DafnyBench: A Benchmark for Formal Software Verification

Chloe Loughridge*
Harvard College
cloughridge@college.harvard.edu

Qinyi Sun*†
Massachusetts Institute of Technology
wendysun@mit.edu

Seth Ahrenbach
seth.ahrenbach@omnifederal.com

Federico Cassano
Northeastern University
cassano.f@northeastern.edu

Chuyue Sun
Stanford University
chuyues@stanford.edu

Ying Sheng
Stanford University
ying1123@stanford.edu

Anish Mudide
Massachusetts Institute of Technology
amudide@mit.edu

Md Rakib Hossain Misu
University of California Irvine
mdrh@uci.edu

Nada Amin
Harvard University
namin@seas.harvard.edu

Max Tegmark
Massachusetts Institute of Technology
tegmark@mit.edu

Abstract

We introduce DafnyBench, the largest benchmark of its kind for training and evaluating machine learning systems for formal software verification. We test the ability of LLMs such as GPT-4 and Claude 3 to auto-generate enough hints for the Dafny formal verification engine to successfully verify over 750 programs with about 53,000 lines of code. The best model and prompting scheme achieved 68% success rate, and we quantify how this rate improves when retrying with error message feedback and how it deteriorates with the amount of required code and hints. We hope that DafnyBench will enable rapid improvements from this baseline as LLMs and verification techniques grow in quality.

1 Introduction

Rapidly improving Large Language Models (LLMs) [1–3] are helping accelerate software development through co-pilots and other program synthesis tools. But how can we ensure that LLM-generated code meets our specifications and reliably does precisely what it is supposed to do? Indeed, this remains a persistent problem even with human-written code: major code-testing efforts failed to prevent e.g. bugs causing an Ariane-V rocket explosion [4] and embarrassing security vulnerabilities in ssh [5] and the Bash shell [6]. The latter was built into the Unix operating system for 25 years before being discovered.

*Equal contribution. Order determined alphabetically.
†Corresponding author.

Preprint. Under review.

Although *formal verification* can guarantee perfect reliability, providing rigorous mathematical proof that software meets specification, it has yet to gain widespread adoption because it is costly. Formally verifying code can easily take more than ten times as much human work as writing it in the first place. Moreover, existing formal-verification tools tend to involve a major learning curve above and beyond just learning to code, greatly reducing the pool of people able to do this work.

The premise of this paper is that AI will soon be able to greatly facilitate formal verification, and hopefully even fully automate it one day. This would drive its cost to near-zero, dramatically increase its adoption and dramatically reduce the prevalence of buggy software. It is easy to imagine formal verification becoming simply a built-in final step of future compilers, which discover code problems and perhaps even fix them automatically. This optimistic premise is based on the close analogy with automated theorem proving, where AI produces formal proofs not about code but about mathematical theorems. Fueled by the advent of benchmarks totaling over 100,000 theorems, AI tools have during the past few years improved their proof success fraction to over 82% [7, 8].

Unfortunately, formal verification sorely lacks correspondingly large benchmarks: the largest of their kind are *Clover* [9] and *dafny-synthesis* [10], containing 66 and 153 programs, respectively. There is room for expanding not only their size, but also their level of difficulty: For example, *Clover* is limited to single-function programs, and sometimes the formal specification for the program directly repeats the implementation of the algorithm (see Appendix F). To support automation of formal verification, the goal of the present paper is to provide such a benchmark expansion. We do so by assembling a suite of formally verified programs written in *Dafny*, a formal verification language that was developed for easy adoption by programmers due to its similarity with popular imperative programming languages such as Python and C++ [11]. In order for formal verification to succeed, most of these programs require supplementary text constituting "hints" to the automated theorem prover.

The rest of this paper is organized as follows. We summarize related work in Section 2, describe our benchmark construction in Section 3, and quantify the ability of current LLMs to solve benchmark verification tasks in Section 4. We summarize our results and discuss promising opportunities for further work in Section 5 . We provide further details on the benchmark construction and evaluation in appendices.

2 Related Work

As summarized in Table 1 below, there is a striking lack of training data for formal verification: while there are hundreds of thousands of training examples for proving mathematical theorems and over ten thousand training examples for synthesizing programs, there are only $66 + 153 = 219$ for proving program correctness. This motivates our work in the current paper to expand the benchmarks from *Clover* and *dafny-synthesis*.

The 66 programs in the *Clover* benchmark are human-written. In contrast, *dafny-synthesis* translates 153 MBPP problems from Python to Dafny using GPT-4. While this method is more efficient than manual translation, it could potentially skew the distribution of represented problems away from real-world Dafny problems that may be too hard for GPT-4 to verify on its own [10]. Our dataset counterbalances this potentially skewed distribution by introducing problems verified by human programmers on GitHub.

Clover proposes the most sophisticated benchmark evaluation strategy to date for formally verifiable software: the authors suggest a six-way consistency check between code, docstrings, and hints. Their checker achieves an 87% acceptance rate of correct implementations on the *Clover* benchmark while rejecting all incorrect implementations [9]. The authors note that equivalence checking with natural language is currently weak, but can hopefully be improved upon [9]. We do not yet implement the full *Clover* evaluation scheme in DafnyBench, and instead deem a benchmark program "solved" if a model can make it pass the Dafny verifier without modifying the `requires` and `ensures` statements in the program and without using `{:verify false}` or `assume false` (see Appendix E for further details).

Table 1: Summary of popular machine-learning benchmark datasets for proving mathematical theorems, synthesizing programs, and formally verifying programs. Size is measured by the number of samples in each dataset. In the formal reasoning datasets, each sample is usually a math problem or a theorem. In the program synthesis and verified software programming benchmarks, each sample corresponds to a program.

