OSVBench: Benchmarking LLMs on Specification Generation Tasks for Operating System Verification

Anonymous ACL submission

Abstract

We introduce OSVBench, a new benchmark for evaluating Large Language Models (LLMs) in generating code for complete specifications pertaining to operating system kernel verification tasks. The benchmark first defines the specification generation problem into a program synthesis problem within a confined scope of syntax and semantics by providing LLMs with the programming model. The LLMs are required to understand the provided verification assumption and the potential syntax and semantics space to search for, then generate the complete specification for the potentially buggy operating system code implementation under the guidance of the high-level functional description of the operating system. This benchmark is built upon a real-world operating system kernel, Hyperkernel, and consists of 245 complex specification generation tasks in total, each is a long context task of about 30,000 tokens. Our comprehensive evaluation of 10 LLMs exhibits the limited performance of the current LLMs on the specification generation tasks for operating system verification. Significant disparities of their performance on the benchmark, differentiating the ability on long context code generation tasks.

1 Introduction

Large Language Models (LLMs) have shown great potential in software engineering tasks, such as code generation, code summarization, and bug repair. However, an important aspect of software engineering remains underexplored: software verification. Software verification uses rigorous mathematical reasoning to prove the absence of bugs in software (Dahl et al., 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 errors could lead to catastrophic economic losses or even endanger human lives. However, manual software verification is challenging and time-consuming. It typically requires a deep understanding of formal methods and program analysis, skills that are usually acquired only at the graduate level. As a result, professionals capable of conducting verification is limited, highlighting the need for automation in this area.

In this paper, we examine the capabilities of LLMs in automating software verification. We introduce a benchmark suite to evaluate the effectiveness of these models in verifying an operating system (OS) kernel, a fundamental component of many critical infrastructures. Verifying an OS kernel is highly non-trivial due to the inherent complexity, concurrency, and hardware interactions involved. For example, the verification of the well-known seL4 microkernel (Klein et al., 2009) required 11 person-years of effort for 10,000 lines of C code, while 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. Automating verification requires generating specifications that characterize the expected properties, a task as complex as developing correctness proofs (Sammler et al., 2021; Leino, 2010; Jacobs and Piessens, 2008; Ma et al., 2024). For example, developing the specification for seL4 (Klein et al., 2009) required 7 person-months of effort.

We introduce OSVBench, a benchmark suite derived from the Hyperkernel project (Nelson et al., 2017), to evaluate the capabilities of LLMs in generating specifications for verifying the functional correctness of an OS kernel. This benchmark aims to facilitate automation in the generation of OS kernel verification specifications. The benchmark comprises of 245 verification specification generation tasks, each of which is a complex and intricate program synthesis task with long context, approximately 30,000 tokens.

Figure 1 illustrates the workflow for each verification specification generation task. This workflow

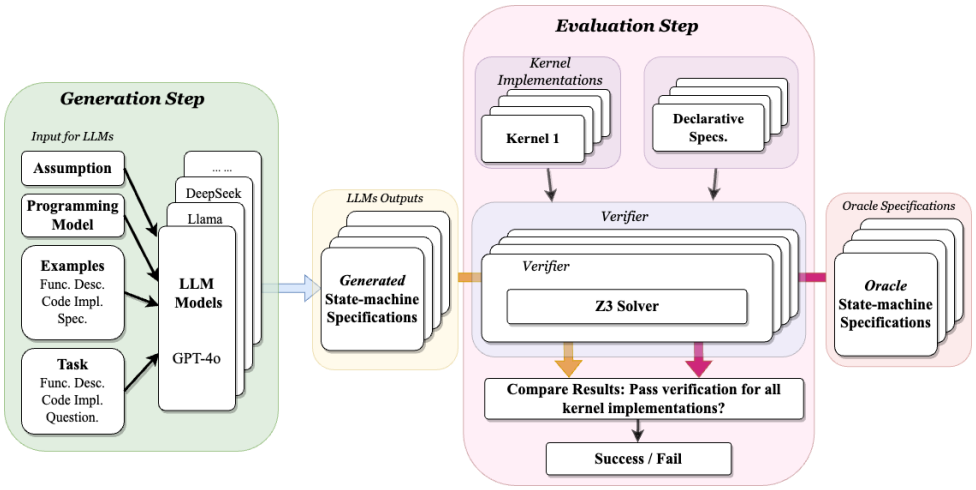


Figure 1: The workflow of OSVBench benchmark suite

is divided into two main stages: the generation stage and the specification quality evaluation stage.

