Agentic Verification of Software Systems

HAOXIN TU, National University of Singapore, Singapore
HUAN ZHAO, National University of Singapore, Singapore
YAHUI SONG, National University of Singapore, Singapore
MEHTAB ZAFAR, National University of Singapore, Singapore
RUIJIE MENG, National University of Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Automatically generated code is gaining traction recently, owing to the prevalence of Large Language Models (LLMs). Further, the AlphaProof initiative has demonstrated the possibility of using AI for general mathematical reasoning. Reasoning about computer programs (software) can be accomplished via general mathematical reasoning; however, it tends to be more structured and richer in contexts. This forms an attractive proposition, since then AI agents can be used to reason about voluminous code that gets generated by AI.

In this work, we present a first LLM agent, AutoRocq, for conducting automatic verification of software systems. Unlike past works, which rely on extensive training of LLMs on proof examples, our agent learns on-the-fly and improves the proof via an iterative refinement loop. The iterative improvement of the proof is achieved by the proof agent communicating with the Rocq (formerly Coq) theorem prover to get additional context and feedback. The final result of the iteration is a proof derivation checked by the Rocq theorem prover. In this way, our proof construction involves autonomous collaboration between the proof agent and the theorem prover. This autonomy facilitates the search for proofs and decision-making in deciding on the structure of the proof tree. Experimental evaluation on SV-COMP benchmarks and on Linux kernel modules shows promising efficacy in achieving automated program verification. As automation in code generation becomes more widespread, we posit that our proof agent can be potentially integrated with AI coding agents to achieve a generate and validate loop, thus moving closer to the vision of *trusted automatic programming*.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Security and privacy → Logic and verification**.

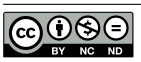
Additional Key Words and Phrases: Formal Methods; Program Verification; Theorem-proving; LLM Agent

ACM Reference Format:
Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. 2026. Agentic Verification of Software Systems. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE157 (July 2026), 24 pages. https://doi.org/10.1145/3808164

1 Introduction

The widespread adoption of AI-generated code has transformed the landscape of software development. Today, more than 25% of the new code at Google is generated by AI [26]. Yet the increasing amounts of code often carry subtle semantic errors or vulnerabilities [52] that may elude human review and testing, necessitating automated reasoning on program behaviors to engender *trust*.

Authors' Contact Information: Haoxin Tu, National University of Singapore, Singapore, haoxin.tu@nus.edu.sg; Huan Zhao, National University of Singapore, Singapore, zhaohuan@comp.nus.edu.sg; Yahui Song, National University of Singapore, Singapore, yahuisong123@gmail.com; Mehtab Zafar, National University of Singapore, Singapore, mehtab@nus.edu.sg; Ruijie Meng, National University of Singapore, Singapore, mengrj.cs@gmail.com; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@nus.edu.sg.



ACM 2994-970X/2026/7-ARTFSE157
https://doi.org/10.1145/3808164

One promising approach to attain strong, machine-checkable evidence about program properties is formal verification, which has been pivotal in certifying high-assurance systems for decades, such as microkernels [32], compilers [36], and databases [44].

Formal verification techniques commonly (1) reduce programs and specifications to logical formulas as proof obligations, and (2) discharge those obligations to automatic or interactive theorem provers (ITPs). Common ITPs, such as Isabelle/HOL and Rocq/Coq, automatically reduce the proof goals by applying user-supplied proof steps called *tactics*. Unfortunately, formal verification sees limited adoption in production systems, as it is an extremely time- and skill-intensive undertaking. Specifically, (1) capturing program behaviors formally requires significant efforts in manually annotating specifications and crafting loop invariants, and (2) generating a mathematically rigorous proof to validate verification conditions demands a high degree of expertise. For instance, the verification of the seL4 microkernel [32] required 22 person-years of effort, while the proofs for the CompCert compiler took 6 person-years and 100,000 lines of Rocq code—eight times longer than the implementation itself [36]. Most of the specifications and proofs are meticulously crafted by the developers. Recently, advances in Large Language Model (LLM) agents have been transformative across various disciplines. In particular, LLMs have demonstrated remarkable capabilities in both code understanding [4] and general mathematical reasoning [25]—two foundational capabilities required by program verification. In this work, we thus ask the following question: *Can LLM agents be leveraged for automatic and end-to-end verification of real-world programs?*

State-of-the-art and what is needed. Since there have been approaches to proof automation that leverage LLMs [41, 65, 67] and other machine-learning-based techniques [55, 57, 76] — let us understand the need for a new approach. First of all, it is reasonable to use machine learning models to learn and predict individual proof steps or tactics in a proof. This has been accomplished in several works, including Proverbot9001 [55]. It is also possible for LLMs to generate whole proofs, albeit potentially incorrect ones, and then have a repair step to correct them, if possible—an approach studied in PALM [41]. Rango [67] goes one step further. Given a sequence of proof steps, it uses an ITP for yes/no validation of the proof steps, thereby allowing the trial of different tactics.

An agentic approach. These approaches all use LLMs, but are not agentic. We aim to develop an approach where an LLM agent can understand the proof structure and autonomously seek help from a theorem prover while constructing the proof. Thus, apart from using a theorem prover for validation of proof steps, the agent can actively collaborate with a theorem prover to prove a program property. As such, we develop a proof automation agent AutoRocq which acts as an *interpreter* of a proof derivation, and collaborates with a theorem prover to extend/improve it. Specifically, AutoRocq *autonomously* interacts with the theorem prover ahead of tactic prediction to retrieve relevant lemmas in the context, and generates tactics smartly guided by tree-shaped proof representations. It also incorporates feedback from the interactive theorem prover and proof histories to refine tactic generation.

Evaluation. We thoroughly evaluate our approach on existing mathematical lemmas [67, 76], as well as verification conditions systematically extracted from SV-COMP programs [8]. These program-derived lemmas capture intricate code logic and properties, and are more representative of program verification tasks. We compare AutoRocq against five baselines, namely Rango [67], PALM [41], Copra [65], QEDCartographer [57], and Proverbot9001 [55], and results show that AutoRocq significantly outperforms these state-of-the-art approaches. Specifically, AutoRocq is capable of proving 48.0% mathematical lemmas and 30.9% verification lemmas, exceeding baseline approaches by 15.2% to 172.8% on mathematical lemmas, and by 10.6% to 204.6% on verification lemmas. Among the successes, 168 lemmas (140 mathematical lemmas and 28 verification lemmas) are uniquely proved by AutoRocq, due to its agentic design and access to proof contexts. We

also conduct a case study to verify the code in Linux kernel modules (i.e., memory management utilities), where AutoRocq automatically verifies 12 lemmas for the function correctness property, compared to only 2–10 lemmas verified by baseline approaches.

Contributions. In summary, the contributions of this paper are as follows.

- We build and make available[1] the first proof automation agent, AutoRocq, which is highly effective in automatic proof generation. It acts as an interpreter of proof representations and actively collaborates with the Rocq prover to construct proofs.
- We showcase the feasibility of automatic and end-to-end verification with LLM agents. AutoRocq is evaluated on the widely used SV-COMP programs in software verification, as well as Linux kernel modules.
- We conduct an empirical study to demonstrate that context- and structure-awareness can help LLM agents reason about software *formally*.

2 Background

2.1 Formal Program Verification

Formal program verification uses mathematically rigorous methods to prove that a program satisfies its specifications. Unlike traditional testing, which only checks specific executions, formal verification provides logical guarantees for all executions within a given semantic model. A prominent paradigm is *deductive verification*, where a program is annotated with formal specifications such as pre/postconditions and loop invariants. These annotations define *formal contracts* for the intended behavior of the program, and the annotated code is translated into logical *proof obligations*. If all proof obligations are discharged, the program is verified to satisfy its specifications.

To facilitate deductive verification, a variety of tools have been developed, including Dafny [35, 48], Verus [34], Viper [49], and Frama-C [31]. These tools support annotating code, generating proof obligations, and attempting to discharge them. Automated solvers [7, 14] can be used to handle simple proof obligations (e.g., quantifier-free linear arithmetic [24]). More complex proof obligations—such as those involving non-linear reasoning, quantifier handling, or sophisticated inductive proofs—require human guidance. In such cases, *interactive theorem provers* (ITPs) are used, where auxiliary *lemmas* are introduced to capture intermediate properties, helping to structure and simplify the proofs of these obligations.

Rocq (formerly Coq) proof assistant [66] is a widely used interactive theorem prover for semi-automated validation of proof obligations. In Rocq, proofs are constructed top-down from a *proof goal*, i.e., the statement of a theorem or lemma. Users iteratively apply *proof tactics*, which transform the current goal into zero or more subgoals and guide the construction of a proof term. This proof term is then type-checked by Rocq's trusted kernel. A proof succeeds when no subgoals remain, and the resulting proof term serves as the formal certificate of validity. The exact sequence of tactics, called the *proof script*, can be replayed to reproduce the proof. Conceptually, a successful proof induces a *proof tree* rooted at the original goal, where nodes are subgoals and edges are tactic applications. This interactive workflow supports incremental and exploratory proof development with frequent inspection and backtracking.

2.2 Machine Learning for Proof Automation

Proof automation aims to automatically synthesize proof scripts that discharge proof obligations, addressing the last mile of program verification. Recent advances in AI have led to a growing body of learning-based approaches. Early works [21, 55, 56, 76] often formulate this task as sequence generation, using representations of syntactic constraints [76], proof states [21], and fine-grained

[1]The tool is released at https://github.com/NUS-Program-Verification/AutoRocq.

proof contexts [56] to generate tactic sequences. However, successfully applying a tactic does not always mean the proof has progressed, since it may produce equivalent or even harder subgoals. This limits standalone prediction models that focus only on next-step tactic generation without a reliable notion of global progress. To address this, another line of work [9, 57] combines tactic prediction with search strategies. For example, QEDCartographer [57] extends Proverbot9001 [55] with a reinforcement-learning-based tactic selection model.

