# OSVBENCH: Benchmarking LLMs on Specification Generation Tasks for Operating System Verification

**Shangyu Li**[1], **Juyong Jiang**[1,2], **Tiancheng Zhao**[3], **Jiasi Shen**[1]

[1]The Hong Kong University of Science and Technology
[2]The Hong Kong University of Science and Technology (Guangzhou)
[3]Georgia Institute of Technology
{sliew,jjiang472}@connect.ust.hk, tzhao350@gatech.edu, sjs@cse.ust.hk

## Abstract

We introduce OSVBENCH, a new benchmark for evaluating Large Language Models (LLMs) on the task of generating complete formal specifications for verifying the functional correctness of operating system kernels. This benchmark is built upon a real-world operating system kernel, Hyperkernel, and consists of 245 complex specification generation tasks in total, each of which is a long-context task of about 20k-30k tokens. The benchmark formulates the specification generation task as a program synthesis problem confined to a domain for specifying states and transitions. This formulation is provided to LLMs through a programming model. The LLMs must be able to understand the programming model and verification assumptions before delineating the correct search space for syntax and semantics and generating formal specifications. Guided by the operating system's high-level functional description, the LLMs are asked to generate a specification that fully describes all correct states and transitions for a potentially buggy code implementation of the operating system. Experimental results with 12 state-of-the-art LLMs indicate limited performance of existing LLMs on the specification generation task for operating system verification. Significant disparities in their performance highlight differences in their ability to handle long-context code generation tasks.

**Code** — https://github.com/lishangyu-hkust/OSVBench
**Extended version** — https://arxiv.org/abs/2504.20964

## 1 Introduction

Large Language Models (LLMs) have shown great potential in software engineering tasks, such as code generation (Austin et al. 2021; Athiwaratkun et al. 2022; Zan et al. 2023; Jiang et al. 2025), code summarization (Ahmed et al. 2024), and bug repair (Jin et al. 2023). An important aspect of software engineering is software verification. Software verification uses rigorous mathematical reasoning to prove the absence of bugs in software (Dahl, Dijkstra, and Hoare 1972), which is essential in ensuring the correctness of software in safety-critical domains such as aerospace, healthcare, and nuclear energy (Klein et al. 2009; Amani et al. 2016; O'Connor et al. 2016), where software bugs could lead to catastrophic economic losses or even endanger human lives.



A class of software for which verification is especially valuable is that of operating system (OS) kernels, which are fundamental components of many critical infrastructures. Yet the inherent complexity, concurrency, and hardware interactions of OS kernels render the verification process highly challenging. Manual software verification requires advanced knowledge of formal methods and program analysis, and there is a shortage of professionals who are capable of conducting such verification (Klein et al. 2009). Verifying the seL4 microkernel (Klein et al. 2009) required 11 person-years of effort for 10k lines of C code, and verifying two operations of the BilbyFs file system required 9.25 person-months of effort (Amani et al. 2016) for 1,350 lines of code. These challenges highlight the need for automation.

Formally verifying an OS kernel often involves (1) defining formal specifications that characterize the properties that must be satisfied by the kernel (Klein et al. 2009; Chajed et al. 2022; Chen et al. 2015, 2017) and (2) constructing formal proofs with a theorem prover to show that the code implementation satisfies the specifications. Existing research focuses mainly on automating proof generation (Chen et al. 2024; Zhang, Lu, and Duan 2024) and often overlooks the crucial process of specification generation (Sammler et al. 2021; Leino 2010; Jacobs and Piessens 2008; Ma et al. 2024). Generating specifications is especially challenging for OS kernel verification because such specifications are often ad hoc (Chen et al. 2017, 2015; Chajed et al. 2022; Sigurbjarnarson et al. 2016) and require substantial domain expertise. For instance, creating the formal specification for seL4 (Klein et al. 2009) required 7 person-months of effort.

We introduce OSVBENCH, a benchmark suite for evaluating the capabilities of LLMs in generating formal specifications for verifying the functional correctness of an OS kernel. OSVBENCH is derived from the Hyperkernel project (Nelson et al. 2017) and consists of 245 specification generation tasks, each of which asks an LLM to generate a specification based on the code implementation of a system call in the OS kernel along with its natural language functional description. To capture realistic scenarios where OS kernels may have vulnerabilities, we inject five types of bugs into the code implementation before providing it to the LLMs.

Each task in OSVBENCH is a sophisticated program synthesis problem with a long context of approximately 20k to 30k tokens. This design enables us to investigate the capabilities of LLMs in understanding and manipulating extensive contextual information and domain knowledge.

We evaluate 12 state-of-the-art LLMs on OSVBENCH. The experimental results show that the LLMs exhibit limited performance in automating formal specification generation. Significant disparities in their performance on the benchmark highlight differences in their ability to handle long-context code generation tasks. We also discuss the impact of varying types and quantities of injected bugs on the quality and effectiveness of the generated specifications.

## 2 Related Work

**LLM for software verification.** Software verification (D'silva, Kroening, and Weissenbacher 2008) ensures that software conforms to specified properties or requirements, playing a critical role in ensuring reliability and correctness. In this domain, operating system kernel verification (Klein et al. 2014) has been a central research goal for ensuring the reliability and security of critical software systems. Early foundational work includes efforts such as UCLA Secure Unix (Walker, Kemmerer, and Popek 1980), PSOS (Feiertag, Levitt, and Robinson 1977), and KIT (Bevier 1989), which laid the groundwork for formal approaches to kernel correctness. Recent progress has expanded to leverage formal methods like theorem proving (Nelson et al. 2017) and model checking (Klein et al. 2009), aiming for high-assurance kernels with mathematically verified properties. Moreover, prior work on leveraging LLMs for software verification mainly focused on generating proofs from specifications (Chen et al. 2024; Zhang, Lu, and Duan 2024), which involves translating one formal semantic representation (specifications, in various forms) into another (proofs expressed in formal languages). Some studies focus on automatically generating verification code from given code snippets and their pre- and postconditions to prove that they satisfy a specified set of properties (Loughridge et al. 2024; Misu et al. 2024), whereas others design a consistency-checking loop among code, docstrings, and formal annotations to filter out incorrect code (Sun et al. 2024). Although some studies have explored the task of specification generation, much of this work has focused on general-purpose specification generation (Ma et al. 2024; Cao et al. 2025), which differs significantly from the generation of OS kernel specifications due to the distinct verification assumptions and requirements encountered in this domain.