In the generation stage, several inputs are provided to guide the process, including the verification assumptions, a programming model with deterministic formal semantics, a set of few-shot examples, and the final task problem. The few-shot examples consist of three components: (1) a precise and accurate functional description in natural language for a specific system call of the OS kernel, (2) the concrete, potentially buggy code implementation of the system call, and (3) the specification required to verify the functional correctness of the system call, which ensures the functional correctness of the OS kernel. The task problem, in turn, is structured to include the functional description and the potentially buggy code implementation of the system call while leaving the specification for LLMs to synthesize.

The second stage, referred to as the specification quality evaluation stage, utilizes a verifier to assess the correctness of the generated specification. This is achieved by verifying multiple OS kernel implementations with different injected bugs, using both the generated specification and the oracle specification as references. If the verification results for any of the implementations differ between the two specifications, the generated specification is deemed incorrect; otherwise, it is considered correct. Further details on the functionality and operation of the verifier are provided in subsection 3.1.

We conducted comprehensive experiments on the formal specification generation from the functional description in natural language and the code implementation of the OS kernel with different types and numbers of vulnerabilities by injecting 5 real-world types of bugs into the OS kernel. The experiment results showcase the potential of LLMs in automating formal specification generation for OS kernel verification. Finally, we conduct rigorous data decontamination (Yang et al., 2023) for the synthetic dataset to remove samples that closely resemble those in the test subset $\mathcal{D}_{test}^{(0)}$ of the coverage dataset. The main contributions of this paper can be summarized as below:

- We initiate an exploration of LLMs in the context of OS verification tasks, which necessitate a deep understanding and manipulation of extensive contextual information and domain-specific knowledge.
- We introduce OSVBench, a benchmark designed for OS verification, to evaluate the capabilities of LLMs in generating specifications for OS kernels.
- We performed a comprehensive evaluation of the most advanced LLMs in generating specifications aimed at verifying the functional correctness of an OS kernel. We also discuss the impact of varying types and quantities of bugs on the quality and effectiveness of the generated specifications.

2 Related Work

2.1 Software Verification

Software verification (D'silva et al., 2008) ensures that software conforms to specified properties or requirements, playing a critical role in guaranteeing software reliability and correctness. Com-

mon techniques include static analysis (Cousot and Cousot, 1977), model checking (Emerson and Clarke, 1980), and theorem proving. Theorem proving can be categorized into interactive theorem proving, such as Isabelle (Isabelle, 2025), Coq (Coq, 2025), and Dafny (Dafny, 2025), and automated theorem proving, which relies on solvers such as Z3 (De Moura and Bjørner, 2008).

Operating system kernel verification (Klein et al., 2014) has been a central research goal in ensuring the reliability and security of critical software systems. Early foundational work includes efforts such as UCLA Secure Unix (Walker et al., 1980), PSOS (Feiertag et al., 1977), and KIT (Bevier, 1989), which laid the groundwork for formal approaches to kernel correctness. Recent progress has expanded to leveraging 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.

Prior work on leveraging LLMs for software verification mainly focused on generating proofs from specifications (Chen et al., 2024; Zhang et al., 2024), which involves translating one deterministic formal semantic representation (specifications, in various forms) into another (proofs expressed in formal languages). Additionally, some studies have explored the task of specification generation. However, much of this work has concentrated on general-purpose specification generation (Ma et al., 2024), which differs significantly from the generation of OS kernel specifications due to the distinct verification assumptions and requirements encountered in this domain.

2.2 LLM for Code Generation

The use of Large Language Models (LLMs) for code generation has gained significant attention in recent years, as these models have demonstrated remarkable capabilities in synthesizing code snippets from natural language descriptions (Austin et al., 2021; Athiwaratkun et al., 2022; Zan et al., 2023; Jiang et al., 2024). Several studies have explored the potential of LLMs in various code generation 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). Despite these advancements, code generation for specific domains, such as operating system (OS) kernel verification, poses unique challenges that are not fully addressed by general-purpose LLMs. The complexity and specificity of the syntax and semantics involved in such domains require models not only to understand programming languages but also to grasp domain-specific knowledge and verification assumptions. This challenge calls for a benchmark that evaluate the capabilities of LLMs in generating specifications for operating system kernels.

3 OSVBench

In this section, we will discuss the benchmark construction details and the specification generation problem formulation.

```
class KernelState(BaseStruct):
    procs = Proc()
    pages = Page()
    files = File()
    pcipages = PCIPage()
    ...
```

3.1 Preliminaries

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. The kernel implementation supports 50 system calls, covering key functionalities such as process management, virtual memory, file descriptors, device interaction, inter-process communication, and scheduling. The entire codebase consists of approximately 18,000 lines of C and assembly, encompassing both the kernel implementation and associated user-space components.

