Automated Lemma Discovery in Agentic Program Verification

Huan Zhao*
National University of Singapore
Singapore
zhaohuan@comp.nus.edu.sg

Haoxin Tu*†
National University of Singapore
Singapore
haoxin.tu@nus.edu.sg

Zhengyao Liu
National University of Singapore
Singapore
zhengyao.liu@u.nus.edu

Martin C. Rinard
Massachusetts Institute of Technology
Cambridge, USA
rinard@csail.mit.edu

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@nus.edu.sg

Abstract

Deductive verification provides strong correctness guarantees for code by extracting verification conditions (VCs) and writing formal proofs for them. The expertise-intensive task of VC proving is the main bottleneck in this process, and has been partly automated owing to recent advances in Large Language Model (LLM) agents. However, existing proof agents are not able to discover helper lemmas – auxiliary lemmas that aid in proving – and thus fall short as programs grow in size and complexity.

In this paper, we argue that VC proving for program verification is more than a purely mathematical task, and benefits considerably from program comprehension. Our key insight is that human proof engineers often discover and apply helper lemmas based on their understanding of the program semantics, which are *not* directly reflected in the VCs produced by VC generators. Inspired by this insight, we propose an LLM agent, LemmaNet, that discovers helper lemmas in two ways. Specifically, the agent first synthesizes lemmas *offline* by directly analyzing the source code and specifications and then relating this semantic understanding to the mechanical, verbose encoding produced by VC generators. As the proof unfolds, LemmaNet then adapts existing helper lemmas *online* to accommodate evolving proof states, enabling the agent to effectively discharge complex VCs on-the-fly.

We implement LemmaNet on top of an existing proof agent AutoRocq for Rocq and the Frama-C ecosystem, and evaluate it on SV-COMP and established real-world subjects, including modules of the Linux kernel, Contiki OS, standard C++ library, and X.509 parser. Our experimental results demonstrate that LemmaNet significantly outperforms state-of-the-art approaches, highlighting the importance of program comprehension-aided lemma discovery in agentic program verification.

CCS Concepts

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

*The first two authors contributed equally to this work and are listed in random order.
†Corresponding Author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834356

Keywords

LLM Agent, Verification, Theorem Proving, Helper Lemma, Rocq

ACM Reference Format:
Huan Zhao, Haoxin Tu, Zhengyao Liu, Martin C. Rinard, and Abhik Roychoudhury. 2026. Automated Lemma Discovery in Agentic Program Verification. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834356

1 Introduction

Trusted software is critical for the safety and security of modern computing systems, especially in safety-critical domains such as aerospace, automotive, and healthcare [8, 31, 32]. In pursuit of this goal, formal verification is a powerful technique that can provide strong guarantees that the software correctly implements its specification [36, 42, 56]. One widely used approach is *deductive verification*, which involves generating mathematical verification conditions (VCs) from source code and specifications, then using interactive theorem provers to generate formal machine-checked proofs of these VCs [1]. Proving these VCs is currently an extremely labor- and expertise-intensive undertaking, which has significantly hindered its adoption in practice [31, 32]. With recent advances in the mathematical reasoning ability of large language models (LLMs) [2, 15], there has been growing interest in leveraging LLM agents to assist in the formal verification process [38, 59, 62].

Limitations of Existing Approaches. Existing theorem-proving approaches formulate VC proving as a purely mathematical exercise [38, 53, 54], where the input to the system is a theorem statement, and the output is a sequence of proof steps that establish its correctness. Recent approaches that adopt this paradigm have therefore focused on improving various aspects of this process [55, 59, 60, 62]. Human proof engineers take a different approach. When discharging complex obligations, they typically start with an understanding of the program semantics and propose helper lemmas that capture relevant aspects of the program structure and properties [18, 49]. These helper lemmas often identify key observations and/or intermediate results that can considerably simplify the proving process, with the helper lemmas often refined and enhanced in light of new information obtained as the proof progresses. This aspect of proof development remains largely overlooked by existing approaches. As such, they struggle to discharge complex VCs that benefit from insights into the program semantics, structure, and properties.

New Angle to VC Proving. We argue that program VC proving is *not* purely a matter of mathematical reasoning, but should be deeply coupled with program comprehension. Our argument is based on the fact that proofs are easier to develop and interpret when they align with program structure and source-level semantics, with the alignment driven by intermediate lemmas that capture important insights into the program structure, semantics, and properties [18, 49]. For example, to prove a loop invariant obligation (e.g., line 10 in Figure 1), a human proof engineer usually first tries to understand the loop's behavior and write a proper semantics-aware VC that is amenable to straightforward proof (e.g., the semantics-aware VC in Figure 2(b)). This informal reasoning process is crucial for effectively discharging complex VCs and is often guided by the program semantics, which are not directly reflected in the proof-targeted VCs (e.g., the proof-targeted VC in Figure 2(a)) generated by existing VC generators (e.g., Frama-C). We therefore propose to mimic this human proof process by developing an LLM agent that discovers helper lemmas informed by program structure and semantics, aiming to bridge the gap between the program semantics and the proof-targeted VCs, and thus facilitate the proving process.

Challenges. Two challenges need to be addressed to enable such a human-like proving process for effective automated verification. First, existing VC generators model the full complexity of the language semantics, including aspects such as integer overflow, pointer arithmetic, and memory addressing. The generated VCs obscure the conceptual structure of the program semantics in low-level details that generators use to model these and other aspects of the underlying language semantics. While a naive LLM-based approach can more clearly reflect the program structure, the approach is fundamentally unsound, and the VCs come with no correctness guarantees. Second, as VC complexity grows, it becomes difficult to foresee *a priori* how proof states will evolve throughout the proving process. As such, proof strategies must dynamically adapt as new information is updated during proving while still preserving the coherence and relevance of the existing reasoning trajectory.

Our Solution. We propose a new solution, LemmaNet, that addresses the above two challenges through a carefully designed mechanism of *helper lemma discovery*. LemmaNet discovers helper lemmas in two stages of the verification workflow. First, LemmaNet performs *offline* analysis before theorem proving starts to extract a semantics-aware VC directly from the annotated source code. To mitigate the potential untrustworthiness of LLM-based analysis, the semantics-aware VC is used merely as a reference for synthesizing helper lemmas to bridge source-level concepts with the proof-targeted VC. These helper lemmas are ultimately used to discharge the VC produced by a trusted VC generator, ensuring soundness throughout the process. Second, LemmaNet performs *online* lemma adaptation as proving progresses. This adaptation is achieved by iteratively refining and enhancing the helper lemmas based on the feedback from the theorem prover. Our results indicate that this online adaptation process often enables the agent to effectively discharge complex, otherwise intractable VCs while maintaining alignment with the program semantics captured by the semantics-aware VC (c.f. Section 5.4). By integrating both offline lemma synthesis and online lemma adaptation, LemmaNet is able to effectively discover and apply helper lemmas that are crucial for VC proving, significantly improving its efficacy in verifying real-world software projects.

Evaluation. Our evaluation on SVCOMP [4] and NTP4VC [68] benchmarks (which include Linux kernel modules, Contiki OS, X.509 parser, etc.) shows that LemmaNet can effectively discharge complex VCs arising from real-world verification tasks, significantly outperforming state-of-the-art proof agents AutoRocq [62] and Copra [59] by 26.8%–51.7%. We also report what helper lemmas are proposed and how they aid in agentic program verification.

Contributions. This paper makes the following contributions:

- It proposes a novel agentic approach for lemma discovery to facilitate program verification.
- It instantiates this framework with LemmaNet[1], which supports both offline lemma synthesis and online lemma adaptation to effectively discharge complex VCs.
- It presents results from an evaluation of LemmaNet on leading benchmarks. These results highlight the effectiveness of LemmaNet in enabling the verification of complex, real-world systems such as Linux kernel modules.

Paper Organization. Section 2 introduces the background on deductive verification and automated theorem proving. Section 3 presents a motivating example that highlights the challenges of existing approaches and motivates our design. Section 4 describes the design and implementation of LemmaNet. Section 5 evaluates LemmaNet on real-world verification benchmarks and analyzes the efficacy of its key components. Section 6 discusses limitations and potential future directions. Section 7 reviews related work, and Section 8 concludes the paper.