**LLM for code generation.** Formal specifications are a special form of source code. Code generation and program synthesis have been extensively studied over the past few decades (Alur et al. 2013; Solar-Lezama 2008; Solar-Lezama et al. 2006; Shen and Rinard 2019; Zhang et al. 2024; Li et al. 2025). Recent uses of LLMs have demonstrated remarkable capabilities of the models in synthesizing code from natural language (Austin et al. 2021; Athiwaratkun et al. 2022; Zan et al. 2023; Jiang et al. 2025). Studies have explored the potential of models in various coding tasks, ranging from simple function generation (Chen et al. 2021; Luo et al. 2023) to more complex programming challenges (Jimenez et al. 2023; Ding et al. 2024; Li et al. 2024; Khatry et al. 2025; Feng et al. 2024; Tan, Jiang, and Shen 2025; Wei et al. 2025b,a). Despite these advances, code generation for specific domains, such as OS kernel verification, poses unique challenges that are not fully addressed by existing LLMs. The complexity and uniqueness of the syntax and semantics involved require models to understand not only programming languages but also domain knowledge and verification assumptions. These challenges call for a benchmark that evaluates the capabilities of LLMs in generating specifications for OS kernel verification.

**LLM for static and dynamic program analysis.** Existing LLM-based static analysis techniques often rely on prompting LLMs to perform source-sink reachability analyses on programs (Wang et al. 2024b,c,a; Li, Dutta, and Naik 2024). However, limited research has explored LLM-based static analysis specifically for the Linux kernel, which presents a significant challenge due to its long-context reasoning requirements. This complexity arises from the kernel's intricate call graphs and alias relationships. In contrast, dynamic analysis techniques, such as fuzzing, have seen broader application of LLMs across various domains, including smart contracts (Shou et al. 2024), the Linux kernel (Yang, Zhao, and Zhang 2023), and universal domains (Xia et al. 2024).

## 3 OSVBENCH

Fig. 1 presents the OSVBENCH workflow. We introduce key concepts and outline how we construct the benchmark tasks.

### 3.1 Preliminaries

**Hyperkernel and its verifier.** Hyperkernel (Nelson et al. 2017) is an OS kernel verification project that includes both a real-world kernel implementation and a verification framework built on the automated theorem prover Z3 (De Moura and Bjørner 2008). The kernel implementation consists of 50 system calls, among which 49 are supported by its verifier. These syscalls cover key functionalities such as process management, virtual memory, file descriptors, device interaction, inter-process communication, and scheduling. The entire codebase, including both the kernel implementation and associated user-space components, consists of approximately 18,000 lines of C and assembly code.

Hyperkernel is particularly suitable as a challenging research vehicle for evaluating specification generation tasks for the following reasons: (1) Hyperkernel adopts a standardized approach (Klein, Sewell, and Winwood 2010) to modeling kernel execution as a state machine. (2) Crafting these specifications requires significant expertise and non-trivial effort. (3) Hyperkernel employs an automated theorem prover, Z3, instead of an interactive theorem prover (Isabelle 2025; Coq 2025; Dafny 2025) to formally verify the functional correctness of the OS kernel. The automated theorem prover streamlines the verification process for performing specification generation tasks.

**Two types of specifications.** To verify a kernel implementation, the verifier for Hyperkernel requires two types of specifications as input to ensure functional correctness: (1) a

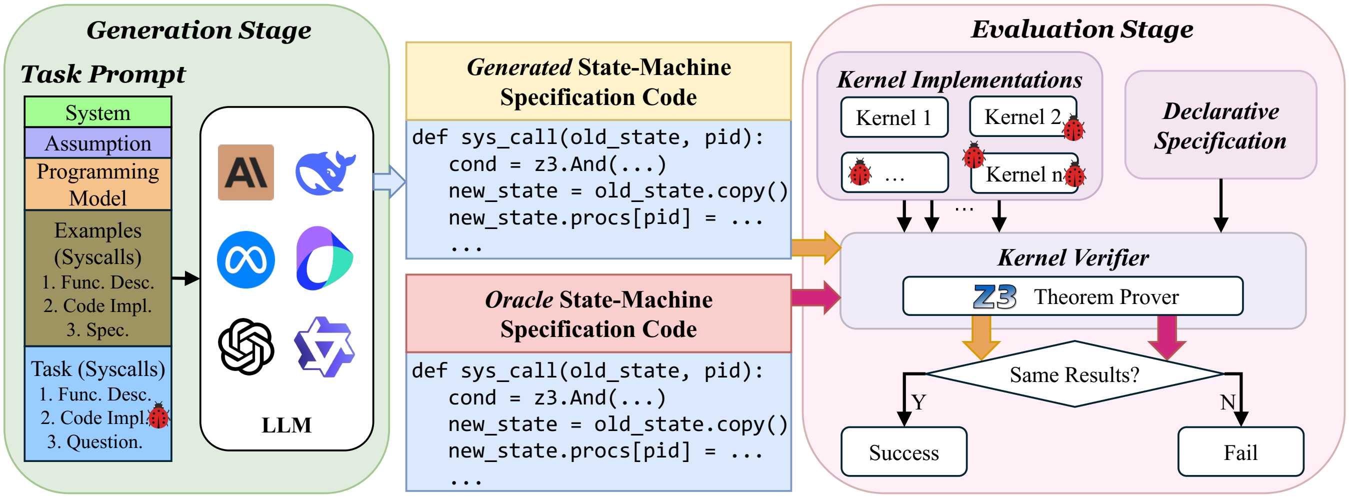


Figure 1: The workflow of the OSVBENCH benchmark suite. The workflow consists of the generation stage and the specification quality evaluation stage. During the generation stage, the input to LLMs includes a system prompt, verification assumptions, a programming model, system call (syscall) examples, and a task question (see Sec. 3.3 for details). Based on this input, the LLMs are tasked with generating the correct state-machine specification for a given syscall. The evaluation stage checks the correctness of the generated state-machine specification by comparing the verification results it produces against those produced by a ground-truth oracle state-machine specification when both are fed into a kernel verifier. In particular, the kernel verifier takes as input a state-machine specification and a formal declarative specification of the kernel (see Sec. 3.1), then uses them to perform verification (see Sec. 3.2) on a potentially buggy kernel implementation (see Sec. 3.3). If both state-machine specifications produce the same verification results for all kernel implementations, the generated one is considered correct. If the two results differ for any kernel implementation, the generated state-machine specification is considered incorrect.