On the other hand, Large Language Models (LLMs) have emerged as a promising alternative due to their strong mathematical reasoning and high-level understanding. A recent study [41] shows that while LLMs can capture the high-level structure of proofs, they often make basic mistakes such as invalid references. Building on this observation, PALM first prompts the LLM to generate a complete proof script, and then applies deterministic repair mechanisms together with CoqHammer [13] to fix errors. Copra [65] adopts an in-context learning approach for Lean/Coq, iteratively proposing tactics, executing them in the ITP, and using error feedback to construct complete proofs under a query budget. More recently, Rango [67] fine-tunes an LLM as a knowledge base for tactic generation. At each proof step, it retrieves relevant proofs and lemmas for the current proof state and then prompts the model to generate the next tactic.

These works have significantly advanced automated proof generation. However, most rely on models trained or fine-tuned on large proof corpora to guide tactic generation. A widely used dataset is CoqGym [76], which contains 71K human-written proofs spanning domains such as mathematics, hardware, and programming languages. While models trained on this dataset are effective for mathematical theorems and libraries, they are less effective for proofs about computer programs, which are often more complex in both structure and semantics (see Section 5.1.4).

3 Motivating Example

Although many automatic theorem-proving approaches have been proposed, to our knowledge, none of them prove lemmas in an *agentic* fashion. To motivate our agentic approach, AutoRocq, we use a concrete example shown in Figure 1. Figure 1(a) shows the proof obligation (`wp_goal`, lines 1-7) extracted from verifying the *function correctness* property of a real-world program `cggmp2005b` in SV-COMP [8], along with the complete proof (lines 9-23) generated by AutoRocq. The proof tree representation that guides AutoRocq's proving process is visualized in Figure 1(b). This example proof goal requires proving that "*i1 = 10%Z*" (line 7) holds for integer-typed (`Z`) variables `i1` and `i` under a complex set of hypotheses involving modular arithmetic, inequalities, and type constraints (lines 2-6). The hypotheses capture program semantics including path conditions (e.g., "*i <= 10%Z*") and non-overflow constraints ("*(-2147483648%Z)%Z <= x*"), all discharged by Frama-C during automated verification (see Section 5.1.3 for more details). This example represents a typical lemma derived from real-world verification tasks. It captures intricate program logic and thus requires sophisticated reasoning about nested quantifiers, integer arithmetic, and logical constraints to handle. The proof tree in Figure 1(b) illustrates how AutoRocq systematically decomposes the problem through strategic tactic application, including case analysis with `destruct` that creates multiple proof branches, and maintains rich contextual information throughout the proving process. Such complex lemmas pose significant challenges to existing approaches (e.g., [41, 55, 57, 67]), and none of them can prove this lemma during our experiments. We highlight these challenges to motivate AutoRocq's agentic design in the following.

Verbose and Crowded Context. The local context of `wp_goal` carries rich semantic information—six hypotheses are present in the statement simultaneously. These hypotheses include conjunctions, disjunctions, inequalities, and type constraints. Conventional approaches, which mostly employ

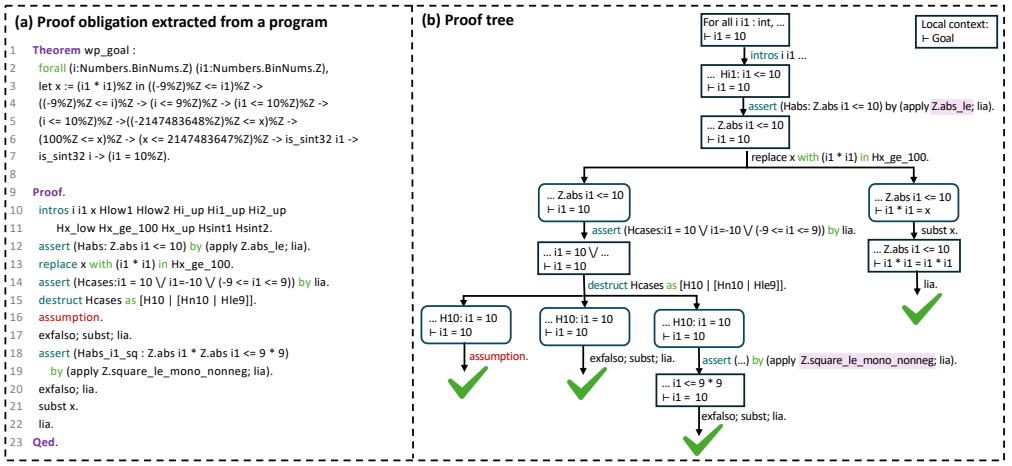


Fig. 1. (a) Proof obligation `wp_goal` extracted from `benchmark52_polynomial` in SV-COMP [8] and its proof generated by AutoRocq, with (b) the proof tree constructed during proving, where lemmas highlighted are retrieved from the global context autonomously by AutoRocq.

naive or similarity-based retrieval, could easily get a surfeit or an insufficiency of contextual information. For example, before proof generation, PALM [41] simply retrieves *all* relevant premises from the global environment. In this case, it would be overwhelmed by hundreds of existing premises on integer arithmetic, most of which are irrelevant in the context. Conversely, Rango [67] retrieves lemmas based on lexical similarities, an imprecise metric that often fails to identify semantically relevant lemmas. This is why Rango fails to prove the example: it cannot pinpoint the specific contextual assumptions on integer bounds and arithmetic properties required for the proof.

In contrast, AutoRocq employs an agentic context search that *autonomously* decides, based on the current proof state, if it should make a tactic prediction, or if it should gather more context. In the latter case, we facilitate its requests for context with fine-grained query commands (see Table 1). For instance, it can leverage "*Search (Z.abs _ <= _).*" to fetch all lemmas and definitions that involve absolute values (`Z.abs`) and match the specified wildcard pattern. As a result, the lemma `Z.abs_le` (line 12 in Figure 1(a)) is retrieved, which is crucial for reasoning about absolute values in the context. This autonomy enables the agent to retrieve additional context from the theorem prover *on demand*, supporting it to reason about lemmas with rich contexts. It is worth noting that our contribution is not on how to extract certain terms or patterns in the theorem prover; rather, it is to build an agentic system that knows *when* to search and *what* patterns to search intelligently.

Understanding of Proving Progress. A complicated lemma such as `wp_goal` requires a delicate sequence of tactics to prove. When generating such sequences, however, existing LLM-based methods lack structured representations of the proof process: they rely on simple goal lists rather than structured proof trees. This textual and linear representation of the proof state makes them overly focus on predicting a single tactic without a holistic view of the proving progress. As a result, they often fail to adapt to the evolving state of the partially generated proof. AutoRocq explicitly maintains a well-structured proof tree representation, as shown in Figure 1(b). It captures the hierarchical structure and dependencies of the proof derivation. The proof tree is built gradually as the proving advances. For instance, AutoRocq carefully decomposes the original goal into multiple subgoals with the `destruct` tactic, resulting in the branching in Figure 1. This structure-aware representation enables strategic reasoning about proof progress, tactical decisions, and effective

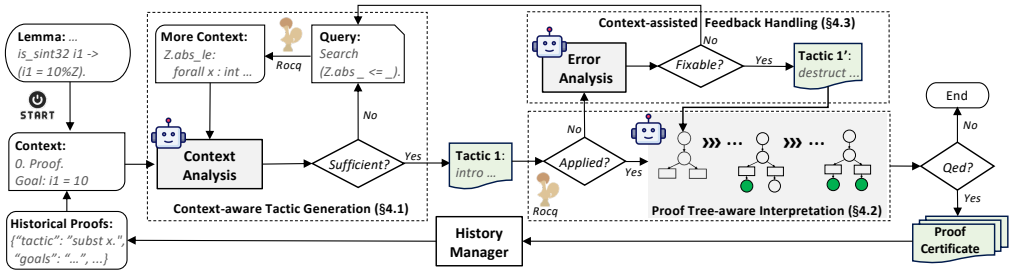


Fig. 2. Overview of AutoRocq, where components involving decision-making by LLMs are highlighted.

tracking of proof dependencies. As such, our proof agent is able to *interpret* the proof derivation from a higher level, thus achieving more effective proof generation.

Harnessing the Feedback. Developing proofs, especially for intricate lemmas, is a trial-and-error process [59], and interactive theorem provers are designed to provide timely feedback to guide the proving process. Nonetheless, existing approaches hardly leverage this opportunity to refine their tactics. For example, PALM [41] only performs deterministic repairs to failed tactics, whereas Rango [67] takes only binary signals from the proof assistant and retries the prediction again. AutoRocq implements an effective feedback mechanism that incorporates the feedback from the prover to refine individual tactics. In addition, it also recognizes persistent errors from the history, and conducts autonomous context searches to collect additional context progressively. This process allows our agent to learn continuously from both failures and successes. In this example, despite 48 failed tactic applications (59 attempts in total) during proving, AutoRocq eventually recovers and synthesizes a complete proof in a short time (i.e., 156.7 seconds).

Collectively, we have an agentic proof system that can autonomously decide when and how to incorporate additional contexts and adapt strategies based on the feedback. A high-level interpretation of the proof derivation from the expressive proof tree supports such an agency. In principle, this process is analogous to how an expert human prover would approach the task. In Section 4, we will explain each of these components in detail.

4 Proof Automation with AutoRocq

Overview. Figure 2 shows the overall workflow of AutoRocq. At a high level, it is an LLM agent that takes autonomous actions to carry out a human-like proving process. Specifically, given a lemma and the initial context (i.e., proof state), AutoRocq performs an agentic context analysis to generate a tactic if the context is sufficient, or a query command if additional context is needed (Section 4.1). Then, AutoRocq generates a sequence of tactics by interpreting formal proof representations (i.e., proof tree), analyzing them, and communicating with the Rocq proof assistant autonomously (Section 4.2). During communication, a context-assisted feedback loop is established to either refine an incorrect tactic application or provide additional context to the LLM if the errors persist, and a on-the-fly learning component is proposed to use successful proofs to help prove new lemmas (Section 4.3). The output of a successful proof is a certificate consisting of a sequence of tactics that can be replayed in the interactive theorem prover to verify the proof.

