

OSVBench: 操作系统验证中规范生成任务的大型语言模型基准测试

Shangyu Li¹ Juyong Jiang^{1,2} Tiancheng Zhao³ Jiasi Shen^{1*}

¹The Hong Kong University of Science and Technology

²The Hong Kong University of Science and Technology (Guangzhou)

³Georgia Institute of Technology

{ sliew, jjiang472 } @connect.ust.hk, tzhao350@gatech.edu
sjs@cse.ust.hk

Abstract

我们引入了 OSVBench，这是一个新的基准，用于评估大型语言模型 (LLM) 在生成涉及操作系统内核验证任务的完整规范代码中的表现。该基准首先通过为 LLM 提供编程模型，将规范生成问题定义为一个在语法和语义的有限范围内的程序综合问题。LLM 需要理解提供的验证假设及其潜在的语法和语义空间，然后生成对潜在有缺陷的操作系统代码实现的完整规范，期间需遵循操作系统的高级功能描述的指导。该基准建立在一个真实世界的操作系统内核，Hyperkernel 之上，总共包含 245 个复杂的规范生成任务，每个任务都是一个约 20k-30k 标记的长上下文任务。我们对 12 种 LLM 的全面评估显示，当前 LLM 在操作系统规格生成任务上的表现有限。它们在基准测试中表现出显著的差异，突出反映了它们在处理长上下文代码生成任务的能力上的不同。评估工具包和基准可在 <https://github.com/lishangyuhkust/OSVBench> 获取。

1 介绍

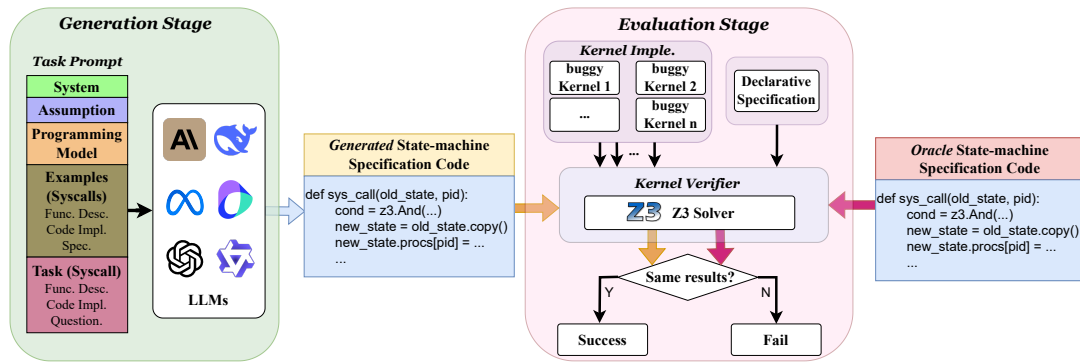
大型语言模型 (LLM) 在软件工程任务中展示了巨大的潜力，如代码生成 (Austin et al., 2021; Athiwaratkun et al., 2022; Zan et al., 2023; Jiang et al., 2024)、代码总结 (Ahmed et al., 2024) 和错误修复 (Jin et al., 2023)。然而，一个重要的软件工程方面仍未得到充分探讨：软件验证。软件验证通过严谨的数学推理来证明软件中不存在错误 (Dahl et al., 1972)，这在确保航空航天、医疗保健和核能等安全关键领域的软件正确性方面至关重要 (Klein et al., 2009; Amani et al., 2016; O'Connor et al., 2016)，在这些领域中，软件错误可能导致灾难性的经济损失甚至危及人类生命。然而，手动软件验证具有挑战性且耗时，要求具备正式方法和程序分析的高级知识。因此，能够进行验证的专业人员有限，这突显了该领域自动化的必要性。在这种背景下，操作系统内核验证仍是一个需要高度专业知识的有价值且重要的利基领域。

在本文中，我们研究了大型语言模型 (LLMs) 在自动化操作系统 (OS) 内核验证中的能力。我们引入了一套基准测试套件以评估这些模型在验证操作系统内核中的有效性，操作系统内核是许多关键基础设施的重要组成部分。正式验证操作系统内核通常涉及首先定义出精确的规范，这些规范概述了内核必须满足的属性 (Klein et al., 2009; Chajed et al., 2022; Chen et al., 2015, 2017)，接下来使用定理证明工具构建形式证明以展示对这些规范的合规性。此外，操作系统内核固有的复杂性、并行性以及硬件交互使得验证过程极具挑战性。例如，著名的 seL4 微内核的验证 (Klein et al., 2009) 需要耗费 11 个工作年才完成其 10k 行 C 代码的验证，而验证 BilbyFs 文件系统的两个操作 (Amani et al., 2016) 则耗费了 9.25 个人月的工时以完成 1,350 行代码的验证。定制化形式证明被广泛认为是一项具有挑战性的任务，现有研究主要集中在自动化生成证明 (Chen et al., 2024; Zhang et al., 2024)，却常常忽视规范开发这一显著的挑战 (Sammler et al., 2021; Leino, 2010; Jacobs and Piessens, 2008; Ma et al., 2024)。在操作系统内核验证的背景下，这一挑战被进一步加剧，因为规范通常是即兴的 (Chen et al., 2017, 2015; Chajed et al., 2022; Sigurbjarnarson et al., 2016)，需具备丰富的领域专业知识。例如，为 seL4 创建形式规范 (Klein et al., 2009) 耗费了 7 个人月的工时。

鉴于大语言模型 (LLMs) 的先进编码能力，值得探索其生成操作系统内核规范的潜力——这是一种正式定义操作系统内核行为的源代码的特殊形式。为此，我们引入了从 Hyperkernel 项目 (Nelson et al., 2017) 衍生出来的基准套件 OSVBench，用于评估 LLMs 在生成用于验证操作系统内核功能正确性的规范方面的能力。该基准旨在促进操作系统内核规范生成的自动化。它包含 245 个规范生成任务，每个任务都是一个复杂且精密的程序合成任务，具有较长的上下文，大约 20000 到 30000 个标记。Figure 1 展示了每个规范生成任务的工作流程。

我们进行了全面的实验，从自然语言的功能

*Corresponding Author.