state-machine specification that defines the intended behavior, especially state transition behavior, of the OS kernel and (2) a higher-level declarative specification that outlines the overarching properties and invariants that any state-machine specification must satisfy. For example, one such invariant is that a page with an owner is not free, denoted as

$$\forall \sigma \in \Sigma_{Spec}, pn \in \sigma, \mathsf{is\_valid}(pn, \sigma) \Rightarrow (\mathsf{is\_valid}(\sigma_{\mathsf{page\_owner}}[pn], \sigma) \Leftrightarrow (\sigma_{\mathsf{page\_type}}[pn] \neq \tau_{free})), \quad (1)$$

where $\sigma$ denotes a specific kernel state defined in the state-machine specification, $pn$ denotes a page number under the current kernel state $\sigma$, $\sigma_{\mathsf{page\_owner}}$ and $\sigma_{\mathsf{page\_type}}$ are mappings from a page to its owning process and its type, respectively, and $\tau_{free}$ represents the type of the freed page. The is_valid function checks the validity of a given process.

The verifier establishes two theorems: (1) the kernel implementation is a refinement of the state-machine specification, where the refinement relation states that a state transition in the kernel implementation is equivalent to a state transition in the specification, and (2) all reachable states in the state-machine specification must satisfy the properties and invariants defined in the declarative specifications, which is expressed as

$$\forall \sigma_{spec}, i, P(\sigma_{spec}) \Rightarrow P(t_{spec}(\sigma_{spec}, i)). \quad (2)$$

Here, $\sigma_{spec}$ denotes a kernel state in the state-machine specification, $i$ represents the input for a state transition $t_{spec}$, and $P$ is the predicate or invariant defined in the declarative specification (Nelson et al. 2017).

### 3.2 Problem Formulation

In the specification generation task, the LLM takes a prompt specifically designed for a syscall and synthesizes a state-machine specification for the syscall. Once the LLM synthesizes the state-machine specification, we feed it into the Hyperkernel verifier. The verifier has access to the declarative specification and a potentially buggy implementation for the same syscall in the kernel. The verifier checks the equivalence between the state-machine specification and the syscall in the kernel implementation by performing symbolic execution (Cadar et al. 2008) on the compiled LLVM IR (Lattner and Adve 2004) of the kernel implementation and invoking the Z3 theorem prover. An equivalent result indicates that the state-machine specification accurately characterizes the functionality of the syscall implementation. An inequivalent result indicates an inconsistency between the two; that is, there may be a bug in the implementation or the state-machine specification may not accurately characterize the syscall functionality. In either case, if the result for an LLM-generated state-machine specification differs from the result for a ground-truth oracle state-machine specification, the generated one is considered incorrect.

**Formulating as a general-purpose program synthesis problem.** It is challenging for LLMs to directly solve the specification generation task due to the inherent complications in formally verifying OS kernels. We therefore reformulate the specification generation task as a general-purpose program synthesis problem:

- We introduce explicit verification assumptions by systematically documenting common assumptions related to the OS kernel, such as hardware behavior and memory layout. This process is important because many such assumptions have been used implicitly by OS kernel verifiers such as Hyperkernel, which makes it infeasible for LLMs to directly perform reliable verification.
- We introduce fixed declarative specifications to enable LLMs to focus solely on generating state-machine specifications. This restriction is important because the declarative specification enforces the correctness of the state-machine specification. If we had instead tasked LLMs with generating both the state-machine specification and the declarative specification, as with the original Hyperkernel design, it would have been more difficult to automatically determine whether the generated specifications are correct—in this case, a successful verification result could have been due to an incorrect declarative specification that fails to identify errors.
- We introduce a deterministic synthesis domain with known constants extracted from the compiled LLVM IR of the kernel implementation, utility functions, Z3 functions, and Python classes to restrict the search space of syntactically correct specifications. Defining this domain is important because the state-machine specifications often use such constants and external functions that are not visible in kernel code implementations. By providing them as known building blocks and making them apparent, we make it more tractable for LLMs to generate a correct specification.

Fig. 2 defines the abstract syntax for the state-machine specifications that LLMs need to synthesize. We define the specification generation task as a general-purpose program synthesis task, which takes a prompt as input and produces a synthesized state-machine specification code as output. The synthesized specification (1) must conform to the specified programming model and (2) is used to verify the functional correctness of an OS kernel.

This synthesis problem is programming-language-agnostic and is particularly challenging in the following aspects: (1) accurately mapping the semantics of functional descriptions to the corresponding regions in the specification, (2) exhaustively considering all possible kernel states when there are a potentially divergent number of conditions, and (3) processing contextual information that averages 20k to 30k tokens, which requires advanced long-context learning capabilities. We present concrete examples of these challenges in the extended version in Appendices A and E.

$$
\begin{aligned}
&\mathsf{Specification} := \mathsf{State}\\
&\mathsf{State} := \mathsf{Param} \mid \mathit{if}\ \mathsf{Cond\ State\ State} \mid\\
&\qquad\quad \mathsf{State}.(\mathit{field}_i \leftarrow \mathsf{Expression})^{+}\\
&\mathsf{Expression} := \mathsf{Value} \mid \mathsf{Expression}\ \mathit{aop}\ \mathsf{Expression} \mid\\
&\qquad\quad \mathit{if}\ \mathsf{Cond\ Expression\ Expression}\\
&\mathsf{Value} := \mathsf{Param} \mid \mathsf{Const} \mid \mathsf{State}.\mathit{field}_i\\
&\mathsf{Cond} := \mathsf{Value}\ \mathit{lop}\ \mathsf{Value} \mid \mathsf{Cond} \wedge \mathsf{Cond} \mid \mathsf{Cond} \vee \mathsf{Cond}\\
&\mathit{aop} := + \mid - \mid \times \mid \div\\
&\mathit{lop} := \texttt{==} \mid \texttt{!=} \mid \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{<=}
\end{aligned}
$$

Figure 2: Abstract syntax for state-machine specifications. A specification is a kernel state. A kernel state may be an input parameter of the system call, a conditional state that branches to one of two existing states according to a condition, or a modified state that assigns new expressions to the attributes of an existing state. An expression may be an existing value, an arithmetic expression, or a conditional expression. A value may be an input parameter, a predefined constant literal, or an attribute of an existing state. A condition may be a logical expression, a conjunction, or a disjunction. Intuitively, a specification defines how to transition the OS kernel to a subsequent state. The specification defines state transition and field assignments, along with the conditions that must be satisfied.