Category	Dataset	Size
Mathematical theorem proving	CoqGym [12]	71,000 proofs
	LeanDojo [13]	98,734 proofs
	PISA [14]	138,000 proofs
	Natural Proofs [15]	15,000 proofs
	Archive of Formal Proofs [16]	1 million lines of code
Unverified program synthesis	APPS [17]	10,000 programs
	HumanEvalX [18, 19]	165 programs
	MBPP [20]	974 programs
	SWEBench [21]	2,294 programs
	LiveCodeBench [22]	grows weekly
Formal software verification	Clover [9]	66 programs
	Dafny-synthesis [10]	153 programs

3 DafnyBench Dataset Construction

3.1 Sourcing Ground Truth Programs

In total, our DafnyBench benchmark contains 782 `ground_truth` stand-alone Dafny programs that compile. These problems come from the following sources:

- **GitHub Scrape**: We scraped all publicly available Dafny files on GitHub published on the before the end of 2023. The relevant files were returned from the GitHub API using the `language:Dafny` search command. We then de-duplicated these files using a minhash de-duplication script written by Chenghao Mou (described in Appendix A). The de-duplication process reduced the number of `.dfy` files from ∼15,000 to ∼5,000. We then attempted to verify each of these remaining files using the `dafny verify` command with a local installation of Dafny 4.3.0, and removed any files that did not verify. At this stage, we removed all of the files from the *Clover* repository [9], which had already been formatted as benchmark files. This left 1,112 files. We found that 374 of these files lacked *ensures* statements, and 459 of lacked `assert` and `invariant` clauses. We removed the union of these sets, which left us with 556 `ground_truth` files. Out of these files, 113 verify without any compiler hints. To mitigate data contamination, models run on our benchmark should ideally not be trained on data from the repositories listed in Appendix D.

- **Clover**: We added 62 ground truth textbook Dafny programs provided by the *Clover* dataset [9]. We formatted these to fit our benchmark style and removed their compiler hints. Out of these files, 23 verify without any compiler hints.

- **Dafny-synthesis**: Finally, we included 164 Dafny programs provided by the *dafny-synthesis* benchmark. These problems have been translated from the MBPP benchmark [10]. Out of these files, 72 verify without any compiler hints.

The `ground_truth` programs in our dataset have on average 2.12 methods, 1.03 functions, and 1.40 lemmas. This places the mean complexity of our examples at a level higher than *Clover* alone, which has only one stand-alone method per example.

Table 2: Mean and maximum values that describe attributes of a DafnyBench test program.

	Mean	Max
# Methods	2.12	42
# Functions	1.03	42
# Lemmas	1.40	35
# Characters	1916.47	28736
# Hint characters	261.23	6019

3.2 Task Design: Fill Hints

We have fully implemented the `fill_hints` task. For this task, we took a `ground_truth` program, removed all of its hints (i.e., all of the `assert` and `invariant` statements in the body of the code), and asked LLM to fill hints back in so that the resulting program could be verified with Dafny.

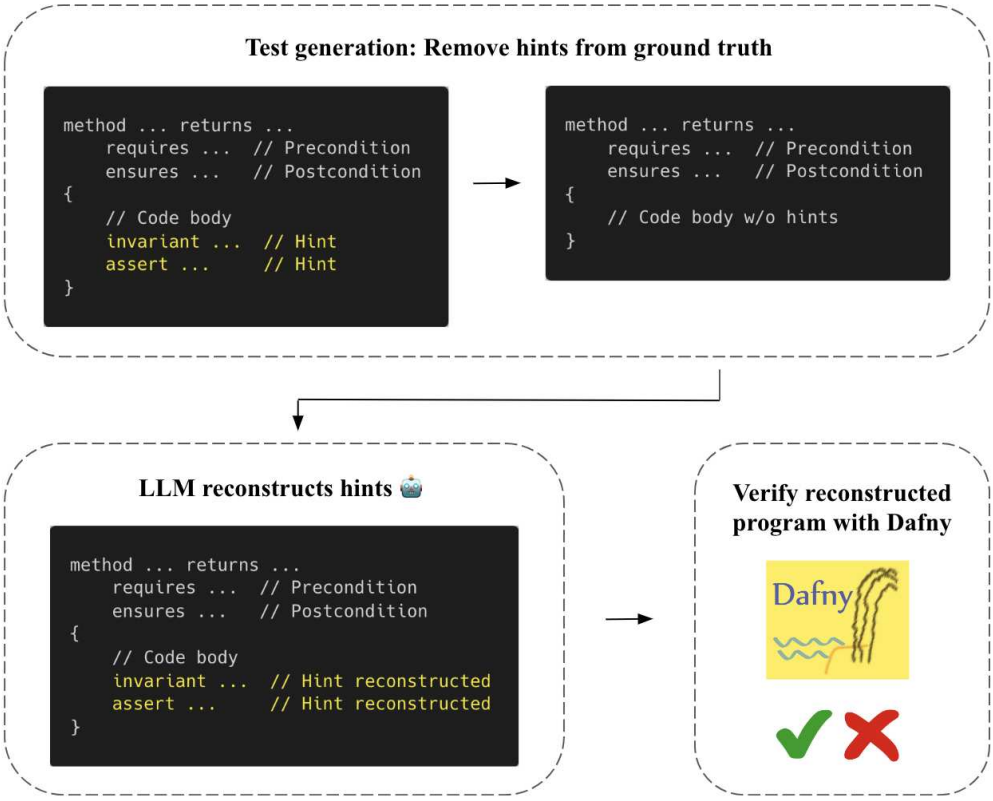


Figure 1: Overview of evaluating LLM on a DafnyBench test program.

We do not demarcate from where these hints have been removed, i.e., we do not insert `/* TODO */` after we remove each annotation, which would make the task easier and not reflective of models utility in real-world use cases.

Evaluation Metric An LLM's attempt to fill hints back in for a test program is counted as a success if all following conditions are satisfied: 1) The reconstructed program is verified with Dafny; 2) LLM preserves all preconditions (`requires` statements) and postconditions (`ensures` statements); and 3) LLM does not use `{:verify false}` or `{assume false}` to "cheat."

```
method LinearSearch<T>(a: array<T>, P: T -> bool) returns (n: int)
    ensures 0 <= n <= a.Length
    ensures n == a.Length || P(a[n])
    ensures forall i :: 0 <= i < n ==> !P(a[i])
{
    n := 0;
    while n != a.Length
        invariant 0 <= n <= a.Length
        invariant forall i :: 0 <= i < n ==> !P(a[i])
    {
        if P(a[n]) {
            return;
        }
        n := n + 1;
    }
}
```

Figure 2: An example `ground_truth` program that is fully verified with Dafny. To create the `fill_hints` task, we would remove the `invariant` lines from the program above.