4.1 Context-Aware Tactic Generation

Previous works [37, 41, 67, 78] have demonstrated that additional information helps LLMs generate valid tactics. Such additional information, which we refer to as the *context*, includes lemmas, definitions, and existing proof steps that are relevant in constructing the proof. Existing approaches, however, tend to augment LLMs with blindly selected context – typically a long list of premises [41]

Table 1. List of supported context query commands in AutoRocq.

Command	Description	Output
`Search <pattern>`	Search for a pattern (e.g., a term).	A list of matched declarations.
`Print <identifier>`	Print the definition of an identifier.	Full definition of the identifier.
`Locate <identifier>`	Locate where an identifier is defined.	Path or location of the identifier.
`About <identifier>`	Show information about an identifier.	Type and summary of the identifier.
`Check <term>`	Type-check a term or expression.	Type of the term or expression.

Table 2. Examples of context query commands and their outputs.

Example Command	Output (reduced version)
`Search (Z.quot _ _).`	`Zquot.Z_mult_quot_le: forall a : int, 0 ÷ a = 0` `Z.quot_unique_exact: forall a : int, a ÷ 0 = 0`
`Print is_sint32.`	`is_sint32=fun x:int => -2147483648 <= x < 2147483648...`
`Locate to_sint64.`	`Constant main_assert.to_sint64.`
`About Z.abs.`	`Z.abs : int -> int.`
`Check why_decidable_eq.`	`why_decidable_eq : forall x y : bool, x = y + x <> y`

or existing proofs [67] ordered by a predefined similarity metric – without a good understanding of the current proof state or the progress of the proof. We argue that, before suggesting a new tactic, an intelligent proof system should be able to analyze the current proof state and the progress of the proof to decide if more context is needed. This system should carefully determine *what* context information is needed and *when* to add it, so that unnecessary noise is reduced to a minimum. To support this autonomy, an agentic context-aware search mechanism is needed. Such a context-aware approach can significantly reduce the amount of context added to the LLM, which could help the LLM focus on the current proof state and generate a more accurate tactic.

To achieve agentic context search, we couple a context analysis module with an agent that autonomously decides when and how to query the proof assistant's database. The agent performs continuous context analysis: if the current context is deemed sufficient, the agent directly generates a tactic to advance the proof; if a key piece of information (e.g., the definition of a term `is_sint32` or a proved lemma in an imported library) is missing, the agent can submit a context query to retrieve it. To retrieve the context, it constructs a precise *Search* command, such as "*Search is_sint32*". This command is forwarded to the Rocq proof assistant, which searches its currently loaded libraries for matching identifiers, lemmas, and definitions. The results of this query are then incorporated into the LLM's context, enriching it for subsequent attempts at tactic generation.

The supported context query commands are listed in Table 1 and the examples of query output are shown in Table 2. By default, AutoRocq supports the standard query commands available in the Rocq GUI (CoqIDE), including *Search*, *Print*, *Locate*, *About*, and *Check*. These commands enable AutoRocq to retrieve critical context information on demand, significantly aiding the proof process. Among these, the *Search* command is the most frequently used (see Section 6.3). Its versatility is the key: it allows for discovery based on name patterns, type signatures, and existing lemma names [54]. This makes *Search* the primary tool for pinpointing relevant declarations without knowing their exact names or forms.

Agentic Context Analysis

Prompt: "Analyze the current {proof tree} and Top-5 {historical tactics}. If sufficient information is available, generate a tactic to proceed. Otherwise, output a query command to retrieve additional context."
Response: `Search (Z.abs _ <= _).`

4.2 Proof Tree-Aware Interpretation

The key component empowering AutoRocq's agency is an expressive proof tree representation that enables high-level interpretation of the proof derivation. Compared to the textual or linear representation of proof states (e.g., a list of goals), the explicit tree structure conduces understanding of the proving process. Formally, we define a proof tree as follows [28, 33].

Definition (*Proof Tree*). Let Σ be the proving context[2], G be the initial proof goal, and S be a sequence of tactics that transforms G under Σ. A proof tree $T(\Sigma, G, S)$ is inductively defined as:

- Each node N_A corresponds to a goal $\Sigma \vdash A$, i.e., a statement A to be proved under Σ.
- The root node is $N_G := \Sigma \vdash G$.
- When a tactic $\tau \in S$ is applied to a node $N_A := \Sigma \vdash A$, it either (1) generates zero subgoals, in which case N_A is a leaf, or (2) generates n subgoals/nodes $\{N_{A_i} := \Sigma \vdash A_i \mid i = 1, \ldots, n\}$, and creates edges $N_A \rightarrow N_{A_i}$.
- A node is a leaf if it is directly derivable under Σ, i.e., no further subgoals are produced.

Note that the S may or may not be a complete tactic sequence that proves G. Intuitively, a proof script S induces a proof tree T in its enclosing context Σ, rooted at the original proof goal G. Each node represents a subgoal to be proved, and a tactic application is represented as an edge. Leaves are subgoals that are trivially true or could be easily proved with a single tactic. The goal is complete when all the sub-goals in the leaves are proved.

AutoRocq explicitly maintains a proof tree as it progresses in tactic generation. After a successful tactic application, it examines the open subgoals from the Rocq proof assistant. If the tactic successfully proves the current subgoal (i.e., a leaf node), AutoRocq marks it as closed, and shifts its focus to another residual subgoal. When the tactic creates multiple subgoals, it creates these nodes in the tree and determines one of the new subgoals to work on. In this way, the proof tree is updated by extending the current node. On the other hand, if the tactic fails, our agent remains at the current node on the proof tree for further attempts. During the entire proving process, the proof states in subgoals and tactic applications are continuously updated until a complete tree is constructed. Figure 1(b) shows an example of a proof tree after all tactics were applied in the motivating example presented in Figure 1(a). From the tree, we could see that four subgoals were generated in total, and the lemmas with a violet background were retrieved from our agentic context search. The proof is completed when all leaves are closed.

Whenever the proof tree is updated, AutoRocq *interprets* the tree structure comprehensively to reevaluate the current progress. It does so by traversing the proof tree to identify the open subgoals and how they relate to the existing proof structure. With a holistic view of the entire derivation process, it then proceeds to generate a sequence of tactics iteratively.

We note that our key contribution is not about formulating the proof tree structure. Instead, our insight lies in the realization that high-level interpretation enables LLM agents to carry out autonomous decision-making and actions. The benefits provided by our proof tree representation are twofold. First, it captures the hierarchical structure of the proof, including the relationships between tactics and subgoals, which provides a richer context for interpreting the proof process. Second, it allows AutoRocq to systematically break down the proof into smaller subgoals and tackle them one by one, which is aligned with how humans approach proving [58, 59].

4.3 Context-Assisted Feedback Handling

To emulate a trial-and-error feedback loop, we augment AutoRocq with feedback handling mechanisms to collaborate with the Rocq ITP. Based on the error that occurs, AutoRocq employs two

[2]Here, Σ captures both the global environment and the local context, as they are not differentiated in our context search.

strategies during tactic generation: (1) revising the tactic based on the error message when a single error occurs, and (2) initiating a context search to retrieve additional context when persistent errors occur. In addition, AutoRocq takes positive feedback from past successes. We delegate each mechanism to a subsection as follows.

4.3.1 Handling a Single Error with Strategic Fixing. When a tactic fails to apply to a subgoal, the ITP provides an error message indicating the reason for the failure. As shown in the following example, AutoRocq captures this error message and feeds it back to the LLM, along with the current proof tree. The LLM then analyzes the error message and the context to generate a revised tactic to fix the issue. Subsequently, the revised tactic is applied again, and the proof tree is updated accordingly.

Agentic Error Handling of a Single Error

Prompt: "The previous {tactic} failed to apply to the current subgoal with the following {error message} from Rocq: {error message}. Analyze the error and generate a corrected tactic."
Response: `destruct Hcases as [H10 | [Hn10 | Hle9]].`

4.3.2 Handling Persistent Errors with Context Search. If the same error persists after several repair attempts (which is treated as *not fixable* as shown in Figure 2), AutoRocq recognizes that the current context may be insufficient to resolve the subgoal. In such cases, AutoRocq initiates a context search by issuing a query command to the ITP to retrieve additional context information, similar to the process described in Section 4.1. The retrieved context is then incorporated into the LLM's input, and the LLM generates a new tactic based on the enriched context. This context-assisted feedback loop continues until the subgoal is successfully solved or a predefined termination condition is met (e.g., maximum number of attempts). For example, the following shows how AutoRocq handles persistent errors. Given the failed tactics and the proof tree, our agent returns a query command to search for what is defined in the integer quotient from the `Z.quot` module.

Agentic Error Handling of Persistent Errors

Prompt: "The agent has repeatedly generated {failed tactics} for the current subgoal multiple times. Analyze the current {proof tree} and determine what additional context is needed to proceed. Output a query command to retrieve the necessary context information."
Response: `Search (Z.quot _ _).`

4.3.3 Managing Historical Proofs for On-the-fly Learning. AutoRocq learns on the fly by maintaining an archive of successful proofs. This archive is accumulated throughout a proving campaign, and AutoRocq adapts its strategy when proving new lemmas. Specifically, when a lemma is successfully proved, AutoRocq stores comprehensive tactic history records that capture the complete context and evolution of each proof step. Each record contains the applied tactic, the proof goals before and after tactic application, the available hypotheses and their changes, along with metadata including the theorem name, and the tactic ID within the proof sequence. This rich historical information serves multiple purposes: (1) it enables AutoRocq to learn from successful proof patterns and reuse effective tactic sequences in similar contexts, (2) it provides detailed examples for context-aware tactic generation by showing how specific goals were transformed, and (3) it supports the feedback handling mechanism by offering concrete instances of successful problem-solving strategies. By maintaining this detailed provenance of proof construction, AutoRocq can build a knowledge base of proven tactics that enhances its capability to tackle new lemmas with similar structural patterns or mathematical properties (the experiment results presented in Section 6.3 also support our claim).

