Can Large Language Models Verify System Software? A Case Study Using FSCQ as a Benchmark.

Jianxing Qin Alexander Du Danfeng Zhang Matthew Lentz Danyang Zhuo
Duke University

Abstract

Large language models (LLMs) have demonstrated remarkable coding capabilities. They excel in code synthesis benchmarks across diverse domains and have become ubiquitous in coding tools. Recently, they have also shown promise in generating mathematical proofs and small software programs. In this paper, we explore their potential to produce proofs for complex system software (e.g., file systems), where verification typically requires substantial manual effort. By automating parts of this process, LLMs could reduce the verification burden and make rigorous proofs for system software more accessible. To evaluate LLMs for system software verification, we use FSCQ, a verified file system, as our benchmark. Our results confirm the promise of this approach: with appropriate proof context and a straightforward best-first tree search, off-the-shelf LLMs achieve 38% proof coverage for theorems sampled from FSCQ. Moreover, for simpler theorems—those with human proofs under 64 tokens, which make up about 60% of all FSCQ theorems—LLMs achieve over 57% coverage. These findings are preliminary, and we anticipate that various techniques can further improve proof coverage.

CCS Concepts

• **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → *Artificial intelligence*.

Keywords

Formal Methods, System Software, Artificial Intelligence

ACM Reference Format:
Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. 2025. Can Large Language Models Verify System Software? A Case Study Using FSCQ as a Benchmark.. In *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730382

HOTOS '25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/2025/05
https://doi.org/10.1145/3713082.3730382

1 Introduction

System software is a critical infrastructure, and ensuring the absence of bugs is essential. Formal methods hold significant promise for system software verification [16, 20, 39], but applying these methods to complex system software requires substantial manual effort. Reducing this proof burden through automation remains a vital area of research [6, 7, 17, 22, 29, 30, 37, 49].

The advent of large language models (LLMs) has opened new possibilities, achieving state-of-the-art performance in various code synthesis tasks. For example, the leading systems in text-to-SQL benchmarks are now exclusively LLM-based [11]. Code generation has become a staple in modern software development, integrated into popular IDEs [2, 15]. Beyond code synthesis, LLMs demonstrate impressive reasoning skills, tackling challenging mathematical problems, including those from the International Mathematical Olympiad (IMO) [34, 35].

In this paper, we pose a fundamental question: *Can LLMs replace or augment the manual process of writing proofs for complex system software*? If successful, this approach could generalize proof automation beyond existing satisfiability modulo theories (SMT)-based techniques [22, 23].

As an initial step toward answering this question, we evaluate the ability of GPT-4o and Gemini variants to generate proofs for FSCQ [10], a formally verified file system whose Coq-based proofs guarantee crash safety. We evaluate on the latest version of FSCQ, which supports deferred writes and the design and proof techniques are described in the DFSCQ paper [9]. Unlike purely mathematical problems, FSCQ introduces unique challenges, as it involves a large, real-world codebase unlike existing formal mathematics benchmarks [3, 48]. Further, FSCQ's proof goals are tightly coupled with system software behavior (i.e., crash safety).

Our preliminary results are highly encouraging. By providing proof context from other parts of the project (emulating human-in-the-loop development) and applying a best-first tree search, off-the-shelf LLMs managed to achieve 38% proof coverage for theorems sampled from FSCQ. For simpler theorems—with human proofs under 64 tokens—these models achieved proof coverage exceeding 57%. Note that this category encompasses about 60% of all FSCQ theorems. Interestingly, in some cases, LLM-generated proofs were even

more concise than human ones. These findings suggest that LLM-based proof generation could significantly reduce the manual verification burden for complex system software.

While these results are promising, open questions remain. First, we did not explore the generation of intermediate lemmas, an intellectually demanding aspect of the verification process. Second, due to GPU resource constraints, we did not investigate the impact of model training or fine-tuning on proof generation. Models fine-tuned specifically on Coq or system software verification may perform even better.

2 Related Work

Traditional Proof Automation Automated theorem provers (ATPs) have long relied on logical rules and heuristics to simplify the development of formal proofs. Interactive theorem provers like Coq, Isabelle, and HOL Light often come with built-in automated search tactics. For example, Coq includes automation tactics such as `auto` and `eauto`, which are heavily used in system software verification [10, 16, 24].

