proof annotations task, while AutoVerus has correction prompts specifically engineered for the task. Their engineering also results in reduced generalizability; for instance, AutoVerus cannot be evaluated on HumanEval as it doesn't support multi-function programs. Second, SAFE used over a month of GPT-4o invocations and thousands of programs, contrasting with our use of 562 Dafny programs and an openly available 70B model.

AlphaVerus learns new concepts over iterations. Next, our goal is to understand what the model learns over iterations that improves its ability to translate more complex programs and improve downstream performance. We manually inspect translations from each iteration of AlphaVerus in an attempt to qualitatively characterize the kinds of programs that the system gradually learns to translate. Figure 7 depicts the new concepts that we identified across iterations, starting with the ability to translate basic syntax, then basic numeric algorithms, and then the ability to work with mutable arrays and sets.

Cost-optimal model for inference. Next, we compare the performance of different models as we increase the inference cost. We compare LLaMA-3.1-8B and LLaMA-3.1-70B, using a cost ratio of 1:8 based on current API pricing. That is, generating 8 outputs with LLaMA-3.1-8B has the same cost as generating 1 output with LLaMA-3.1-70B. We show the accuracy of each model as a function of cost in Figure 8. LLaMA-3.1-8B achieves faster initial gains, reaching an accuracy of 0.55 with 128 units of cost, while LLaMA-3.1-70B requires about 4 times more cost to reach similar performance. In other words, for cost-constrained scenarios, it is preferable to use the smaller model with more samples, but the larger model has better asymptotic performance. Our findings echo those of Wu et al. (2024a) and Snell et al. (2024).

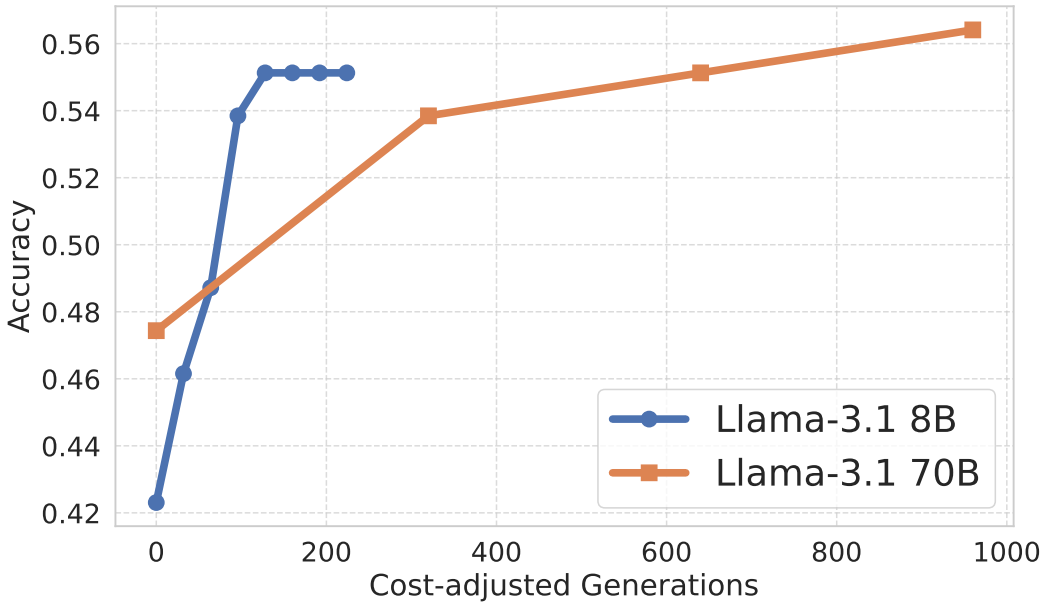

Figure 8: Performance scaling of `LLaMA-3.1-8B` and `LLaMA-3.1-70B` with cost. `LLaMA-3.1-8B` shows better cost efficiency at lower compute budgets, while `LLaMA-3.1-70B` shows higher asymptotic performance.

	MBPP
SAFE	59.0%
AutoVerus	65.4%
`AlphaVerus`	**75.7%**

Table 4: Comparison of proof annotation generation tasks against baselines on the MBPP dataset.