4.4 Implementation

We implemented AutoRocq in approximately 8k lines of Python code with a modular architecture that separates the core proving logic, theorem prover interface, and supporting utilities. The main Rocq backend is implemented using the CoqPyt (v1.0.0) library [10] to interact with the Rocq proof assistant. The implementation follows an iterative workflow as shown in Figure 2. The modular design enables easy extension and maintenance while supporting the three key components of our approach: context-aware tactic generation, proof tree management, and feedback handling. Historical proof data is stored in JSON format to enable learning from successful proof patterns for future lemmas. For the LLM backend, we use GPT-4.1 as the backbone model for all LLM interactions. We set the temperature to 0 for reproducibility.

5 Experimental Setup

To evaluate the effectiveness of AutoRocq on program verification tasks, we seek to answer the following research questions (RQs):

RQ.1: Is AutoRocq effective at proving mathematical lemmas from CoqGym?
RQ.2: Is AutoRocq effective at proving lemmas from program verification tasks?
RQ.3: How does each component of AutoRocq contribute to its effectiveness?
RQ.4: How do the proofs generated by AutoRocq compare with human written proofs?

In this section, we first present our experimental setup. We then analyze the detailed results in Section 6. We also conduct case studies on verifying individual Linux kernel modules in Section 7.

5.1 Preparation of Benchmark Lemmas

5.1.1 Lemmas from Existing Datasets. CoqGym [76] and CoqStoq [67] are two well-known datasets for evaluating neural theorem provers in Rocq, and have been extensively studied in the community [9, 41, 67]. For fair and comprehensive evaluation, we include only the lemmas that come from the *intersection* of CoqStoq and CoqGym testing sets. This is because other projects from CoqGym's test set are included in CoqStoq's training set, and thus may cause inflated results. Specifically, these include seven projects (i.e., `dblib`, `zfc`, `hoare-tut`, `huffman`, `buchberger`, `PolTac`, and `zorns-lemma`), which represent common mathematical and human-written lemmas, such as logical tautology, coding algorithms, and theory formalizations. In total, there are 1,717 mathematical lemmas selected to evaluate comparative approaches.

5.1.2 Lemmas for Program Verification Tasks. While CoqGym [76] and CoqStoq [67] provide a valuable corpus of human-written theorems, they are drawn primarily from mathematical libraries and do not capture the complexity of program verification lemmas found in real-world software. Since AutoRocq is designed to automate program verification, its evaluation requires a benchmark of lemmas extracted from real programs. To our knowledge, no such dataset currently exists, leaving a critical gap in the program verification field.

To address this gap, we construct a new benchmark by systematically extracting lemmas from real-world C programs. This benchmark enables the evaluation of AutoRocq and facilitates future research in automatic program verification. We source our subject programs from SV-COMP [8] for two key reasons: their widespread adoption in the formal verification community ensures they represent a diverse set of C language constructs, and their inclusion of ground-truth specifications provides meaningful proof obligations. Our selection focused on the most common property types in SV-COMP: functional correctness, defined by the unreachability of error calls, and non-overflow. The final criteria mandated that a program must (1) be verified for both properties and (2) exhibit deterministic behavior (e.g., no multi-threading). Applying these criteria yielded a final benchmark

of 131 C programs from SV-COMP. The programs have an average of 43.06 lines of code, with the largest (the utility `basename` in BusyBox [1]) comprising 428 lines.

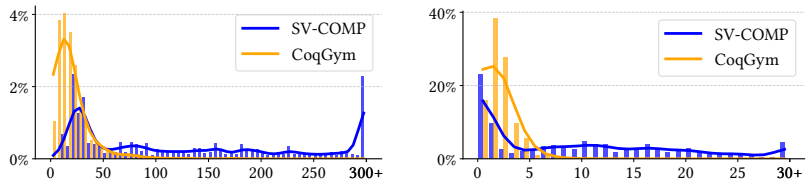
5.1.3 Lemma Extraction Methodology. We designed a systematic method to automatically extract lemmas from the selected SV-COMP programs. We focus on generating non-trivial lemmas that necessitate reasoning about program semantics, as opposed to simple lemmas that can be discharged with basic tactics (e.g., `auto.`). Our approach utilizes Frama-C [31] to generate the necessary proof obligations from the program code.

Technically, we first use the RTE plug-in [3] to annotate the source code with formal contracts in the ANSI/ISO C Specification Language (ACSL), including preconditions and postconditions. As Frama-C cannot automatically infer loop invariants, we address this by employing LLMs to generate candidate invariants based on the loop's context and structure. Each candidate is validated through property testing with the Eva plug-in [2]; unsuccessful candidates are refined iteratively until a verifiably correct invariant is established. Note that while property testing can falsify incorrect invariants by discovering counterexamples, it cannot formally prove the correctness of an invariant – that it holds for all possible program executions. We justify this design choice as loop invariant inference is a long-standing, challenging problem [23, 43] that is orthogonal to our core focus on proof automation. Once the code is fully annotated and verified, we employ Frama-C's WP plug-in [22] to generate proof obligations. These obligations are automatically translated into lemmas, which can be discharged in the Rocq proof assistant. The overall framework completes an end-to-end automated workflow from C programs to verifiable Rocq lemmas.

As a result, we collected 641 lemmas extracted from the verification of two types of target properties: non-overflow and functional correctness. Since both property types often require reasoning about loops, a significant portion of the generated proof obligations are related to loop invariants. These are essential intermediate lemmas needed to conclude that either a non-overflow or a functional correctness property holds for the entire program. Consequently, the 641 lemmas comprise three categories, reflecting their role in the proof: (1) non-overflow lemmas (203, 31.7%): direct proof obligations for non-overflow properties; (2) functional correctness lemmas (153, 23.9%): direct proof obligations for functional properties; and (3) loop invariant lemmas (285, 44.5%): Supporting lemmas required to prove the loop invariants necessary for establishing both non-overflow and functional correctness properties. Together, these lemmas represent a diverse set of challenging proof obligations that arise in real-world program verification tasks.

5.1.4 Comparison of Extracted Lemmas and Existing Lemmas. To quantify the complexity and difficulty of verification lemmas extracted in Section 5.1.3, we compare them against existing mathematical lemmas pooled in Section 5.1.1. To do this, one commonly used proxy is the difficulty of proving a lemma, since more sophisticated lemmas generally necessitate longer proofs involving more cases and derivation steps. Unfortunately, this is infeasible for our purpose, since no ground-truth proofs are readily available for our extracted lemmas. As such, we directly examine these lemmas themselves. Specifically, we propose two metrics: (a) term count: the number of terms (variables, quantifiers, operators) to gauge structural complexity, and (b) hypothesis count: the number of assumptions to measure the richness of the semantic context.

In Figure 3, we report the complexity distribution of the lemmas by measuring the percentage of lemmas (y-axis) with a given complexity metric (term or hypothesis count, x-axis). For both metrics, the lemmas from CoqGym (orange) densely concentrate on the left end of the axis, suggesting uniformly lower complexity. In contrast, lemmas extracted from SV-COMP programs (blue) exhibit a more even distribution, with a lower peak and a long right tail, reflecting generally higher complexity. Concretely, 48% of these lemmas from SV-COMP programs consist of >100 terms, and 52% of them are associated with >7 hypotheses; < 1% mathematical lemmas satisfy either constraint.



(a) Term Count (b) Hypothesis Count

Fig. 3. Histogram of different complexity metrics: lemmas from SV-COMP programs (blue) vs. CoqGym (orange). Less than 1% of lemmas from CoqGym involve more than 100 terms or 7 hypotheses.

The stark contrast across both metrics suggests that the obligations extracted from real-world programs, which often capture intricate logic in the code, tend to have much higher complexity than their benchmark counterparts. Below shows an example lemma from `hoare_tut`, a project in CoqGym. It states that the non-equality function `Zneq_bool` correctly returns false only if `x = y`. It is much simpler than lemmas from program verification tasks, as shown in Figure 1(a).

```
Lemma Zneq_bool_false: forall x y, Zneq_bool x y=false → x=y.
```

In summary, we use the two benchmark sets for our evaluation in Section 6, namely (1) 1,717 mathematical lemmas from the intersection of the test partition of CoqGym [76] and CoqStoq [67], and (2) 641 verification-derived lemmas from SV-COMP programs [8]. In total, we have 2,358 lemmas across different domains.

5.2 Comparative Baselines

We compare AutoRocq with state-of-the-art tools developed for proof automation in Rocq. We choose five baselines, namely Rango [67], PALM [41], Copra [65], QEDCartographer [57] (or QEDC for short), and Proverbot9001 [55] (or P9001 for short). These tools have shown exceptional results on proof automation tasks, and they employ different machine-learning techniques: standard LLMs, fine-tuned LLMs, reinforcement learning, and recurrent neural networks (RNNs), respectively. AutoRocq, PALM, and Copra, which invoke closed-source LLM through APIs, use GPT-4.1 as the backend with temperature 0. For the other three baselines, we use the pre-trained weights made available in their artifacts. These include a fine-tuned version of DeepSeek-Coder used by Rango, a reinforcement-learning-based tactic selection agent employed by QEDC, and a custom tactic prediction network in P9001. They also have access to a single Nvidia A40 GPU with 48GB of VRAM if needed. We use the default settings for all the tools. Rango and Copra times out after 10 minutes, and QEDC is limited to 512 generation steps. PALM and P9001 are executed until completion. We set up all the baselines in Docker on Ubuntu 22.04. Other than PALM, which uses older Rocq 8.12 for compatibility reasons, all the other tools are configured with Rocq 8.18. We delay the discussions on the use of different models and Rocq versions to Section 6.5.