Another important category of ATP methods involves SMT solvers, which convert proof goals into satisfiability problems. Verification tools like Dafny [23], Verus [22], and F* [38] simplify the use of SMT solvers and have been used successfully for system software verification [17, 30, 37].

Other popular ATP techniques include superposition, resolution, and term rewriting [19, 21]. These methods leverage logical rules to automate reasoning tasks.

LLMs for Mathematical Proofs The emergence of LLMs reframed theorem proving as a text generation task. Because generating entire proofs at once is challenging, state-of-the-art methods typically produce proofs incrementally, by generating individual tactics and verifying each tactic with a proof assistant. GPT-f [35] pioneered this approach, training an LLM to predict tactics and using cumulative log probabilities to estimate the likelihood of proof completion. It discovered new, shorter proofs adopted by the Metamath library. Polu et. al [34] extended GPT-f with a proofsize objective and expert iteration, enabling LLMs to solve some IMO problems. Since these early efforts, substantial progress has been made in applying LLMs to solve increasingly complex mathematical problems, including model pretraining [4, 36, 43], finetuning [46], autoformalization [42] and context retrieval [45].

LLMs for Formal Verification Given the success of LLMs in theorem proving, several recent efforts have explored their application to formal verification. Early work focused on generating entire proofs in a single pass, rather than predicting individual tactics [12, 27]. Selene [47] used seL4 [20] as a benchmark for whole-proof generation and explored context augmentation strategies by including related lemmas. In contrast, our approach adopts tactic-level prediction and incorporates both related lemmas and related proofs as contextual information. FVEL [25] also used tactic-level prediction, but focused primarily on small programs. In contrast, our work targets system software, and combines tactic-level proof search with richer context selection.

Concurrent to our work, Rango [40] similarly applies LLM-based proof search to software verification. While Rango uses a finetuned small model with trial-and-error linear search on CompCert, our work applies off-the-shelf LLMs with best-first tree search on FSCQ. Our work focuses on the FSCQ codebase, we compare the LLM generated proofs and original manual proofs (§4.2) and analyze the failing cases when LLMs cannot complete a proof (§4.3).

3 Best-First Proof Search for Coq

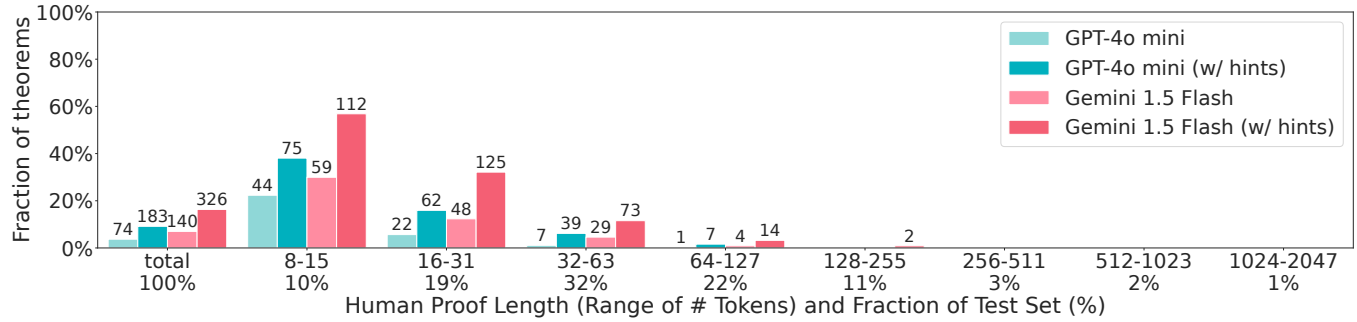
We introduce a system, inspired by GPT-f [35], that automates proof search in Coq by querying an LLM. To systematically explore the space of potential proofs, we employ a best-first search algorithm, similar to the approach used by GPT-f and other related works [34, 42, 45]. Utilizing Coq's low-level state transition machine interface [5] and SerAPI [13], we built a custom Coq proof checker integrated with this search algorithm.