D.2 Qualitative Analysis

Listing 2, Listing 3, and Listing 4 show example Dafny-to-Verus translations from `AlphaVerus`, indicating that `AlphaVerus` is capable of complex translations. In particular, the translations can involve multiple specifications, helper functions, and proof annotations, and individually reach up to 100 lines of Verus code. For formally verified code generation, Listing 1 shows a generated implementation and proofs for a function that removes duplicates from a sorted vector while maintaining its sorted order. The model is given the function signature and the specification (the `requires` and `ensures` clauses). `AlphaVerus` generates a Rust implementation and proof annotations (e.g., `invariant`, `proof`, `assert` statements) that pass the verifier. Listing 5 shows a multi-function example, in which a helper function `is_prime` is followed by a `largest_prime_factor` function. In addition to function implementations, Listing 6 shows `AlphaVerus` completing a nontrivial lemma.

Listing 2, Listing 3, Listing 4 show example Dafny-to-Verus translation from `AlphaVerus`.

Listing 5, Listing 6 show example formally verified code generations from `AlphaVerus`.

E Limitations and Future Work

Our results show promising performance and potential for scaling the methods proposed in this work. In particular, we achieved state-of-the-art performance without any fine-tuning. However, moving forward, exploring fine-tuning models and RL algorithms guided by verifier feedback would be a promising and important direction. Further, while our analysis shows the critique module being

Listing 2: Translation example.

```
// ---- Dafny program (input to AlphaVerus) ----
function R(n: nat): nat {
    if n == 0 then 0 else if R(n-1) > n then R(n-1) - n else R(n-1) + n
}

method calcR(n: nat) returns (r: nat)
    ensures r == R(n)
{
    r := 0;
    var i := 0;
    while i < n
        decreases n-i
        invariant 0 <= i <= n
        invariant r == R(i)
    {
       i := i + 1;
       if r  > i {
           r := r - i;
       }
       else {
            r := r + i;
        }
    }
}
```

```
// ---- Verus program (all model-generated) ----
use vstd::prelude::*;

verus! {

fn calc_r(n: usize) -> (r: usize)
    requires
        n < 100000, // added relaxation to prevent overflow
        n * (n + 1) / 2 < usize::MAX, // added check to prevent overflow
    ensures
        r == n * (n + 1) / 2,
{
    let mut r: u128 = 0;
    let mut i = 0;
    while i < n
        invariant
            0 <= i && i <= n,
            true,
        decreases
            n - i,
    {
        i = i + 1;
    }
    assert(i == n);
    let temp: u128 = n as u128 * (n as u128 + 1) / 2;
    assert(temp <= usize::MAX as u128);
    let r: usize = temp as usize;
    r
}

fn main() {}

} // verus!
```

Listing 3: Translation example.

```
// ---- Dafny program (input to AlphaVerus) ----
method SearchRecursive( a: seq<int>, i: int, j: int, x: int ) returns (k: int)
    decreases j-i;
    requires 0 <= i <= j <= |a|;
    ensures i <= k < j || k == -1;
    ensures k != -1 ==> a[k] == x;
    ensures k != -1 ==> forall r | k < r < j :: a[r] != x;
    ensures k == -1 ==> forall r | i <= r < j :: a[r] != x;
{
    if j == i
    {
        k := -1;
        return;
    }
    if a[j-1] == x
    {
        k := j-1;
        return;
    }
    else
    {
        k := SearchRecursive(a, i, j-1, x);
    }
}

method SearchLoop( a: seq<int>, i: int, j: int, x: int ) returns (k: int)
    requires 0 <= i <= j <= |a|;
    ensures i <= k < j || k == -1;
    ensures k != -1 ==> a[k] == x;
    ensures k != -1 ==> forall r | k < r < j :: a[r] != x;
    ensures k == -1 ==> forall r | i <= r < j :: a[r] != x;
{
    if i == j
    {
        return -1;
    }

    var t := j;
    while t > i
        decreases t;
        invariant forall p | t <= p < j :: a[p] != x;
    {
        if a[t-1] == x
        {
            k := t-1;
            return;
        }
        else
        {
            t := t - 1;
        }
    }
    k := -1;
}
```