4 Experiments

In this section, we report success rates for different models on the `fill_hints` task, as well as provide some insight into current LLMs' capabilities at writing hints for formal verification.

4.1 Prompts & Hyperparameters

We tried to keep prompts and hyperparameters mostly the same across models in order to reduce the difference between model performances that is caused by hyperparameters. However, the prompts are not fully identical. For example, when we ask LLM to simply return the hints-filled program without any explanation, Claude 3 tends to add explanations that interfere with Dafny compilation. Thus, we had to adjust some prompts slightly to fit each model's peculiarities.

For hyperparameters, we set `max_tokens` $= 4096$, which corresponds to the lowest max output token limit among all the evaluated models, and we set `temperature` $= 0.3$. We gave each model up to $n = 10$ attempts at a given file. If it succeeded on an attempt before the n^{th}, it would be early stopped. If the model failed on any of the intermediate attempts, it received the Dafny error message and was asked to filled in the hints again with the error message taken into consideration. If it failed on all n attempts, it was considered to fail on that specific test program.

4.2 Basic Results

We tested GPT-4o, GPT-4 Turbo [23], GPT-3.5 Turbo [24], Claude 3 Opus [2], and CodeLlama-7b-Instruct-hf [25] on the 782-program benchmark. Table 3 shows that Claude 3 Opus performed best, achieving a success rate $\sim 68\%$.

4.3 Difficulty Utilizing Dafny Error Messages

Figure 3 shows how the cumulative success rate improved with more attempts n. We see that the best models succeeded on the first try about 54%, with rapidly diminishing returns after that, approaching a plateau about 65% for $n \sim 5$. This suggests that the LLMs are not great at taking Dafny error messages into consideration, or struggle to cope with the underlying task.

Model	% Success
No LLM	26.9
GPT-3.5 Turbo	44.0 ± 1.8
GPT-4 Turbo	59.8 ± 1.8
GPT-4o	59.3 ± 1.8
Claude 3 Opus	**67.8** ± 1.7
CodeLlama-7b-Instruct-hf	28.0 ± 1.6

Table 3: Models' success rates at writing formally verifiable hints for DafnyBench, with $n = 10$ attempts given. Dafny succeeds in auto-verifying some programs even without hints, corresponding to the "No LLM" 26.9% success rate baseline.

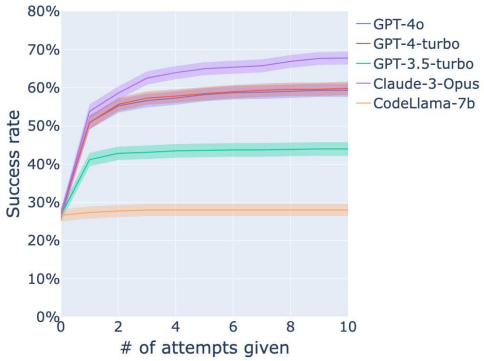


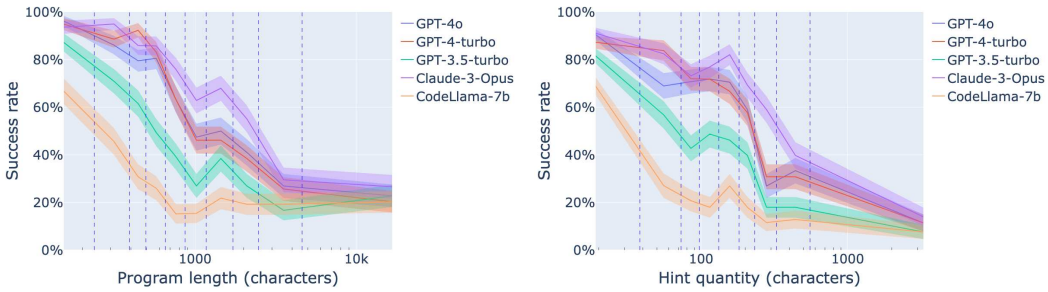
Figure 3: Success rate vs. number of attempts given.

4.4 Difficulty Grows with Program Size

Figure 4a show that the success rate drops with program size. An obvious explanation could be that there is more to verify and more hints needed. Also, as a program gets longer, there may be more dependencies among variables, functions, methods, and classes, increasing the overall verification difficulty level.

4.5 Difficulty Grows with Hint Quantity

Figure 4b shows that the success rate drops with the hint quantity, defined as the number of characters in the lines of compiler hints. In other words, the success rate drops with the amount of work that the LLM needs to do (the amount of text that it needs to insert in the right places).



(a) (b)

Figure 4: **Mean success rate of each bin vs. program length (a)**, and **mean success rate of each bin vs. hint quantity (b)**. The vertical lines indicate the bin boundaries used, where the bins have an almost uniform distribution of the programs. Note that the bins are different for the two metrics. For better visual clarity, the scales are adjusted for both plots and their x-axes do not start at 0 character.

5 Discussion and Conclusions

We have assembled the largest machine-learning benchmark to date for formal software verification and made it publicly available on GitHub at `https://github.com/sun-wendy/DafnyBench`. We also tested five large language models on this benchmark, including one open source model.

We found that Claude 3 Opus achieved $\sim 68\%$ accuracy on our benchmark, with even better success on programs that were shorter or involved less hint text than the benchmark average. GPT-4 Turbo came second with $\sim 60\%$ accuracy. Meanwhile, CodeLlama-7b-Instruct-hf only achieved a marginal

improvement in accuracy compared to our "No LLM" baseline. While in certain cases it succeeds in copying and lightly modifying programs that already verify without compiler hints, it fails to add compiler hints to programs that don't verify without them.