2 Background

2.1 Deductive Verification Practices

Deductive verification formally checks if the code conforms to its formal specification. In contrast to testing, which can prove the *existence* of bugs, deductive verification can prove their *absence* by reasoning over all possible executions under a well-defined semantic model. The workflow starts with augmenting the source code with specification annotations either manually or automatically [41]. These annotations are formal contracts that are attached to functions and program locations, and specify key properties such as pre/post-conditions, invariants, or (partial or full) correctness. Figure 1 presents example source code with ACSL [30] annotations. The annotated source code is then transformed into a collection of formal *verification conditions* (VCs) [1, 20], also known as proof obligations, via the use of a VC generator that applies a weakest precondition (or strongest postcondition) calculus to systematically produce logical formulas. The validity of all VCs implies that the program meets its specification. This style of verification has been widely adopted in mature verification frameworks including Dafny [36, 44], Verus [34], Viper [47], and Frama-C [30].

While some VCs can be discharged by automated solvers such as SMT solvers [3, 13], Interactive Theorem Provers (ITPs) are required for more complex obligations that involve sophisticated reasoning [11, 14, 50]. The Rocq proof assistant (formerly Coq) [11]

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

is a prominent example of ITPs. To discharge a VC as the proof *goal* in Rocq, one applies a sequence of *tactics*: commands that transform the current goal into simpler sub-goals. Each successful tactic application refines a *proof term* that Rocq's kernel type-checks for soundness. The proof concludes when no sub-goal remains, yielding a certified derivation. To enable proofs of complex obligations, human provers often introduce auxiliary *lemmas* that encapsulate reusable intermediate facts, which is a well-supported feature in modern ITPs such as Rocq.

2.2 Theorem Proving for Verification

Machine Learning for Proof Automation. Automating proof construction is a key step toward closing the verification gap. Recent advances in machine learning have spurred numerous approaches to this problem. Pioneering efforts [53, 70] frame proof synthesis as sequence prediction, employing neural networks trained on syntactic patterns [70], proof-state encoding [23], and contextual features [54]. These prediction models later became the cornerstone of more sophisticated approaches that employ various search algorithms to steer tactic selection toward genuine progress [5, 55].

Large Language Models (LLMs) offer strong mathematical reasoning capabilities, making them attractive for proof automation [15]. Recent approaches have leveraged LLMs for whole-proof generation [38] or retrieval-augmented tactic generation (RAG) using a curated database of historical proof states [60]. Notably, LLM agents show great promise in proof generation. Copra [59] iteratively proposes tactics, executes them, and incorporates feedback from the proof assistant to refine subsequent suggestions. AutoRocq [62] allows for a more dynamic workflow, enabling the agent to autonomously gather additional context or traverse the proof tree.

Proof Automation for VCs. Despite notable progress, transferring these techniques to program-derived theorems remains difficult. Existing neural theorem provers [38, 53, 59, 60] are trained and evaluated primarily on mathematical lemmas with human-written ground truth [70]. Prior work [62] has shown that verification conditions arising from real-world programs tend to be more complex than their mathematical counterparts. This complexity compounds as programs grow in scale, increasingly straining existing approaches for automated discharge of verification conditions.

A critical bottleneck lies in *helper lemma discovery* [33, 58, 71]. Complex VCs often cannot be proved directly; instead, they require auxiliary lemmas that capture intermediate facts about program semantics, data structure properties, or arithmetic relationships. Existing approaches largely overlook this challenge, focusing on tactic generation and lemma retrieval from existing libraries only. However, in the context of VC proving, key lemmas are often *not* available in the proof context. In practice, identifying and synthesizing the right helper lemmas demands a nuanced understanding of both the program under verification and the proof structure.

3 Motivating Example

In this section, we illustrate the challenges in the existing VC proving workflow through a motivating example, and present the key insights that motivate our solution.

```
1  int hex_to_bin(char ch) {...}
2
3  /*@ requires ...; ensures ... */
4  int hex2bin(u8 *dst, const char *src, size_t count)
5  {
6      //@ ghost size_t ocount = count;
7      //@ ghost char *osrc = src;
8      /*@
9          loop invariant 0 <= count <= ocount;
10         loop invariant osrc <= src;
11         loop invariant \forall char *p; ...
12     */
13     while (count--) {
14         int hi = hex_to_bin(*src++);
15         int lo = hex_to_bin(*src++);
16         if ((hi < 0) || (lo < 0))
17             return -1;
18         //@ assert 0 <= ((hi << 4) | lo) <= 255;
19         *dst++ = (hi << 4) | lo;
20     }
21     //@ assert count == ((size_t)-1);
22     return 0;
23 }
```

Figure 1: Source code of `hex2bin.c` from [20] with ACSL annotation. The loop invariant to prove is highlighted.

Illustration of a Proof-targeted VC. Figure 1 shows a simplified C program from `hex2bin.c` in the Linux kernel codebase, which converts a string representing a hexadecimal number to its binary representation. Guarded by `//@` and `/*@...*/` are ACSL [30] annotations that declare the specifications to be proved. Highlighted is a key loop invariant, whose validity underpins the verification of the program. Specifically, it states an invariant that variable `src` is always at least as large as `osrc` throughout the `while` loop, where `osrc` is a ghost variable recording the original value of `src`. Intuitively, the preservation of this simple invariant holds almost trivially, as `src` is monotonically incremented exactly by two in each iteration.

When passed through a VC generator such as Frama-C's WP plugin [25], however, this simple invariant yields a surprisingly complex VC as shown in Figure 2(a). We denote it as the *proof-targeted VC* ϕ_C. This proof-targeted VC, bearing little resemblance to the original program or the annotation, is a sprawling, low-level formula that encodes tool-specific details such as Frama-C's internal memory model for C, pointer arithmetic, and integer overflow semantics. Consequently, even state-of-the-art theorem proving agents such as [62] struggle to synthesize a proof for ϕ_C.

Illustration of a Semantics-aware VC. If one formally expresses the high-level understanding that `src` is monotonically increased by two in each iteration, a proof engineer may produce a VC as shown in Figure 2(b). The lemma at line 6, denoted as the *semantics-aware VC* ϕ_A, states that, for all `osrc` and `src`, $\texttt{osrc} \leq_{\texttt{addr}} \texttt{src} \rightarrow \texttt{osrc} \leq_{\texttt{addr}} \texttt{src} + 2$. Here, the notion $\leq_{\texttt{addr}}$ refers to a custom, intuitive model of address comparison as defined in line 2. Compared to Figure 2(a), ϕ_A encodes and proves the same loop invariant but is instead expressed directly in terms of source-level concepts drawn from the program and its specification, making it much more compact and amenable to straightforward proof. Indeed, ϕ_A could be easily proved by existing approaches, or by simply consulting an LLM.

```
1  Definition is_sint32 (x:Numbers.BinNums.Z) : Prop := ...
2
3  (* Proof-targeted Verificaion Condition *)
4  Theorem wp_goal :
5    forall (t: Z → Z) (t1:addr → Z) (t2:Z → Z)
6    (a:addr) (i:Z) (i1:Z) (i2:Z) (a1:addr) (a2:addr),
7    let x := (16 * i) in let x1 := lor i1 x in
8    let x2 := offset a2 in let x3 := offset a1 in
9    let x4 := t1 a2 in let x5 := t2 (to_uint8 x4) in
10   let x6 := t1 (shift a2 1) in let x7 := t2 (to_uint8 x6) in
11   let x8 := land 68 x5 in let x9 := land 68 x7 in
12   ~ (i2 = 0) → (x = (lsl i 4)) → ((i1 + x) = x1) →
13   ((Z.quot (x2 + ((-1) * x3)) 2) = 0) →
14   ((Z.rem (x3 + ((-1) * x2)) 2) = 0) →
15   (0 <= i2) → (0 <= i1) → (0 <= i) →
16   ((region (base a1)) <= 0) → ((region (base a)) <= 0) →
17   (0 <= x1) → ((to_uint64 ((-1) + i2)) <= i2) →
18   (i1 <= 15) → (i <= 15) →
19   (x1 <= 255) → IsArray_uint8 t2 → linked t → sconst t1 →
20   is_sint32 i1 → is_sint32 i → is_uint64 i2 → addr_le a1 a2 →
21   addr_le a1 a1 → addr_le a a → is_sint8 x4 → is_uint8 x5 →
22   is_sint8 x6 → valid_rw t (shift a 0) (1 + i2) →
23   valid_rd t (shift a1 0) (1 + (2 * i2)) → is_uint8 x7 →
24   ((x8 = 0) → (i = (-1))) → (~(x8 = 0) → ((L_hex_to_bin x4) = i)) →
25   ((x9 = 0) → (i1 = (-1))) → (~(x9 = 0) → ((L_hex_to_bin x6) = i1)) →
26   (forall (a3:addr), addr_lt a3 a2 → addr_le a1 a3 →
27   ~ ((land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
28   (forall (a3:addr), addr_lt a3 a1 → addr_le a1 a3 →
29   ~ ((land 68 (t2 (to_uint8 (t1 a3)))) = 0)) →
30   addr_le a1 (shift a2 2).
31 Proof.
32   intros t t1 i t2 a i1 i2 a1 a2; cbn. ...
33   apply (HL1_addr_le_shift_same_base a1 a2 2).
34   apply (HL2_addr_le_same_base _ _ Hle_a1a2).
35   all: try (exact Hle_a1a2); try lia.
36 Qed.
```