As demonstrated in the specification quality evaluation section in Figure 1, the verifier for Hyperkernel requires two types of specifications as input. The first is a state-machine specification, which defines the functional correctness by describing the intended behavior of the OS kernel. The second is a higher-level declarative specification, which outlines overarching properties and invariants that the state-machine specification must satisfy. For example, one such property ensures that the number of children for any given process is always equal to the total number of processes identifying that process as their parent. In our formulation of the verification specification generation task, the declarative specification is provided as input, and the objective is to synthesize the corresponding state-machine specification that formally defines the functional behaviors of the kernel.

The verifier establishes two theorems. The first one proves that the kernel implementation is a re-

finement of the state-machine specification, expressed as $\forall \sigma_{Impl} \in \Sigma_{Impl}, \forall c \in C, \exists \sigma_{Spec} \in \Sigma_{Spec}$, such that $\sigma_A(c) = \sigma_B(c)$, which states that for every kernel state in the implementation, under any condition, there exists a corresponding kernel state defined in the state-machine specification that is equivalent under the same condition. The second theorem demonstrates that the state-machine specification satisfies the properties and invariants defined within the declarative specifications.

In our formulation of the specification generation task, the declarative specifications are provided and the role of the LLM is to synthesize the state-machine specifications. Once the state-transition specification is prepared, its verifier performs symbolic execution on the compiled LLVM IR (Lattner and Adve, 2004) of the OS kernel and invokes the z3 (De Moura and Bjørner, 2008) solver to perform the equivalence checking on the real kernel state transitions in the concrete implementation and those defined in the state-transition specification. Additionally, the verifier ensures that the state-machine specification adheres to the high-level declarative specifications. Any detected inconsistency indicates either the presence of a bug in the OS kernel code implementation or that the state-transition specification fails to accurately describe the intended functionality of the system call.

We select the Hyperkernel as the benchmark for the following reasons: 1) Hyperkernel's specifications are well-defined and systematically modeled using Python classes with deterministic semantics. Crafting these specifications requires significant expertise and non-trivial effort, making it a suitable and challenging benchmark for evaluating specification generation tasks. 2) Hyperkernel employs an automated theorem prover, specifically the Z3 solver, to formally verify the functional correctness of the OS kernel instead of using the interactive theorem provers, such as Isabelle, Coq, or Dafny. By utilizing an automated solver, the focus can remain on the specification generation tasks, streamlining the verification process.

3.2 Problem Formulation

Verification of operating systems requires extensive domain-specific expertise. To enable large language models (LLMs) to address the verification specification problem by leveraging their pre-trained knowledge, we reformulate the problem as a program synthesis task within a domain defined by deterministic formal semantics based on two observations. The first is that the kernel behavior is well-modeled using a set of Python classes with deterministic semantics, which serves as the programming model. Second, as outlined in the subsection 3.1, the state-machine specification of the Hyperkernel defines the kernel state transitions. Specifically, it specifies the kernel state to which the OS will transition under specific conditions upon the completion of a corresponding system call. Thus, the synthesis domain for the state-machine specification synthesis task can be formally defined, as illustrated in Equation 1. In this context, the State represents the modeled class `KernelState`, as detailed in Listing 3.

Synthesize correct specification within the scoped semantics. In this context, the task of the LLMs is to first comprehend the semantics of the programming model described in Listing 3. Subsequently, the LLMs perform state-machine specification synthesis guided by the high-level functional description of the system call. Finally, within the constrained domain of formal semantics defined in Equation 1, the LLMs search for the correct implementation of the specification.

To synthesize the correct specification from an informal functional description in natural language, the process must ensure accurate field access, appropriate constant selection, and precise condition selection. Specifically, to achieve deterministic specification synthesis, the synthesis process should operate within a symbolic triple relation, denoted as $< Desc, Impl, Model >$. This rule entails that, for synthesizing any given statement in the specification, the LLM should first identify the relevant functional description (Desc), then locate the corresponding concrete kernel implementation (Impl). Finally, it must interpret the intended semantics and synthesize the correct specification by referencing the appropriate model (Model) as defined within the programming model.

```
int sys_close(pid_t pid, int fd) {
    if (!is_pid_valid(pid)) // Cond1.
        return -ESRCH;
    if (!is_fd_valid(fd))   // Cond2.
        return -EBADF;
    ...
    clear_fd(pid, fd);
    ...
}
static inline void clear_fd(pid_t pid,
    int fd) {
    ...
    file = get_file(get_fd(pid, fd));
    proc->ofile[fd] = 0;
    --proc->nr_fds;
```

```
    if (--file->refcnt == 0) { // Cond3
        ...
    }
}
```

For example, Listing 3.2 shows the implementation of the system call sys_close. As the functional description indicates: This involves updating the process's file descriptor table to mark the descriptor as unused and decrementing the count of open file descriptors for the process. Additionally, the system updates the file's reference count to reflect the decrease of the file references.