The search iterates through a tree structure, starting with the root goal, and operates in two steps:

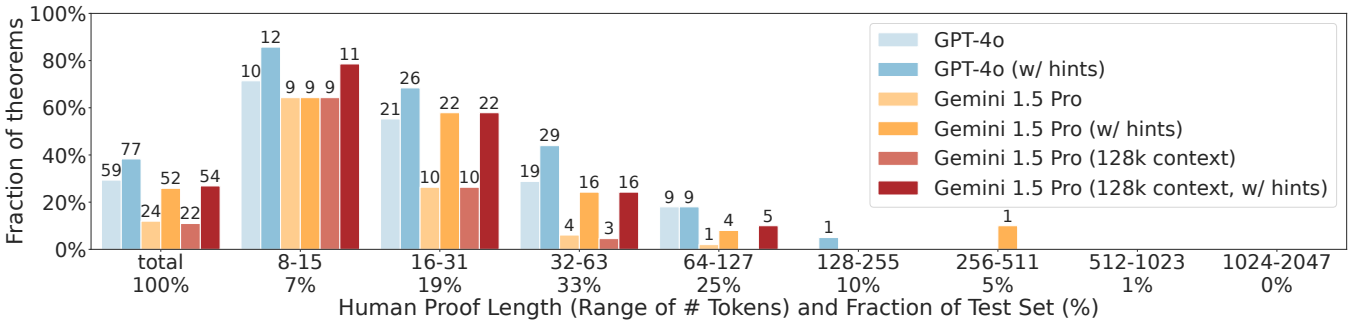
- **Selection.** Select the goal with the highest score from the set of unexpanded goals. The score is defined as the cumulative log probability of the tactics used to reach that goal from the root goal.
- **Expansion.** Query the model for possible next tactics based on the selected goal. Mark the selected goal as expanded and independently append each predicted tactic to it. Each valid tactic either completes the selected goal or creates a subgoal as its child.

A tactic is considered invalid if it: (1) is rejected by Coq, (2) results in a proof state that has already been encountered in the search tree, or (3) takes longer than 5 seconds to execute. To manage computational overhead, we also impose a limit on the number of model queries. The search succeeds if, at any point, the proof is completed (i.e. all goals are proven); it fails if no unexpanded goal remain or if the query limit is reached before the proof is complete.

Our approach diverges from GPT-f in two main ways. First, GPT-f limits the model's context to the active proof goals, which is insufficient for a large codebase like FSCQ, where components are highly interdependent. To address this problem, we provide the model with an extended proof context that may include definitions, theorem statements, and proof steps in the current file and imported files up to (but not beyond) the active proof goals.



(a) Using Small LLMs (GPT-4o mini, Gemini 1.5 Flash) on 50% of the FSCQ codebase.



(b) Using Large LLMs (GPT-4o, Gemini 1.5 Pro) on 5% of the FSCQ codebase.

Figure 1: Proof coverage analysis.

Second, because of cost constraints, we rely on pre-trained, off-the-shelf LLMs to estimate the likelihood of proof completion, rather than training or fine-tuning a dedicated model. Training a specialized LLM involves significant effort and remains a promising direction for future work.

4 Results

Model choices. We evaluate four off-the-shelf LLMs: GPT-4o mini, GPT-4o, Gemini 1.5 Flash, and Gemini 1.5 Pro, selected for their strong performance on established chatbot evaluation benchmarks [1]. To study the effect of context length on proof generation, we evaluate Gemini 1.5 Pro with both a 1M-token context window and a truncated 128k-token context window. For all other models we evaluate, we use their default context window.

Best-first search's hyperparameters. We set the search width to 8, as Gemini currently supports a maximum of 8 outputs per query. The model query limit is set to 128, consistent with the configuration used in GPT-f.

Prompt design. We test two prompt settings: a vanilla setting and a hint setting. In the vanilla setting, the model receives a proof context containing definitions and theorem statements only, but no proof steps. However, we hypothesize that FSCQ proofs might share recurring patterns or structural similarities that could guide tactic generation. To explore this, the hint setting augments the proof context with human proofs taken from 50% of the theorems, which are selected at random and remain consistent across all experiments. For each step, we include the preceding proof context in its file, as well as those from imported files. When the prompt exceeds the model's context window, we retain the portions closer to the next tactic.

Data. Our experiments use theorems from the FSCQ codebase. For smaller models (GPT-4o mini, Gemini 1.5 Flash), we evaluate their performance on all theorems not selected for the hint setting. For larger models, (GPT-4o, Gemini 1.5 Pro), we reduce the evaluation data to 10% of the theorems not selected for the hint setting, representing 5% of all FSCQ theorems. This sampling is due to budget constraints. Note that all theorems tested for larger models are also in the test set for smaller models.