描述中生成形式规格，并通过向操作系统内核中注入 5 种真实世界的漏洞类型来实现具有不同类型和数量漏洞的操作系统内核代码。这些实验结果展示了大型语言模型在自动化生成用于操作系统内核验证的形式规格方面的潜力。最后，我们对合成数据集进行了严格的数据净化 (Yang et al., 2023b)，以去除与覆盖数据集测试子集 $D_{test}^{(0)}$ 中内容紧密相似的样本。本文的主要贡献可以总结如下：

- 我们开始在操作系统验证任务的背景下探索大规模语言模型，这些任务需要深入理解和操作大量上下文信息和领域特定知识。
- 我们推出 OSVBench，这是一个为操作系统验证设计的基准，用于评估大型语言模型 (LLM) 在为操作系统内核验证生成规范方面的能力。
- 我们对最先进的大型语言模型在生成旨在验证操作系统内核功能正确性的规格方面进行了全面评估。我们还讨论了不同类型和数量的错误对生成规格的质量和有效性的影响。

2 相关工作

用于软件验证的 LLM。软件验证 (D’silva et al., 2008) 确保软件遵循指定的属性或要求，在保证软件的可靠性和正确性方面起到至关重要的作用。在该领域，操作系统内核的验证 (Klein et al., 2014) 一直是确保关键软件系统可

靠性和安全性的核心研究目标。早期的基础性工作包括像 UCLA 安全 Unix (Walker et al., 1980)、PSOS (Feiertag et al., 1977) 和 KIT (Bevier, 1989) 这样的努力，为内核正确性的形式化方法奠定了基础。最近的进展扩展到利用形式化方法，如定理证明 (Nelson et al., 2017) 和模型检验 (Klein et al., 2009)，旨在实现具有数学验证属性的高保障内核。先前利用 LLM 进行软件验证的工作主要集中在从规格生成证明 (Chen et al., 2024; Zhang et al., 2024)，这涉及将一种确定性的形式语义表示（各种形式的规格）翻译成另一种（使用形式语言表达的证明）。此外，一些研究探索了规格生成任务。然而，大部分工作专注于通用规格生成 (Ma et al., 2024)，这与操作系统内核规格的生成有显著不同，因为在该领域中遇到的验证假设和要求截然不同。

代码生成的 LLM。近年来，LLM 用于代码生成已受到广泛关注，因为这些模型在从自然语言描述中合成代码片段方面表现出显著的能力 (Austin et al., 2021; Athiwaratkun et al., 2022; Zan et al., 2023; Jiang et al., 2024)。多项研究探索了 LLM 在各种代码生成任务中的潜力，从简单功能生成 (Chen et al., 2021; Luo et al., 2023) 到更复杂的编程挑战 (Jimenez et al., 2023; Ding et al., 2024)。尽管取得了这些进展，针对特定领域（如操作系统内核验证）的代码生成面临着一般用途的 LLM 尚未完全解决的独特挑战。所涉及的语法和语义的复杂性和特异性要求模型不仅要理解编程语言，还要掌握领域特定的

知识和验证假设。这一挑战需要一个基准来评估 LLM 在生成操作系统内核验证规范方面的能力。

用于静态/动态程序分析。现有的基于 LLM 的静态分析方法主要依赖于提示 LLM 对程序 (Wang et al., 2024b,c,a) 进行源到汇的可达性分析。然而，专门针对 Linux 内核的基于 LLM 的静态分析研究较少，因为它对长上下文推理有很高的要求，这是一个重大挑战。这一复杂性来自于内核复杂的调用图和别名关系。相比之下，动态分析技术（如模糊测试）在包括智能合约 (Shou et al., 2024)、Linux 内核 (Yang et al., 2023a) 和通用领域 (Xia et al., 2024) 在内的各种领域中得到了更广泛的 LLM 应用。

3 OSVBench

在本节中，我们将讨论规范生成问题的表述以及基准构建的细节。

3.1 预备知识

Hyperkernel (Nelson et al., 2017) 是一个操作系统内核验证项目，它包括一个实际的内核实现和一个构建于自动定理证明器 Z3 之上的验证框架。内核实现支持 50 个系统调用，涵盖了进程管理、虚拟内存、文件描述符、设备交互、进程间通信和调度等关键功能。整个代码库由大约 18,000 行 C 语言和汇编语言组成，包含内核实现和相关的用户空间组件。

正如在 Figure 1 的规格质量评估部分所展示的那样，Hyperkernel 的验证器需要两种规格作为输入。第一种是状态机规格，通过描述 OS 内核的预期行为来定义功能正确性。第二种是高层声明式规格，其概述了状态机规格必须满足的总属性和不变量。例如，其中一个这样的属性确保了任何给定进程的子进程数总是等于识别该进程为其父进程的总进程数。

验证者建立了两个定理。第一个证明内核实现是状态机规格的细化，表示为 $\forall \sigma_{Impl} \in \Sigma_{Impl}, \forall c \in C, \exists \sigma_{Spec} \in \Sigma_{Spec}$, such that $\sigma_{Impl}(c) = \sigma_{Spec}(c)$ ，这表明对于实现中的每个内核状态 σ_{Impl} ，在任何条件 C 下，存在一个在相同条件 c 下等效的状态机规范中定义的相应内核状态 σ_{Spec} 。第二个定理表明状态机规范满足在声明性规范中定义的性质和不变量。

在我们对规格生成任务的表述中，提供声明性规格，LLM 的作用是合成状态机规格，具体详见 subsection 3.2。一旦状态机规格准备好，其验证器就会对操作系统内核编译后的 LLVM IR (Lattner and Adve, 2004) 执行符号执行 (Cadar et al., 2008)，并调用 z3 (De Moura

and Bjørner, 2008) 求解器在具体实现中的真实内核状态转换和状态机规格中定义的状态转换上执行等价性检查。此外，验证器还确保状态机规格遵循高级声明性规格。任何检测到的不一致性都表明操作系统内核代码实现中存在错误，或者状态机规格未能准确描述系统调用的预期功能。

我们选择 Hyperkernel 作为基准的原因如下：1) Hyperkernel 采用标准化方法 (Klein et al., 2010)，将内核执行建模为一个状态机，通过一组具有确定语义的 Python 类表示。编写这些规范需要大量专业知识和不小的努力，使其成为评估规范生成任务的合适且具有挑战性的基准。2) Hyperkernel 使用自动定理证明器，特别是 Z3 求解器，来正式验证操作系统内核的功能正确性，而不是使用交互式定理证明器，例如 Isabelle (Isabelle, 2025)、Coq (Coq, 2025) 和 Dafny (Dafny, 2025)。通过使用自动求解器，可以专注于规范生成任务，简化验证过程。

3.2 问题公式化

形式化验证操作系统内核本质上是复杂的，需要协调众多组件，这使得大语言模型 (LLMs) 难以直接解决且难以评估生成规格的准确性。具体来说，Hyperkernel 中的固有问题削弱了 LLMs 在合成正确规格方面的有效性：