### 3.3 Benchmark Task Construction

**Prompt design.** As illustrated in Fig. 1, the prompt for each task is specifically designed for a particular syscall in the OS kernel. Example task prompts are available in Appendix D of the extended version. The prompt is structured into five components:

- The system prompt describes the layout of contents and sections in the prompt and the general task to be solved.
- The verification assumptions define the assumptions required to formally verify an OS kernel, such as the kernel running on a uniprocessor system and not providing multicore support.
- The programming model follows the standardized approach (Klein, Sewell, and Winwood 2010) to modeling kernel execution as a state machine, using a set of Python classes and constants extracted from the compiled LLVM IR of Hyperkernel.
- Few-shot syscall examples are provided, each of which consists of a functional description, a code implementation, and a corresponding oracle state-machine specification. These examples are carefully selected to ensure their representativeness.
- The task question consists of a syscall's functional description, a potentially buggy code implementation, and a question that asks the LLM to generate the corresponding state-machine specification.

**Constructing functional descriptions.** Note that both the few-shot syscall examples and the task question contain a functional description of the syscall. For each of the 49 system calls in Hyperkernel, we manually drafted multiple versions of its functional description based on potential understandings of its intended behavior, purpose, and interactions within the OS kernel. Then, we selected the best version based on clarity and technical precision (Hao et al. 2023).

**Systematic bug injection.** We started with a correct implementation of the OS kernel and systematically generated

| Institution | Model | Incorrect Pointer | Incorrect Privilege | Memory Leak | Buffer Overflow | Bounds Checking | Correct | Total |
|---|---|---|---|---|---|---|---|---|
| OpenAI | o1*△ | 12.68 | 21.43 | 13.51 | 20.37 | 23.15 | 28.57 | 23.67 |
| | o3-mini*△ | 19.72 | 18.75 | 18.92 | 12.96 | 15.74 | 26.53 | 22.04 |
| | GPT-4o△ | 33.80 | 34.82 | 32.43 | 33.33 | 36.11 | 42.86 | 38.78 |
| DeepSeek | DeepSeek-R1* | 32.39 | 21.43 | 13.51 | 20.37 | 23.15 | 42.86 | 40.82 |
| | DeepSeek-Chat | 38.02 | 39.29 | 36.49 | 44.44 | 43.52 | 51.02 | 46.53 |
| Meta | Llama-3.1-70B-Instruct | 12.68 | 18.75 | 12.16 | 16.67 | 22.22 | 22.45 | 22.45 |
| | Llama-3.1-8B-Instruct | 0.00 | 11.61 | 0.00 | 12.96 | 9.26 | 10.20 | 10.61 |
| Qwen Team | QwQ-32B-Preview* | 14.08 | 23.21 | 20.27 | 20.37 | 23.15 | 22.45 | 24.08 |
| | Qwen2.5-72B-Instruct | 25.35 | 26.79 | 24.32 | 25.93 | 30.56 | 34.69 | 32.24 |
| | Qwen2.5-Coder-7B-Instruct | 0.00 | 8.04 | 0.00 | 3.70 | 5.56 | 4.08 | 4.90 |
| Anthropic | Claude-3.5-sonnet△ | 39.44 | 41.96 | 39.19 | 48.15 | 39.81 | 46.94 | 44.90 |
| ByteDance | Doubao-1.5-pro△ | 50.70 | 48.21 | 45.95 | 40.74 | 52.78 | 63.27 | 55.10 |

Table 1: Performance comparison (*Pass@1 %*) of various models with a 5-shot prompt. *Pass@1* evaluates performance in the single-attempt setting: the model generates one specification, and success is recorded only if that specification is correct. Models marked with * are reasoning LLMs, while △ denotes closed-source models; all models without △ are open-source. The columns from **Incorrect Pointer** to **Bounds Checking** correspond to specific types of bugs injected into the syscall code within the task prompts. The column **Correct** corresponds to cases where the provided code implementations are bug-free. Lastly, the column **Total** presents the overall *Pass@1* rate across all 245 tasks.

| Model | Syntax Error | Semantic Error | Incorrect Pointer | Incorrect Privilege | Memory Leak | Buffer Overflow | Bounds Checking | Correct | Total |
|---|---|---|---|---|---|---|---|---|---|
| GPT-4o | 16.47 (1.25) | 12.31 (2.08) | 17.02 (5.13) | 17.81 (3.33) | 6.12 (2.17) | 8.33 (3.03) | 17.39 (1.75) | 17.86 (0.00) | 14.67 (1.56) |
| DeepSeek-Chat | 8.57 (6.25) | 14.75 (0.00) | 2.27 (4.65) | 8.82 (1.61) | 6.52 (0.00) | 10.00 (0.00) | 6.56 (3.51) | 20.83 (0.00) | 11.45 (2.59) |

Table 2: Comparison of two-round self-repair performance (repair success rate %) for models using a 5-shot prompt. The columns **Syntax Error** and **Semantic Error** present the repair success rates for specifications that produce the types of errors. The columns from **Incorrect Pointer** to **Correct** present the repair success rates for specifications derived from the task prompt, categorized by the types of buggy code implementations in the prompt. The **Total** column presents the overall repair success rate for all erroneous specifications addressed in each round. Values outside parentheses are the success rates in the first repair round, while values within parentheses are the success rates in the second round.

a set of buggy implementations to reflect real-world scenarios where OS kernel implementations are not guaranteed to be correct and may contain various types and numbers of bugs. In particular, we randomly injected five types of real-world bugs that were previously found in the xv6 kernel (Cox, Kaashoek, and Morris 2011):

- Incorrect pointer: This bug means the kernel uses an unintended pointer value to access or update critical structures. It will cause incorrect state transitions, context corruption, and potential crashes.
- Incorrect privilege: This bug means privilege separation is broken, allowing user-space to perform privileged operations (e.g., I/O) directly. It will cause privilege escalation, device misuse, and bypass of kernel controls.
- Memory leak: This bug means allocated memory becomes unreachable and is not freed. It will cause cumulative resource loss, fragmentation, and eventual exhaustion.
- Buffer overflow: This bug means reads or writes exceed the bounds of a buffer or array. It will cause out-of-bounds memory access, corruption of kernel data, instability, and security vulnerabilities.
- Bounds checking: This bug means input validation fails to prevent invalid indices or sizes in corner cases. It will cause bypass of checks, out-of-bounds access, and undefined behavior.