4.1 Proof Coverage

Figure 1 illustrates the fraction of theorems that each LLM can solve directly, grouped by the length (in tokens) of their human proofs in FSCQ. Notably, the hinted GPT-4o model proves 38% of all FSCQ theorems and 57% of simpler theorems (those with human proofs under 64 tokens)—a category that constitutes about 60% of all FSCQ theorems.

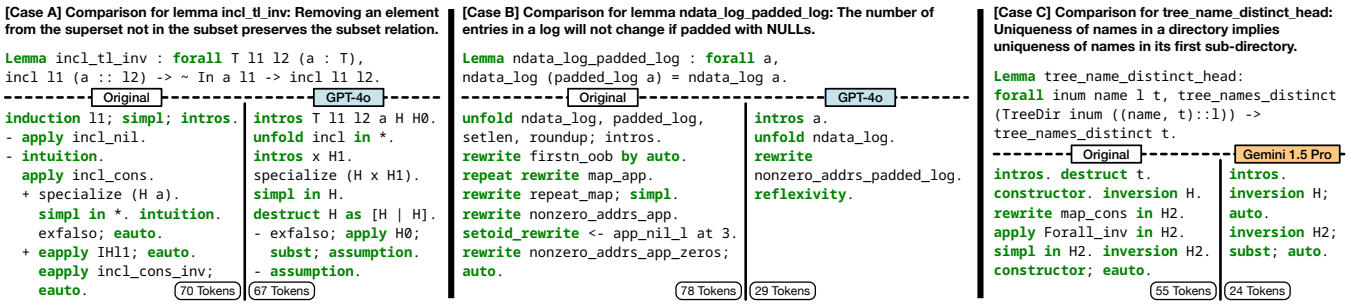


Figure 2: Selected examples comparing human proofs with LLM-generated proofs.

Model	Utilities	CHL	File System
GPT-4o	40.0% / 36.0%	43.3% / 32.3%	15.6% / 24.4%
GPT-4o (w/ hints)	57.8% / 46.6%	51.7% / 42.2%	20.8% / 32.0%

Table 1: Proof coverage across different categories of theorems (actual / expected based on token length).

Although LLMs excel at shorter proofs, their coverage degrades considerably when confronted with theorems requiring longer proofs. No model has succeeded in proving theorems whose human proofs exceed 512 tokens.

Figure 1 shows that when models are prompted with hints (ie. proofs of previous theorems), the proof coverage improves substantially. One might expect a larger context window to yield even better results. Surprisingly, as Figure 1b shows, reducing the Gemini 1.5 Pro context window from 1M tokens to 128k tokens does not improve its coverage. This observation suggests that simply feeding the model more context is not necessarily optimal, and better context-selection strategies may be needed moving forward.

To provide a different perspective based on FSCQ's verification structure, we investigate proof coverage for different categories: Utilities, which are helper lemmas generally useful in Coq (e.g. `ListUtils.v`); CHL, representing lemmas in Crash Hoare Logic; and File System, which refers to lemmas in the file system components of FSCQ. For each category, we compute two proof coverage values. The *actual* coverage is based on the LLM's ability to prove the lemmas in a category. The *expected* coverage captures category-agnostic behavior based on the lengths of the human proofs; for each lemma in a category, we use the proof coverage result from Figure 1 for the bin of human proof lengths that the lemma falls in.

Table 1 presents this analysis using GPT-4o. These results suggest that the model exceeds expected coverage in the Utilities and CHL categories but performs worse than expected in the File System category. Notably, with hints, the model proves over half of the CHL lemmas, indicating its understanding of CHL gained from the provided proof context. This demonstrates that the model can adapt to custom proof systems like CHL, which suggests that the model may be able to handle specialized verification tasks with appropriate context. In contrast, we observe that LLMs perform worse than expected for the File System category; we suspect this is from an increase in dependent theorems and custom tactics, but further analysis is needed to understand this difference. Additionally, when using LLMs to augment human proof effort, these results highlight the potential utility in delegating different classes of lemmas to LLMs.

4.2 Is it Reasoning or Memorization?

Case Studies Because the FSCQ codebase was publicly available on GitHub before the tested models' knowledge cutoffs, one might worry that the generated proofs reflect memorization rather than genuine verification capability. To address this concern, we manually compared several LLM-generated proofs with their human-written counterparts to confirm that the generated proofs are not mere duplicates. In fact, in some cases, LLM-generated proofs are more concise than their human-written versions.