6 Experimental Results

6.1 RQ.1: Efficacy of AutoRocq on Mathematical Lemmas

For this research question, we investigate AutoRocq's effectiveness in proof generation on mathematical lemmas. We report the number of proved lemmas from each tool and a detailed breakdown in Figure 4. The bar plot (left) shows the total number of lemmas successfully proved by each tool, whereas the Venn diagram (right) presents the detailed breakdown of the proved lemmas. We note

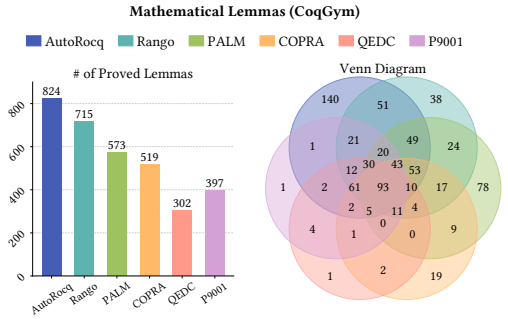


Fig. 4. [RQ.1] Mathematical lemmas from CoqGym proved by each tool and their breakdown.

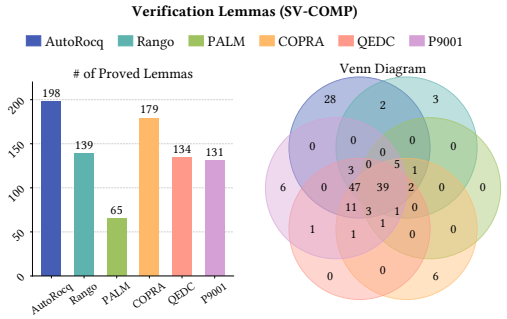


Fig. 5. [RQ.2] Verification lemmas from SV-COMP programs proved by each tool and their breakdown.

that PALM's results may not reflect its full capability, as several lemmas in the benchmarks are incompatible with Rocq 8.12.

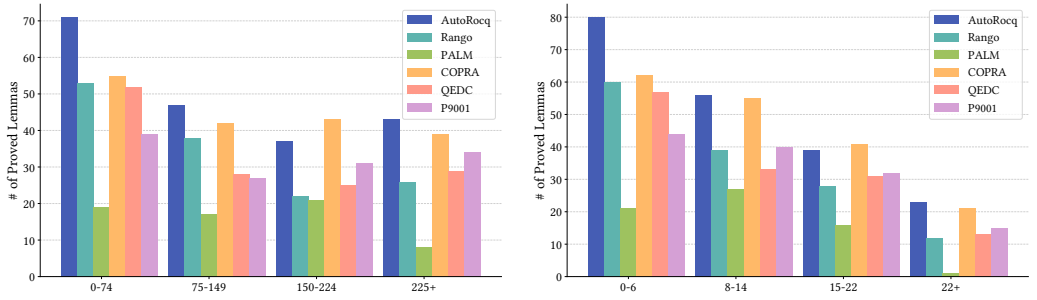
The bar plot (Figure 4, left) shows that, across all 1,717 lemmas, AutoRocq successfully proves 824 (48.0%) of them, outperforming other tools by a large margin of 15.2% (compared with Rango) to 172.8% (compared with QEDC). Among the baseline tools, Rango performs the best on this dataset, synthesizing proofs for 715 (41.6%) of the lemmas on CoqGym. The two baselines using GPT-4.1 as the backend model, PALM and Copra, can both prove more than 500 mathematical lemmas. The Venn diagram (Figure 4, right) also demonstrates that AutoRocq proves the most lemmas (140) uniquely. Our tool is followed by PALM and Rango, which account for 78 and 38 uniquely proved lemmas, respectively. Copra is able to find proofs for 19 lemmas uniquely. The results suggest that AutoRocq is highly effective in automatic proof generation.

6.2 RQ.2: Efficacy of AutoRocq on Verification Lemmas

More importantly, we study complex proof obligations that represent real verification workflows, and present the results in Figure 5, with the bar plot (left) and its breakdown in the form of a Venn diagram (right). Overall, AutoRocq manages to prove 198 lemmas (30.9%) from the benchmark, outperforming the best baseline, Copra, by 10.6%. Results produced by Rango, QEDC, and P9001 lie in close proximity to one another, with each producing proofs for around 135 lemmas. AutoRocq outperforms them by 42.4%, 47.8%, and 51.1%, respectively. The number of lemmas proved by AutoRocq is also 204.6% higher than that of PALM.

The Venn diagram (Figure 5, right) shows that AutoRocq generates proof scripts for 28 unique lemmas (owing to its agentic workflow) that *no* other tool can prove. In contrast, *no* other tool is able to generate proofs uniquely for more than 6 lemmas. In fact, only 19 lemmas proved by *any* other baselines elude AutoRocq! We note that there is a large overlap among the successes of baseline tools. 113 lemmas are proved commonly by at least three of the baselines, suggesting a strong convergence in capabilities among them.

Compared to the results in Figure 4, the verification lemmas indeed pose a greater challenge to *all* neural theorem provers (including AutoRocq), as evidenced by their lower success rates overall. The relative degradation in performance is the most significant for PALM, indicating that whole-proof generation scales poorly as the lemmas grow in complexity. Similarly, Rango is the best-performing baseline on mathematical lemmas, but is outperformed by Copra in the SV-COMP benchmark. This performance decline can be attributed to a distribution mismatch: its underlying model was fine-tuned primarily on CoqStoq, and consequently struggles when encountering the more complex verification lemmas that lie outside this training data. In *both* benchmarks,



(a) Breakdown by term count. (b) Breakdown by hypothesis count.

Fig. 6. [RQ.2] # of verification lemmas proved from SV-COMP programs: a breakdown by lemmas' complexity.

AutoRocq proves the most lemmas *and* has the most unique successes, strongly suggesting its efficacy and robustness in automated proof generation.

Answer to RQ.1&2: Overall, AutoRocq is more effective in proving both mathematical and verification-derived lemmas, outperforming baselines by 15.2% to 172.8% on CoqGym and 10.6%–204.6% on SV-COMP. Notably, AutoRocq proves 168 lemmas (140 on CoqGym and 28 on SV-COMP) that none of the other approaches can prove.

To better understand the remarkable efficacy of AutoRocq, we further correlate the number of successfully proved SV-COMP lemmas with their complexity. Figure 6 details the breakdown of successful attempts from different tools, grouped into five complexity buckets. We present the breakdown for both the term count (Figure 6(a)) and the hypothesis count (Figure 6(b)) in the original goal. Buckets on the right correspond to lemmas containing more terms/hypotheses, and thus are more structurally and contextually complicated. Results reveal an emerging pattern: as the original goal becomes more verbose and context-rich, the number of successfully proved lemmas generally decreases. However, AutoRocq retains its relative effectiveness in proving longer, more complex goals, resulting in an almost uniform lead across different complexity levels. Among the baselines, Copra scales with increased complexity better, and even manages to prove slightly more lemmas than AutoRocq on moderately difficult lemmas (150–224 terms or 15–22 hypotheses). This further demonstrates the advantage of feedback-guided proof generation in program verification.

Additionally, we report the breakdown by the categories of lemmas, as shown in Figure 7. Results show that AutoRocq is particularly effective in proving non-overflow-related lemmas in the program, outperforming the best baselines by 37%. On lemmas related to loop invariants, AutoRocq is comparable with Copra and outperforms other baselines; whereas on lemmas specifying functional correctness, all approaches except PALM perform comparably. Results indicate that AutoRocq's performance is consistent across different properties, demonstrating its versatility and generality.

We also compare the efficiency of different tools in terms of the time taken to generate a successful proof on the SV-COMP lemmas. On average, AutoRocq takes 21.3 seconds to generate a successful proof, expending less time than other LLM-based approaches, namely Rango (105.5 seconds), PALM (45.4 seconds), and Copra (49.1 seconds). AutoRocq's result is also comparable to P9001 (22.6 seconds). Notably, QEDC finishes extremely fast (5.1 seconds) when it does succeed, thanks to its well-optimized tactic-prediction model for the tactic generation.

We hypothesize that AutoRocq's remarkable efficacy and efficiency stem from its agentic access to the proof context, strategic tactic fixing, and effective communication between the LLM

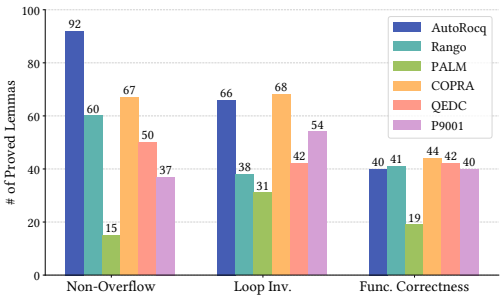


Fig. 7. [RQ.2] # of lemmas proved from SV-COMP programs: a breakdown by the category of lemmas.

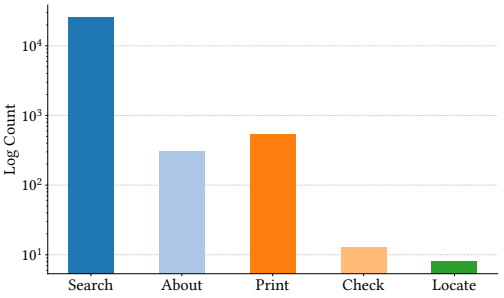


Fig. 8. [RQ.3] The log frequency of invocation for context search commands on SV-COMP programs.

and ITPs. These collectively enable the agent to discover and apply relevant facts and lemmas dynamically during the proof search. These capabilities could be particularly beneficial for complex proof obligations, which often require non-trivial reasoning steps and the application of specific lemmas or theorems. In the next research question (Section 6.3), we evaluate the design decisions of AutoRocq to gain a deeper understanding of the system.