- 各种验证假设。Hyperkernel 包含许多隐式和显式的验证假设，包括与用于证明的定理、硬件行为和内存布局等相关的假设。
- 状态机规范与声明式规范之间的相互依赖关系。在 Hyperkernel 的设计中，用户需要定义声明式规范和状态机规范。如前所述，验证器确保状态机规范满足声明式规范，后者封装了一组属性和不变量。然而，这种相互依赖关系使得对 LLM 生成的规范的验证变得复杂。即使验证器产生了成功的验证结果，也不能保证内核实现的正确性，因为初始的声明式规范自身可能包含错误。
- 无限的搜索空间。状态机规格不仅依赖于内核行为建模，还依赖于一组隐含常数和外部函数。这些包括从内核实现的编译 LLVM IR 中派生的常数、Z3 函数、实用函数和其他函数。任何这些组件的遗漏或不完整定义都会导致不完整的综合域，这可能导致错误的规格以及一个无边界的、无限的搜索空间。

为了解决上述挑战并确保 LLMs 的可处理性，我们通过引入以下内容，将规范生成重新

```

Specification := State
State := if cond State, State' | State |
        State.(fieldi ← Expression)*
Expression := if cond Expression, Expression' | Param |
        Expression op Expression | Const | State.fieldi
cond := and cond, cond | or cond, cond | Param op Const |
        State.fieldi op Const | State.fieldi op Param
op := + | - | × | ÷ | == | != | > | < | >= | <=

```

Figure 2: 用于规格合成的抽象领域。在该领域中，State 表示内核状态，而 Param 表示提供给系统调用的参数。Const 指代可用的常量，op 则包括算法和逻辑运算符的集合。Expression 表示使用常量、内核状态字段、参数和算法运算符形成的表达式，而 cond 指的是通过将逻辑运算符应用于内核状态字段、常量或参数值，并使用合取 (and) 或析取 (or) 进行组合而形成的条件。规格定义了基于当前状态的系统转换后产生的后续内核状态。内核状态的转换或字段赋值将根据不同的条件进行。

表述为一个程序合成问题：(1) 显式验证假设：系统性地记录相关假设，以指导 LLMs 实现可靠的验证。(2) 固定的声明式规范：限制声明式规范，使 LLMs 能够专注于生成状态机规范，从而简化验证过程。(3) 确定性合成领域：定义常量、外部函数和类以限制搜索空间，并确保合成过程对于 LLMs 保持可处理性。具体而言，Figure 2 正式定义了抽象合成领域，促进 LLMs 生成状态机规范。

规范生成任务被正式定义为一个代码生成任务，旨在综合完整的规范代码以验证操作系统内核的功能正确性，受编程模型范围语义的约束。此过程由系统调用的精确高级功能描述及其可能有缺陷的实现指导，具有固定的验证假设和声明性规范。在正式定义了问题之后，解决综合准确规范的挑战成为大型语言模型 (LLMs) 的责任。这些挑战与编程语言无关，源于以下因素：(a) 将功能描述的语义准确映射到相应的规范代码，例如附录 A 中所示，(b) 解决由于内核状态合成分歧带来的挑战，这在附录 ?? 中也有示例，以及 (c) 处理和理解平均为 20k 到 30k 个符号的上下文信息，这需要先进的长上下文学习能力，以确保准确的规范生成。

3.3 基准任务构建

根据规格生成问题的定义，每个任务都需要功能描述及其可能有缺陷的实现。因此，从 Hyperkernel 的 49 个系统调用开始，我们根据对它们在操作系统内核中的预期行为、目的和交互的潜在理解，手动起草了每个系统调用的

多个功能描述版本。然后，我们根据清晰度和技术准确性选择了最佳版本 (Hao et al., 2023)。此外，我们从一个正确的操作系统内核实现开始，并系统地生成一组故障实现，这反映了在现实世界中操作系统内核实现不一定是正确的，并可能包含各种类型和数量的错误。这是通过随机引入五种源自 xv6 内核的现实世界错误到 Hyperkernel 代码库中实现的，并在 Appendix B 中做了示例。这种方法使我们能够评估各种漏洞对生成准确状态机规格的 LLMs 性能的影响。

最后，我们创建了总计 245 个规范生成任务，每个任务包括一个系统调用的正确高级功能描述，以及其对应的可能有漏洞的代码实现。在这 245 个代码实现中，部分是正确的，而其他则包含不同数量的错误，从一个到五个不等。

4 评估

4.1 实验设置

最先进的 LLM。我们对由六家领先机构开发的当前最先进的大型语言模型 (LLM) 进行了评估：OpenAI、DeepSeek、Meta、Anthropic、字节跳动和 Qwen 团队。具体来说，我们的评估包括来自 OpenAI 的 o1、o3-mini 和 GPT-4o 模型；来自 DeepSeek 的 DeepSeek-R1 和 DeepSeek-Chat 模型；来自 Meta 的 Llama-3.1-70B-Instruct 和 Llama-3.1-8B-Instruct 模型；来自 Qwen 团队的 QwQ-32B-Preview、Qwen2.5-72B-Instruct 和 Qwen2.5-Coder-7B-Instruct 模型；来自 Anthropic 的 Claude-3.5-sonnet；以及来自字节跳动的 Doubao-1.5-pro。评估利用 OSVBench 框架系统地评估这些模型在生成 OS 内核验证规范任务中的性能。这些 LLM 在参数数量、开源可用性、数据截止日期和预训练目标等关键特征上有所不同。对于所有模型，我们使用贪婪搜索解码策略与 pass@1 进行评价的一致性。提示设计。如 Figure 1 所示，每个任务的提示是针对操作系统内核的特定系统调用而专门设计的。提示结构分为四个关键组件：系统验证假设、编程模型、少量示例和任务问题，如 Appendix D 所示。少量示例由操作系统内核验证专家精心挑选，以确保其代表性。

规范质量指标。为了系统地评估 LLM 在任务上的表现，我们定义了几个指标。如 Figure 1 中的规范质量评估阶段所述，我们有意创建了多个具有已知、人工插入的错误的操作系统内核实现。指标 Pass@N 表示生成的 N 个规范中至少有一个是正确的，这意味着它在所有操作系统内核实现中识别的不一致与由预言机规范检测到的不一致相吻合。语法错误指标表示生成的规范未能正确执行或因异常而终止的情

Table 1: 使用 5-shot 提示的各种模型性能比较 (通过率 @1%)。标有 * 的模型表示推理 LLM，而 Δ 则表示闭源模型；所有未标记的模型都是开源的。从 Incorrect Pointer 到 Bounds Checking 的列对应于在任务提示中注入到系统调用代码中的特定类型的漏洞。列 Correct 表示所提供的代码实现没有漏洞的情况。最后，列 Total 报告了所有 245 个任务中总体的通过率 @1。