Figure 2 presents three such examples. In Case A, the human-written proof uses induction unnecessarily, while the LLM (GPT-4o) bypasses induction entirely. In Case B, the human proof expands the reasoning through multiple rewrites, whereas the LLM (GPT-4o) simplifies it by applying a single lemma. Finally, in Case C, the human proof involves redundant applications of lemmas, while the LLM (Gemini 1.5 Pro) inverts both hypotheses and directly uses `subst` and `auto`. These examples show that the human proofs tend to rely on additional lemmas, sometimes unnecessarily, while LLMs leverage Coq's default tactics more efficiently.

Proof similarity More broadly, we quantified the similarity between LLM-generated proofs and human proofs using normalized Levenshtein distance (ranging from 0 to 1, where

Model	Proved	Failed		Qualitative Metrics	
		Stuck	Fuelout	Similarity	Length
GPT-4o mini	4.2% → 9.1%	94.8% → 90.0%	1.0% → 0.9%	0.460 → 0.582	97.4% → 113.7%
GPT-4o	**29.2% → 38.1%**	65.8% → 57.9%	5.0% → 4.0%	0.546 → 0.605	101.6% → 100.7%
Gemini 1.5 Flash	7.1% → 16.3%	91.7% → 81.7%	1.2% → 2.0%	0.529 → 0.598	100.6% → 98.7%
Gemini 1.5 Pro	11.9% → 25.7%	88.1% → 73.3%	0.0% → 1.0%	0.565 → 0.660	98.7% → 92.5%
Gemini 1.5 Pro (128k context)	10.9% → 26.7%	89.1% → 72.8%	0.0% → 0.5%	0.579 → 0.683	111.2% → 109.1%

Table 2: Percentage of proved lemmas and failed lemmas (by failure mode). Additionally, qualitative metrics for average similarity and length of generated proofs relative to the corresponding human proofs. Each result is shown *without hints* → *with hints*.

1 denotes an exact match and 0 indicates complete dissimilarity). As Table 2 shows, the average similarity typically remains below 0.6, with a maximum of 0.683, indicating that LLM-generated proofs are not replicas memorized verbatim. Note that randomly sampled proofs from FSCQ (with completely different proof goals) have an average similarity of 0.360.

Proof length We also compared the average length (in tokens) of the generated proofs to that of the human proofs. As Table 2 shows, the average length of the generated proofs is similar to that of the human proofs.

4.3 When and Why do LLMs Fail?

Table 2 presents the distribution of the two failure modes in our system: "stuck" refers to cases where no more unexpanded goals remain, while "fuelout" indicates that the model query limit has been reached. Notably, "fuelout" events are much less frequent than "stuck" events, suggesting that an increase to the query limit alone is unlikely to yield significant improvements. Instead, the model's inherent reasoning capability appears to be a more critical factor limiting proof coverage.

Context selection In some cases, the LLMs struggle to prove very simple theorems, even when they could be resolved by applying a nearby lemma. We suspect this difficulty arises because the lengthy prompts contain many potential lemmas, making it challenging for the LLMs to identify the relevant ones. As a result, the models often attempt to reconstruct the proof from scratch, which is significantly more difficult. To confirm this intuition, we manually crafted prompts for specific failed theorems (those with human proofs under 16 tokens). For each selected theorem, we examined its dependencies and included only the necessary definitions, lemmas, and tactics in the prompt. With this reduced context, the LLMs were able to successfully complete the proofs.

Reasoning models Recent developments in reasoning models, such as OpenAI's o1 variants, have demonstrated remarkable performance on mathematical problem-solving tasks [31]. These models were not included in our experiments because they currently do not provide the log probabilities required for best-first search. Instead, we attempted whole-proof generation for selected theorems using o1 variants, which highlighted several challenges when applying these reasoning models to system software verification:

- **Lack of interaction with the proof assistant.** Current reasoning models, which operate on natural language, struggle when the problem and proof are formalized in a strict formal language like Coq. Specifically, these models seem to lack awareness of the proof progress during intermediate steps. For example, the models may incorrectly assume that a subgoal is simple enough to be closed by built-in automation (e.g. `auto`) when it actually requires several additional steps. Addressing these errors requires at least several rounds of human interaction, which is less efficient and less automated than best-first search.
- **High token usage and long inference times.** The internal reasoning process of these models typically involves iterative generation and feedback, which increases both token usage and inference time. This makes reasoning models less practical for tasks that involve long contexts.