Kernel code implementation:

```
    proc->ofile[fd] = 0;
    --proc->nr_fds;
    --file->refcnt;
```

By searching in the programming model and the synthesis domain Equation 1:

```
class Proc(Struct):
    ...
    ofile = Map(pid_t, fd_t, fn_t)
    nr_fds = Refcnt(pid_t, fd_t, size_t)
class File(Struct):
    ...
    refcnt = Refcnt(fn_t, (pid_t, fd_t),
         size_t)
```

The synthesized specification is:

```
    new.procs[pid].ofile[fd] = z3.
        BitVecVal(0, dt.fn_t)
    new.procs[pid].nr_fds[fd] -= 1
    new.files[fn].refcnt[(pid, fd)] -= 1
```

$$
\begin{aligned}
&\text{Specification} := \text{State}\\
&\text{State} := \text{State}.(field_i \leftarrow \text{Expression})^* \mid \text{State}\\
&\text{Expression} = if\ \text{cond}\ \text{Expression}, \text{Expression}' \mid\\
&\qquad \text{Expression}\ op\ \text{Expression} \mid \text{Constant}\\
&\text{cond} = and\ \text{cond}, \text{cond} \mid or\ \text{cond}\ \text{cond} \mid\\
&\qquad \text{State}.field_i\ op\ \text{Constant} \mid\\
&\qquad \text{State}.field_i\ op\ \text{State}.field_{i'}\\
&op = + \ - \ \times \ \div \ == \ > \ < \ >= \ <= \qquad (1)
\end{aligned}
$$

Diverged kernel statesKernel state conditions synthesis. To synthesize an accurate state-machine specification, it is essential to determine the kernel state to which the system transitions under specific conditions according to the synthesis domain Equation 1. This process requires LLMs to perform advanced reasoning, as many system calls involve cascading and interdependent conditions. Figure 2 illustrates the diverged kernel states resulting from

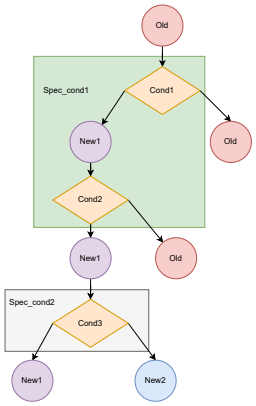


Figure 2: Diverged kernel states of sys_close

the system call **sys_close**. Consequently, the kernel state specified in the synthesized specification should be as follows:

```
def sys_close(old, pid, fd):
    spec_cond1 = z3.And(
        z3.And(pid > 0, pid < dt.NPROC),
        z3.And(fd >= 0, fd < dt.NOFILE),
        ...)
    new1 = old.copy()
    ...
    spec_cond2 = z3.And(new.files[fn].
        refcnt() == 0)
    new2 = new1.copy()
    ...
    new3 = util.If(spec_cond2,new2,new1)
    return spec_cond1, util.If(
        spec_cond1, new3, old)
```

The specification Listing 3.2 illustrates the kernel's state transitions under different conditions. Specifically, the kernel transitions to new_state1 when both $spec\_cond1$ and $spec\_cond2$ are satisfied. If $spec\_cond1$ is satisfied but $spec\_cond2$ is not, the kernel transitions to new_state2. Finally, if neither $spec\_cond1$ nor $spec\_cond2$ is satisfied, the kernel remains in old_state.

3.3 Verification Specification Generation

In real-world scenarios involving the verification of OS kernels, the kernel implementation is not guaranteed to be correct and may contain various types and numbers of bugs. The primary objective of verification is to identify and pinpoint these bugs within the kernel's concrete code implementation. So the verification specification generation task is defined as synthesizing the comprehensive specifications to verify the functional correctness of an OS kernel under the guidance of the correct high-level functional descriptions and potentially buggy code implementations of the system calls.

We begin by engaging three OS specialists to manually draft detailed functional descriptions of each system call in natural language. We then employ a voting strategy to determine the most accurate and representative functional description for each system call to ensure clarity and precision.

Additionally, to simulate real-world conditions, we begin with the correct code implementation of the OS kernel and systematically construct a set of incorrect implementations. This is achieved by randomly introducing five types of real-world bugs, derived from the xv6 kernel (Cox et al., 2011), into the codebase of Hyperkernel. This approach allows us to evaluate the impact of various vulnerabilities on the performance of large language models (LLMs) in generating accurate state-machine specifications. The definition of each type of buggy code implementation is provided in Appendix A.

Finally, we create a total of 245 verification specification generation tasks upon 49 of the system calls of Hyperkernel, each consisting of a correct high-level functional description of a system call paired with its corresponding potentially buggy code implementation. Among the 245 code implementations, some are correct, while others contain varying numbers of bugs, ranging from one to five.