Institution	Model	Incorrect Pointer	Incorrect Privilege	Memory Leak	Buffer Overflow	Bounds Checking	Correct	Total
OpenAI	o1 $^{*\Delta}$	12.68	21.43	13.51	20.37	23.15	28.57	23.67
	o3-mini $^{*\Delta}$	19.72	18.75	18.92	12.96	15.74	26.53	22.04
	GPT-4o Δ	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	0.00	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 Δ	39.44	41.96	39.19	48.15	39.81	46.94	44.90
ByteDance	Doubao-1.5-pro Δ	50.70	48.21	45.95	40.74	52.78	63.27	55.1

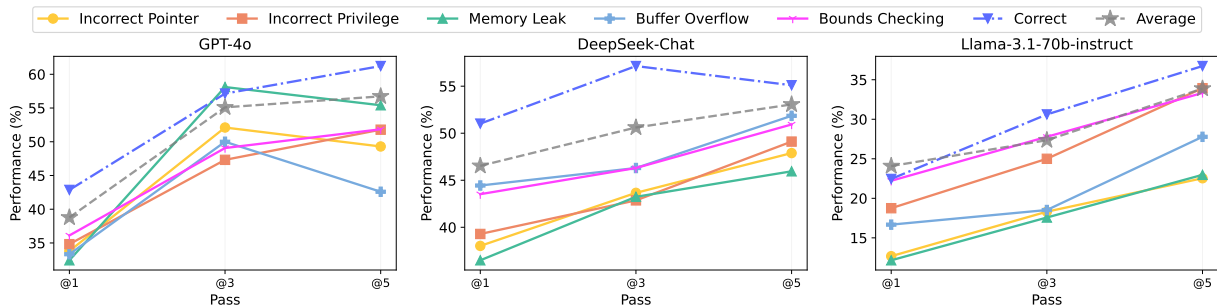


Figure 3: 各种模型在 Pass@1、Pass@3 和 Pass@5 上的性能比较。随着 N 在 Pass@N 中增加，模型的平均性能表现出持续改进。

况。最后，语义错误指的是验证器成功地将规范翻译成 SMT (De Moura and Bjørner, 2008) 并对操作系统内核实现进行验证，但指出的不一致与预言机规范指定的不一致不同的情况。

4.2 主要结果

Table 1 展示了大语言模型在不同机构和错误分类中的性能结果。表现最佳的闭源大语言模型 Doubao-1.5-pro 以最高的平均 pass@1 通过率 (55.1%) 和生成正确规范的出色能力 (63.27%)，超过了表现最佳的开源大语言模型 DeepSeek-Chat，展示了在所有错误类型中的强大性能。

一般来说，参数规模较大的模型往往表现优于其较小的同类模型。例如，Llama-3.1-8B-Instruct 和 Qwen2.5-Coder-7B-Instruct 的性能明显逊色于其较大的同类模型，如拥有超过 700 亿参数的 Llama-3.1-70B-Instruct 和 Qwen2.5-72B-Instruct。结果还突出了由于存在各种类型的错误而导致的性能下降，其影响因模型和错

误类别而异。例如，内存泄漏错误对 DeepSeek-R1 模型的影响最为显著，而不正确的指针错误对 o1 模型的影响最大。

令人惊讶的是，广泛认可的推理模型，如 o1 和 DeepSeek-R1，在此任务中并未持续优于其他模型。特别是，o1 模型表现较弱，甚至不如 QwQ-32B-Preview 模型，这对某些推理模型在这些任务中的优越性假设提出了挑战。我们推测，所使用的先进推理模型产生了冗长的推理链，这可能会对 OS 验证场景中的长上下文学习能力构成挑战。

Pass@k 性能。Pass@k 的性能使用 GPT-4o、DeepSeek-Chat 和 Llama-3.1-70b-instruct 进行评估，如 Figure 3 所示。结果显示，当 k 从 1 增加到 3 再到 5 时，平均通过率有了一致的提升。此外，GPT-4o 在 k = 3 和 k = 5 时优于 DeepSeek-Chat。

Table 2: 由 LLM 生成的规范中的语法和语义错误率 (%)，涵盖所有 245 个任务。* 表示推理 LLM。较低的错误率表明性能更好。△ 表示闭源模型，而未标记的模型为开源模型。

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

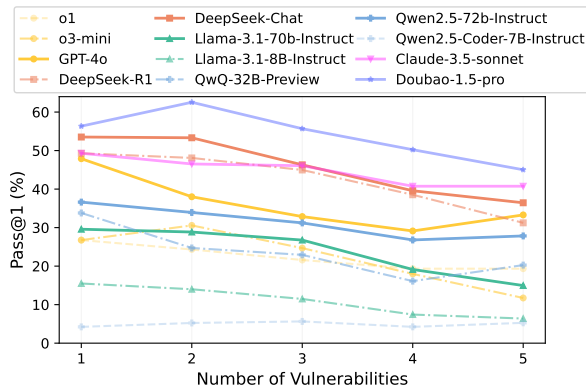


Figure 4: 12 个大型语言模型在使用注入不同数量漏洞的系统调用实现进行规范生成任务上的性能比较 (Pass@1 %)。

4.3 错误分析和自我修复

Table 2 展示了各种大型语言模型的语法和语义错误率。值得注意的是，表现最差的模型 Qwen2.5-Coder-7B-Instruct 相比于表现最好的模型 Doubao-1.5-pro 更容易产生语法错误。这体现在 Doubao-1.5-pro 的语义到语法错误率比例 (21.22 % / 23.67 %) 相对于 Qwen2.5-Coder-7B-Instruct (11.02 % / 86.12 %) 更高。

为了进一步评估 LLMs 的错误处理能力，使用 GPT-4o 和 DeepSeek-Chat 进行了两轮自我修复过程，正如 Table 3 中所展示的那样。在第一轮中，模型使用自己生成的规范以及相应的验证错误作为修复输入。如果第一轮规范仍然不正确，模型则利用所有错误版本及其相关验证错误进行第二轮修复。值得注意的是，自我修复方法始终提高了针对操作系统内核验证的规范生成任务的性能。然而，随着修复轮数的增加，修复成功率下降。此外，我们在