(a) Proof-targeted VC ϕ_C in Rocq, generated by Frama-C (simplified) and the complete proof generated by LemmaNet. Helper lemmas are applied in lines 33-34.

```
1  (* Custom model of address comparison *)
2  Definition addr_le_model (p q:addr) : Prop :=
3    base p = base q ∧ offset p <= offset q.
4
5  (* Semantics-aware Verificaion Condition *)
6  Lemma goal_osrc_le_src_preserved :
7    forall osrc src : addr,addr_le_model osrc src →
8      addr_le_model osrc (shift src 2).
9  Proof.
10   intros osrc src [Hb Hoff]. split.
11   (* base preserved *)
12   - unfold shift, base in *; simpl in *; exact Hb.
13   (* offset incremented *)
14   - unfold shift, offset in *; simpl in *; lia.
15 Qed.
```

(b) Semantics-aware VC ϕ_A expressed and proved in source-level concepts.

```
1  Lemma base_shift :
2    forall p k, base (shift p k) = base p. Proof... Qed.
3
4  Lemma offset_shift :
5    forall p k, offset (shift p k) = (offset p + k)%Z. Proof... Qed.
6
7  (* Helper Lemma 1 *)
8  Lemma HL1_addr_le_shift_same_base :
9    forall p q k, base p = base q → addr_le p q
10     → (0 <= k)%Z → addr_le p (shift q k).
11 Proof.
12   rewrite base_shift. rewrite offset_shift... (* steps omitted *)
13 Qed.
14
15 (* Helper Lemma 2 *)
16 Lemma HL2_addr_le_same_base :
17   forall p q, addr_le p q → base p = base q. Proof... Qed.
```

(c) Helper lemmas discovered by LemmaNet, which are used to bridge the gap between semantics-aware and proof-targeted VCs.

Figure 2: (a) Proof-targeted VC ϕ_C v/s (b) semantics-aware VC ϕ_A for the highlighted loop invariant in Figure 1. (c) Helper lemmas are discovered by LemmaNet, and are useful in generating a complete proof for the proof-targeted verification condition.

Key Insights. Our key observation is that the inefficiency of existing VC proving approaches stems *not* from lagging mathematical reasoning capability, but from the burial of semantic signal under layers of verbose, tool-specific encoding that has no direct correspondence to how the program is understood. As such, approaches that treat VC proving as a purely mathematical endeavor and reason directly about ϕ_C struggle even on simple programs and properties such as the one in Figure 1. This observation is consistent with the fact that human proof engineers often discover and apply helper lemmas that are informed by their understanding of the program semantics [18, 49]. On the other hand, reasoning exclusively about the semantics-aware VC ϕ_A is *insufficient* for soundness reasons, as there is no formal argument that ϕ_A's modeling of C constructs is faithful. In other words, the verifier must therefore ultimately discharge ϕ_C produced by the (trusted) VC generator, but without being overwhelmed by its low-level encoding. This calls for a verification framework that can leverage the insights from ϕ_A to facilitate the proof of ϕ_C, while ultimately targeting the final proof towards ϕ_C to guarantee soundness.

Two Challenges. Two challenges need to be addressed to realize the above vision. First, the gap between ϕ_A and ϕ_C is non-trivial, and it is not clear how to systematically derive helper lemmas that can bridge the gap between them. Second, as the proof state evolves during the proving process, the applicability of the helper lemmas may change. It is thus difficult to derive the exact helper lemmas that are useful to the proving process, calling for a dynamic approach that adapts the helper lemmas as proof states change.

Our Solution. We introduce mechanisms to discover helper lemmas in two stages. First, before the VC proving process starts, an *offline* synthesizer discovers lemmas to bridge the gap between ϕ_A and ϕ_C. This is achieved by analyzing the annotated program to derive ϕ_A and then generating lemmas that are aligned with both ϕ_A and ϕ_C. These lemmas can be used to discharge ϕ_C while remaining faithful to the program semantics captured by ϕ_A. Second, as the proof unfolds, an *online* adapter refines these lemmas if they are inapplicable to the proof of ϕ_C. The adaptation is based on real-time feedback from the proof assistant, and typically involves strengthening the lemma statement or revising conflicting representations that the lemma depends on. As a result, the agent can effectively discharge complex VCs that are otherwise intractable, while maintaining the alignment with the program semantics captured by ϕ_A. As we can see in Figure 2(c), the offline synthesizer discovers two important lemmas (i.e., `HL1_addr_le_shift_same_base` and `HL2_addr_le_same_base`), with which ϕ_C (in Figure 2(a)) can be easily discharged.

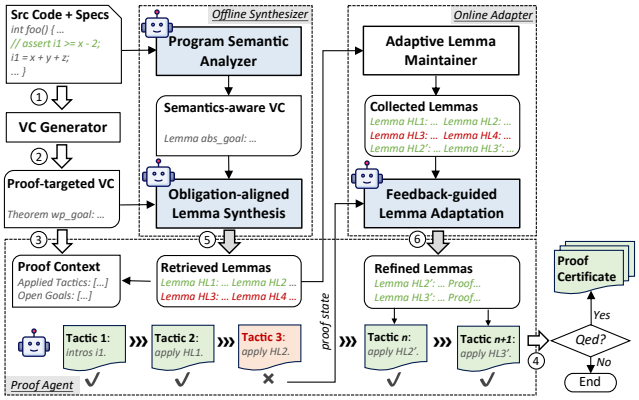


Figure 3: System overview of LemmaNet. In addition to a typical workflow that directly reasons about and proves the VC produced by a VC Generator (①→②→③→④), LemmaNet adopts (1) an offline lemma synthesizer that extracts helper lemmas directly from the source code and specifications ⑤, and (2) an online lemma adapter that refines helper lemmas based on the proof state and existing lemma contexts ⑥.

4 The Design of LemmaNet

4.1 System Overview

Figure 3 presents an overview of LemmaNet. Built on top of an existing tactic-by-tactic proof agent that directly reasons about the proof-targeted VC (①→②→③→④), LemmaNet's novelties lie in two new components: (1) an *offline lemma synthesizer* that synthesizes helper lemmas that are aligned with the program semantics before the prover is invoked (Section 4.2, ⑤), and (2) an *online lemma adapter* that iteratively refines the helper lemmas and their supporting obligations to accommodate evolving proof states during the proving process (Section 4.3, ⑥). Concretely, the offline lemma synthesizer extracts program semantics from the source code and specification via a *program semantic analyzer* (Section 4.2.1), and generates helper lemmas that are aligned with both the semantics-aware VC (derived from program semantics) and the proof-targeted VC (derived from the verification tool). While the bridging may fail due to mismatches between the semantics-aware VC and the proof-targeted VC, the online lemma adapter maintains a dynamic library of helper lemmas via an *adaptive lemma maintainer* (Section 4.3.1), and refines the lemmas on-the-fly in response to evolving proof states through *feedback-guided lemma adaptation* (Section 4.3.2). The two components work in tandem to enable powerful reasoning and adaptability, enabling the agent to effectively discharge complex, otherwise intractable VCs.

4.2 Offline Lemma Synthesizer

To generate helper lemmas that are aligned with the program semantics, we need to have a better understanding of the program semantics from the source code and specification, as well as how the semantics are reflected in the proof obligations. The following subsections explain how we achieve them through an agentic program semantic analyzer and an obligation-aligned lemma synthesizer.

> ***Prompt:*** Your task is to analyze the annotated source code with ACSL annotations and write a complete Rocq file that intuitively proves what the annotation states. Focus only on the highlighted property to prove... Available context:
> - Property name: `hex2bin_loop_invariant_2`
> - Location: function `hex2bin` at line 10 of file `hex2bin.c`
> - Annotated source code: <Figure 1>
>
> ***Response:*** Here is the Rocq file: <Figure 2(b)>

Figure 4: Prompt for program semantic analyzer.

4.2.1 Program Semantic Analyzer. In traditional deductive program verification, human proof engineers usually manually write formal proof obligations based on program semantics and representations [18, 49]. Inspired by this tradition, we ask an LLM to generate a proof obligation by utilizing its capability both in code comprehension [48] and formal reasoning [38]. Such a proof obligation can be captured in an *semantics-aware VC* that is expressed in terms of source-level concepts and is amenable to straightforward proof, which can be easily discharged by existing theorem proving techniques or by simply consulting an LLM.