These bugs are also prevalent in other OS kernels, such as the Linux kernel (linux kernel CVE team 2025). We present example bugs in Appendix B of the extended version.

To evaluate the impact of bugs on the performance of LLMs in generating accurate state-machine specifications, we injected bugs into task questions. Specifically, we created a total of 245 specification generation tasks, each using a correct high-level functional description of a syscall paired with its potentially buggy code implementation. Among the 245 code implementations, 49 are correct, while the others contain 1–5 injected bugs. We chose this small number of injected bugs because the number of severe vulnerabilities in mature OS kernels is often relatively small.

# 4 Evaluation

We use OSVBENCH to evaluate the performance of a range of state-of-the-art large language models (LLMs) in the task of generating OS kernel verification specifications.

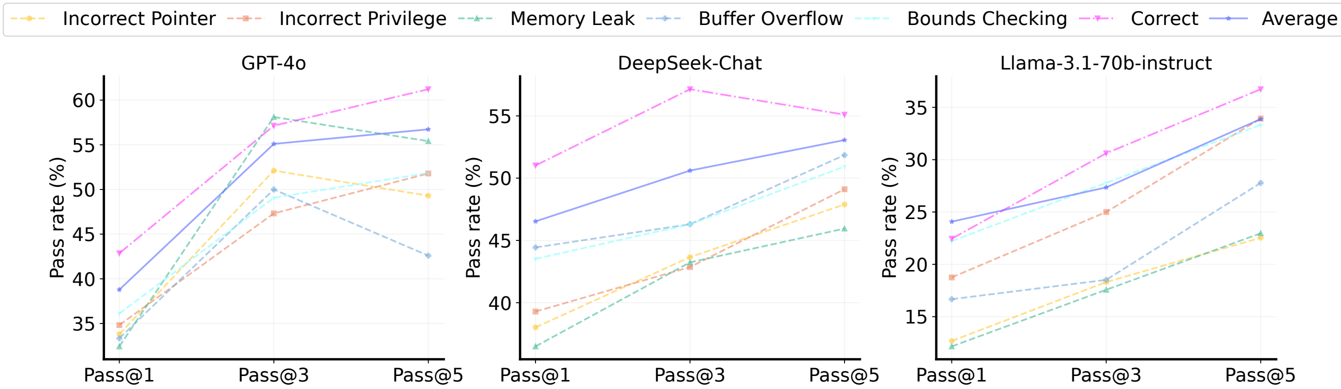


Figure 3: Performance comparison of *Pass@1*, *Pass@3*, and *Pass@5* of various models. The average performance of the models exhibits consistent improvement as k increases in *Pass@k*.

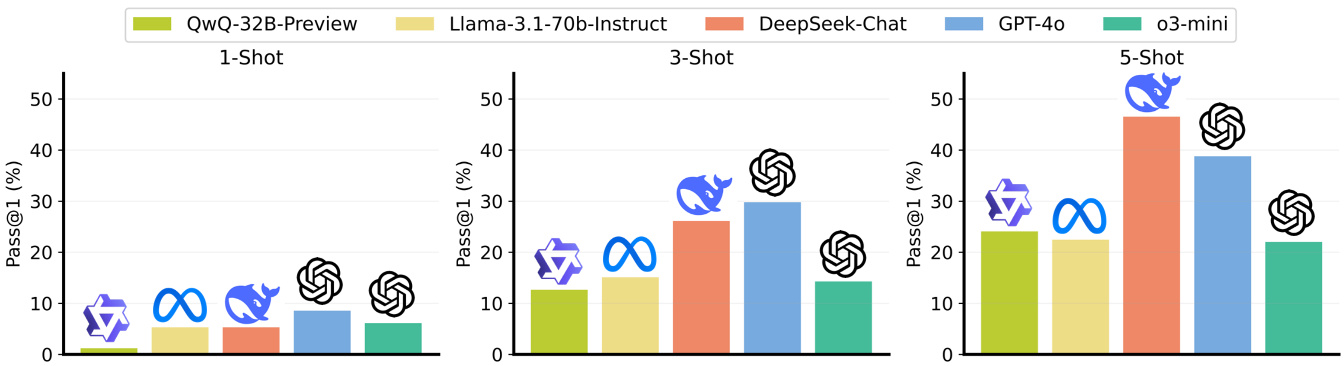


Figure 4: Performance of various models with different numbers of few-shot examples.

### 4.1 Experimental Setup

**State-of-the-art LLMs.** We evaluate existing LLMs from six leading institutions: OpenAI, DeepSeek, Meta, Anthropic, ByteDance, and the Qwen Team. Our evaluation includes the o1, o3-mini, and GPT-4o models from OpenAI; the DeepSeek-R1 and DeepSeek-Chat models from DeepSeek; the Llama-3.1-70B-Instruct and Llama-3.1-8B-Instruct models from Meta; the QwQ-32B-Preview, Qwen2.5-72B-Instruct, and Qwen2.5-Coder-7B-Instruct models from the Qwen Team; Claude-3.5-sonnet from Anthropic; and Doubao-1.5-pro from ByteDance. These LLMs vary in key characteristics such as the number of parameters, open-source availability, data cutoff dates, and pretraining objectives. For all models, we use a greedy search decoding strategy with *Pass@1* to ensure the consistency of the evaluation. To ensure a fair comparison, we use official APIs to access the models, measuring *Pass@1* for each model three times and reporting the average results.

**Specification quality metrics.** We quantify LLM performance with the following metrics. *Pass@k* checks if at least one of k generated specifications is correct. A *syntax error* occurs when the generated specification fails to execute or terminates with an exception. A *semantic error* occurs when there are no syntax errors but the pinpointed inconsistencies differ from those identified by the oracle specification.

### 4.2 Main Results

Table 1 presents the performance of LLMs across institutions and bug categories. The best-performing closed-source LLM, Doubao-1.5-pro, outperforms the best-performing open-source LLM, DeepSeek-Chat, with the highest average *Pass@1* rate (55.10%) and a superior ability to generate specifications for correct implementations (63.27%), showcasing robust performance across all bug types.

In our experiments, models with more parameters outperform their smaller counterparts. For instance, Llama-3.1-8B-Instruct and Qwen2.5-Coder-7B-Instruct exhibit significantly weaker performance compared to their larger counterparts, Llama-3.1-70B-Instruct and Qwen2.5-72B-Instruct. In 53 of 60 (88.3%) experiments with injected bugs, the performance results are lower than those without bugs. For example, memory leak has the most pronounced effect on the DeepSeek-R1 model, while incorrect pointer bugs most significantly impact the o1 model. These results indicate that the presence of bugs caused performance degradation.