6.3 RQ.3: Design Choices of AutoRocq

To better understand the interplay between AutoRocq's components, we evaluate how different design choices affect the overall effectiveness by implementing 6 variants of AutoRocq. We conduct the evaluation on 70 randomly sampled verification lemmas from SV-COMP programs. Specifically, we examine the following variant approaches:

- $\neg CS$: No context search (Section 4.1). Instead, tactics are directly generated through prompting.
- $\neg PT$: No proof tree awareness (Section 4.2). Instead, proof states are encoded as plain texts.
- $\neg EF$, $\neg HM$: No error feedback or history manager for the on-the-fly learning (Section 4.3).
- Err_1, Err_3, and Err_5: The maximum number of errors encountered before context search. We evaluate three settings of the maximum number of errors encountered before seeking context search. We found that the value 3 is the most effective setting, and is used in AutoRocq. This is expected, as more frequent context queries (Err_1) may confuse the agent with unnecessary information, whereas fewer queries (Err_5) may not supply sufficient context.

We report the success rates and AutoRocq's relative improvement, of different variants in Table 3. Results show that all variants only prove a fraction of lemmas compared to full-fledged AutoRocq. In particular, AutoRocq sees the most significant improvement from the ablative version $\neg CS$, which replaces the context search component with a direct request of tactic through prompting. This indicates that agentic access to the proof context contributes the most to our approach's effectiveness. Intriguingly, querying for contexts much more frequently (Err_1) or less frequently (Err_5) also affects the results negatively, suggesting that the amount of contexts supplied to the agent is paramount in practice. Removing other components in our approach, namely the proof-tree representation ($\neg PT$) or the feedback mechanisms ($\neg EF$ and $\neg HM$), each reduces the efficacy moderately by $\sim 10\% - 15\%$. We conclude that all components of AutoRocq are useful.

To visualize how AutoRocq conducts the context search, we summarize the frequency of invocation for different search commands provided by our framework (Table 1). We plot these frequencies in log scale, as shown in Figure 8. Statistics show that *Search* is the most frequently used command. Its primacy stems from its unique ability to solve the fundamental problem of discovery: it allows users to find relevant lemmas and theorems without prior knowledge of their names by searching for patterns. In essence, *Search* directly facilitates the core intellectual challenge

Table 3. [RQ.3] Success rates of different variants of AutoRocq on proving sampled verification lemmas.

	AutoRocq	¬*CS*	¬*PT*	¬*EF*	¬*HM*	Err_1	Err_5
Success Rate	44.3%	30.0%	40.0%	38.6%	41.4%	37.1%	38.6%
Relative Improv.	–	+47.6%	+10.7%	+14.8%	+6.9%	+19.2%	+14.8%

of finding the right fact at the right time. It also reflects the fact that many proof obligations in our benchmarks require non-trivial reasoning steps, which often hinge on the application of specific lemmas or theorems. We dig into the specific patterns searched by AutoRocq, and find that they often involve complex expressions with multiple operators and operands, e.g., "*(_ * _ <= _ * _)*", "*Z.abs (_ + _)*", and "*(_ + _ <= _)*", which are challenging to write for inexperienced Rocq users. In short, the above facts highlight that AutoRocq's agentic access to the proof context through invocation of *Search* and other queries is pivotal in its efficacy.

Answer to RQ.3: Each component of AutoRocq contributes to its overall effectiveness. AutoRocq's agentic context search enhances its efficacy most significantly.

6.4 RQ.4: Comparison to Human-written Proofs

Since there is no ground truth for the lemmas extracted from SV-COMP programs, we cannot directly compare the proofs synthesized by AutoRocq with human-written proofs. In this subsection, we closely examine a proof synthesized by AutoRocq, and compare it to a manually crafted proof by a Rocq expert. The expert has more than six years of formal verification experience and one year of practical experience with Rocq. We use the same theorem `wp_goal` as presented in Figure 1(a), and report the human-written proof in Figure 9.

To construct the proof in Figure 9(b), the expert had first to recognize that the problem's core reduces to the mathematical principle that, for any integer a, $a^2 \geq 100$ implies $|a| \geq 10$. This insight is non-trivial, as it requires moving beyond a direct case-based enumeration of values to grasp the underlying structural inequality. Only after achieving this conceptual leap could the expert elegantly bifurcate the problem based on the sign of `i1` and construct the two custom, symmetric helper lemmas (lines 9–15 in Figure 9(a)) to complete the solution. Crafting this proof takes the Rocq expert 20 minutes of time.

In contrast, let us consider again the proof generated automatically by AutoRocq in lines 9–23 in Figure 1(a). It took AutoRocq only 156.7 seconds to synthesize a proof for the same lemma. At a high level, it achieves the same realization as the human prover, and focuses its proof on an explicit, enumerative case analysis for the value of `i1`. It also quickly connects the goal to properties of absolute values (`Z.abs`). Crucially, AutoRocq leverages two key lemmas from the global context to assist its reasoning on absolute values, namely `Z.abs_le` at line 12 and `Z.square_le_mono_nonneg` at line 19. This not only allows it to prove the goal successfully, but also proves it in noticeably fewer steps than the human expert. Our log reveals that AutoRocq autonomously invokes multiple context searches during its proving. Specifically, searching for the pattern "*(_ * _ <= _ * _)*" surfaces the crucial lemma used at line 19, which eventually helps AutoRocq to succeed. Unsurprisingly, *no* other baseline tools are able to prove this lemma due to their lack of agency.

Answer to RQ.4: With the help of context queries, AutoRocq is able to prove challenging lemmas, sometimes in even fewer steps than human experts. It also takes much less time.

```
Theorem wp_goal :
  forall (i:Numbers.BinNums.Z) (i1:Numbers.BinNums.Z),
  let x := (i1 * i1)%Z in ((-9%Z)%Z <= i1)%Z →
  ((-9%Z)%Z <= i)%Z → (i <= 9%Z)%Z → (i1 <= 10%Z)%Z →
  (i <= 10%Z)%Z → ((-2147483648%Z)%Z <= x)%Z →
  (100%Z <= x)%Z → (x <= 2147483647%Z)%Z → is_sint32 i1 →
  is_sint32 i → (i1 = 10%Z).

(* Helper lemmas written by the human prover*)
Lemma square_le_mono_nonneg :
  forall a b, 0 <= a → 0 <= b → a * a <= b * b → a <= b.
Proof. intros a b Ha Hb Hsq. nia. Qed.
Lemma square_le_mono_neg :
  forall a b, a < 0 → b < 0 → a * a <= b * b → b <= a.
Proof. intros a b Ha Hb Hsq. nia. Qed.
```

(a) wp_goal in Figure 1(a) with human helper lemmas.

```
Proof.
  intros. subst x. assert (Hcases: 0 <= i1 ∨ i1 < 0) by lia.
  destruct Hcases as [Hge0 | Hlt0].
  - assert (H10: 0 <= 10) by lia.
    pose square_le_mono_nonneg.
    specialize (l 10 i1 H10 Hge0).
    simpl in l.
    specialize (l H5). lia.
  - pose square_le_mono_neg.
    specialize (l (-10) i1).
    assert (Hn10: -10 < 0) by lia.
    specialize (l Hn10 Hlt0).
    simpl in l.
    specialize (l H5). lia.
Qed.
```

(b) Main proof script written by a human Rocq expert.

Fig. 9. [RQ.4] Proof for wp_goal in Figure 1(a), written by a human expert prover in 20 minutes.

6.5 Threats to Validity

Trustworthiness and Cost of LLMs. LLMs may give incomplete or factually wrong responses to questions. As a result, verifying programs with untrusted LLMs may seem off-putting at first glance. However, we note that the certificate essentially comes from the trusted kernel of Rocq, which ensures that only derivations conforming to the formal rules of the proof assistant are accepted [63]. Therefore, all the generated proof scripts are *true positives* and are trustworthy, as any erroneous or misleading outputs from the LLMs will simply fail to be verified. Throughout our experiments, it takes $1.22 on average to prove a single lemma without token caching. We deem this cost acceptable, compared to the costly human involvement in lemma-proving activities.

Data Leakage. LLMs are trained on a large corpus of data; thus, they may have been exposed to open-sourced Rocq projects. As such, LLM-based approaches, including AutoRocq, may report inflated results on CoqGym, which comprises only lemmas and proofs in the public domain. In our evaluation, we mitigate this risk by including obligations from SV-COMP programs. These lemmas are extracted by us automatically (see more details in Section 5.1.3), and their ground-truth proofs are not available. In fact, it is unclear if these lemmas are provable at all until we find a correct proof, so we believe the risk of data leakage is minimal.

Lemma Selection. In our evaluation, we include established benchmarks and verification targets, namely CoqGym and SV-COMP programs. The lemmas are selected systematically as detailed in Section 5.1.2. We study (un)reachability and overflow freedom as examples of correctness and safety properties, respectively, due to their universality. We also study the loop invariants proposed by LLMs. However, certain types of programs or properties may pose systematic challenges to our approach that we are unaware of. We plan to investigate more types of programs (e.g., multi-threaded) or properties (e.g., memory safety) in future work.

Impact of LLMs Used. Since AutoRocq adopts a general-purpose LLM (i.e., GPT-4.1) while some baseline approaches adopt custom ones, we want to study the impact of the underlying model. We conducted an additional comparison between AutoRocq and a variant of Rango (the best-performing baseline with a custom model) named Rango-GPT-4.1. In this variant, the fine-tuned LLM was replaced with GPT-4.1. Experimental results show that Rango-GPT-4.1 can only prove 11.43% of the verification lemmas used in Section 6.3, whereas AutoRocq proves 44.29%. These results strongly indicate that AutoRocq's relative efficacy over competing tools comes from the agentic layer rather than from a powerful choice of LLM.

Impact of Rocq Versions. During evaluation, we used a different Rocq 8.12 for PALM, which is different from other comparative tools (i.e., Rocq 8.18), raising concerns about how different Rocq versions may affect the experimental results. In theory, Rocq version differences do not materially

Table 4. [Cast Study] Verifying functional correctness in Linux kernel modules: # of lemmas proved, and the average time (in seconds) and steps (in # of tactics) expended on proved lemmas.