To this end, we design an agentic program semantic analyzer (PSA) that performs a comprehensive analysis of the code to be verified, with the prompt template shown in Figure 4. The prompt provides detailed information such as the source code, specifications (e.g., pre/post-conditions, loop invariants, and assertions), and other metadata to derive a semantics-aware VC that captures the high-level semantics of the program and the proof obligation. To ease the practical concerns of input size and to focus the analysis on the most relevant parts of the code, we selectively examine portions of the code that are most pertinent to the proof obligations via slicing. We instruct the LLM to document its understanding of the target property as a complete file in Rocq, including the statement *and* its proof with all the supporting definitions. This is possible because the semantics-aware VC is expressed in terms of source-level concepts and is amenable to straightforward proof. We apply a refinement process of up to five iterations to fix syntax errors or incorrect proofs until the output can be compiled with the Rocq compiler. The semantics-aware VC is eventually given to the LLM as part of the prompt to produce potentially useful helper lemmas (see Section 4.2.2) whose proofs are constructed separately. As such, we note that the semantics-aware VC only serves as *guidance*, and its correctness does not affect the soundness of our approach. We ensure unproven statements are never imported into the proving context. In the example from Figure 1, the PSA would analyze the loop structure and the specifications to produce the semantics-aware VC as shown in Figure 2(b)) that captures the key invariant that `src` is monotonically increased by two in each iteration.

4.2.2 Obligation-aligned Lemma Synthesis. The PSA component generates a semantics-aware VC that is guaranteed to be valid as it is checked by the proof assistant. However, there is no formal argument that the LLM-generated semantics-aware VC is faithful to the program semantics, and reasoning exclusively about this VC is inherently *unsound*. As such, helper lemmas are synthesized to bridge the gap between the semantics-aware VC and the proof-targeted

Prompt: Analyze an *equivalent* goal ϕ_C that has been directly discharged from Frama-C using the same code and annotations: <Figure 2(a)>. Based on the previously proved lemma ϕ_A, propose *strong enough helper lemmas* to help prove ϕ_C. Note that you *do not* need to prove ϕ_C itself... Important guidelines:

(1) For each helper lemma proposed, provide its proof.
(2) Helper lemmas can be very specific to the goal (e.g., you may use specific constants).
(3) Describe a step-by-step plan detailing where each helper lemma can be applied.

Response: Here are the helper lemmas: <Figure 2(c)>

Figure 5: Prompt for obligation-aligned lemma synthesis.

Prompt: Here is the current proof state:

(1) Applied tactics: `Proof. intros i1. apply HL1.`
(2) Open goal: `forall i i1 x y: int, …`
(3) Error feedback: Tactic `apply HL2.` failed because term `HL2` has type `x<i` while it is expected to have type `x<i1`.

Here is a list of helper lemmas. If a critical lemma exists but is not applicable, propose a refined version with corrected types:

```
1  Lemma HL1: …  Proof… Qed. (* imported *)
2  Lemma HL2: …  Proof… Qed. (* imported *)
3  Lemma HL3: …  Proof… Qed. (* conflict *)
4  Lemma HL4: …  Proof… Qed. (* conflict *)
```

Response: Here is the refined lemma: `Lemma HL2': …`

Figure 6: Prompt for feedback-guided lemma adaptation.

VC, so that the verifier ultimately discharges the VC produced by the VC generator.

To do so, we design an obligation-aligned lemma synthesis strategy that generates helper lemmas that are specifically tailored to bridge the representation gap between the two VCs, thus simplifying the proving of the proof-targeted VC while ensuring soundness. Figure 5 illustrates the prompt template for this process. The input includes both the semantics-aware VC and the proof-targeted VC. In principle, these two lemmas should have encoded the same program and the same property, albeit produced by different means. The LLM is instructed to compare them and output a set of helper lemmas that are aligned with both VCs and can be used to discharge the proof-targeted VC while remaining faithful to the program semantics captured by the semantics-aware VC. Similar to the PSA component, here the LLM is also required to generate a complete Rocq file, i.e., each helper lemma being proposed comes with a formal argument of correctness. For example, given the input of the semantics-aware VC ϕ_A and the proof-targeted VC ϕ_C from Figure 1, the agent synthesizes helper lemmas such as those shown in Figure 2(c). To avoid generating an excessive number of trivial lemmas, we also export a proof plan that details how each helper lemma should be used to discharge the proof-targeted VC.

We note that the LLM may fail to produce a compilable Rocq file during offline synthesis for properties that involve sophisticated reasoning. When that happens, we discard the lemmas causing compilation errors and recursively remove those dependent on them. All the remaining well-formed helper lemmas and the proof plan are imported into the context of the proof agent to assist the online proving process.

4.3 Online Lemma Adapter

The lemmas synthesized in the offline synthesizer may not necessarily be sufficient to discharge the original verbose VCs. As the synthesizer is not able to *a priori* predict how the proof unfolds, the discovered lemmas may become inapplicable as the proof states evolve. The following subsections explain how we design an online lemma adapter that iteratively refines the helper lemmas and their supporting obligations to accommodate evolving proof states during the proving process, by either strengthening the lemma statement or revising the conflicting representation that the lemma depends on through feedback-guided lemma adaptation.

4.3.1 Adaptive Lemma Maintainer. During the proving process, certain important information (e.g., retrieved helper lemmas) may be potentially useful for discharging the original, verbose VCs, but they are not directly applicable due to various reasons. The adaptive lemma maintainer (ALM) is responsible for maintaining a corpus of helper lemmas and providing them to the agent during the proof process. The helper lemmas maintained can be from various sources, including the initial lemmas synthesized from the previous section, the lemmas proved in the history proofs that these lemmas depend on (often implicitly through the program's control/data dependencies and specification structure), and the refined lemmas from the feedback-guided lemma adaptation in the next section. In principle, other sources (e.g., program source code and specification) can also be provided to the agent as additional information for lemma refinement. Here we particularly focus on the lemmas maintained in ALM as they are more directly related to the proof states and can be more easily reused and refined based on the evolving proof context.

4.3.2 Feedback-guided Lemma Adaptation. Inapplicable lemmas can occur due to type mismatch, i.e., a helper lemma defined over one type (e.g., integer `Z`) is inapplicable on terms of another type (e.g., `addr`). Another common reason is naming conflict, where definitions in the helper lemmas are inconsistent with the ones in the proof state, e.g.,`isint32` is defined with $\leq$ other than a strict inequality. However, these inapplicable lemmas still provide valuable insights and can often become usable with minor adjustments. As such, we design a feedback-guided lemma adaptation approach that refines the helper lemmas and their supporting obligations to accommodate the evolving proof states. Concretely, with access to the helper lemmas maintained in ALM and the current proof state, the agent proceeds with the following three strategies accordingly.

First, the best-case scenario is when the existing helper lemmas in ALM can be applied to the proof-targeted VC. In that case, since the helper lemmas are aligned with both the program semantics and the proof-targeted VC, the agent attempts to apply them directly.

Second, the application of these lemmas may fail due to the gap between the semantics-aware VC and the proof-targeted VC. In this case, the agent refines the helper lemmas based on the feedback from the proof process, by either strengthening the lemma

statement or revising the conflicting representation, and then attempts to apply the refined lemmas to discharge the proof-targeted VC. This is the common case we observed in practice: the initial helper lemmas synthesized in the offline phase may not be directly applicable for normalization or on-the-fly induction, and may require refinement based on the evolving proof states to effectively discharge the proof-targeted VCs. Concretely, Figure 6 shows the prompt template used for the lemma adaptation step, which guides the agent to refine the helper lemmas and their supporting obligations based on the feedback from the proof assistant.

Third, if no lemmas in ALM can be applied to discharge the proof-targeted VC, the agent may propose new helper lemmas that are most relevant to the proof state, which can be synthesized from the agent's understanding of the proof progress, and then attempt to apply them to discharge the proof-targeted VC. Through this iterative process of lemma refinement and application, the agent can effectively discharge complex VCs that are otherwise intractable, while maintaining the alignment with the program semantics captured by the semantics-aware VC.

Running Example. To recap the overall design of LemmaNet as shown in Figure 3, we walk through the example from Figure 2 again to illustrate how the offline lemma synthesizer and the online lemma adapter work together to discharge the proof-targeted VC. In addition to feeding the source code and specification to the VC generator (①→②), LemmaNet involves an offline lemma that digests these artifacts through the PSA component to derive the semantics-aware VC ϕ_A, and then refers to ϕ_C to synthesize helper lemmas to bridge the representation gap. These lemmas are then added to the proof context to assist the proof agent (⑤), and are collected by the adaptive lemma maintainer. Then, during the proving process, the online lemma adapter iteratively refines the helper lemmas based on the feedback from the proof assistant so that lemmas become applicable in the evolved proof context (⑥). Collectively, the two stages of lemma discovery systematically encode semantic information from the program, and enable the proof agent to retrieve them on-the-fly as reusable artifacts.