Contrary to common belief, widely regarded reasoning models, such as o1 and DeepSeek-R1, do not consistently outperform other models in this task. In particular, o1 demonstrates weak performance, performing worse than QwQ-32B-Preview, which challenges assumptions about the superiority of certain reasoning models in these tasks. We

| Model | Syntax Error | Semantic Error |
|---|---|---|
| o1*△ | 52.65 | 23.67 |
| o3-mini*△ | 51.02 | 26.94 |
| GPT-4o△ | 35.10 | 26.53 |
| DeepSeek-R1* | 32.65 | 26.53 |
| DeepSeek-Chat | 31.02 | 24.90 |
| Llama-3.1-70B-Instruct | 44.90 | 32.65 |
| Llama-3.1-8B-Instruct | 67.76 | 23.67 |
| QwQ-32B-Preview* | 66.53 | 9.39 |
| Qwen2.5-72B-Instruct | 42.25 | 25.31 |
| Qwen2.5-Coder-7B-Instruct | 86.12 | 11.02 |
| Claude-3.5-sonnet△ | 22.45 | 32.65 |
| Doubao-1.5-pro△ | 23.67 | 21.22 |

Table 3: Syntax and semantic error rates (%) in the specifications generated by LLMs across all 245 tasks. * denotes reasoning LLMs. A lower error rate indicates better performance. △ denotes closed-source models.

speculate that the advanced reasoning models used in our experiments produce lengthy chains of reasoning traces, which could pose challenges to the long-context learning capabilities in OS verification scenarios.

**Pass@k performance.** We evaluate *Pass@k* for GPT-4o, DeepSeek-Chat, and Llama-3.1-70B-Instruct. Fig. 3 presents the results. For all three models, the average pass rates improve when k increases from 1 to 3 to 5. Although *Pass@1* for GPT-4o is lower than that for DeepSeek-Chat, GPT-4o outperforms DeepSeek-Chat at k = 3 and k = 5.

## 4.3 Error Analysis and Self-Repair

Table 3 presents the syntax and semantic error rates during the experiments in Table 1. The worst-performing LLM, Qwen2.5-Coder-7B-Instruct, is more prone to syntax errors than the best-performing LLM, Doubao-1.5-pro. This is evidenced by the higher semantic-to-syntax error rate ratio for Doubao-1.5-pro (21.22% / 23.67%) relative to Qwen2.5-Coder-7B-Instruct (11.02% / 86.12%).

To further evaluate the error-handling capabilities of LLMs, we conduct a two-round self-repair process for GPT-4o and DeepSeek-Chat. In the first round of self-repair, the models use a prompt that consists of the original prompt, the models' own generated specifications, an instruction to fix errors, and the error messages from the kernel verifier. If the first round still produces incorrect specifications, in the second round of self-repair, the models use a prompt that consists of the prompt for the first round, the generated specifications for the first round, an instruction to fix errors, and the error messages from the kernel verifier for the first round.

Table 2 presents the results. In total, GPT-4o repairs 14.67% of erroneous specifications for the first round and 1.56% for the second round, while DeepSeek-Chat repairs 11.45% for the first round and 2.59% for the second round. In our experiments, self-repair consistently improves the performance of LLMs in generating specifications for OS kernel verification. However, the repair success rate declines

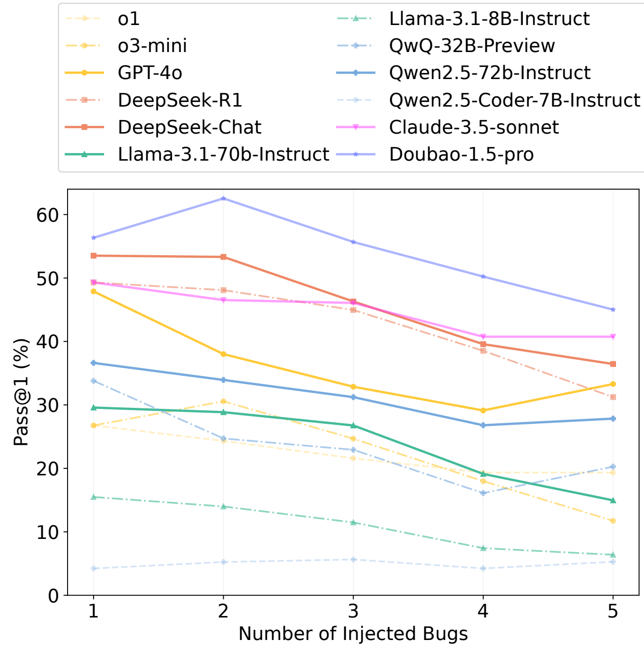


Figure 5: Performance comparison (*Pass@1 %*) of 12 LLMs on specification generation tasks using syscall implementations injected with varying numbers of bugs.

with additional rounds of repair. Appendix C in the extended version presents a detailed case study on two representative error cases to investigate the root causes of these errors.

## 4.4 Impact of Number of Demonstrations

Recent studies indicate that in-context learning (ICL) significantly improves the ability of LLMs to acquire new tasks from a limited set of examples (Brown et al. 2020; Dong et al. 2022). In OS verification, which is inherently complex, the provision of examples that illustrate the generation of specifications from functional descriptions and code implementations greatly impacts performance.

We conduct experiments with five LLMs in various ICL settings, specifically 0-shot, 1-shot, 3-shot, and 5-shot learning. We omit experiments with the other LLMs due to their prohibitive cost and time-intensive nature.

The 0-shot setting results in complete task failure, with a success rate of 0% across all models, indicating the importance of demonstrations in OS verification contexts.

Fig. 4 presents the results for the 1-, 3-, and 5-shot settings. As the number of demonstrations increases, the performance across all five models improves by a large margin. As anticipated, the *Pass@1* performance of LLMs improves with the provision of additional demonstrations in our experiments. Notably, DeepSeek-Chat appears to derive greater benefits from increased demonstrations. While the reasoning model o3-mini surpasses the non-reasoning model DeepSeek-Chat in the 1-shot context, it underperforms DeepSeek-Chat in the 3- and 5-shot scenarios. We hypothesize that advanced reasoning models used in our experiments generate extensive chains of reasoning traces, which may challenge the long-context learning capabilities in OS verification scenarios.