4 Evaluation

4.1 Experimental Setup

State-of-the-art LLMs. We conduct an evaluation of the current state-of-the-art large language models (LLMs) developed by four leading institutions: OpenAI, DeepSeek, Meta, and the Qwen Team. Specifically, 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; and the QwQ-32B-Preview, Qwen2.5-72B-Instruct, and Qwen2.5-Coder-7B-Instruct models from the Qwen Team. The evaluation leverages the OS-VBench framework to systematically assess the performance of these models in the task of generating OS kernel verification specifications. These LLMs differ in key characteristics such as the number of parameters, open-source availability, data cutoff dates, and pretraining objectives. For all models, we employ a greedy search decoding strategy with pass@1 for consistency in evaluation.

Prompts As illustrated in Figure 1, the prompt for each task is specifically designed for a particular system call within the OS kernel. The prompt is structured into four key components: the system verification assumptions, the programming model, the few-shot examples, and the task question. The task question includes the correct functional description, a potentially buggy code implementation, and instructions for generating the specification required to verify the functional correctness of the system call. The few-shot examples are carefully selected by OS kernel verification experts to ensure they are representative for the task.

Specification quality metrics. To systematically evaluate the performance of LLMs on the tasks, we define several metrics. As outlined in the specification correctness evaluation stage in Figure 1, we deliberately created multiple OS kernel implementations with known, artificially inserted bugs. The metric *Pass@1* is used to indicate that the generated specification is correct, meaning it precisely identifies the vulnerabilities in the same manner as the oracle specification across all buggy OS kernel implementations. The *Syntax Error* metric denotes cases where the generated specification fails to execute correctly or terminates with an exception. Finally, the *Semantic Error* refers to instances where the verifier successfully translates the specification into SMT (De Moura and Bjørner, 2008) and performs verification on the OS kernel implementation, but the pinpointed vulnerability differs from that specified by the oracle specification.

4.2 Main Results

Table 1 presents results of the performance of LLMs across institutions and bug categories. DeepSeek-Chat stands out with the highest average pass@1 rate (46.53%) and a superior ability to generate correct specifications (51.02%), showcasing robust performance across all bug types.

In general, models with larger parameter sizes tend to 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, such as Llama-3.1-70B-Instruct and Qwen2.5-72B-Instruct, which have over 70 billion parameters. The results also highlight the performance degradation caused by the presence of various bug types, with the impact varying across models and bug categories. For example, memory leak bugs have the most pronounced effect on the DeepSeek-R1 model, while incorrect pointer bugs most signifi-

Table 1: Performance comparison of various models with 5-shots prompt. * denotes reasoning LLMs.

Institution	Model	Incorrect Pointer	Incorrect I/O Privilege	Memory Leak	Buffer Overflow	Bounds Checking	Correct	Avg.
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	40.00
	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	11.61	0	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	8.04	0	3.70	5.56	4.08	4.90

Table 2: Syntax and semantic errors of various models. * denotes reasoning LLMs.

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

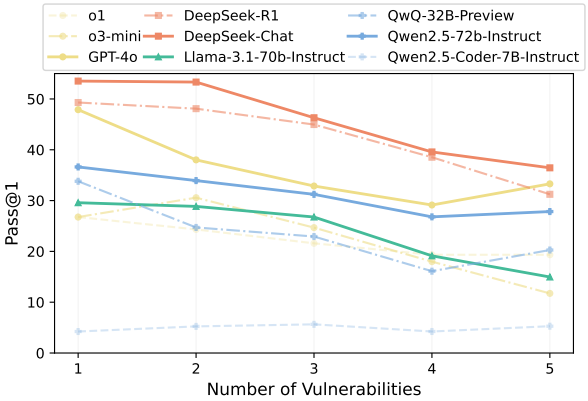


Figure 3: Performance comparison of bug number.

cantly impact the o1 models.

Surprisingly, widely regarded reasoning models, such as o1 and DeepSeek-R1, do not consistently outperform other models in this task. In particular, the o1 model demonstrates weak performance, performing worse than the QwQ-32B-Preview model, which challenges assumptions about the superiority of certain reasoning models in these tasks. We speculate that the advanced reasoning models being utilized produce lengthy chains of reasoning traces, which could pose challenges to the long-context learning capabilities in OS verification scenarios.