4.4 Implementation

We implement LemmaNet on top of an existing proof agent, AutoRocq [62], in Python. The system is dependent on the Rocq proof assistant and the Frama-C ecosystem, specifically, the ACSL [30] and the WP plugin [25]. The offline lemma synthesizer is implemented as a separate module that interacts with an LLM and checks its output in the Rocq proof assistant. The offline synthesizer simply imports generated helper lemmas and a proof plan into the context of the proof agent without interacting with it directly. The online lemma adapter is implemented as a dynamic tool for the LLM [51]. The proof agent interacts with it on demand by invoking the function and supplying the necessary arguments. Refined lemma statements are inserted as `assert` embeddings in the proof context, and require separate subproofs to be constructed by the proof agent. The integration of these components allows LemmaNet to effectively discover and apply helper lemmas that are crucial for proving complex VCs. The underlying LLM for the proof agent and all lemma discovery steps is configurable and is set to gpt-5.2-2025-12-11 with temperature 0 for better reproducibility.

5 Evaluation

5.1 Evaluation Setup

To evaluate the efficacy of LemmaNet, we design a comprehensive evaluation that answers the following research questions (RQs):

[RQ1] Is LemmaNet effective in proving verification conditions from real-world C programs?

[RQ2] How does each component of LemmaNet contribute to its overall efficacy?

[RQ3] To what extent are the helper lemmas useful? How are they used in successful proofs?

Benchmarks. We evaluate LemmaNet on a total of 941 verification conditions (VCs) extracted from real-world C programs, which are collected from two existing sources:

- 641 VCs extracted from the **SV-COMP** benchmarks [4], which have been used in the evaluation of [62]. These VCs are extracted from 131 C programs, with an average size of 43 lines of code. The largest base program is a BusyBox utility spanning 428 lines.
- 300 VCs from the test partition of **NTP4VC** [68], the largest benchmark to date in real-world verification conditions. Specifically, these theorems are collected from eight critical C projects, including the Linux kernel (scheduler [35], memory management [20], and string utilities [10]), Contiki OS [6], C++ standard library [9], X.509 parser [19], the UAV autopilot [52], and more. These programs are considerably larger than the SV-COMP benchmarks, with several exceeding 1,000 lines of code. The largest among them, X.509 parser, comprises 5,044 LoC.

Proving VCs from these benchmarks involves reasoning about various language constructs, such as pointers, floating-point arithmetic, and custom data structures. In particular, theorems from NTP4VC tend to be much more complex than those from SV-COMP due to the larger program sizes, posing significant challenges to existing approaches [68]. Regarding the type of properties being proved, the 941 real-world VCs can be broadly grouped into four categories. A large portion of them (391, 41.6%) are related to loops, including loop invariants/variants and loop assignments; 237 (25.2%) VCs specify RTE-freeness properties such as non-overflow and valid memory accesses; 163 (17.3%) VCs encode functional correctness through assertions; finally, 150 (15.9%) VCs are extracted from functional contracts, including pre/post-conditions and behavioral partitioning (e.g., `disjoint` and `complete`). Collectively, these benchmarks form an ideal testbed for evaluating VC proving tools.

Comparative Approaches. We compare LemmaNet with the following approaches that are applicable for proving program VCs:

- **AutoRocq** [62], an LLM agent that collaborates with the Rocq ITP and retrieves additional context on demand to conduct proof search. It is the base proof agent upon which LemmaNet is built.
- **Copra** [59], an LLM agent that generates proofs through in-context learning, searching, and knowledge retrieval.
- **CoqHammer** [12], a popular tool that uses machine-learned heuristics and SMT solving to automate theorem proving.

We include AutoRocq and Copra as they are state-of-the-art LLM-based theorem provers, as evidenced by prior work [62]. We include CoqHammer as the non-agentic baseline that has been reported to

Table 1: Results on proving verification conditions by different approaches and ablative variants: the number of successfully proved VCs among SV-COMP (max. 641) and NTP4VC (max. 300) benchmarks, as well as the relative improvement (Improv.) of LemmaNet.

Tools	SV-COMP	NTP4VC	Total	Improv.
LemmaNet	298	66	364	–
AutoRocq [62]	247	40	287	+26.8%
Copra [59]	209	31	240	+51.7%
CoqHammer [12]	85	38	123	+195.9%
LemmaNet (¬Ofl)	273	55	328	+11.0%
LemmaNet (¬Onl)	246	56	302	+20.5%
LemmaNet (¬PSA)	253	48	301	+20.9%

outperform naive LLM-based approaches [68]. We additionally include three ablative variants of LemmaNet in Section 5.3 to evaluate the contribution of individual components. Unfortunately, existing tools in automated lemma discovery [7, 43, 57] target mathematical theorems only, and cannot be applied in our setting. We delay the detailed discussion and qualitative comparison to Section 7. In our evaluation, we use the same backend LLM gpt-5.2-2025-12-11 for all tools, with temperature set to 0. LemmaNet, AutoRocq, and Copra are run for a budget of 10 minutes or 100 steps, following the setup in [62]. CoqHammer, which does not involve an LLM, is run with the default setting and the same 10-minute timeout.

5.2 RQ1: Efficacy in VC Proving

For this research question, we investigate LemmaNet's effectiveness in proving real-world verification conditions. We report the number of proved VCs from each comparison tool in Table 1.

For the SV-COMP benchmark, LemmaNet is able to prove 298 out of 641 VCs, outperforming the best-performing baseline, AutoRocq, by 20.6%. Copra is able to prove 209 VCs. CoqHammer only proves 85 VCs within the 10-min timeout, lagging behind three agent-based proving techniques by a large margin. In contrast, the success rates across the NTP4VC benchmark drop across all tools due to increased complexity in the VC statements. LemmaNet is able to prove 66 out of 300 VCs, outperforming AutoRocq (40) substantially by 60%. AutoRocq is closely followed by CoqHammer, which manages to prove 38 VCs. Across all evaluated VCs, LemmaNet improves over comparison tools by 26.8%–195.9%, demonstrating its remarkable efficacy in proof generation relative to existing approaches.

In addition, we present Figure 7, a Venn diagram that visualizes the relation among the set of VCs proved by each tool. The results are aggregated across both benchmarks. The diagram shows that LemmaNet proves the largest number of VCs uniquely (78), significantly outperforming comparison tools. LemmaNet is followed by CoqHammer, which obtains 26 unique successes despite proving the least number of VCs overall, highlighting the ability of SMT-based approaches to effectively complement agentic approaches.

To better understand the efficacy of LemmaNet, we categorize the successfully proved VCs along two dimensions and report the resulting breakdown for each tool in Figure 8. Figure 8(a) correlates

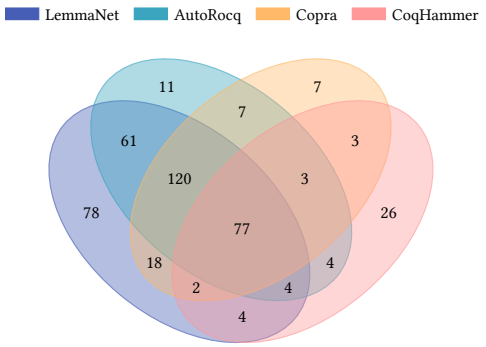
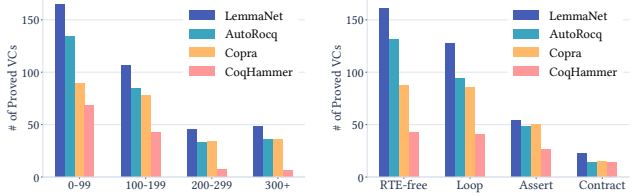


Figure 7: Venn diagram of the number of lemmas proved by each tool. LemmaNet uniquely proves the most VCs.



(a) Breakdown by term complexity. (b) Breakdown by property type.

Figure 8: The number of VCs proved by different tools: a breakdown by complexity (# of terms) and property type.

the number of proved VCs with their complexity. Here, the complexity of a VC is measured by the number of terms contained in the theorem statement, and a larger count (counts increase from left to right on the x-axis) indicates higher complexity [62]. On the other hand, Figure 8(b) groups proved VCs by the type of property they denote, as discussed in Section 5.1. In both plots, LemmaNet consistently leads across *all* buckets. These results highlight the effectiveness of LemmaNet in comparison with other tools in proving VCs from real-world programs. The results also indicate that helper lemma discovery is widely applicable across the full range of VC complexities and property types.