Appendix C 中对两个典型的错误案例进行了详细的案例研究，以进一步调查这些错误的根本原因。

4.4 漏洞数量的影响

在实际应用中，操作系统内核通常只有少量潜在的漏洞，并且严重漏洞的实例较少。因此，我们进一步探讨了不同数量漏洞对规格合成性能的影响，如图 4 所示。我们的观察得出以下几个见解：1) 随着漏洞数量的增加，大型语言模型的 pass@1 性能趋于下降。这种下降可能是因为内核实现中的漏洞增多使模型难以准确理解功能描述。2) 先进的推理模型相比于传统的跟随指令模型表现不佳。例如，GPT-4o 在所有漏洞级别上都始终优于 o1 和 o3-mini。同样，DeepSeek-R1 不如 DeepSeek-Chat。这些发现与表 1 中的观察结果一致。因此，在操作系统验证场景中，由于长时间上下文的限制，增强推理的模型可能会面临更大的挑战。

4.5 示范次数的影响

最近的研究表明，情境学习 (ICL) 显著增强了大型语言模型 (LLM) 从有限的例子集中获取新任务的能力 (Brown et al., 2020; Dong et al., 2022)。在本质上复杂的操作系统验证领域，提供从功能描述和代码实现中生成规范的示例对性能结果有着重大影响。在本研究中，我们探索了各种 ICL 设置，具体为零样本、单样本、三样本和五样本学习，如图 5 所示。值得注意的是，由于其高昂的成本和耗时的性质，我们排除了对 o1 和 DeepSeek-R1 的讨论。我们的观察表明，零样本设置 (未在图中显示) 导致任务完全失败，所有模型的成功率为 0%，这强调了在操作系统验证背景下演示的重要性。正如预期的那样，随着提供更多的演示，LLM 的 pass@1 性能趋于提高。值得注意的是，DeepSeek-Chat 似乎从增加的演示中获得了更大的好处。虽然在单样本情境下 o3-mini 优于 DeepSeek-Chat，但在三样本和五样本情境中，其表现不及 GPT-4o 和 DeepSeek-Chat。我们假设当前使用的高级推理模型生成了大量的推理痕迹，这可能对操作系统验证场景中的长情境学习能力构成挑战。

5 结论

我们引入了 OSVBench，一个用于评估 LLM 在生成验证操作系统内核的规范方面性能的强大基准。通过将规范生成框架化为程序合成问题，该基准挑战 LLM 在长上下文任务中导航复杂的语法和语义。我们对 12 个强大 LLM 的全面评估揭示了其当前在有效处理这些任务方面的局限性，并且不同模型的性能存在显著差

Table 3: 对使用 5-shot 提示的模型进行的两轮自修复性能比较（修复成功率%）。列 Syntax error 和 Semantic error 表示生成错误类型的规格的修复成功率。从 Incorrect Pointer 到 Correct 的列表从任务提示中导出的规格的修复成功率，这些规格按照提示中的有缺陷代码实现类型分类。Total 列报告每轮处理中所有错误规格的总体修复成功率。括号外的值表示第一次修复轮次的成功率，而括号内的值表示第二轮的成功率。

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 (0.00)	6.56 (3.51)	20.83 (0.00)	11.45 (2.59)

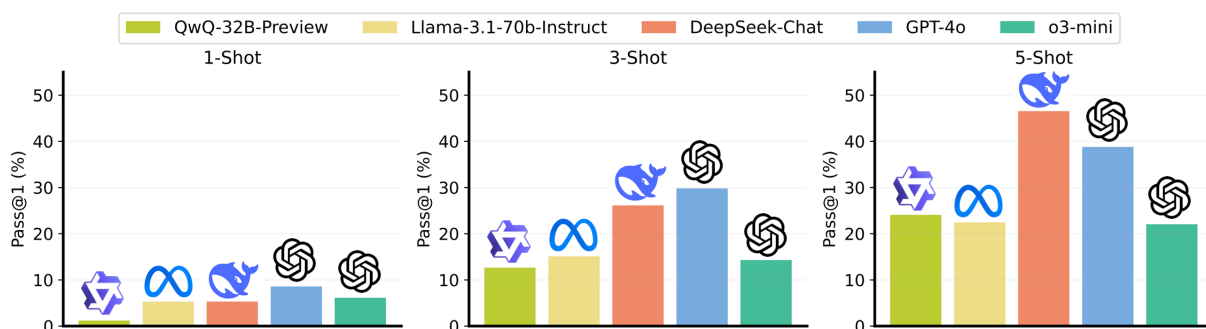


Figure 5: 不同模型在不同数量的小样本示例上的表现。随着示例数量的增加，五种最先进模型的性能有了大幅提高。

异。这些发现强调了在 LLM 技术中进一步发展的必要性，以增强其在复杂领域内的理解和生成能力。OSVBench 不仅突显了现有的空白，还作为指导未来研究的重要工具，旨在改善操作系统开发中的验证过程。

尽管我们的 OSVBench 在评估用于操作系统内核验证任务的 LLMs 方面提供了显著进步，但仍需考虑几个局限性。该基准测试是围绕 Hyperkernel 操作系统特定设计的，这可能无法涵盖内核体系结构的全部多样性，可能限制结果推广到其他系统的能力。任务的复杂性，每个大约包含 20k 到 30k 个 token，在上下文管理方面对 LLMs 提出了重大挑战，可能会掩盖其他能力，如逻辑推理。此外，基准测试中语法和语义的范围有限，可能无法充分反映真实世界操作系统开发环境的动态特性。当前评估指标可能无法反映成功规范生成的定性方面，例如可读性和适应性，这对实际实施至关重要。此外，该基准测试缺乏真实世界的反馈循环，例如迭代测试和调试，限制了其模拟现实开发条件的能力。最后，鉴于任务的固定性质和依赖于单个内核，存在 LLMs 过拟合特定任务而非发展更广泛的、适应性理解的风险，这可能限制对其一般能力的洞见。这些局限性可以指导未来的努力，以增强用于评估 LLMs 在复杂的、真实世界编程和验证任务中的基准测试。

我们感谢钟思哲和谭惠日的宝贵意见和反馈。此外，我们感谢张济鹏对初始问题表述的深刻讨论。

References

- 2025. Linux cve announcement. <https://lore.kernel.org/linux-cve-announce/>. Accessed: 2025-2-14.
- Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*, pages 1–13.
- 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.
- Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224.
- Tej Chajed, Joseph Tassarotti, Mark Theng, M Frans Kaashoek, and Nikolai Zeldovich. 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)*, pages 447–463.
- Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nikolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286.
- Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nikolai Zeldovich. 2015. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37.
- 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.
- 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.
- 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.
- Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardan Amiri Sani. 2023. Syszdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE.
- 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*.
- Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 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*, pages 1646–1656.
- 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.
- Gerwin Klein, Thomas Sewell, and Simon Winwood. 2010. Refinement in the formal verification of the sel4 microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 323–339. Springer.
- 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, Xibo 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.
- Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108*.
- Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. {Push-Button} verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16.
- 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.
- Chengpeng Wang, Yifei Gao, Wuqi Zhang, Xuwei Liu, Qingkai Shi, and Xiangyu Zhang. 2024a. Llmsa: A compositional neuro-symbolic approach to compilation-free and customizable static analysis. *arXiv preprint arXiv:2412.14399*.
- Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024b. Llmdfa: Analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems*, 37:131545–131574.
- Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. 2024c. Sanitizing large language models in bug detection with data-flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3790–3805.
- Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.
- Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023a. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*.
- Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E Gonzalez, and Ion Stoica. 2023b. 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*.