4.3 Syntax and Semantic Error Analysis

Given the definitions of syntax and semantic errors provided in section 4.1, we analyze the error rates of the evaluated LLMs. By definition, models with better overall performance are expected to exhibit lower error rates, with semantic error rates comparatively higher than syntax error rates. This is because, when generating specifications, LLMs must first ensure syntactic validity according to the programming model before attempting to perform verification on the OS kernel to pinpoint vulnerabilities. As observed in Table 2, the best-performing model, DeepSeek-Chat, aligns with this expectation, showing a comparatively higher semantic error rate relative to its syntax error rate. In contrast, the worst-performing model, Qwen2.5-Coder-7B-Instruct, exhibits the opposite trend, with a notably low semantic error rate but an exceptionally high syntax error rate, indicating significant challenges in producing syntactically valid specifications. To further investigate the root causes of these errors, we conducted a detailed case study in Appendix B on two representative error cases, analyzing how the specifications generated by the LLMs lead to syntax and semantic errors. This analysis aims to provide deeper insights into the limitations of the models and potential areas for improvements.

4.4 Impact of Number of Vulnerabilities

In practical applications, OS kernels typically exhibit a limited number of potential vulnerabilities, with few instances of severe vulnerabilities. Consequently, we further explore the impact of varying

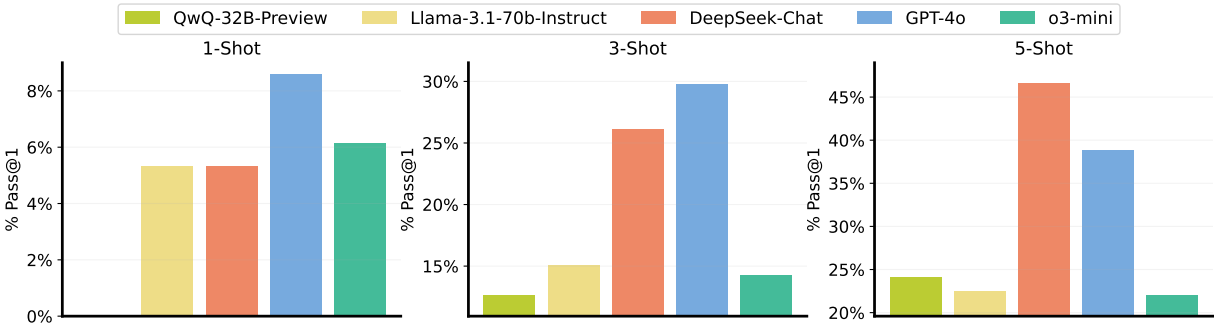


Figure 4: Performance of various models on different number of few-shot examples. * denotes reasoning LLMs.

numbers of vulnerabilities on the performance of specification synthesis, as illustrated in Figure 3. Our observations yield several insights: 1) The *pass@1* performance of LLMs tends to decline as the number of vulnerabilities increases. This decline is likely because the presence of more vulnerabilities in the kernel implementation 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 observations presented in Table 1. Therefore, reasoning-enhanced models may encounter greater challenges due to the long-context limitations inherent in OS verification scenarios.

4.5 Impact of Number of Demonstrations

Recent studies have demonstrated that in-context learning (ICL) significantly enhances the ability of LLMs to acquire new tasks from a limited set of examples (Brown et al., 2020; Dong et al., 2022). In the realm of OS verification, which is inherently complex, the provision of examples illustrating the generation of specifications from functional descriptions and code implementations exerts a substantial influence on performance outcomes. In this study, we explore various ICL settings, specifically zero-shot, one-shot, three-shot, and five-shot learning, as illustrated in Figure 4. It is noteworthy that we exclude discussions on o1 and DeepSeek-R1, owing to their prohibitive cost and time-intensive nature. Our observations reveal that the zero-shot setting (without being shown in the Figure 4) results in complete task failure, with a success rate of 0% across all models, underscoring the critical importance of demonstrations in OS verification contexts. As anticipated, the *pass@1* performance of LLMs tends to improve with the provision of additional demonstrations. Notably, DeepSeek-Chat appears to derive greater benefits from increased demonstrations. While o3-mini surpasses DeepSeek-Chat in the one-shot context, it underperforms compared to GPT-4o and DeepSeek-Chat in the three- and five-shot scenarios. We hypothesize that the advanced reasoning models currently in use generate extensive chains of reasoning traces, which may challenge the long-context learning capabilities in OS verification scenarios.

5 Conclusion

We introduce OSVBench, a robust benchmark for evaluating the performance of LLMs in generating specifications for verifying OS kernels. By framing the specification generation as a program synthesis problem, the benchmark challenges LLMs to navigate complex syntax and semantics within long-context tasks. Our comprehensive evaluation of 10 powerful LLMs reveals limitations in their current ability to handle these tasks effectively, with notable disparities in performance across models. These findings underscore the need for further advancements in LLM technology to enhance their understanding and generation capabilities in complex domains. OSVBench not only highlights existing gaps but also serves as a valuable tool for guiding future research aimed at improving verification processes in operating system development.