5 Discussions

Improving search and reasoning algorithms. In designing our tool, we chose to use best-first search, a strategy which has been well-explored in prior work. However, there are many other search strategies to consider, such as Monte Carlo Tree Search [14], as well as a range of prompting techniques, such as Chain-of-Thought [41] and Self-Reflection [18]. We have seen newer LLMs incorporate reasoning, such as o1 [31], which could enable proof strategies

beyond tactic-by-tactic generation. It remains an open research problem to continue exploring these alternate algorithms.

Off-the-shelf versus fine-tuned LLMs. Off-the-shelf LLMs are trained on a broad dataset sourced from the internet, which primarily consists of text unrelated to system software verification proofs (though those proofs may be a subset). Training LLMs on domain-specific data, or fine-tuning off-the-shelf LLMs on such data, could significantly improve their success rate in completing proofs [4, 34, 35, 42, 45].

LLMs augmenting, not replacing, human proof effort. LLMs also have the potential to augment developers in writing proofs. We saw that hints, derived from human effort in proving other theorems, improved the proof coverage of LLMs. We also observed that LLM performance varies across different categories of theorems. While our work explores one approach to augmenting human proof effort, other paradigms remain to be explored; for instance, LLMs might assist by completing partial proofs or suggesting next steps. Future usability studies will be valuable in understanding how to best integrate LLM into developer workflows.

Improving context retrieval. As we have shown in §4, context selection plays a crucial role in improving LLMs' ability to complete proofs. Our evaluation used a simple strategy—randomly sampling 50% of the available proofs—to supply additional context. While this broad inclusion of information proved beneficial, prior work has shown that excessive context can actually degrade model performance [26]. Investigating methods to retrieve more relevant and targeted context may significantly enhance LLM proof generation capabilities.

Constructing intermediate lemmas. In our study, we only invoke LLMs to generate proofs for existing theorem and lemma statements. However, a challenging aspect of verification involves the development of useful intermediate lemmas to break down theorems. We believe recent work on autoformalization [42] and induction invariant synthesis [8, 32, 33] are promising, but further research is needed.

Efficacy for different types of theorem provers. Some recent work has investigated the integration of LLMs with automated theorem provers that use SMT solvers, such as Dafny and Verus [28, 44]. These tools differ in how proofs are constructed and in the feedback they provide to the user. In Coq, the proof goals are clearly displayed and updated with each tactic, making it easy to understand progress or identify failure. In contrast, intermediate goals in Verus are not clear, including the effects of additional `assert` statements beyond whether the assertion failed (or not). Currently, there is no benchmark that allows us to compare the efficacy of LLMs across these different types of theorem provers, (particularly in the context of large-scale software systems). It remains an open research question which of these environments is more amenable to LLMs.

What is the metric to demonstrate that the LLM is not merely memorizing and paraphrasing existing proofs? We use normalized Levenshtein distance to measure the similarity of LLM generated proofs and the original FSCQ proofs. However, LLMs are excellent at paraphrasing: their output may be a paraphrased version of their memorized proofs, and normalized Levenshtein distance cannot capture the similarity between paraphrased proofs. Ideally, the true test would involve evaluating LLMs on entirely unseen proofs; however, in the context of existing verified systems, this means we need to pre-train LLMs from scratch to make sure it has never seen the manual proofs before, preventing the use of all off-the-shelf LLMs (or their fine-tuned versions). Currently, to the best of our knowledge, all the existing LLM-for-proof works [40, 47] demonstrate capabilities on existing systems that already have manual proofs. What the appropriate evaluation methodology is for off-the-shelf LLMs or their fine-tuned versions in proof synthesis remains an open question.

6 Conclusion

Using LLMs to generate verification proofs is a promising approach to reduce the human effort required in formal verification. Our results show that, with appropriate proof context and a straightforward best-first tree search, LLMs achieve a proof coverage of 38% for theorems sampled from FSCQ. Moreover, for simpler theorems—those where human proofs used fewer than 64 tokens—LLMs achieve a proof coverage exceeding 57%. Our results also identify a few key areas for improving proof coverage, particularly for large-scale software systems, including better context construction, improved reasoning algorithms, and using proof steps from similar theorems as hints.