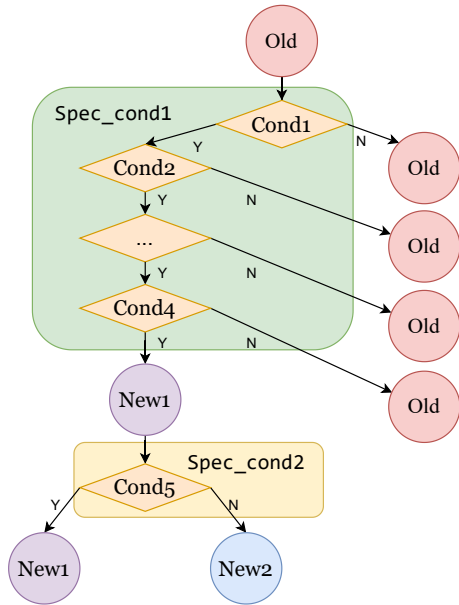


Figure 6: 状态机规范中分歧的内核状态 (Figure 7 的 sys_close 系统调用 (Figure 9))。标记为“Old”的节点表示系统调用执行前的内核状态，而标记为“New1”或“New2”的节点则描述了成功完成系统调用后的潜在内核状态。菱形形状的节点表示规范中定义的条件。

A 合成挑战

合成具有相应语义的正确规范。在我们的表述中，LLMs 的任务是在编程模型的语义范围内，由系统调用的高级功能描述和代码实现指导，合成状态机规范。为了从非正式的自然语言功能描述中合成正确的规范，这个过程必须确保准确的字段访问、适当的常量选择和精确的条件选择。具体来说，为了实现确定性的规范合成，合成过程应在一个符号三元关系内操作，记为 $\langle \text{Desc}, \text{Impl}, \text{Model} \rangle$ 。这一规则要求，为了合成规范中的任何给定语句，LLM 应首先识别相关的功能描述 (Desc)，然后找到相应的具体内核实现 (Impl)。最后，它必须通过参考编程模型中定义的适当模型 (Model) 来解释预期的语义并合成正确的规范。

例如，Figure 9 显示了对系统调用 sys_close 的部分实现。功能描述指出：“这涉及更新进程的文件描述符表以标记描述符为未使用，并减少进程的打开文件描述符计数。此外，系统更新文件的引用计数以反映文件引用的减少。”然后，LLM 在 Figure 9 的 L12-14 中定位到相关的内核代码实现，如 Listing 2 所示。

通过在编程模型 Figure 8 和综合域 Figure 2 中进行搜索，综合得到的状态机规格应对应于如 Listing 1 所示的 Figure 7 中的 L13-15。

在这里，LLM 将诸如“进程的文件描述符表”之类的术语解释为“new1.procs[pid].ofile”，“进

Figure 7: 系统调用 sys_close Figure 9 的状态机规范。在这一上下文中，“旧”表示初始内核状态，而“新 1”到“新 3”表示随后的内核状态。

```

1 def sys_close(old, pid, fd):
2     spec_cond1 = z3.And(
3         z3.And(pid > 0, pid < dt.NPROC),
4             # cond1.
5         z3.And(fd >= 0, fd < dt.NOFILE),
6             # cond2.
7         z3.Or(
8             pid == old.current,
9             old.procs[pid].state == dt.
10                proc_state.PROC_ZOMBIE),
11            # cond3.
12         z3.And(z3.UGT(old.procs[pid].
13                ofile(fd), 0), z3.ULT(old.
14                procs[pid].ofile(fd), dt.
15                NOFILE)) # cond4.
16     )
17     new1 = old.copy()
18     fn = new1.procs[pid].ofile(fd)
19     new1.procs[pid].ofile[fd] = z3.
20         BitVecVal(0, dt.fn_t)
21     new1.procs[pid].nr_fds[fd] -= 1
22     new1.files[fn].refcnt[(pid, fd)] -=
23         1
24     spec_cond2 = z3.And(new1.files[fn].
25         refcnt() == 0) # cond5.
26     new2 = new1.copy()
27     new2.files[fn].type = dt.file_type.
28         FD_NONE
29     new2.files[fn].value = z3.BitVecVal
30         (0, dt.uint64_t)
31     new2.files[fn].offset = z3.BitVecVal
32         (0, dt.off_t)
33     new2.files[fn].omode = z3.BitVecVal
34         (0, dt.uint64_t)
35     new3 = util.If(spec_cond2, new2,
36         new1)
37     return spec_cond1, util.If(
38         spec_cond1, new3, old)

```

程打开文件描述符的数量”解释为“new1.procs[pid].nr_fds[fd]”，而“文件的引用计数”则解释为“new1.files[fn].refcnt[(pid, fd)]”。在这个上下文中，“new1”表示内核状态，而“new1.procs[pid]”则指定了进程，等等。

分歧内核状态合成。为了合成一个准确的状态机规格，必须根据合成域 Figure 2 确定系统在特定条件下转变到的内核状态。这个过程需要 LLMs 执行高级推理，因为许多系统调用涉及级联和相互依赖的条件。Figure 6 展示了由于系统调用 sys_close 导致的内核状态分歧。具体来说，它表明当条件 spec_cond1 和条件 spec_cond2 都满足时，内核转换到状态 new1。如果条件 spec_cond1 满足但条件 spec_cond2 未满足，内核转换到状态 new2。最后，如果条件 spec_cond1 和条件 spec_cond2 都不满足，内核保持在状态 old。