5.1 Opportunities for Larger Benchmarks

It will be valuable to further expand formal verification benchmarks, which still remain more than two orders of magnitude smaller than corresponding benchmarks for mathematical theorem proving. One convenient way to expand the number of available problems may involve incorporating Dafny programs from GitHub that have dependencies spread across multiple files (while *DafnyBench* encompasses increasingly complex multi-step programs, its programs each fit in a single file, avoiding the intricacies associated with distributed files or the integration of external libraries).

Perhaps models that perform especially well on this initial benchmark can later be used to expand it by translating existing Python benchmark problems into Dafny, Rust [26] or other popular formal verification languages.

A subset of the programs we scraped from GitHub do not have appropriate docstrings. By building a benchmark with better code documentation, models may be able to leverage helpful contextual information to better constructing verification hints.

5.2 Benchmark Evaluation Limitations

Data contamination emerges as a potentially significant limitation for evaluating LLMs on this benchmark. Scraping data from platforms such as GitHub introduces risks of leveraging previous models' training data into the benchmark evaluation, potentially artificially inflating the abilities of certain models.

Another limitation emerges in that this benchmark does not assess a model's competence in translating natural language into concise formal specifications. Arguably, this conversion is a demanding and crucial skill we seek from language models: the capacity to validate, beyond merely verifying code. The pivotal question is whether a model can assist in identifying the essential properties an algorithm must fulfill. Currently, evaluating this ability presents significant challenges. The *Clover* paper stands as a prominent example in this area, highlighting the complexity of translating natural language descriptions into formal specifications that can be effectively used for validation. This provides an exciting frontier for future work, which we begin to brainstorm in Appendix C.

5.3 Opportunities for Improved LLM Results

It will be interesting to test this benchmark on additional LLMs, both existing ones such as Gemini [3] and Grok [27] and upcoming ones. Furthermore, we evaluated the models with a fixed temperature setting and a max output token limit of 4096, and we used prompts that were manually but not very systematically tuned for effectiveness (see Appendix B) — all of these choices probably leave room for improvement.

We do not yet provide an official training dataset or models custom-trained to do well on the DafnyBench evaluation set. However, we do provide the full json file produced by the GitHub scrape, and we separately provide the names of the files we use for the evaluation benchmark. Hence it is possible for researchers to use files from the Github scrape that are not used in the benchmark as training data, though we cannot at this time provide strong guarantees on similarity between such training problems and the benchmark problems. Pre-training on this type of data may boost large language model performance on DafnyBench.

We also see great opportunity for LLM-related innovation on the algorithmic side: out-of-the-box LLMs provide a floor but not a ceiling for possible performance on this benchmark. For example, fine-tuning or search-based inference-time algorithms might boost models' performances on this benchmark [28].

5.4 The Promise of Better LLM-Powered Verifiers

LLMs also have potential to improve formal verification in more profound ways than mentioned above, when used in combination with other AI tools. For example, they can help automate the

identification of sub-goals and hints, exponentially reducing the search space for automated theorem provers and SAT solvers. A good software developer is likely able to specify the high level assurance properties of a piece of code. However, in trying to prove that the given code satisfies these high level properties, numerous, sub-goals must be identified, proven, and leveraged correctly in the broader context. Software developers often lack familiarity with the complexities of proof sub-goals and hints. LLMs offer a way to bridge this gap between software developers and formal verification.

Achieving this requires benchmarks suitable for improving the performance and generality of LLMs with respect to software verification. Bigger, more general benchmarks can be used to train LLMs to specify sub-goals and hints in formats most useful to the presently available provers and solvers. Benchmarks covering broad ground, from cryptography, lambda calculus, embedded systems, and avionics, in a variety of widely used programming languages suitable for verification, will help create LLMs that can take real-world software, automatically process and serve it to verification tools, and inform the developer in near real time about the correctness of the code. The problem is analogous to that solved by existing automated theorem provers and model checkers in the domain of mathematics. They address the problem, when given a set of constraint formulas or background theorems, whether a candidate formula is satisfiable or derivable. Many clever algorithms have increased the degree of automation available to mathematical theorem proving over time. LLMs should be able to help similarly improve automation for software verification. For a survey on the application of deep learning to automated theorem proving, see [29].

In order to formally specify a correctness property for a programming language, some formalization of the lower level language's semantics must be represented in a higher level specification language. A lower level language with well-defined semantics to begin with makes this easier. For languages lacking well-defined semantics, such as C, JavaScript, and Python, a well-defined subset may suffice [30]. Programming languages fall on a spectrum of well-defined semantics, with higher level languages like Haskell on one end, and C on the other. Rust falls in a particularly nice intermediate place, with a strongly typed, functional semantics and macros for achieving side effects. An ecosystem of formal verification tools has begun to emerge for Rust, due to its nice semantics and popularity as a practical programming language [31]. A benchmark leveraging this ecosystem for LLMs would likely compound on this progress dramatically. Multiple formal verification tools compile to Rust or extract correct Rust code. For example, Dafny can compile to Rust, and other tools for extracting Rust from Coq exist [32, 33]. In this case, Rust would be considered the low level language, and Dafny and Coq would serve as candidate specification languages. A workflow might be possible such that a developer working in Rust could have a LLM assistant that identifies correctness properties for the code, either automatically or provided at a high level by the developer, and produces appropriate artifacts for verifying correctness via multiple tools for improved assurance.

5.5 The Promise of Auto-Verifying Program Synthesis

Above we discussed the challenge of verifying existing pre-programs. Anther promising approach is use program-synthesis techniques that produce not only programs but also proofs of their correctness, all at the same time. This makes intuitive sense, since when a human programmers writes code, they typically have an informal proof in their head for why this code is correct.