Answer to RQ1: In comparison with the other tools in the study, LemmaNet is highly effective in proving VCs from real-world programs. It uniquely proves the largest number of VCs and consistently outperforms comparison tools across the full range of VC complexities and types.

5.3 RQ2: Ablation Studies

To answer RQ2, we conduct ablation studies to understand how much each LemmaNet component contributes to the overall effectiveness by comparing LemmaNet with the following variants:

- **LemmaNet (¬Ofl)**, a variant with only the online adapter, i.e., the offline synthesizer is disabled;

- **LemmaNet (¬Onl)**, a variant with only the offline synthesizer, i.e., the online adapter is disabled;
- **LemmaNet (¬PSA)**, a variant that removes the program semantic analyzer (PSA) component, i.e., offline helper lemmas are generated directly from the proof-targeted VC without semantic knowledge about the annotated source code.

Each of the first two variants, ¬Ofl and ¬Onl, individually disables one stage of the lemma discovery process, namely ⑤ and ⑥ in Figure 3. We include the third ablative variant, ¬PSA, to evaluate the effectiveness of program-guided offline lemma discovery.

We evaluate each variant on the entire benchmark set of 941 VCs. The last section of Table 1 presents results from these variants. These results show that all LemmaNet components make meaningful contributions to its overall effectiveness. In particular, the comparison with variant LemmaNet (¬PSA) highlights the importance of program comprehension in generating semantics-aligned helper lemmas that are conducive to online proving.

Answer to RQ2: The offline lemma synthesizer, the online lemma adapter, and the program semantic analyzer all contribute to the overall effectiveness of LemmaNet.

5.4 RQ3: Understanding Helper Lemmas

We next delve into the details of the helper lemmas to better understand the lemmas that are proposed and how they are used in successful proofs. To this end, we first report aggregated statistics for LemmaNet's lemma discovery phases along four dimensions, and then present two case studies on how they are used in the corresponding proofs.

5.4.1 Quality. We first study *how good* the discovered helper lemmas are. Unfortunately, a direct, similarity-based measure of their quality [7, 57] is infeasible in this case due to the lack of human-written ground truths. As such, we resort to an indirect measure by counting how many successful proofs make use of the proposed lemmas (e.g., via `apply` or `rewrite`). We report the counts in Table 2 for all 364 VCs proved by LemmaNet and 78 uniquely proved VCs. Among the 78 unique successes, only 9 (11.5%) of the proofs do not require an offline or online lemma. 32 (41.0%) require offline, but not online lemmas, 17 (21.8%) require online, but not offline lemmas, and 20 (25.7%) require both kinds of lemmas. These numbers highlight the usefulness of these lemmas. The percentages are lower when considering all successes, as many simpler theorems can be directly proved without relying on helper lemmas. Overall, offline lemmas synthesized tend to be used more often, indicating the importance of program awareness in lemma discovery.

5.4.2 Cardinality. We next report statistics on *how many* helper lemmas are discovered by LemmaNet and subsequently used in successfully proved VCs (max. 212=57+95+60). We present the results in Figure 9, where the x-axis lists the VCs ordered by the number of lemmas discovered (i.e., y-axis). A few patterns are visible from the diagram. First, in general, more helper lemmas are discovered in the offline stage (blue) compared to the online phase (orange). Second, discharging the proof-targeted VC typically only requires a few key helper lemmas, as only a small fraction of lemmas discovered offline are being directly used in proofs. Third, we observe that

Table 2: Quality of helper lemmas: # of successful proofs that make use of both, either, or none of online/offline lemmas, reported for all and unique VCs proved by LemmaNet.

	Both	Offline	Online	None	Total
All	57	95	60	152	364
	15.7%	26.1%	16.5%	41.7%	100%
Unique	20	32	17	9	78
	25.7%	41.0%	21.8%	11.5%	100%

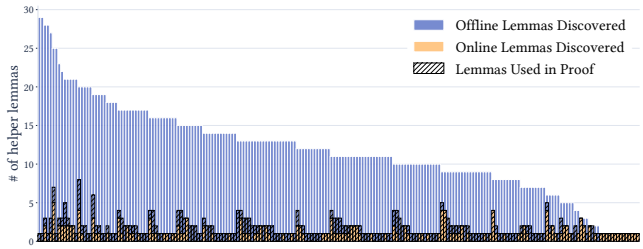


Figure 9: Cardinality of helper lemmas: # of offline/online lemmas discovered and used for successfully proved VCs.

the lemmas discovered online, although fewer than offline lemmas, are often immediately useful to the proof. This is expected as the offline discovery phase is unaware of proof states, whereas the online adaptation is driven by real-time feedback from Rocq.

5.4.3 Utility Taxonomy. We report *what purpose* the helper lemmas serve in Table 3 by categorizing them based on keyword matching. We report the categorization results for (1) the list of all helper lemmas discovered, and (2) the ones being used in successful proofs. The most common category for both cases is memory-related helper lemmas that contain keywords such as `addr` and `ptr`, with Figure 2(c) being concrete examples. 28.1% of all discovered lemmas and 30.2% of used lemmas fall into this category. This is followed by simplification lemmas that rewrite, reduce, or split a large theorem into smaller or simpler (sub-)goals that are easier to reason about. Other popular categories include typing bridges (e.g., mapping a `uint32` variable to its range as integer `Z`) and arithmetic lemmas (e.g., expressing left-shifting as multiplication), each accounting for over 10% of the lemmas.

5.4.4 Proof Strategy Taxonomy. We further inspect all VCs that are uniquely proved by LemmaNet with the help of helper lemmas (max. 69=20+32+17). We manually categorize how these successful proofs make use of helper lemmas based on the proof strategies they enable. We identify four primary strategies:

- *BRIDGE*: connecting representations across semantic levels,
- *REDUCE*: search-space reduction,
- *REWRITE*: normalization or term rewriting, and
- *STRENGTHEN*: deriving stronger usable facts.

Among the 69 proofs that utilize helper lemmas, BRIDGE is the most prevalent, appearing in 61 proofs (88.4%), underscoring the critical role of helper lemmas in connecting tool-specific representations to program-level semantics. REDUCE follows closely at 40 proofs

Table 3: Utility taxonomy of helper lemmas: a breakdown by category. Results are reported for both discovered (% Disc.) and used (% Used) lemmas. Categories are determined through keyword matching on lemma names.

Category	% Disc.	% Used	Example Keywords
Memory	28.1%	30.2%	`addr, base, ptr, mem`
Simplification	25.6%	22.0%	`simpl, rewrite, split`
Typing	19.3%	18.8%	`uint32, float, INT_MAX`
Arithmetic	13.5%	17.2%	`mul, div, mod, lxor`
Data Structure	7.8%	9.0%	`array, list, heap, map`
String	3.5%	2.4%	`str, char, tolower`
Others	2.2%	0.4%	–

(58.0%), while REWRITE and STRENGTHEN appear in 34 (49.3%) and 33 (47.8%) proofs, respectively. Other strategies such as case-splitting, contradiction-exposing, and modularization are used less frequently (under 6% each). Notably, most proofs employ multiple strategies—averaging 2.5 per proof—demonstrating that effective VC proving requires helper lemmas serving complementary roles.

5.4.5 Case Studies. We present representative case studies that illustrate how helper lemmas facilitate VC proving. The theorem statements (i.e., proof-targeted VCs) have been simplified for clarity.

Case I: Refined helper lemma from failed proofs (Listing 1). This VC from the Linux kernel scheduler requires proving `valid_rd t a_5 1`, where `a_5 = shift a_4 0`. Although `valid_rd t a_4 1` is in the hypotheses, Rocq cannot automatically conclude validity for `a_5`. The online adapter synthesizes `HL_valid_rd_shift0` when it fails to apply `HL_valid_rd_rewrite`, stating that shifting by zero preserves memory validity, bridging the gap and completing the proof.

```
Lemma HL_valid_rd_rewrite :
  forall (t : Z → Z) (a b : addr) (n : Z),
    valid_rd t a n → a = b → valid_rd t b n.
Proof. intros t a b n H Hab. eapply valid_rd_ext_addr; eauto. Qed.

Theorem wp_goal_reduced
  (a_1 : addr) (t_4 : addr → addr) (t : Z → Z) :
  let a_3 : addr := shift a_1 4%Z in let a_4 : addr := t_4 a_3 in
  let a_5 : addr := shift a_4 0%Z in valid_rd t a_4 1%Z →
  valid_rd t a_5 1%Z.
Proof.
  intros; cbn.
  assert (HL_valid_rd_shift0: forall (t:Z→ Z) (p:addr) (n:Z),
      valid_rd t p n → valid_rd t (shift p 0%Z) n). {
  intros t0 p n Hrd Hpos; destruct (Hrd Hpos) as [Hb [Hoff Hle]];
  repeat split; try assumption; simpl; lia. }
  exact (HL_valid_rd_shift0 t a_4 1 H22).
Qed.
```

Listing 1: Refined helper lemma from failed proofs.