Limitations

While our OSVBench offers a significant advancement in evaluating LLMs for operating system kernel verification tasks, several limitations must be considered. The benchmark is specifically designed around the Hyperkernel operating system, which may not capture the full diversity of kernel architectures, potentially limiting the generalizabil-

ity of results to other systems. The complexity of tasks, consisting of approximate 30,000 tokens each, poses significant challenges in context management for LLMs, possibly overshadowing other capabilities like logical reasoning. Additionally, the confined scope of syntax and semantics within the benchmark may not fully reflect the dynamic nature of real-world operating system development environments. Current evaluation metrics may not capture qualitative aspects of successful specification generation, such as readability and adaptability, which are crucial for practical implementation. Furthermore, the benchmark lacks real-world feedback loops, such as iterative testing and debugging, limiting its ability to simulate realistic development conditions. Lastly, given the fixed nature of tasks and reliance on a single kernel, there's a risk of LLMs overfitting to specific tasks rather than developing broader, adaptable understanding, which can constrain insights into their general capabilities. These limitations can guide future efforts to enhance benchmarks for evaluating LLMs in complex, real-world programming and verification tasks.

References

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. 2016. Cogent: Verifying high-assurance file system implementations. *ACM SIGARCH Computer Architecture News*, 44(2):175–188.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, 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.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252.

Russ Cox, M Frans Kaashoek, and Robert Morris. 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.

Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. 1972. *Structured programming*. Academic Press Ltd.

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*, pages 337–340. Springer.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 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.

E Allen Emerson and Edmund M Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming: Seventh Colloquium Noordwijkerhout, the Netherlands July 14–18, 1980 7*, pages 169–181. Springer.

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

Isabelle. 2025. Isabelle: A generic proof assistant. https://isabelle.in.tum.de/. Accessed: 2025-2-14.

Bart Jacobs and Frank Piessens. 2008. The verifast program verifier. Technical report, Technical Report CW-520, Department of Computer Science, Katholieke ....

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

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

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807*.

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA. Association for Computing Machinery.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. 2016. Refinement through restraint: Bringing down the cost of verification. *ACM SIGPLAN Notices*, 51(9):89–102.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 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*, pages 158–174.

Bruce J Walker, Richard A Kemmerer, and Gerald J Popek. 1980. Specification and verification of the ucla unix security kernel. *Communications of the ACM*, 23(2):118–131.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E Gonzalez, and Ion Stoica. 2023. Rethinking benchmark and contamination for language models with rephrased samples. *arXiv preprint arXiv:2311.04850*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 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)*, pages 7443–7464.

Lichen Zhang, Shuai Lu, and Nan Duan. 2024. Selene: Pioneering automated proof in software verification. *arXiv preprint arXiv:2401.07663*.

A Types of Buggy Code Implementations

Incorrect pointer. The incorrect pointer bug occurs because the switchuvm() function is intended to switch the Task State Segment (TSS) and page table to the process p that is passed as an argument. However, instead of using p to access the kstack field, the function erroneously references the global variable proc. This misuse of the pointer results in an incorrect kernel state transition.

Bounds checking. The bounds checking bug occurs because the second condition fails to validate if size is negative. Without this check, a negative size can cause an integer underflow in (uint)i + size, bypassing the bounds check and allowing invalid memory access. This results in potential memory violations, including accessing or modifying out-of-bounds memory, leading to undefined behavior.

Memory leak. This memory leak bug occurs because the function incorrectly skips over page table entries (PTEs) when encountering a zero page directory entry (PDE). The faulty statement fails to move to the next page table, causing valid PTEs to be missed and their corresponding physical memory not being freed. This leads to leaked memory, which remains allocated but unusable, potentially exhausting system resources over time.

Incorrect I/O privilege. This I/O privilege bug occurs because the kernel fails to set the iomb field in the TSS, leaving it at its default value (0), which allows user-space processes to execute I/O instructions directly. This violates privilege separation, enabling malicious processes to bypass kernel control, access hardware, corrupt device state, or destabilize the system.

Buffer overflow. This buffer overflow bug occurs because the kernel incorrectly assumes that cpu->id (APIC IDs) are consecutive and start from zero, using it directly as an index into the cpus array. If APIC IDs are sparse or non-consecutive, this results in out-of-bounds memory access, potentially corrupting kernel memory, causing system instability, or introducing security vulnerabilities.

B Case Study

We next conduct a case study to further investigate the root causes of these errors.