In other words, in addition to bridging the gap from low level implementation to high level specification in the upward direction, LLMs can offer assistance in generating provably correct low level code from high level specifications via program synthesis. Current approaches to program synthesis enable engineers to encode a desired specification in a high level language, and then through a (hopefully) verified correct compiler generate correct low level code in a language like VHDL [34] or Verilog [35] for hardware synthesis. Indeed, the compilation of Dafny code to Rust or Python is an example of program synthesis. Program synthesis is limited by the need for a special purpose language or compiler to be constructed and verified correct in its own right. For example, ReWire is a domain specific language defined as a subset of Haskell [36]. Using ReWire, engineers can specify hardware properties and then through the Haskell compiler, synthesize VHDL that is guaranteed to satisfy the specifications. ReWire itself was manually verified correct using the Coq Interactive Theorem Prover. In order to add a new high to low path, a new language or compiler must be defined and verified. If an engineer needs to synthesize correct Verilog rather than VHDL, they must first learn Caisson [37].

LLMs offer a way to generalize this approach. Starting with a high level language, an engineer might be able to specify a system and then leverage a LLM to generate low level code with the

corresponding loop invariants, weakest pre-conditions, strongest post-conditions, etc, included. In the limit, an engineer might be using a natural language to describe the system and its desired assurance properties, with the LLM performing translation, annotation, and even suggesting additional correctness properties. Early results indicate that an LLM that is able to converse with a human when producing a program can reduce the error rate against a simple programming benchmark by half [20]. If instead of receiving feedback from a human, the LLM were to interact with a suite of formal verification tools, we expect further improvements. We could avoid hallucination problems by relying on the LLM to generate the code and formal specification, but relying on an established verification tool to perform the model checking or proof verification itself. The LLM's translation process need not itself be verified, because it can try multiple times to produce a verifiable output. The LLM must be capable of generating code that is appropriately annotated for theorem proving, which is exactly the skill assessed by test benches like that described here. The more theorem proving tools and programming languages that LLMs are trained and assessed on, the more auto-verifying program synthesis options become available. To return to the previous example, a LLM proficient at ReWire, Caisson, and myriad other software verification techniques, might be given a ReWire specification as input and told to produce correct Verilog as output. The ReWire specification contains the high level correctness properties that must be satisfied. The task is to synthesize Verilog code that satisfies those same correctness properties specified in Caisson. A strong ability to reason about code properties and to express them in multiple languages is exactly what is called for here, and what diverse LLM test benches help to enable.

In summary, there are good reasons for optimism that automated formal verification will soon be greatly improved.

Acknowledgements: The authors wish to thank Clark Barrett, Rustan Leino, Daniel Windham, David Brandfonbrener, William Byrd, Josh Engels, and Anastasiya Kravchuk for helpful discussions.

References

[1] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[2] Anthropic. The claude 3 model family: Opus, sonnet, haiku. Technical report, Anthropic, 2024.

[3] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[4] European Space Agency. Flight 501 failure, 1996. URL `https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`. Accessed: 2024-06-04.

[5] Heartbleed. The heartbleed bug. `https://heartbleed.com/`, 2024. Accessed: 2024-06-04.

[6] Wikipedia contributors. Shellshock (software bug). `https://en.wikipedia.org/wiki/Shellshock_(software_bug)`, 2024. Accessed: 2024-06-04.

[7] Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving. *arXiv*, September 2020. doi: 10.48550/arXiv.2009.03393.

[8] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. HyperTree Proof Search for Neural Theorem Proving. *arXiv*, May 2022. doi: 10.48550/arXiv.2205.11491.

[9] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *ICLR 2024 Conference*, 2024. URL `https://openreview.net/forum?id=oSuVEv4X7w`.

[10] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE), 2024. doi: 10.1145/3643763. URL `https://doi.org/10.1145/3643763`.

[11] K Rustan M Leino. *Program Proofs*. MIT Press, 2023.

[12] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants, 2019.

[13] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023.

[14] Pisa: Isabelle proofs dataset. `https://aitp-conference.org/2021/abstract/paper_17.pdf`, 2021.

[15] Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. Naturalproofs: Mathematical theorem proving in natural language. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL `https://openreview.net/forum?id=Jvxa8adr3iY`.

[16] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 3–17, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20615-8.

[17] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[18] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021.

[20] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[21] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, Karthik Narasimhan, et al. Swe-bench: Can language models resolve real-world github issues?, 2023.

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

[23] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan,

Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[24] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[25] Code llama: Fine-tuning llama for code generation. `https://huggingface.co/Phind/Phind-CodeLlama-34B-v2`, 2022.

[26] Steve Klabnik and with contributions from the Rust Community Carol Nichols. *The Rust Programming Language*. No Starch Press, second edition, 2021. URL `https://doc.rust-lang.org/stable/book/`.

[27] XAI-ORG. Grok-1 Model on Hugging Face. `https://huggingface.co/xai-org/grok-1`, 2024. Accessed: 2024-06-05.

[28] David Brandfonbrener, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Jianang Yang, Simon Henniger, William E. Byrd, Robert Zinkov, and Nada Amin. Verified multi-step synthesis using large language models and monte carlo tree search, 2023.

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

[30] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL `http://xavierleroy.org/publi/Clight.pdf`.

[31] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[32] The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[33] DANIL ANNENKOV, MIKKEL MILO, JAKOB BOTSCH NIELSEN, and BAS SPITTERS. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming*, 32:e11, 2022. doi: 10.1017/S0956796822000077.

[34] Design Automation Standards Committee and Automatic Test Program Generation Subcommittee. Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019. doi: 10.1109/IEEESTD.2019.8938196.

[35] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Springer Publishing Company, Incorporated, 5th ed. edition, 2008. ISBN 0387849300.

[36] Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. Semantics driven hardware design, implementation, and verification with rewire. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, LCTES'15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332576. doi: 10.1145/2670529.2754970. URL https://doi.org/10.1145/2670529.2754970.

[37] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 109–120, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993512. URL https://doi.org/10.1145/1993498.1993512.

[38] Wikipedia contributors. Jaccard index — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Jaccard_index, 2024. Accessed: 2024-03-27.

[39] Wikipedia contributors. N-gram — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/N-gram, 2024. Accessed: 2024-03-27.

[40] Chenghao Mou, Chris Ha, Kenneth Enevoldsen, and Peiyuan Liu. Chenghaomou/text-dedup: Reference snapshot, September 2023. URL https://doi.org/10.5281/zenodo.8364980.