Case II: Combining offline and online helper lemmas (Listing 2). This VC from a Contiki OS memory allocator requires proving `-2147483648 <= i * x_1` (no signed underflow), given `0 <= i` and `is_uint16 x_1`. LemmaNet employs two lemmas: `HL_uint16_nonneg` (from offline), extracting the fact `0 <= x_1` from the type predicate, and `HL_mul_nonneg_of_nonneg` (from online), concluding a non-negative product from non-negative factors. This illustrates offline-online synergy: domain-specific type knowledge combined with arithmetic reasoning.

```
Theorem wp_goal_reduced (a : addr) (i : Z) (t_1 : addr → Z) :
  let x_1 : Z := t_1 (shift a 0%Z) in 0%Z <= i →
  is_uint16 x_1 → -2147483648%Z <= i * x_1.
Proof.
  intros; cbv zeta.
  assert (HL_Hx1_nonneg : 0 <= x_1) by (apply HL_uint16_nonneg; exact H9).
  assert (HL_Hmul_nonneg : 0<=i*x_1) by (apply HL_mul_nonneg_of_nonneg;
      [exact H0 | exact HL_Hx1_nonneg]). lia.
Qed.
```

Listing 2: Combining offline and online helper lemmas.

Answer to RQ3: The helper lemmas discovered by LemmaNet are highly useful in VC proving. They bridge representation gaps and provide key intermediate results, enabling reasoning steps that are otherwise inaccessible to the agent.

6 Discussion

Experimental Costs. We report the LLM API costs for running LemmaNet. For offline synthesis, the mean cost per VC is $0.12 (SV-COMP) and $0.17 (NTP4VC), with medians of $0.10 and $0.14, respectively. For the proof agent phase (which includes online adaptation), the mean cost is $0.36 (SV-COMP) and $1.03 (NTP4VC), with medians of $0.07 and $0.37. The higher costs for NTP4VC reflect its greater VC complexity. Overall, these costs are comparable to AutoRocq [62] and Copra [59], and are modest given the significant improvements in proving capability. The average proving time is 165.2 seconds per VC in our evaluation, making LemmaNet a practical tool for real-world verification tasks.

Soundness. LemmaNet is sound by design: if a proof is successfully produced, the target VC generated by the VC generator is eventually discharged, with all its dependent helper lemmas formally stated and machine-checked in Rocq. The trusted computing base includes the small Rocq and Frama-C kernels — the underlying LLM is not part of the trusted computing base, and the soundness guarantees hold in the face of arbitrary LLM behavior.

Failure Analysis. We manually examined all (25=11+7+3+4) VCs, as shown in Figure 7, that are proved by AutoRocq but not by LemmaNet, and classified the root cause as follows:

- *Import errors* (12/25): importing offline lemmas into the proof context fails due to conflicts such as inconsistent definitions.
- *Over-fixation on lemmas* (7/25): LemmaNet expends too many steps on adapting/proving new lemmas online, whereas AutoRocq finds similar, existing lemmas in the context and applies them directly.
- *Divergent strategies* (6/25): the additional context confuses the LLM, steering LemmaNet toward a different proving strategy altogether and leading to early aborts.

Threats to Validity. Potential biases in benchmark selection may threaten the internal validity of our evaluation. To mitigate this, we work with two benchmark sets from prior work covering a range of different verification domains, including both SV-COMP programs and complex real-world C programs, ranging across OS modules, parsers, standard libraries, and algorithms. The external validity of our results may be limited by the specific configurations and versions of the tools used in our evaluation. To mitigate this threat, we have used the same version of Rocq (i.e., 8.18) and the

same configurations (i.e., gpt-5.2-2025-12-11 with temperature 0) to ensure a fair comparison. There is also a potential threat of data leakage, as the benchmarks used in our evaluation may have been seen by the underlying LLM during training, potentially leading to bloated results. To the best of our knowledge, no ground-truth proofs of either benchmark are publicly available, so we evaluate the risk of data leakage as minimal.

Limitations and Future Work. While LemmaNet demonstrates significant improvements in proving complex VCs, it has limitations. First, offline lemmas in our evaluation rarely require substantial proofs, because each lemma typically captures a single utility (Table 3) and is therefore straightforward to prove. As such, the offline synthesizer may fail to produce useful lemmas for VCs requiring deep domain knowledge or intricate reasoning. Future work could incorporate domain-specific heuristics or richer program context to improve lemma quality. Second, the online adapter currently depends on existing lemmas and proof assistant feedback to guide refinement, and can be invoked anytime during the proving process as the agent sees fit. This sometimes leads to over-fixation on lemma adaptation, as suggested by the failure analysis. More principled adaptation strategies may warrant further investigation.

7 Related Work

Lemma Discovery. LemmaNet is closely related to prior work on automated helper lemma synthesis [28] to assist automated theorem proving. Existing approaches broadly fall into two categories. *Bottom-up* approaches, also referred to as theory exploration, iteratively conjecture propositions from terms and theorems available in the proving context [29, 33, 43, 45]. On the other hand, *top-down* approaches synthesize lemmas by analyzing a given proof state [7, 57], typically through generalization or rewriting. These tools are often used in conjunction with neural theorem provers to recover from a failed proof. For both approaches, the conjectures are often constructed from templates and validated through counterexample checking [16]. To the best of our knowledge, however, existing approaches are limited to mathematical lemmas in practice due to their inability to instantiate counterexamples involving non-primitive types. LemmaNet similarly features a bottom-up phase, through interpretation of both the semantics-aware and proof-targeted VC offline (Section 4.2), and a top-down phase, through on-the-fly refinement to accommodate evolving proof states online (Section 4.3). LemmaNet synthesizes both the statement and the proof of helper lemmas in Rocq to ensure their validity, instead of searching for counterexamples opportunistically.

Failure Recovery in Proof Automation. Recovering from failed proof attempts is critical for proof automation systems. Earlier works take binary signals from the proof assistant and retry in case of failures [53, 55, 60], whereas others leverage more granular diagnostic messages from the ITP [38, 59, 62]. These error messages can be helpful for both whole-proof generation [24] and heuristic repair of tactics [38, 39]. Agentic approaches have introduced more flexibility into the recovery stage by optionally restoring to a prior state [59] or gathering additional context [62]. LemmaNet continues this line of work on leveraging the feedback from the ITP. However, rather than focusing on failed tactics, LemmaNet is able to recover at the level of helper lemmas through different means of adaptation. This allows LemmaNet to refine lemmas curated from other sources (e.g., by the offline synthesizer) and apply them in new proof contexts, significantly reducing the proving efforts.

Agentic Software Quality Assurance. Beyond theorem proving, LLMs have also been applied to upstream stages of the deductive verification pipeline, such as automated inference of specifications [21, 41, 63, 64] and assertion hints [46, 69]. More broadly, it has transformed a wide spectrum of software quality assurance techniques [22, 27], e.g., model checking [65, 72], test generation [17, 40, 61, 67], and static analysis [26, 37, 66]. Among these approaches, deductive verification offers unmatched mathematical rigor, making it indispensable for safety-critical systems such as the Linux kernel. LemmaNet advances this frontier by enabling scalable, semantics-driven proof automation that can discharge complex VCs from real-world C programs, addressing a longstanding challenge in the field.

8 Conclusion

We presented LemmaNet, a semantic proof agent that synthesizes and adapts helper lemmas for proving verification conditions from real-world C programs. LemmaNet integrates an offline lemma synthesizer, which derives helper lemmas from program semantics, with an online lemma adapter that refines lemmas through feedback-guided adaptation during proof search, thus bridging high-level program semantics and low-level tool-specific details. More broadly, LemmaNet enables semantics-driven proof automation by grounding LLMs' informal reasoning over program semantics to rigorous helper lemmas. As AI-generated code proliferates, scalable automated verification becomes essential; we envision this paradigm enabling trustworthy deployment of AI-synthesized programs through formal correctness guarantees.

Acknowledgments

We would like to sincerely thank all the anonymous reviewers for their valuable feedback and insights. This research is supported by the National Research Foundation, Singapore, under its Artificial Intelligence (AI)-for-Science (AI4S) Challenge Grant (Award No. NRFAI4SCH-2025-0003), called "AI for Program Reasoning". Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation.