Figure 8: 编程模型的一部分。

```

1 class KernelState(BaseStruct):
2     procs = Proc()
3     pages = Page()
4     files = File()
5     pcipages = PCIPage()
6     pci = PCI()
7     ...
8 class Proc(Struct):
9     ...
10    ofile = Map(pid_t, fd_t, fn_t)
11    nr_fds = Refcnt(pid_t, fd_t, size_t)
12 class File(Struct):
13    ...
14    refcnt = Refcnt(fn_t, (pid_t, fd_t),
15                    size_t)
16 class PCIPage(Struct):
17    owner = Map(pn_t, devid_t)
18    valid = Map(pn_t, bool_t)
19 class PCI(Struct):
20    ...
    owner = Map(devid_t, pid_t)

```

Listing 1: Relevant state-machine specification.

```

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

```

B 五种错误类型的示例

我们从 xv6 内核 (Cox et al., 2011) 中选择了五种类型的错误注入到内核实现中，如下所示。这些类型的漏洞代表了在操作系统内核领域中经常观察到的常见且关键的漏洞类型，例如 Linux 内核 (lin, 2025)。

不正确的特权。如 Figure 10 (a) 所示，不正确的特权错误发生是因为内核未能在任务状态段 (TSS) 中设置 iomb 字段，使其保持默认值 (0)，这允许用户空间进程直接执行 I/O 指令。这违反了特权分离，使恶意进程能够绕过内核控制，访问硬件，损坏设备状态或使系统不稳定。

边界检查。正如 Figure 10 (b) 所示，边界检查错误的发生是因为第二个条件未能验证 size 是否为负数。缺少该检查，一个负的 size 会导致 (uint)i + size 中的整数下溢，绕过边界检查并允许无效的内存访问。这可能导致内存违规，包括访问或修改超出边界的内存，从而导致未定义的行为。

内存泄漏。如 Figure 10 (c) 中所示，内存泄漏错误是由一个错误的语句引起的。当在寻找可释放的现有页表项 (PTEs) 时，遇到零页目录项 (PDE)，搜索将从下一个页表的第一个条目重新开始，而不是当前条目。这导致内存泄

Figure 9: 系统调用 sys_close 的部分代码实现。

```

1 int sys_close(pid_t pid, int fd) {
2     if (!is_pid_valid(pid)) // Cond1.
3         return -ESRCH;
4     if (!is_fd_valid(fd)) // Cond2.
5         return -EBADF;
6     ...
7     clear_fd(pid, fd);
8     ...
9 }
10 static inline void clear_fd(pid_t pid,
11                             int fd) {
12     ...
13     file = get_file(get_fd(pid, fd));
14     proc->ofile[fd] = 0;
15     --proc->nr_fds;
16     if (--file->refcnt == 0) { // Cond5.
17         ...
18     }
19 }

```

Listing 2: Relevant potentially buggy code implementation according to the functional description.

```

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

```

漏，这些内存仍然被分配但不可用，可能随着时间的推移耗尽系统资源。

不正确的指针。如 Figure 10 (d) 所示，错误指针 bug 发生是因为 switchvm() 函数旨在切换任务状态段 (TSS) 和页表到作为参数传递的进程 p。然而，函数没有使用 p 来访问 kstack 字段，而是错误地引用了全局 proc。这种指针的误用导致内核状态的不正确转换。

缓冲区溢出。如 Figure 10 (e) 所示，缓冲区溢出错误发生的原因是内核错误地假设 cpu->id (APIC IDs) 是连续的并且从零开始，直接将其用作 cpus 数组的索引。如果 APIC IDs 是稀疏的或不连续的，这将导致越界内存访问，可能会破坏内核内存，引起系统不稳定，或引入安全漏洞。

C 案例研究

接下来，我们进行一个案例研究，以进一步调查这些错误的根本原因。

语法错误。首先，我们展示一个语法错误的例子。这个错误发生在系统调用 sys_map_pci_page 的状态机规范合成期间。该系统调用的功能描述如下：它验证当前进程是否拥有指定的 PCI 页。相应的代码片段在 Listing 3 中展示。

LLM 生成如 Listing 4 所示的规格说明。

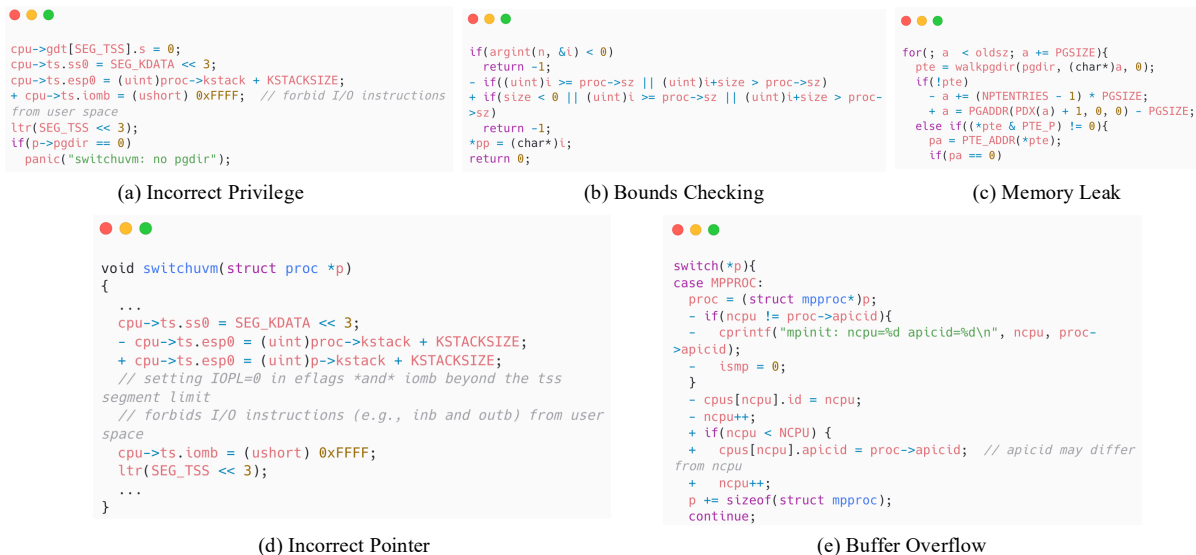


Figure 10: 从 xv6 内核中提取的五种错误类型的示例。

Listing 3: Part of implementation of the system call sys_map_pcpage.

```

1 int sys_map_pcpage(pn_t pt, size_t
2   index, pn_t pcpin, pte_t perm) {
3     if (!is_pcpin_pid(pcpin, pid))
4       return -EACCES;
5 }
6 static int is_pcpin_pid(pn_t pcpin,
7   pid_t pid) {
8   ...
9   return pci_table[pcipages[pcpin].
10     devid] == pid;
11 }

```

Listing 4: Part of incorrect state-machine specification of system call sys_map_pcpage.

```

1 ...
2 old.pcpages[pcpin].devid == old.current
3 ...