[41] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

A The Minhash Deduplication Algorithm

We can think about deduplicating a set of files by finding groups of "similar"files and then choosing only one file representative from each group to form our final deduplicated set of files. To do this, we can use the Jaccard similarity metric to decide whether one document is a duplicate of another.

The Jaccard similarity metric provides a way to quantify the similarity of two sets. It is defined as [38]:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

In the application to code files, we could consider each file to be a set of n-grams, where an n-gram is defined as a sequence of n adjacent symbols in a particular order [39], and then apply the Jaccard score as a similarity metric for our files. To directly calculate this Jaccard score, we would need to run string comparison on every n-gram, which would have time complexity $O\left(nm^2\right)$ if we have n n-grams each with max length m characters. This turns out to be an inefficient method for representing each code file as a set. Instead, the minhash deduplication algorithm approximates the Jaccard similarity between two documents by shingling the documents and comparing the minhash representation of each set of shingles (i.e. we compare fingerprints of documents instead of full

documents). The minhash representation of a document is a way to represent a text document as a set of numbers that is faithful to the structure of its content but with a fixed set size that is smaller than the total number of n-grams in the document (i.e. the minhash representation of the document is a form of numerical fingerprint of the document). In Figure 5 below, we provide the pseudocode for the minhash algorithm used, based entirely on the script in [40]:

```
function minhash_deduplication(documents, num_permutations, threshold)
    :
    # Preprocess the documents
    for each document in documents:
        tokenize the document into n-grams (shingles)
        hash each n-gram using a hash function (e.g., xxHash or SHA-1)
        store the hashed n-grams in a set

    # Generate permutations
    for i from 1 to num_permutations:
        generate random coefficients a and b
        create a permutation function: (a * x + b) % prime_modulus

    # Create minhash signatures
    signatures = []
    for each document in documents:
        signature = []
        for each permutation function:
            min_hash = INFINITY
            for each hashed n-gram in the document:
                permuted_hash = apply permutation function to hashed n
                    -gram
                min_hash = min(min_hash, permuted_hash)
            append min_hash to signature
        append signature to signatures

    # Perform Locality-Sensitive Hashing (LSH)
    # We use 250 permutations, so to achieve Jaccard similarity
       threshold of 0.5
    # We really only need one band (i.e. one hash table)
    num_bands = choose number of bands
    rows_per_band = num_permutations / num_bands
    candidate_pairs = []
    for each band:
        create an empty hash table
        for each document signature:
            band_signature = subset of signature for the current band
            hash_bucket = hash(band_signature)
            add document to the corresponding hash bucket
        for each hash bucket:
            if number of documents in the bucket > 1:
                generate all pairs of documents in the bucket
                add pairs to candidate_pairs
```

Figure 5: Pseudocode for the minhash deduplication algorithm.

Note that the probability two files have the same min hash value under the same hash function is equivalent to their Jaccard similarity. Concretely, for file A and file B:

$$\Pr\left[\min h_i(A) = \min h_i(B)\right] = J(A, B)$$

where $\min h_i()$ denotes taking the minimum hash value under hash function h_i. This makes sense because, assuming negligible hash collision, $\Pr\left[\min h_i(A) = \min h_i(B)\right]$ is equivalent to the probability that the first n-gram hash of A under h_i is equal to the first n-gram hash of B under h_i. If h_i is a good hash function, then it uniformly distributes the hash values of the original n-gram hashes over the range of h_i. Let c denote the number of n-grams with equivalent hashes; let a denote the number

```
    # Use a union-find datastructure to track groups of duplicates
    duplicates = UnionFind()
    for each band:
        for each row in hashtable:
            for each hash_bucket:
                if size(hash_bucket) <= 1:
                    continue
                else:
                    cluster_id = min(hash_bucket)
                    for x in hash_bucket:
                        duplicates.union(x, cluster_id)

    # Perform deduplication
    deduplicated_documents = []
    for each document in documents:
        if duplicates.find_root(document) = document:
            add document to deduplicated_documents

    return deduplicated_documents
```

Figure 5: Pseudocode for the minhash deduplication algorithm (continued).

of n-grams from A with smaller hash values than the hash value of corresponding n-gram from B; let b denote the reverse of the previous category. Then, $\Pr\left[\min h_i(A) = \min h_i(B)\right] = \frac{c}{a+b+c}$, given the uniformity of h_1. Note that $\frac{c}{a+b+c} = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$.

B Prompt Engineering for Hint Reconstruction

We based our prompts on the prompts used in the *Clover* benchmark [9].

B.1 GPT Model Famly

```
SYSTEM_PROMPT = "You are an expert in Dafny. You will be given tasks dealing
                 with Dafny programs including precise annotations."

USER_PROMPT = "Given a Dafny program with function signature, preconditions,
               postconditions, and code, but with annotations missing.
               Please return a complete Dafny program with the strongest
               possible annotations (loop invariants, assert statements,
               etc.) filled back in. Do not explain. Please use exactly the
               same function signature, preconditions, and postconditions.
               Do not ever modify the given lines. Below is the program:"
```

B.2 Claude 3 Opus

```
SYSTEM_PROMPT = "You are an expert in Dafny. You will be given tasks dealing
                 with Dafny programs including precise annotations. You should
                 only return code body in all circumstances. No text is allowed."

USER_PROMPT = "Given a Dafny program with function signature, preconditions,
               postconditions, and code, but with annotations missing.
               Please return a complete Dafny program with the strongest
               possible annotation (loop invariants, assert statements,
               etc.) filled back in. Do not explain or output any text. If
               you have to explain, put all explanations in comments form.
               There should only be code body in your output. Please use
               exactly the same function signature, preconditions, and
               postconditions. Do not ever modify the given lines. Below
               is the program:\n‘‘‘dafny\n"
```

B.3 CodeLlama-7b-Instruct-hf

The prompts for CodeLlama-7b-Instruct-hf are the same as those in B.2.