Syntax error. First, we present an example of a syntax error. This error occurs during the synthesis of the state-machine specification for the system call **sys_map_pcipage**. The functional description of this system call is as follows: it verifies that the current process owns the specified PCI page. The corresponding code snippet is provided below:

```
int sys_map_pcipage(pn_t pt, size_t
    index, pn_t pcipn, pte_t perm) {
    if (!is_pcipn_pid(pcipn, pid))
        return -EACCES;
}
static int is_pcipn_pid(pn_t pcipn,
    pid_t pid) {
    ...
    return pci_table[pcipages[pcipn].
        devid] == pid;
}
```

And the model generate the specification as below:

```
old.pcipages[pcipn].devid == old.current
```

However, based on the provided programming model, *old.pcipages[pcipn]* is intended to retrieve a modeled instance of a PCIPage, which does not include a modeled *devid* attribute. This error results in an attempt to access a non-existent class field in Python, leading to the syntax error.

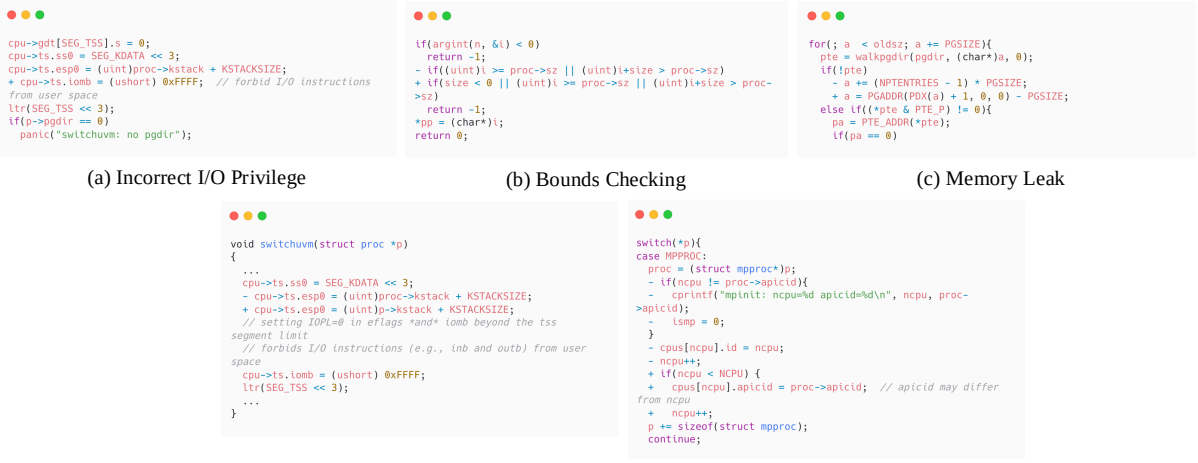
```
class PCIPage(Struct):
    owner = Map(pn_t, devid_t)
    valid = Map(pn_t, bool_t)
```

```
class KernelState(BaseStruct):
    pci = PCI()
    pcipages = PCIPage()

class PCIPage(Struct):
    ...
    owner = Map(pn_t, devid_t)

class PCI(Struct):
    owner = Map(devid_t, pid_t)
```

Based on the programming model described above, the appropriate reasoning process to determine the process identifier associated with the PCIPage, starting from its page number, can be outlined as follows: By analyzing the types of the fields involved, we can deduce that the first step is to retrieve the PCIPage instance corresponding to the given page number. Next, we should identify its owner, represented by the device identifier (devid). Using this device identifier, the associated PCI instance can then be obtained. Finally, from the PCI instance, the process identifier can be determined. Hence, the correct specification should be:

```
old.pci[old.pcipages[pcipn].owner].owner
    == old.current,
```

The LLM fails to accurately reason about the access chain required to retrieve the correct object.



(a) Incorrect I/O Privilege (b) Bounds Checking (c) Memory Leak

(d) Incorrect Pointer (e) Buffer Overflow

Figure 5: Five types of bugs.

This failure may be attributed to the long context, which might cause the LLM to lose track of the provided programming model and instead generate a specification resembling the concrete code implementation of the system call. However, it is crucial to emphasize that the kernel's abstract modeling differs fundamentally from its code implementation.

Semantic error. Next, we present another example to demonstrate the occurrence of semantic errors. This example involves a mistake in the synthesis of the state-machine specification for the system call **sys_map_proc**. The functional description of this system call is as follows: it verifies the specified permissions and rejects the operation if the permissions include write access. However, the code implementation contains an injected bug:

```
if (pte_writable(perm)) [correct]
if (!pte_writable(perm)) [bug injected]
    return -EACCES;
static inline bool pte_writable(
    uintptr_t x)
{
    return x & PTE_W;
}
```

In this case, the LLM is expected to identify the injected vulnerability by adhering to the functional description and generate the correct specification as follows:

```
perm & dt.PTE_W != 0,
```

However, the large language model (LLM) fails to accurately interpret the functional description and instead generates an incorrect specification:

```
perm & dt.PTE_W == 0,
```

This incorrect condition causes the kernel state, as defined in the specification, to transition into an inconsistent state that deviates from the correct operating system implementation, ultimately resulting in a semantic error.