Tools	AutoRocq	Rango	PALM	Copra	QEDC	P9001	AutoRocq-w/H
Proved? (max. 60)	**12**	3	10	7	3	2	**18**
Avg. Time	29.9	96.2	39.4	65.1	**2.0**	160.8	65.9
Avg. Steps	**14.3**	32.0	–	50.0	32.0	979.0	16.7

affect the comparison between the PALM and AutoRocq, as most of the version changes concern syntax, parsing, or library organization, rather than the proof engine that our method relies on. Practically, the impact would be minimal, as AutoRocq dynamically retrieves and queries lemmas from the current Rocq environment and thus does not depend on version-specific assumptions.

7 Case Study: Verifying Linux Kernel Modules

AutoRocq has demonstrated remarkable efficacy in automatically synthesizing rigorous proofs for lemmas related to smaller programs or a mathematical context. In this case study, we assess if AutoRocq can handle the complexity of real-world software. To this end, we build on earlier efforts [17, 69] to reason about properties in the Linux kernel. Specifically, we examine 60 lemmas that formalize functional correctness in Linux kernel code. These lemmas reason about the correctness of intricate program behaviors, and come from various source files, such as memory management (e.g., `memmove`) and utilities such as string operations (e.g., `strcpy`) and type conversion (e.g., `hex2bin`). These lemmas tend to be significantly more involved and may contain up to hundreds of terms.

We report the statistics in Table 4. Results show that AutoRocq is the most effective, being able to synthesize the proofs for 12/60 (20%) lemmas. Copra manages to prove 7 lemmas as well. In contrast, Rango, QEDC, and P9001 perform poorly on these tasks, which only manage to succeed in 3, 3, 2 lemmas, respectively. We also note that QEDC runs exceptionally fast when it is able to generate a proof, being 15x faster than AutoRocq, which is the second fastest. This suggests that QEDC's search heuristic performs extremely well on certain cases. Nonetheless, such cases seem rare when verifying real, complex software, as it only succeeds on 3 lemmas.

Notably, PALM also achieves remarkable effectiveness and proves 10/60 lemmas. Upon manual examination, we note that *all* successful proofs generated by PALM rely on heavy use of CoqHammer [13], a plug-in that directly discharges remaining subgoals to automated theorem provers (ATPs). CoqHammer is invoked by PALM as a last resort when *none* of its deterministic tactics repairs or resolves the encountered errors. Based on this finding, we create a new variant, AutoRocq-w/H, by augmenting our approach with basic CoqHammer integration. Specifically, the hammer is invoked with its default settings *once* per proof state upon the first failed tactic application. Such integration yields significant efficacy gains, with 18 lemmas (30%) proved, indicating that access to additional tools could further enhance the capabilities of our proof agent.

> AutoRocq can be applied to verify properties in complex software such as the Linux kernel. Access to external automated theorem provers (e.g., CoqHammer) further improves AutoRocq.

8 Related Work

8.1 Deductive Program Verification

Deductive verification formally verifies the correctness of software systems by extracting and proving a collection of mathematical proof obligations from both the program and its specification [20]. The automation of the verification process is largely underpinned by verification-oriented

languages, including Frama-C [31], Dafny [35], Verus [34], and Viper [49], which provide an end-to-end verification flow that integrates specification and proof directly into the programming workflow through the use of annotations. These tool chains generate proof obligations automatically, thereby reducing the problem of program verification to theorem proving. The proof obligations are sent to either Automated Theorem Provers (ATPs) or Interactive Theorem Provers (ITPs) to verify. ATPs such as CoqHammer [13] attempt to discharge the obligations automatically through constraint solving [7, 14], while ITPs such as Isabelle/HOL [51], Lean [15], and Rocq [66] allow users to interactively construct proofs using tactics. Due to ATPs' limited capabilities in handling complex proofs [50] and occasional correctness issues [45, 70], ITPs have become the *de facto* standard for verifying sophisticated software systems. Indeed, many pioneering efforts have successfully verified critical software systems [11, 32, 36, 44], network protocols [79], and microprocessor designs [30] with ITPs. However, ITPs require significant human effort to guide the proof construction, which limits their scalability and applicability. AutoRocq specifically focuses on reducing the manual effort in proving complex residual obligations, thus advancing towards an automatic, end-to-end verification workflow. As such, it is related but complementary to prior work automating other aspects of the workflow, including inference of specifications [18] and loop invariants [23, 40, 60, 61], discovery of helper lemmas [62], and direct synthesis of verified methods [48].

8.2 Automated Techniques in Theorem Proving

Our work is closely related to the substantial research devoted to automating mathematical theorem proving [25, 39], spanning the complete pipeline from premise selection and tactic generation to proof search and recovery mechanisms. Our focus on proving program-derived lemmas leads to unique design choices in AutoRocq, which we detail below.

Premise Selection. Premise selection is the task of identifying relevant lemmas, definitions, or axioms from a large library to assist in proving a given theorem [47, 77]. Early approaches treat premise selection as a binary classification task, where the goal is to predict whether a given premise is relevant to the theorem at hand [5]. More recent approaches such as CoqHammer [13], PALM [41], and Rango [67] leverage machine learned weights or lexical similarity measures [64] to rank premises statically. In contrast, AutoRocq adopts on-demand premise retrieval with the support of pattern-based context search.

Tactic Generation. Tactic generation is the core component in interactive theorem proving, where a proof step is proposed to transform the current proof state [39]. Early works rely on hand-crafted heuristics and domain-specific knowledge to guide generation [27, 32, 36]. More recently, machine learning techniques have been applied to learn step-wise generation strategies from large corpora of human-written proofs [9, 21, 25, 55, 67, 72, 76]. AutoRocq follows the same paradigm and queries the underlying LLM at each step with a tree-based proof state representation.

Proof Search. Proof search concerns the *sequence* of tactic applications, and has been a long-standing challenge due to an inherently large search space and sparse rewards [12, 33]. Whole-proof generation approaches such as PALM [41] circumvent the search process by predicting the entire tactic sequence in a single pass. Step-by-step proving techniques conventionally employ beam search to sample multiple tactic predictions with breadth-first [6], depth-first [76], or best-first heuristics [53, 67] to traverse the search space. Recent works have leveraged other exploration mechanisms such as reinforcement learning [57, 72] and in-context learning [65]. AutoRocq adapts the search strategy through its agency, enabled by timely feedback, high-level interpretation of the proving progress, and on-the-fly learning from successful histories.

Recovery Mechanisms. Due to the trial-and-error nature of theorem proving, proof automation systems need to effectively recover proof attempts and explore alternative strategies [41, 65]. Yet this issue is often not addressed explicitly. Rather, a large body of work takes only binary signals

from ITPs and simply retries tactic generation in case of failures [57, 67]. Others, such as Copra [65] and Proverbot9001 [55], are able to restore to an earlier proof state if necessary. PALM [41], on the other hand, performs opportunistic tactic repairs and delegates the open subgoal to an ATP [13] as the last resort. In contrast, AutoRocq exploits detailed diagnostic errors from ITPs to rectify failed tactic applications, and handles persistent errors through autonomous context enrichment.

8.3 LLM for Software Quality Assurance

Besides formally verifying correctness through theorem proving as is done in AutoRocq, a wide spectrum of techniques for software quality assurance have benefited from recent advances in Large Language Models [19, 29]. For example, model checking techniques also offer strong correctness guarantees by encoding a software system as a finite state machine and iteratively checking its states and transitions. LLMs have been employed to automate the construction of both the model [80] and the formal specification [71]. Testing techniques, on the other hand, try to prove the *presence* of bugs by synthesizing test cases that trigger unintended behaviors, and have benefited significantly from LLMs' capability in input generation [16, 46, 74] and program understanding [42, 68]. Static analysis leverages LLMs to identify code issues and vulnerabilities faster and earlier in the development process [38, 73]. Recent work also applies agentic hybrid (static–dynamic) analysis to security tasks such as cross-platform rule conversion [75]. While each technique above presents unique trade-offs, deductive verification stands out for its mathematical rigor and stringent guarantees on correctness, making it particularly well-suited for safety-critical systems [17].

9 Perspectives

We present an agentic proving system that automatically generates machine-checkable certificates for proof obligations. At its core, our agent is an autonomous process that retrieves relevant contexts on demand, incorporates feedback from the proof assistant, and generates tactics adaptively – all guided by interpreting the proof derivation tree. Together with a lemma extraction process, our agent can achieve effective push-button verification that takes source code in C (i.e., SV-COMP programs and Linux kernel modules) and automatically generates proofs without any human effort.

As AI-generated code becomes increasingly prevalent in software development, the need for automated verification becomes more pressing. Our work demonstrates that LLM agents can bridge the gap between automatic code generation and formal verification, moving closer to the vision of trusted automatic programming. The agentic nature of our approach suggests a paradigm shift where verification tools act as intelligent collaborators rather than passive validators, capable of autonomously navigating complex proof spaces and adapting to diverse verification challenges. This represents a crucial step toward making formal verification accessible for real-world software systems. We can thus move closer to the vision of AI-based Verification and Validation (V & V) of AI-generated code. This will alleviate the problem of humans reviewing AI-generated code.

Acknowledgments

We would like to sincerely thank all the anonymous reviewers for their valuable comments. This work was partially supported by a research project "AI for Program Reasoning", project AI4SCH-2025-0084 under AI for Science program of National Research Foundation (NRF) Singapore. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

Data Availability

We release AutoRocq, including its implementation, benchmarks, and replication instructions, for academic use at the following link: https://github.com/NUS-Program-Verification/AutoRocq.