Disclosure of Generative AI Usage

LLMs were used in the work as they form a fundamental part of the approach. In particular, ChatGPT was used as the backend by LemmaNet and comparison tools in the evaluation. ChatGPT were also used by the authors during the development of LemmaNet, and for grammatical edits of the paper. The authors validated the results as needed and take full responsibility for the final content.

Data Availability Statement

We release LemmaNet, including its implementation, benchmarks, and replication instructions, for academic use at the following link:

https://github.com/NUS-Program-Verification/LemmaNet.

References

[1] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. 2010. Deductive verification of cryptographic software. *Innovations in Systems and Software Engineering* 6, 3 (2010), 203–218. doi:10.1007/S11334-010-0127-Y

[2] Yoav Alon and Cristina David. 2025. Integrating Large Language Models and Reinforcement Learning for Non-Linear Reasoning. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 957–977. doi:10.1145/3715761

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 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

[4] 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, Cham, 151–186. doi:10.1007/978-3-031-90660-2_9

[5] 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

[6] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. Ghosts for lists: a critical module of Contiki verified in Frama-C. In *NASA Formal Methods Symposium*. Springer, 37–53. doi:10.1007/978-3-319-77935-5_3

[7] Ana Brendel, Aishwarya Sivaraman, and Todd Millstein. 2025. Synthesizing Implication Lemmas for Interactive Theorem Proving. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 2254–2278. doi:10.1145/3763131

[8] Yuriy Brun, Saikat Chakraborty, Claire Le Goues, Corina Păsăreanu, and Adish Singla. 2026. Automatically Engineering Trusted Software: A Research Roadmap. *ACM Transactions on Software Engineering and Methodology* (March 2026). doi:10.1145/3779132

[9] Jochen Burghardt, Jens Gerlach, and Timon Lapawczyk. 2015. ACSL by example. https://publica.fraunhofer.de/handle/publica/297476

[10] Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, and Aaron Tomb. 2014. Formal Verification of kLIBC with the WP Frama-C Plug-in. In *NASA Formal Methods Symposium*. Springer, 343–358. doi:10.1007/978-3-319-06200-6_29

[11] Projet Coq. 1996. The Coq proof assistant: reference manual. *INRIA Rocquencourt and ENS Lyon, version* 5 (1996), 7–1. https://rocq-prover.org/doc/V8.2pl3/refman/Reference-Manual001.html

[12] 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

[13] 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

[14] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388. doi:10.1007/978-3-319-21401-6_26

[15] Google DeepMind. 2024. AlphaProof. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/

[16] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*, Vol. 125. 126. https://catalin-hritcu.github.io/students/topics/2014/quick-chick.html

[17] 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 (ICSE)*. doi:10.1145/3597503.3623343

[18] Josiah Dodds and Andrew W Appel. 2013. Mostly sound type system improves a foundational program verifier. In *International Conference on Certified Programs and Proofs*. Springer, 17–32. doi:10.1007/978-3-319-03545-1_2

[19] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a RTE-free X.509 parser. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2019)*, Vol. 186. 1–30. https://github.com/ANSSI-FR/x509-parser

[20] 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*. Springer, 216–234. doi:10.1007/978-3-030-03421-4_15

[21] 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:10.1145/3660791

[22] 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 (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[23] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428299

[24] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023-11-30) *(ESEC/FSE 2023)*. Association for Computing Machinery, 1229–1241. doi:10.1145/3611643.3616243

[25] Frama-C Developers. 2026. Weakest Precondition (WP) Plug-in in Frama-C. Accessed: 2026-03-23. https://www.frama-c.com/fc-plugins/wp.html

[26] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2026. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In *Forty-second International Conference on Machine Learning*. 1–18. doi:10.48550/arXiv.2501.18160

[27] 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). doi:10.1145/3695988

[28] Moa Johansson. 2019. Lemma discovery for induction: a survey. In *International Conference on Intelligent Computer Mathematics*. Springer, 125–139. doi:10.1007/978-3-030-23250-4_9

[29] Moa Johansson, Lucas Dixon, and Alan Bundy. 2011. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning* 47, 3 (2011), 251–289. doi:10.1007/s10817-010-9193-y

[30] 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

[31] 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

[32] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. 2014. Formal C Semantics: CompCert and the C Standard. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 543–548. doi:10.1007/978-3-319-08970-6_36

[33] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: e-graph guided lemma discovery for inductive equational proofs. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 818–844. doi:10.1145/3674653

[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 (extended version). (2023). https://doi.org/10.48550/arXiv.2303.05491

[35] Julia Lawall, Keisuke Nishimura, and Jean-Pierre Lozi. 2024. Should we balance? Towards formal verification of the Linux kernel scheduler. In *International Static Analysis Symposium*. Springer, 194–215. doi:10.1007/978-3-031-74776-2_8

[36] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference*, Vol. 6355. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[37] 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

[38] 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*. 1509–1520. doi:10.1145/3691620.3695521

[39] Minghai Lu, Zhe Zhou, Danning Xie, Songlin Jia, Benjamin Delaware, and Tianyi Zhang. 2025. *Adaptive Proof Refinement with LLM-Guided Strategy Selection*. arXiv:2510.25103 [cs] doi:10.48550/arXiv.2510.25103

[40] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. 2026. Agentic Concolic Execution. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 37–55. doi:10.1109/sp63933.2026.00003

[41] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 16–28. doi:10.1109/icse55347.2025.00129

[42] 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*. 237–248. doi:10.1145/1706299.1706329

[43] Roy L McCasland, Alan Bundy, and Patrick F Smith. 2017. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence* 47, 3 (2017), 585–606. doi:10.1007/s10489-017-0954-8

[44] 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

[45] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. 2012. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications* 39, 2 (2012), 1637–1646. doi:10.1016/j.eswa.2011.06.055

[46] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking automated verification with Large Language Models. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1519–1545. doi:10.1145/3720499

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

[48] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639187

[49] Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 355–369. doi:10.1007/978-3-540-70545-1_34

[50] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer. doi:10.1007/3-540-45949-9

[51] OpenAI. 2025. *Function Calling Developer Documentation*. https://developers.openai.com/api/docs/guides/function-calling

[52] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux. 2021. Verifying the mathematical library of an UAV autopilot with Frama-C. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 167–173. doi:10.1007/978-3-030-85248-1_10

[53] 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* (London, UK) *(MAPL 2020)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3394450.3397466

[54] 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* (2023). doi:10.1145/3593374

[55] Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Dylan Zhang, Talia Ringer, and Yuriy Brun. 2024. QEDCartographer: Automating formal verification using reward-free Reinforcement Learning. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 405–418. doi:10.1109/ICSE55347.2025.00033

[56] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. 2023. *Formal verification: an essential toolkit for modern VLSI design*. Elsevier. doi:10.1016/C2013-0-18672-2

[57] 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:10.1145/3563306

[58] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2017. Automated lemma synthesis in symbolic-heap separation logic. *Proceedings of the ACM on Programming Languages* 2, POPL (2017). doi:10.1145/3158097

[59] 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 (COLM)*. doi:10.48550/arXiv.2310.04353

[60] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, Joao F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 347–359. doi:10.1109/icse55347.2025.00161

[61] 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 *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2064–2082. doi:10.1109/sp63933.2026.00110

[62] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. 2026. Agentic Verification of Software Systems. *Proceedings of the ACM on Software Engineering* FSE (2026). doi:10.1145/3808164

[63] Zhongyi Wang, Tengjie Lin, Mingshuai Chen, Haokun Li, Mingqi Yang, Xiao Yi, Shengchao Qin, Yixing Luo, Xiaofeng Li, Bin Gu, Liqiang Lu, and Jianwei Yin. 2026. A Tale of 1001 LoC: Potential Runtime Error-Guided Specification Synthesis for Verifying Large-Scale Programs. *Proc. ACM Program. Lang.* OOPSLA1 (2026). doi:10.1145/3798268

[64] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by Large Language Models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[65] 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 (ASE)*. 406–417. doi:10.1145/3691620.3695014

[66] 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

[67] 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*. doi:10.1145/3597503.3639121

[68] Qiyuan Xu, Xiaokun Luan, Renxi Wang, Joshua Ong Jun Leang, Peixin Wang, Haonan Li, Wenda Li, and Conrad Watt. 2026. Neural Theorem Proving for Verification Conditions: A Real-World Benchmark. In *The Fourteenth International Conference on Learning Representations*. 1–27. doi:10.48550/arXiv.2601.18944

[69] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 396 (2025), 29 pages. doi:10.1145/3763174

[70] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994. doi:10.48550/arXiv.1905.09381

[71] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma synthesis for automating induction over algebraic data types. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 600–617. doi:10.1007/978-3-030-30048-7_35

[72] 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 2026-03-26; accepted 2026-06-18