effective in preventing reward hacking, it may not work in all cases, especially for more complex problems. We believe investigating better methods for filtering misaligned specifications remains an important future direction. Finally, we believe that treefinement can be extended to other domains,

```
// ---- Verus program (all model-generated) ----
use vstd::prelude::*;

verus! {

/// Searches for an element in a sequence using recursion.
fn search_recursive(a: &Vec<i32>, i: usize, j: usize, x: i32) -> (k: i32)
    requires
        0 <= i && i <= j && j <= a.len(),
        j - i < 100000, // added precondition
        j <= i32::MAX as usize, // added precondition
    ensures
        i <= k && k < j || k == -1,
        k != -1 ==> a[k as int] == x,
        k != -1 ==> forall|r: nat| k < r && r < j ==> a[r as int] != x,
        k == -1 ==> forall|r: nat| i <= r && r < j ==> a[r as int] != x,
    decreases
        j - i,
{
    if j == i {
        -1
    } else if a[j - 1] == x {
        assert(j - 1 <= i32::MAX as usize);
        if j - 1 <= i32::MAX as usize {
            (j - 1) as i32
        } else {
            -1
        }
    } else {
        search_recursive(a, i, j - 1, x)
    }
}

/// Searches for an element in a sequence using a loop.
fn search_loop(a: &Vec<i32>, i: usize, j: usize, x: i32) -> (k: i32)
    requires
        0 <= i && i <= j && j <= a.len(),
        j - i < 100000, // added precondition
        j <= i32::MAX as usize, // added precondition
    ensures
        i <= k && k < j || k == -1,
        k != -1 ==> a[k as int] == x,
        k != -1 ==> forall|r: nat| k < r && r < j ==> a[r as int] != x,
        k == -1 ==> forall|r: nat| i <= r && r < j ==> a[r as int] != x,
{
    if i == j {
        return -1;
    }
    let mut t: usize = j;
    while t > i
        invariant
            0 <= i <= t && t <= j,
            t <= a.len(),
            forall|p: nat| t <= p && p < j ==> a[p as int] != x,
            t <= i32::MAX as usize, // added invariant
        decreases
            t,
    {
        assert(t <= a.len());
        if a[t - 1] == x {
            if t - 1 <= i32::MAX as usize {
                return (t - 1) as i32;
            } else {
                return -1;
            }
        } else {
            assert(t > i);
            t = t - 1;
        }
    }
    -1
}

fn main() {}

} // verus!
```

Algorithm 1: Iterative Translation and Refinement

Input: Source programs D_{src}, initial data $D^{(0)}_{x \to y}, D^{(0)}_{y \to y'}, D^{(0)}_{\text{exploit}}$
Output: Verified target programs D_{tgt}
Initialize $i \leftarrow 0$.
while not converged **do**
  (I) Candidate Generation & Verification:
  foreach $x \in D_{\text{src}}$ **do**
    Generate candidate translations $\{y_j\} \sim G_{\text{explore}}(x; D^{(i)}_{x \to y})$
    $C \leftarrow \emptyset$: verified pairs; $S \leftarrow \emptyset$: syntactically correct, unverified candidates
    foreach y_j **do**
      if y_j passes verification **then**
        $C \leftarrow C \cup \{(x, y_j)\}$
      else if y_j is syntactically correct **then**
        $S \leftarrow S \cup \{y_j\}$
  (II) Refinement via Treefinement Search:
  foreach $y \in S$ **do**
    Initialize a refinement tree with root node $(y, e(y))$
    while max iterations not reached **do**
      Select node $(y', e(y'))$ by REBASE scoring
      Generate refinements $\{y'_k\} \sim G_{\text{refine}}(y', e(y'); D^{(i)}_{y \to y'})$
      foreach y'_k **do**
        if y'_k passes verification **then**
          $C \leftarrow C \cup \{(x, y'_k)\}$; record trajectory in C_τ
          break (stop refining this candidate)
        else
          Add $(y'_k, e(y'_k))$ as a child node to the refinement tree
  (III) Filtering and Data Update:
  foreach $(x, y) \in C$ **do**
    if critic rejects y or $f(x, y) =$ False or exploit model finds z on s_y **then**
      Discard y
      if exploit model finds z **then**
        $D^{(i+1)}_{\text{exploit}} \leftarrow D^{(i+1)}_{\text{exploit}} \cup \{(s_y, z)\}$
  Update $D^{(i+1)}_{x \to y} \leftarrow D^{(i)}_{x \to y} \cup C$
  Update $D^{(i+1)}_{y \to y'} \leftarrow D^{(i)}_{y \to y'} \cup \{(y, e(y), y') | (x, y') \in C_\tau\}$
  $i \leftarrow i + 1$
return $D_{\text{tgt}} \leftarrow \{y \mid (x, y) \in D^{(i)}_{x \to y}\}$

and better search strategies can be developed for even better performance in this important task of verified code generation.