C Proposals for Evaluating Strength of Generated Specifications

The evaluation of models' capability to generate formal specifications might be enhanced by integrating the process with the creation of positive and negative test cases for each Dafny implementation. This approach proposes a reward system where models are evaluated based on the number of positive test cases their formal specifications support and the number of negative test cases they successfully reject. However, this method introduces a new challenge: ensuring the test cases accurately reflect the comprehensive meaning intended in the natural language descriptions. The consistency and validity of these test cases become critical, raising questions about the methods used to generate and verify them.

D Repositories of Scraped Dafny Code

We provide a full list of all repositories whose data we used in the scraped portion of DafnyBench in Tables 4, 5, 6. When reporting the license information, "Renamed so N/A" implies that the original repository we scraped in December 2023 no longer exists under that name. Otherwise, the repositories have either Microsoft open-source licenses, MIT licenses, GNU General Public License v3.0 licenses, Creative Commons Zero v1.0 Universal, Apache 2.0 licenses, or "Other" (which is secretly an MIT License in a strange format, which has been checked manually). In light of this, we release our derivative DafnyBench repository under an Apache 2.0 license and a GNU General Public License v3.0. We note explicitly here that all files from repositories with the Apache 2.0 license have been modified from their original form.

E Dafny Verification Examples

We take one example test program from DafnyBench, and consider four possible results for the corresponding LLM-reconstructed program: successfully verifies, fails to verify, cheats by including `assume false`, and cheats by including `{:verify false}`. The last three cases are all considered a fail by the DafnyBench evaluation metric.

E.1 Successful Example

Figure 6 shows a Dafny program that is considered to have successfully verified without cheating.

Dafny verifier message: Dafny program verifier finished with 3 verified, 0 errors.

E.2 Failed Example

Figure 7 shows a Dafny program that fails to be verified.

Dafny verifier message: (20,11): Error: index out of range. (30,4): Error: a postcondition could not be proved on this return path. (11,28): Related location: this is the postcondition that could not be proved. Dafny program verifier finished with 2 verified, 2 errors.

E.3 Cheat Example

Figure 8 shows that a Dafny program cheats by including `assume false`, which DafnyBench evaluation would count as a fail.

Dafny verifier message: Dafny program verifier finished with 3 verified, 0 errors.

E.4 Another Cheat Example

Figure 9 shows that another Dafny program cheats by including `{:verify false}`, which DafnyBench evaluation would count as a fail.

Table 4: Repositories from which DafnyBench utilizes scraped code (no particular order).

Repository Name	**License**
dafl	No license provided
Dafny-Grind75	No license provided
feup-mfes	MIT License
Dafny	GNU General Public License v3.0
nitwit	MIT License
Dafny-experiences	No license provided
Formal_Verification_With_Dafny	No license provided
SENG2011	No license provided
M2	No license provided
assertive-programming-assignment-1	No license provided
t1_MF	No license provided
dafny-exercise	Other
dafny-learn	No license provided
software-specification-p1	No license provided
FMSE-2022-2023	The Unlicense
fv2020-tms	No license provided
type-definition	No license provided
laboratory	No license provided
dafny	GNU General Public License v3.0
TFG	GNU General Public License v3.0
SiLemma	MIT License
dafny-training	No license provided
FormalMethods	No license provided
dafny_misc	MIT License
vmware-verification-2023	No license provided
CSU55004—Formal-Verification	No license provided
MIEIC_mfes	MIT License
Dafny-programs	No license provided
MFES_2021	MIT License
DafnyPrograms	No license provided
cs357	No license provided
formal-methods-in-software-engineering	No license provided
Dafny_ProgrammingLanguages	No license provided
CSC8204-Dafny	No license provided
BPTree-verif	No license provided
tangent-finder	No license provided
Trab1-Metodos-Formais	No license provided
verified-using-dafny	MIT License
Metodos_Formais	No license provided
lets-prove-blocking-queue	Creative Commons Zero v1.0 Universal
Dafny_Programs	No license provided
dafny-workout	MIT License
Dafny-Projects	No license provided
VerifiedMergeSortDafny	No license provided
dafny_projects	No license provided
pucrs-metodos-formais-t1	No license provided
specTesting	No license provided
QS_BoilerPlate1	No license provided
dafny-sandbox	No license provided
Formal-Verification	No license provided
dafny-duck	No license provided
FlexWeek	No license provided
703FinalProject	No license provided

Table 5: Repositories from which DafnyBench utilizes scraped code (no particular order), continued.

Repository Name	**License**
MFS	No license provided
dafny-mini-project	No license provided
Software-Verification	No license provided
circular-queue-implemetation	No license provided
Final-Project-Dafny	No license provided
DafnyProjects	No license provided
bbfny	No license provided
Formal-methods-of-software-development	No license provided
Software-building-and-verification-Projects	No license provided
software_analysis	No license provided
cs245-verification	No license provided
dafny-aoc-2019	No license provided
ProjectosCVS	No license provided
MFDS	MIT License
groupTheory	No license provided
dafny-language-server	Other
Invoker	Apache License 2.0
formal-verification	No license provided
dafny-programs	No license provided
ironsync-osdi2023	Other
verified-isort	No license provided
paxos_proof	No license provided
se2011	No license provided
Dafny_Verify	No license provided
Formal-Methods-Project	No license provided
630-dafny	No license provided
dafny_examples	MIT License
Workshop	No license provided
Dafny-Practice	MIT License
CVS-handout1	No license provided
CS494-final-project	No license provided
iron-sync	Other
stunning-palm-tree	Creative Commons Zero v1.0 Universal
sat_dfy	No license provided
verification-class	MIT License
AssertivePrograming	No license provided
Dafny-VMC	MIT License
libraries	Other
cmsc433	No license provided
Correctness	No license provided
CVS-Projto1	No license provided
dafleet	MIT License
dafny-rope	MIT License
protocol-verification-fa2023	No license provided
vfag	No license provided
Dafny_Learning_Experience	Apache License 2.0
summer-school-2020	No license provided
BinarySearchTree	Renamed so N/A
llm-verified-eval	MIT License
Programmverifikation-und-synthese	Renamed so N/A
Prog-Fun-Solutions	Renamed so N/A
CO3408-Advanced-Software-Modelling-Assignment...	Renamed so N/A

Table 6: Repositories from which DafnyBench utilizes scraped code (no particular order), continued.