### 4.5 Impact of Number of Injected Bugs

We further investigate the impact of varying numbers of injected bugs on the performance of specification synthesis. Fig. 5 presents the results. Our observations yield several insights: 1) In general, the *Pass@1* performance of LLMs declines as the number of vulnerabilities increases. We attribute this decline to the presence of more vulnerabilities in the kernel implementation, which complicates the models' ability to accurately comprehend the functional descriptions. 2) Advanced reasoning models underperform compared to traditional instruction-following models. For instance, GPT-4o consistently outperforms o1 and o3-mini across all levels of vulnerability. Similarly, DeepSeek-R1 is less effective than DeepSeek-Chat. These findings align with the results presented in Table 1. These results indicate that reasoning models may encounter greater challenges due to the long-context limitations inherent in OS verification scenarios.

## 5 Conclusion

We introduce OSVBENCH, a benchmark for evaluating the performance of LLMs in generating specifications for verifying OS kernels. It formulates the specification generation task as a program synthesis problem and challenges LLMs to navigate complex syntax and semantics within long-context tasks. Experimental results with 12 state-of-the-art LLMs reveal limitations in their ability to handle these tasks effectively, with notable disparities across models. These findings underscore the need to enhance LLMs' capabilities for understanding and generation in complex domains. OSVBENCH highlights existing gaps and serves as a valuable tool for guiding future research aimed at improving verification processes in OS development.

## Limitations

OSVBENCH is designed around Hyperkernel, which might not capture the full diversity of OS kernel architectures, potentially causing LLMs to overfit and limiting the generalizability of results to other systems. The complexity of tasks, each consisting of approximately 20k to 30k tokens, poses significant challenges for LLMs in context management, which might overshadow other capabilities like logical reasoning. The confined scope of syntax and semantics within the benchmark might not fully reflect the dynamic nature of real-world OS development environments. Current evaluation metrics might not capture qualitative aspects such as readability and adaptability. These limitations can guide future efforts to enhance benchmarks for evaluating LLMs in complex, real-world programming and verification tasks.

## Acknowledgments

We thank the anonymous reviewers, Sizhe Zhong, Huiri Tan, and Jipeng Zhang for their insightful comments. This work was supported in part by the Hong Kong Research Grants Council (Project No. 26216025) and Alibaba Group (through the Alibaba Innovative Research Program).

## References

Ahmed, T.; Pai, K. S.; Devanbu, P.; and Barr, E. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, 1–13.

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 1–8. IEEE.

Amani, S.; Hixon, A.; Chen, Z.; Rizkallah, C.; Chubb, P.; O'Connor, L.; Beeren, J.; Nagashima, Y.; Lim, J.; Sewell, T.; et al. 2016. Cogent: Verifying high-assurance file system implementations. *ACM SIGARCH Computer Architecture News*, 44(2): 175–188.

Athiwaratkun, B.; Gouda, S. K.; Wang, Z.; Li, X.; Tian, Y.; Tan, M.; Ahmad, W. U.; Wang, S.; Sun, Q.; Shang, M.; et al. 2022. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations*.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Bevier, W. R. 1989. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11): 1382–1396.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Cadar, C.; Dunbar, D.; Engler, D. R.; et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, 209–224.

Cao, J.; Lu, Y.; Li, M.; Ma, H.; Li, H.; He, M.; Wen, C.; Sun, L.; Zhang, H.; Qin, S.; Cheung, S.-C.; and Tian, C. 2025. From Informal to Formal – Incorporating and Evaluating LLMs on Natural Language Requirements to Verifiable Formal Proofs. In Che, W.; Nabende, J.; Shutova, E.; and Pilehvar, M. T., eds., *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 26984–27003. Vienna, Austria: Association for Computational Linguistics. ISBN 979-8-89176-251-0.

Chajed, T.; Tassarotti, J.; Theng, M.; Kaashoek, M. F.; and Zeldovich, N. 2022. Verifying the {DaisyNFS} concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 447–463.

Chen, H.; Chajed, T.; Konradi, A.; Wang, S.; İleri, A.; Chlipala, A.; Kaashoek, M. F.; and Zeldovich, N. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 270–286.

Chen, H.; Ziegler, D.; Chajed, T.; Chlipala, A.; Kaashoek, M. F.; and Zeldovich, N. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 18–37.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, T.; Lu, S.; Lu, S.; Gong, Y.; Yang, C.; Li, X.; Misu, M. R. H.; Yu, H.; Duan, N.; Cheng, P.; et al. 2024. Automated Proof Generation for Rust Code via Self-Evolution. *arXiv preprint arXiv:2410.15756*.

Coq. 2025. The Coq Proof Assistant. https://coq.inria.fr/. Accessed: 2025-2-14.

Cox, R.; Kaashoek, M. F.; and Morris, R. 2011. Xv6, a simple Unix-like teaching operating system.

Dafny. 2025. Dafny: The Dafny Programming and Verification Language. https://dafny.org/. Accessed: 2025-2-14.

Dahl, O.-J.; Dijkstra, E. W.; and Hoare, C. A. R. 1972. *Structured programming*. Academic Press Ltd.

De Moura, L.; and Bjørner, N. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Ding, Y.; Wang, Z.; Ahmad, W.; Ding, H.; Tan, M.; Jain, N.; Ramanathan, M. K.; Nallapati, R.; Bhatia, P.; Roth, D.; et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Dong, Q.; Li, L.; Dai, D.; Zheng, C.; Ma, J.; Li, R.; Xia, H.; Xu, J.; Wu, Z.; Liu, T.; et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

D'silva, V.; Kroening, D.; and Weissenbacher, G. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7): 1165–1178.

Feiertag, R. J.; Levitt, K. N.; and Robinson, L. 1977. Proving multilevel security of a system design. *ACM SIGOPS Operating Systems Review*, 11(5): 57–65.

Feng, J.; Liu, J.; Gao, C.; Chong, C. Y.; Wang, C.; Gao, S.; and Xia, X. 2024. ComplexCodeEval: A Benchmark for Evaluating Large Code Models on More Complex Code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, 1895–1906. New York, NY, USA: Association for Computing Machinery. ISBN 9798400712487.

Hao, Y.; Li, G.; Zou, X.; Chen, W.; Zhu, S.; Qian, Z.; and Sani, A. A. 2023. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, 3262–3278. IEEE.