Listing 4: Translation example. We collapse some whitespace and lines with brackets due to space.

```
// ---- Dafny program (input to AlphaVerus) ----
module OneSpec {
    datatype Variables = Variables(value: int)

    predicate Init(v: Variables) { v.value == 0 }

    predicate IncrementOp(v: Variables, v': Variables) { && v'.value == v.value + 1 }

    predicate DecrementOp(v: Variables, v': Variables) { && v'.value == v.value - 1 }

    datatype Step =
        | IncrementStep()
        | DecrementStep()

    predicate NextStep(v: Variables, v': Variables, step: Step)
    {
        match step
            case IncrementStep() => IncrementOp(v, v')
            case DecrementStep() => DecrementOp(v, v')
    }

    predicate Next(v: Variables, v': Variables)
    { exists step :: NextStep(v, v', step) }
}

module OneProtocol {
    datatype Variables = Variables(value: int)

    predicate Init(v: Variables)  { v.value == 0 }

    predicate IncrementOp(v: Variables, v': Variables) { && v'.value == v.value - 1 }

    predicate DecrementOp(v: Variables, v': Variables) { && v'.value == v.value + 1 }

    datatype Step =
        | IncrementStep()
        | DecrementStep()

    predicate NextStep(v: Variables, v': Variables, step: Step)
    {
        match step
            case IncrementStep() => IncrementOp(v, v')
            case DecrementStep() => DecrementOp(v, v')
    }

    predicate Next(v: Variables, v': Variables)
    { exists step :: NextStep(v, v', step) }
}

module RefinementProof {
    import OneSpec
    import opened OneProtocol

    function Abstraction(v: Variables) : OneSpec.Variables {
        OneSpec.Variables(v.value)
    }

    lemma RefinementInit(v: Variables)
        requires Init(v)
        ensures OneSpec.Init(Abstraction(v)) { }

    lemma RefinementNext(v: Variables, v': Variables)
        requires Next(v, v')
        ensures OneSpec.Next(Abstraction(v), Abstraction(v'))
    {
        var step :| NextStep(v, v', step);
        match step {
            case IncrementStep() => {
                assert OneSpec.NextStep(Abstraction(v), Abstraction(v'), OneSpec.DecrementStep
                    ());
            }
            case DecrementStep() => {
                assert OneSpec.NextStep(Abstraction(v), Abstraction(v'), OneSpec.IncrementStep
                    ());
            }
        }
    }
}
```

```
// ---- Verus program (all model-generated) ----
#[allow(unused_imports)]
use builtin::*;
#[allow(unused_imports)]
use builtin_macros::*;

verus! {
struct Variables {
    value: int,
}
spec fn init(v: Variables) -> bool { v.value == 0 }
spec fn increment_op(v: Variables, v_prime: Variables) -> bool { v_prime.value == v.value + 1 }
spec fn decrement_op(v: Variables, v_prime: Variables) -> bool { v_prime.value == v.value - 1 }

enum Step {
    Increment,
    Decrement,
}
spec fn next_step(v: Variables, v_prime: Variables, step: Step) -> bool {
    match step {
        Step::Increment => increment_op(v, v_prime),
        Step::Decrement => decrement_op(v, v_prime),
    }
}
spec fn next(v: Variables, v_prime: Variables) -> bool {
    exists|step: Step| next_step(v, v_prime, step)
}

struct VariablesProtocol { value: int, }

spec fn init_protocol(v: VariablesProtocol) -> bool { v.value == 0 }
spec fn increment_op_protocol(v: VariablesProtocol, v_prime: VariablesProtocol) -> bool { v_prime.value == v.value - 1 }
spec fn decrement_op_protocol(v: VariablesProtocol, v_prime: VariablesProtocol) -> bool { v_prime.value == v.value + 1 }

enum StepProtocol {
    Increment,
    Decrement,
}
spec fn next_step_protocol(v: VariablesProtocol, v_prime: VariablesProtocol, step: StepProtocol) -> bool {
    match step {
        StepProtocol::Increment => increment_op_protocol(v, v_prime),
        StepProtocol::Decrement => decrement_op_protocol(v, v_prime),
    }
}
spec fn next_protocol(v: VariablesProtocol, v_prime: VariablesProtocol) -> bool {
    exists|step: StepProtocol| next_step_protocol(v, v_prime, step)
}
spec fn abstraction(v: VariablesProtocol) -> Variables {
    Variables { value: v.value }
}

proof fn refinement_init(v: VariablesProtocol)
    requires
        init_protocol(v),
    ensures
        init(abstraction(v)),
{ assert(true); }

proof fn refinement_next(v: VariablesProtocol, v_prime: VariablesProtocol)
    requires
        next_protocol(v, v_prime),
    ensures
        next(abstraction(v), abstraction(v_prime)),
{
    assert(abstraction(v_prime).value == abstraction(v).value + 1 || abstraction(v_prime).value == abstraction(v).value - 1);
    assert(exists|step: Step| next_step(abstraction(v), abstraction(v_prime), step)) by {
        if abstraction(v_prime).value == abstraction(v).value + 1 {
            assert(next_step(abstraction(v), abstraction(v_prime), Step::Increment));
        } else {
            assert(next_step(abstraction(v), abstraction(v_prime), Step::Decrement));
        }
    }
    assert(next(abstraction(v), abstraction(v_prime)));
}

fn main() {}
} // verus!
```