Our code, along with all the proofs generated, is open-sourced at https://github.com/QDelta/LLM-for-FSCQ.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. Our work is partially supported by NSF grants 2238665 and 2402696, as well as by gifts from Amazon, Google, and Meta.

References

[1] 2025. *Chatbot Arena (formerly LMSYS) Free AI Chat to Compare & Test Best AI Chatbots.* https://lmarena.ai/

[2] 2025. *Cursor - The AI code editor.* https://www.cursor.com/

[3] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. 2025. ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics. In *Conference on Neural Information Processing Systems (NeurIPS).*

[4] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. 2023. Llemma: An Open Language Model for Mathematics. In *International Conference on Learning Representations (ICLR).*

[5] Bruno Barras, Carst Tankink, and Enrico Tassi. 2015. Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface. In *International Conference on Interactive Theorem Proving (ITP).*

[6] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. 2024. Practical Verification of System-Software Components Written in Standard C. In *ACM Symposium on Operating Systems Principles (SOSP).*

[7] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS Concurrent and Crash-Safe File System with Sequential Reasoning. In *Symposium on Operating Systems Design and Implementation (OSDI).*

[8] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Empirical Methods in Natural Language Processing (EMNLP).*

[9] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *ACM Symposium on Operating Systems Principles (SOSP).*

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *ACM Symposium on Operating Systems Principles (SOSP).*

[11] Papers With Code. 2025. *spider Benchmark (Text-To-SQL).* https://paperswithcode.com/sota/text-to-sql-on-spider

[12] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *ACM International Conference on the Foundations of Software Engineering (FSE).*

[13] Emilio Jesús Gallego Arias. 2016. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. *HAL preprint HAL:01384408* (2016).

[14] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: Learning to Prove with Tactics. *Journal of Automated Reasoning* 65, 2 (2021).

[15] GitHub. 2025. *GitHub Copilot · Your AI pair programmer.* https://github.com/features/copilot

[16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Symposium on Operating Systems Design and Implementation (OSDI).*

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *ACM Symposium on Operating Systems Principles (SOSP).*

[18] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards Mitigating LLM Hallucination Via Self Reflection. In *Empirical Methods in Natural Language Processing (EMNLP).*

[19] John Arnold Kalman. 2001. *Automated Reasoning with Otter.* Rinton Press Princeton NJ.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles (SOSP).*

[21] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *International Conference on Computer Aided Verification (CAV).*

[22] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *ACM Symposium on Operating Systems Principles (SOSP).*

[23] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR).*

[24] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009).

[25] Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. 2024. FVEL: Interactive Formal Verification Environment with Large Language Models via Theorem Proving. In *Conference on Neural Information Processing Systems (NeurIPS).*

[26] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024).

[27] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE).*

[28] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-assisted Synthesis of Verified Dafny Methods. In *ACM International Conference on the Foundations of Software Engineering (FSE).*

[29] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *ACM Symposium on Operating Systems Principles (SOSP).*

[30] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles (SOSP).*

[31] OpenAI. 2025. *Learning to reason with LLMs.* https://openai.com/index/learning-to-reason-with-llms/

[32] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *International Conference on Machine Learning (ICML).*

[33] Muhammad AA Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C Cordeiro. 2024. LLM-Generated Invariants for Bounded Model Checking without Loop Unrolling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE).*

[34] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2023. Formal Mathematics Statement Curriculum Learning. In *Conference on Neural Information Processing Systems (NeurIPS).*

[35] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393* (2020).

[36] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y. Wu, and Guo Daya.

2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300* (2024).

[37] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[38] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[39] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[40] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *International Conference on Software Engineering (ICSE)*.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[42] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[43] An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024. Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement. *arXiv preprint arXiv:2409.12122* (2024).

[44] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. In *International Conference on Learning Representations (ICLR)*.

[45] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2024. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[46] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2024. MetaMath: Bootstrap Your Own Mathematical Questions for Large Language Models. In *International Conference on Learning Representations (ICLR)*.

[47] Lichen Zhang, Shuai Lu, and Nan Duan. 2024. Selene: Pioneering Automated Proof in Software Verification. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

[48] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. miniF2F: A Cross-System Benchmark for Formal Olympiad-Level Mathematics. In *International Conference on Learning Representations (ICLR)*.

[49] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *ACM Symposium on Operating Systems Principles (SOSP)*.