Repository Name	**License**
DafnyExercises	No license provided
test-generation-examples	No license provided
HATRA-2022-Paper	No license provided
veri-sparse	No license provided
Formal-Verification-Project	No license provided
formal_verication_dafny	No license provided
Simulink-To_dafny	No license provided
dafny_experiments	No license provided
cs686	No license provided
Program-Verification-Dataset	MIT License
Dafny-demo	No license provided
dafny-exercises	No license provided
metodosFormais	No license provided
CS5232_Project	No license provided
Dafny-Exercises	No license provided

```
function sorted(a: array<int>) : bool
    reads a
{
   forall i,j : int :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}

method BinarySearch(a: array<int>, x: int) returns (index: int)
    requires sorted(a)
    ensures 0 <= index < a.Length ==> a[index] == x
    ensures index == -1 ==> forall i : int :: 0 <= i < a.Length ==> a[
        i] != x
{
    var low := 0;
    var high := a.Length - 1;
    var mid := 0;

    while (low <= high)
        invariant 0 <= low <= high + 1 <= a.Length
        invariant x !in a[..low] && x !in a[high + 1..]
    {
        mid := (high + low) / 2;
        if a[mid] < x {
            low := mid + 1;
        }
        else if a[mid] > x {
            high := mid - 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}
```

Figure 6: An example response that successfully fills hints back in and verifies without cheating.

Dafny verifier message: Dafny program verifier finished with 3 verified, 0 errors.

```
function sorted(a: array<int>) : bool
    reads a
{
   forall i,j : int :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}

method BinarySearch(a: array<int>, x: int) returns (index: int)
    requires sorted(a)
    ensures 0 <= index < a.Length ==> a[index] == x
    ensures index == -1 ==> forall i : int :: 0 <= i < a.Length ==> a[
        i] != x
{
    var low := 0;
    var high := a.Length - 1;
    var mid := 0;

    while (low <= high)
    {
        mid := (high + low) / 2;
        if a[mid] < x {
            low := mid + 1;
        }
        else if a[mid] > x {
            high := mid - 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}
```

Figure 7: An example response that fails to be verified, as it lacks necessary `invariant` statements.

F Overdetailed Specification

Figures 10 and 11 show two example programs `update_array_strong.dfy` and `triple_strong.dfy` from the *Clover* benchmark [9], in which the formal specification closely echoes the program implementation.

G Ethics Statement

In creating DafnyBench, we took care to use only data that was publicly available on GitHub, and we reference every repository from which we acquired this data, along with their licenses, in Appendix D. Furthermore, we cite the existing verifiable programming benchmarks that we subsume in DafnyBench (i.e. *Clover* [9] and *dafny-synthesis* [10]), and we asked explicit permission from their authors in order to do so. Finally, we cite all models that were used for evaluations on this benchmark [23, 24, 2, 25]. We used these models in accordance with the policies set forth in their API and model card documentation.

H Reproducibility Statement

Our benchmark contains the 782 `ground_truth` programs and the corresponding `hints_removed` programs. Additionally, we include full metadata on all of these files and the evaluation scripts necessary for running the listed models on them. By using the OpenAI and Anthropic APIs, others looking to reproduce this work should not expect to spend more than $300 for a full run of GPT4-o on DafnyBench, $300 for a full run of Claude3 on DafnyBench, $500 for a full run of GPT4-turbo on DafnyBench, and $400 for a full run of GPT-3.5 on DafnyBench. We used the `sglang` package

```
function sorted(a: array<int>) : bool
    reads a
{
   forall i,j : int :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}

method BinarySearch(a: array<int>, x: int) returns (index: int)
    requires sorted(a)
    ensures 0 <= index < a.Length ==> a[index] == x
    ensures index == -1 ==> forall i : int :: 0 <= i < a.Length ==> a[
        i] != x
{
    assume false;
    var low := 0;
    var high := a.Length - 1;
    var mid := 0;

    while (low <= high)
    {
        mid := (high + low) / 2;
        if a[mid] < x {
            low := mid + 1;
        }
        else if a[mid] > x {
            high := mid - 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}
```

Figure 8: An example response that cheats by including `assume false`.

[41] to efficiently query the models. All evaluations were completed on a Linux cluster with an A100 Nvidia GPU.

```
function sorted(a: array<int>) : bool
    reads a
{
   forall i,j : int :: 0 <= i < j < a.Length ==> a[i] <= a[j]
}

method {:verify false} BinarySearch(a: array<int>, x: int) returns (
    index: int)
    requires sorted(a)
    ensures 0 <= index < a.Length ==> a[index] == x
    ensures index == -1 ==> forall i : int :: 0 <= i < a.Length ==> a[
        i] != x
{
    var low := 0;
    var high := a.Length - 1;
    var mid := 0;

    while (low <= high)
    {
        mid := (high + low) / 2;
        if a[mid] < x {
            low := mid + 1;
        }
        else if a[mid] > x {
            high := mid - 1;
        }
        else {
            return mid;
        }
    }
    return -1;
}
```

Figure 9: An example response that cheats by including `{:verify false}`.

```
method UpdateElements(a: array<int>)
  requires a.Length >= 8
  modifies a
  ensures old(a[4]) +3 == a[4]
  ensures a[7]==516
  ensures forall i::0 <= i<a.Length ==> i != 7 && i != 4 ==> a[i] ==
      old(a[i])
{
  a[4] := a[4] + 3;
  a[7] := 516;
}
```

Figure 10: An example program `update_array_strong.dfy` from the *Clover* benchmark [9], in which the formal specification closely echoes the program implementation.

```
method Triple (x:int) returns (r:int)
  ensures r==3*x
{
  r:= x*3;
}
```

Figure 11: Another example program `triple_strong.dfy` from the *Clover* benchmark [9], in which the formal specification closely echoes the program implementation.