```

然而，基于提供的编程模型 Figure 8，old.pcpages[pcpin] 旨在检索一个建模的 PCI-Page 实例，该实例不包括一个建模的 devid 属性。这个错误导致尝试在 Python 中访问一个不存在的类字段，从而导致语法错误。

这种失败可能归因于长上下文，可能导致 LLM 无法追踪所提供的编程模型，而是生成了一个类似于系统调用具体代码实现的规范。然而，必须强调的是，内核的抽象建模与其代码实现有所不同。

语义错误。接下来，我们展示另一个例子来说明语义错误的发生。这个例子涉及在系统调用 sys_map_proc 的状态机规范合成中的错误。该系统调用的功能描述如下：它验证指定的权限，并在权限包含写入访问时拒绝操作。然而，

Listing 5: Part of implementation of the system call sys_map_proc.

```

1 - if (pte_writable(perm)) [correct]
2 + if (!pte_writable(perm)) [bug injected]
3   return -EACCES;
4 static inline bool pte_writable(
5   uintptr_t x)
6 {
7   return x & PTE_W;
8 }

```

代码实现中存在一个注入的错误，如 Listing 5 所示。

在这种情况下，期望 LLM 通过遵循功能描述并生成正确的规范来识别注入的漏洞，如 Listing 6 所示。

Listing 6: Part of correct state-machine specification of system call sys_map_proc.

```

1 ...
2 perm & dt.PTE_W != 0,
3 ...

```

然而，LLM 未能准确解释功能描述，导致生成了错误的规范，如 Listing 7 所示。

Listing 7: Part of incorrect state-machine specification of system call sys_map_proc.

```

1 ...
2 perm & dt.PTE_W == 0,
3 ...

```

这个错误的条件导致内核状态转变为与正确操作系统实现不一致的状态，最终导致一个语义错误。

D 任务示例

Programming Model

Constants available:

```
intremap_state = {
  IR_FREE: z3.BitVecVal(0, 64),
  IR_FORCE_WIDTH: z3.BitVecVal(
    18446744073709551615, 64),
  IR_ACTIVE: z3.BitVecVal(1, 64)}
...
```

Class models:

```
class KernelState(BaseStruct):
  procs = Proc()
  dmapages = DMAPage()
  files = File()
  pci = PCI()
  ...

class File(Struct):
  type = Map(fn_t, file_type_t)
  offset = Map(fn_t, size_t)
  ...

class Proc(Struct):
  state = Map(pid_t, proc_state_t)
  killed = Map(pid_t, bool_t)
  ...
...
```

External functions: Other than the model above, z3 datatypes and methods are included in the model, such as z3.ULT(), z3.UGT(), z3.ULE(), z3.UGE(), etc. ...

...

Verification Assumption

The kernel requires user space to explicitly make resource allocation decisions and sidesteps concurrency issues that arise from I/O devices. It also does not support threads, copy-on-write fork, shared pages, or Unix permissions. Additionally, the kernel runs on a uniprocessor system and does not provide multicore support. ...

System

You will be provided with four sections displayed below, including verification assumption, programming model, a few examples, and a task question. ... Your task

Examples

Example 1: Given a system call `sys_close`.
Functional Description : The `sys_alloc_port` system call is designed to allocate an I/O port to a process, allowing it to perform input/output operations on that port.

...

Code Implementation :

Its corresponding code implementation that may contain bugs is as follows:

```
int sys_alloc_port(uint16_t port)
{
    struct proc *proc;
    if (io_table[port])
        return -EBUSY;
    proc = get_proc(current);
    if (!proc->use_io_bitmap)
        return -EACCES;
    io_table[port] = current;
    ...
    return 0;
}
...
```

Specification : Based on the detailed functional description and the potentially buggy code implementation of the system call `sys_alloc_port` provided above, the state-machine specification of the system call is deduced as follows:

```
def sys_alloc_port(old, port):
    cond = z3.And(
        old.io[port].owner == 0,
        old.procs[old.current].
            use_io_bitmap,
    )
    new = old.copy()
    new.io[port].owner = old.current
    new.procs[old.current].
        nr_ports[port] += 1
    ...
    return cond,util.If(cond, new, old)
```

Example n: ...

Task Question

Now, given the system call `sys_close`.

Functional Description : The `sys_close` system call is responsible for closing a file descriptor associated with a specific process, thereby releasing the resources tied to that descriptor.

...

The system call begins by validating the process identifier to ensure it corresponds to a valid process.

...

Code Implementation : Its corresponding code implementation that may contain bugs is as follows:

```
int sys_close(pid_t pid, int fd)
{
    if (!is_pid_valid(pid))
        return -ESRCH;
    if (!is_fd_valid(fd))
        return -EBADF;
    if (pid != current &&
        !is_proc_state(pid, PROC_ZOMBIE))
        return -EACCES;
    if (get_fd(pid, fd) == 0)
        return -EBADF;
    ...
    return 0;
}
...
```

Specification : Based on the detailed functional description and the potentially buggy code implementation of the system call `sys_close` provided above, the state-machine specification of the system call is deduced as follows:

给定上面的任务提示，期望 LLM 合成如下的系统调用 `sys_close` 的状态机规格：

LLM Output

```
def sys_close(old, pid, fd):
    spec_cond1 = z3.And(
        z3.And(pid > 0, pid < dt.NPROC),
        z3.And(fd >= 0, fd < dt.NOFILE),
        z3.Or(
            pid == old.current,
            old.procs[pid].state ==
                dt.proc_state.PROC_ZOMBIE),
        z3.And(z3.UGT(old.procs[pid]
            .ofile(fd), 0), z3.ULT(
                old.procs[pid].ofile(fd),
                dt.NOFILE))
    )
    new1 = old.copy()
    fn = new1.procs[pid].ofile(fd)
    new1.procs[pid].ofile[fd] =
        z3.BitVecVal(0, dt.fn_t)
    new1.procs[pid].nr_fds[fd] -= 1
    new1.files[fn].refcnt[(pid, fd)]
        -= 1
    spec_cond2 = z3.And(new1.files[fn].
        refcnt() == 0)
    new2 = new1.copy()
    new2.files[fn].type = dt.file_type
        .FD_NONE
    new2.files[fn].value = z3.BitVecVal
        (0, dt.uint64_t)
    new2.files[fn].offset = z3.
        BitVecVal(0, dt.off_t)
    new2.files[fn].omode = z3.
        BitVecVal(0, dt.uint64_t)
    new3 = util.If(spec_cond2, new2
        , new1)
    return spec_cond1, util.If
        (spec_cond1, new3, old)
```