Isabelle. 2025. Isabelle: A Generic Proof Assistant. https://isabelle.in.tum.de/. Accessed: 2025-2-14.

Jacobs, B.; and Piessens, F. 2008. The VeriFast program verifier. Technical report, Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. R. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations*.

Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; and Svyatkovskiy, A. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1646–1656.

Khatry, A.; Zhang, R.; Pan, J.; Wang, Z.; Chen, Q.; Durrett, G.; and Dillig, I. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. arXiv:2504.15254.

Klein, G.; Andronick, J.; Elphinstone, K.; Murray, T.; Sewell, T.; Kolanski, R.; and Heiser, G. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1): 1–70.

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 207–220.

Klein, G.; Sewell, T.; and Winwood, S. 2010. Refinement in the Formal Verification of the seL4 Microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, 323–339. Springer.

Lattner, C.; and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, 75–86. IEEE.

Leino, K. R. M. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, 348–370. Springer.

Li, L.; Geng, S.; Li, Z.; He, Y.; Yu, H.; Hua, Z.; Ning, G.; Wang, S.; Xie, T.; and Yang, H. 2024. Infibench: Evaluating the question-answering capabilities of code large language models. *Advances in Neural Information Processing Systems*, 37: 128668–128698.

Li, S.; Zhang, Z.; Zhong, S.; Zhou, D.; and Shen, J. 2025. A Sound Static Analysis Approach to I/O API Migration. *Proc. ACM Program. Lang.*, 9(OOPSLA2).

Li, Z.; Dutta, S.; and Naik, M. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*.

linux kernel CVE team. 2025. Linux CVE announcement. https://lore.kernel.org/linux-cve-announce/. Accessed: 2025-2-14.

Loughridge, C.; Sun, Q.; Ahrenbach, S.; Cassano, F.; Sun, C.; Sheng, Y.; Mudide, A.; Misu, M. R. H.; Amin, N.; and Tegmark, M. 2024. DafnyBench: A Benchmark for Formal Software Verification. *arXiv preprint arXiv:2406.08467*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations*.

Ma, L.; Liu, S.; Li, Y.; Xie, X.; and Bu, L. 2024. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. *arXiv preprint arXiv:2401.08807*.

Misu, M. R. H.; Lopes, C. V.; Ma, I.; and Noble, J. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.*, 1(FSE).

Nelson, L.; Sigurbjarnarson, H.; Zhang, K.; Johnson, D.; Bornholt, J.; Torlak, E.; and Wang, X. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 252–269. New York, NY, USA: Association for Computing Machinery. ISBN 9781450350853.

O'Connor, L.; Chen, Z.; Rizkallah, C.; Amani, S.; Lim, J.; Murray, T.; Nagashima, Y.; Sewell, T.; and Klein, G. 2016. Refinement through restraint: Bringing down the cost of verification. *ACM SIGPLAN Notices*, 51(9): 89–102.

Sammler, M.; Lepigre, R.; Krebbers, R.; Memarian, K.; Dreyer, D.; and Garg, D. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 158–174.

Shen, J.; and Rinard, M. C. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 269–285. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367127.

Shou, C.; Liu, J.; Lu, D.; and Sen, K. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108*.

Sigurbjarnarson, H.; Bornholt, J.; Torlak, E.; and Wang, X. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 1–16. Savannah, GA: USENIX Association. ISBN 978-1-931971-33-1.

Solar-Lezama, A. 2008. *Program synthesis by sketching*. Ph.D. thesis, University of California, Berkeley.

Solar-Lezama, A.; Tancau, L.; Bodik, R.; Seshia, S.; and Saraswat, V. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 404–415.

Sun, C.; Sheng, Y.; Padon, O.; and Barrett, C. 2024. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, 134–155. Springer.

Tan, H.; Jiang, J.; and Shen, J. 2025. ProfiX: Improving Profile-Guided Optimization in Compilers with Graph Neural Networks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Walker, B. J.; Kemmerer, R. A.; and Popek, G. J. 1980. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2): 118–131.

Wang, C.; Gao, Y.; Zhang, W.; Liu, X.; Shi, Q.; and Zhang, X. 2024a. LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis. *arXiv preprint arXiv:2412.14399*.

Wang, C.; Zhang, W.; Su, Z.; Xu, X.; Xie, X.; and Zhang, X. 2024b. LLMDFA: Analyzing Dataflow in Code with Large Language Models. *Advances in Neural Information Processing Systems*, 37: 131545–131574.

Wang, C.; Zhang, W.; Su, Z.; Xu, X.; and Zhang, X. 2024c. Sanitizing Large Language Models in Bug Detection with Data-Flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, 3790–3805.

Wei, A.; Cao, J.; Li, R.; Chen, H.; Zhang, Y.; Wang, Z.; Liu, Y.; Teixeira, T. S. F. X.; Yang, D.; Wang, K.; and Aiken, A. 2025a. EquiBench: Benchmarking Large Language Models' Reasoning about Program Semantics via Equivalence Checking. In Christodoulopoulos, C.; Chakraborty, T.; Rose, C.; and Peng, V., eds., *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 33856–33869. Suzhou, China: Association for Computational Linguistics. ISBN 979-8-89176-332-6.

Wei, A.; Wu, Y.; Wan, Y.; Suresh, T.; Tan, H.; Zhou, Z.; Koyejo, S.; Wang, K.; and Aiken, A. 2025b. SATBench: Benchmarking LLMs' Logical Reasoning via Automated Puzzle Generation from SAT Formulas. In Christodoulopoulos, C.; Chakraborty, T.; Rose, C.; and Peng, V., eds., *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 33820–33837. Suzhou, China: Association for Computational Linguistics. ISBN 979-8-89176-332-6.

Xia, C. S.; Paltenghi, M.; Le Tian, J.; Pradel, M.; and Zhang, L. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Yang, C.; Zhao, Z.; and Zhang, L. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*.

Zan, D.; Chen, B.; Zhang, F.; Lu, D.; Wu, B.; Guan, B.; Yongji, W.; and Lou, J.-G. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 7443–7464.

Zhang, B.; Chen, W.; Yao, P.; Wang, C.; Tang, W.; and Zhang, C. 2024. SIRO: empowering version compatibility in intermediate representations via program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 882–899.

Zhang, L.; Lu, S.; and Duan, N. 2024. Selene: Pioneering Automated Proof in Software Verification. *arXiv preprint arXiv:2401.07663*.