References

[1] 2026. *BusyBox*. Retrieved April 10, 2026 from https://busybox.net/

[2] 2026. *Evolved Value Analysis (Eva) in Frama-C*. Retrieved April 10, 2026 from https://frama-c.com/fc-plugins/eva.html

[3] 2026. *Runtime Error (RTE) in Frama-C*. Retrieved April 10, 2026 from https://www.frama-c.com/fc-plugins/rte.html

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, 2655–2668. doi:10.18653/v1/2021.naacl-main.211

[5] Alexander A. Alemi, François Chollet, Niklas Een, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath - deep sequence models for premise selection. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2243–2251. https://dl.acm.org/doi/10.5555/3157096.3157347

[6] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 454–463. https://proceedings.mlr.press/v97/bansal19a.html

[7] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[8] Dirk Beyer and Jan Strejček. 2025. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, 151–186. doi:10.1007/978-3-031-90660-2_9

[9] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The Tactician: A Seamless, Interactive Tactic Learner and Prover for Coq. In *International Conference on Intelligent Computer Mathematics*. Springer, 271–277. doi:10.1007/978-3-030-53518-6_17

[10] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F. Ferreira, and Emily First. 2024. CoqPyt: Proof Navigation in Python in the Era of LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Association for Computing Machinery, 637–641. doi:10.1145/3663529.3663814

[11] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 447–463. https://www.usenix.org/conference/osdi22/presentation/chajed

[12] Łukasz Czajka. 2020. Practical Proof Search for Coq by Type Inhabitation. In *Proceedings of the 10th International Joint Conference of Automated Reasoning*. Springer, 28–57. doi:10.1007/978-3-030-51054-1_3

[13] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (2018), 423–453. doi:10.1007/s10817-018-9458-4

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[15] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Proceedings of the 28th International Conference on Automated Deduction*. Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models Are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623343

[17] Denis Efremov, Mikhail Mandrykin, and Alexey Khoroshilov. 2018. Deductive Verification of Unmodified Linux Kernel Library Functions. In *International Symposium on Leveraging Applications of Formal Methods*, Vol. 11247. Springer, 216–234. doi:10.1007/978-3-030-03421-4_15

[18] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912. doi:10.1145/3660783

[19] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[20] Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Proceedings of the 19th International Conference on Computer Aided Verification*. Springer, 173–177. doi:10.1007/978-3-540-73368-3_21

[21] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, 1–31. doi:10.1145/3428299

[22] Frama-C Developers. 2026. *Weakest Precondition (WP) in Frama-C*. Retrieved April 10, 2026 from https://www.frama-c.com/fc-plugins/wp.html

[23] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 499–512. doi:10.1145/2837614.2837664

[24] Yeting Ge and Leonardo De Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*. Springer, 306–320. doi:10.1007/978-3-642-02658-4_25

[25] Google DeepMind. 2024. *AlphaProof*. Retrieved April 10, 2026 from https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

[26] Matthew Griffin. 2024. *AI Generated Code in Google*. Retrieved April 10, 2026 from https://www.311institute.com/google-ceo-says-25-of-all-new-google-code-is-ai-generated/

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[28] Hendrik. 2025. *Proof Tree Visualization for Proof General*. Retrieved April 10, 2026 from https://askra.de/software/prooftree/

[29] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[30] Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark D. Aagaard, and Thomas F. Melham. 2001. Practical Formal Verification in Microprocessor Design. *IEEE Design & Test of Computers* 18, 4 (2001), 16–25. doi:10.1109/54.936246

[31] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. doi:10.1007/s00165-014-0326-7

[32] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, 207–220. doi:10.1145/1629575.1629596

[33] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems* 35 (2022), 26337–26349. https://dl.acm.org/doi/10.5555/3600270.3602180

[34] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 286–315. doi:10.1145/3586037

[35] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Vol. 6355. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[36] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert: A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 1–8. http://xavierleroy.org/publi/erts2016_compcert.pdf

[37] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474. https://dl.acm.org/doi/abs/10.5555/3495724.3496517

[38] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499. doi:10.1145/3649828

[39] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A Survey on Deep Learning for Theorem Proving. In *First Conference on Language Modeling*. 1–27. https://openreview.net/pdf?id=zlw6AHwukB

[40] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2025. DafnyBench: A Benchmark for Formal Software Verification. *Transactions on Machine Learning Research* 2025 (2025), 1–21. https://openreview.net/pdf?id=yBgTVWccIx

[41] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 1509–1520. doi:10.1145/3691620.3695521

[42] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. 2026. Agentic Concolic Execution. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 37–55. https://doi.ieeecomputersociety.org/

10.1109/SP63933.2026.00003

[43] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of IEEE/ACM 47th International Conference on Software Engineering*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[44] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 237–248. doi:10.1145/1706299.1706329

[45] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 701–712. doi:10.1145/3368089.3409763

[46] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model Guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. 1–17. doi:10.14722/ndss.2024.24556

[47] Maciej Mikuła, Szymon Tworkowski, Szymon Antoniak, Bartosz Piotrowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. MagnusHammer: A Transformer-Based Approach to Premise Selection. In *37th Conference on Neural Information Processing Systems (NeurIPS 2023) Workshop on MATH-AI*. 1–16. https://openreview.net/forum?id=oYjPk8mqAV

[48] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 812–835. doi:10.1145/3643763

[49] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, Vol. 9583. Springer, 41–62. doi:10.1007/978-3-662-49122-5_2

[50] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2007. Challenges in Satisfiability Modulo Theories. In *International Conference on Rewriting Techniques and Applications*. Springer, 2–18. doi:10.1007/978-3-540-73449-9_2

[51] Lawrence C. Paulson. 1994. *Isabelle: A Generic Theorem Prover*. Springer. doi:10.1007/BFb0030541

[52] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (2025), 96–105. doi:10.1145/3672458

[53] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393* (2020). https://arxiv.org/abs/2009.03393

[54] Rocq Development Team. 2025. *Search Command in Rocq Documentation*. Retrieved April 10, 2026 from https://rocq-prover.org/doc/V9.0.0/refman/proof-engine/vernacular-commands.html#coq:cmd.Search

[55] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating Correctness Proofs with Neural Networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10. doi:10.1145/3394450.3397466

[56] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Transactions on Programming Languages and Systems* 45, 2 (2023), 1–30. doi:10.1145/3592457

[57] Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Dylan Zhang, Talia Ringer, and Yuriy Brun. 2025. QED-Cartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE, 405–418. doi:10.1109/ICSE55347.2025.00160

[58] Gabriel Scherer. 2017. *Bullets Are Good for Your Coq Proofs*. Retrieved April 10, 2026 from https://prl.khoury.northeastern.edu/blog/2017/02/21/bullets-are-good-for-your-coq-proofs/

[59] Jessica Shi, Cassia Torczon, Harrison Goldstein, Benjamin C. Pierce, and Andrew Head. 2025. QED in Context: An Observation Study of Proof Assistant Users. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 337–363. doi:10.1145/3720426

[60] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. *Advances in Neural Information Processing Systems* 31 (2018), 7762–7773. https://dl.acm.org/doi/abs/10.5555/3327757.3327873

[61] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2Inv: A Deep Learning Framework for Program Verification. In *Proceedings of the 32nd International Conference on Computer Aided Verification*, Vol. 12225. Springer, 151–164. doi:10.1007/978-3-030-53291-8_9

[62] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-Driven Lemma Synthesis for Interactive Proofs. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 505–531. doi:10.1145/3563308

[63] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (2025), 74 pages. doi:10.1145/3706056

[64] Karen Sparck Jones. 1972. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation* 28, 1 (1972), 11–21. doi:10.1108/eb026526

[65] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *First Conference on Language Modeling*. 1–27. https://openreview.net/forum?id=V7HRrxXUhN

[66] The Coq Development Team. 2023. *The Coq Proof Assistant – Reference Manual.* Retrieved April 10, 2026 from https://flint.cs.yale.edu/cs430/coq/pdf/Reference-Manual.pdf

[67] 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 *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE, 347–359. doi:10.1109/ICSE55347.2025.00161

[68] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. 2026. Cottontail: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2064–2082. https://doi.ieeecomputersociety.org/10.1109/SP63933.2026.00110

[69] Grigoriy Volkov, Mikhail Mandrykin, and Denis Efremov. 2018. Lemma Functions for Frama-C: C Programs as Proofs. In *2018 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 31–38. doi:10.1109/ISPRAS.2018.00012

[70] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 718–730. doi:10.1145/3385412.3385985

[71] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-Symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417. doi:10.1145/3691620.3695014

[72] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*. 9330–9342. https://dl.acm.org/doi/10.5555/3540261.3540975

[73] Yin Wu, Xiaofei Xie, Chenyang Peng, Dijun Liu, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. 2024. AdvScanner: Generating Adversarial Smart Contracts to Exploit Reentrancy Vulnerabilities Using LLM and Static Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1019–1031. doi:10.1145/3691620.3695482

[74] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639121

[75] Ming Xu, Hongtai Wang, Yanpei Guo, Zhengmin Yu, Weili Han, Hoon Wei Lim, Jin Song Dong, and Jiaheng Zhang. 2026. ARuleCon: Agentic Security Rule Conversion. arXiv:2604.06762 [cs.CR] https://arxiv.org/abs/2604.06762

[76] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97. PMLR, 6984–6994. http://proceedings.mlr.press/v97/yang19a.html

[77] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *Advances in Neural Information Processing Systems* 36 (2023), 21573–21612. https://dl.acm.org/doi/abs/10.5555/3666122.3667066

[78] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1592–1604. doi:10.1145/3650212.3680384

[79] Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2023. Automated Verification of an In-Production DNS Authoritative Engine. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*. ACM, New York, NY, USA, 80–95. doi:10.1145/3600006.3613154

[80] Xinyue Zuo, Yifan Zhang, Hongshu Wang, Yufan Cai, Zhe Hou, Jing Sun, and Jin Song Dong. 2025. PAT-Agent: Autoformalization for Model Checking. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2122–2133. doi:10.1109/ASE63991.2025.00176

Received 2026-02-08; accepted 2026-03-24