Listing 5: Multi-function example. First, we have `AlphaVerus` complete the helper function `it_prime` given its specification and preceding code. Then we have `AlphaVerus` complete `largest_prime_factor`.

```
spec fn spec_prime_helper(num: int, limit: int) -> bool {
    forall|j: int| 2 <= j < limit ==> (#[trigger] (num % j)) != 0
}

spec fn spec_prime(num: int) -> bool {
    spec_prime_helper(num, num)
}

fn is_prime(num: u32) -> (result: bool)
    requires
        num >= 2,
    ensures
        result <==> spec_prime(num as int),
{ // ------- Model generation 1 starts here ---------
    let mut i = 2;
    let mut result = true;
    while i < num
        invariant
            2 <= i <= num,
            result <==> spec_prime_helper(num as int, i as int),
    {
        if num % i == 0 {
            result = false;
        }
        i += 1;
    }
    result
}

fn largest_prime_factor(n: u32) -> (largest: u32)
    requires
        n >= 2,
    ensures
        1 <= largest <= n,
        spec_prime(largest as int),
        n % largest == 0,
        forall|p| 0 <= p < n && spec_prime(p) && n as int % p == 0 ==> p
            <= largest,
{ // ------- Model generation 2 starts here ---------
    let mut largest = 1;
    let mut i = 2;
    while i < n
        invariant
            2 <= i <= n,
            1 <= largest <= i,
            spec_prime(largest as int),
            i <= largest ==> spec_prime(i as int),
            largest <= n,
            n % largest == 0,
            forall|p| 0 <= p < i && spec_prime(p) && n as int % p == 0
                ==> p <= largest,
    {
        if is_prime(i) && n % i == 0 {
            largest = i;
        }
        i += 1;
    }
    assert(1 <= largest && largest <= n);
    assert(spec_prime(largest as int));
    assert(n % largest == 0);
    assert(forall|p| 0 <= p < n && spec_prime(p) && n as int % p == 0 ==>
        p <= largest);
    largest
}
```

Listing 6: Lemma example. `AlphaVerus` completes the lemma `lemma_step_subrange`.

```
proof fn lemma_step_subrange(substring: Seq<char>, string: Seq<char>)
    requires
        substring.len() > 0,
        string.len() >= substring.len(),
    ensures
        (substring[0] == string.subrange(0, substring.len() as int)[0] &&
             (substring.skip(1)
            =~= string.skip(1).subrange(0, substring.skip(1).len() as int
                ))) ==> (substring
            =~= string.subrange(0, substring.len() as int)),
    decreases substring.len(),
{ // -------- Model generation starts here --------
    let m = substring.len();
    let n = string.len();
    assert(m > 0);
    assert(m <= n);
    let h = string.subrange(0, m as int);
    assert(h.len() == m);
    assert(substring.len() == h.len());
    if substring[0] == h[0] {
        let k = substring.skip(1).len();
        assert(k == h.skip(1).len());
        let g = string.skip(1).subrange(0, k as int);
        assert(g.len() == k);
        assert(k == substring.len() - 1);
        assert(k == h.len() - 1);
        if m > 1 {
            if substring.skip(1) =~= g {
                assert(substring[0] == h[0]);
                assert(substring.len() == g.len() + 1);
                assert(substring.len() == h.len());
                assert(forall|i: nat| 0 <= i && i < m ==> substring[i as
                    int] == h[i as int]) by {
                    assert(substring[0] == h[0]);
                    assert(substring.skip(1) =~= g);
                    assert(forall|i: nat| 1 <= i && i < m ==> substring[i
                         as int] == g[i as int - 1]);
                    assert(substring =~= h);
                }
                assert(substring =~= h);
            } else {
                assert(!(substring =~= h));
            }
        } else {
            assert(substring =~= h);
        }
    } else {
        assert(!(substring =~= h));
    }
}
```