Bridging Kernel Drivers and Virtual Device Models with LLM-Powered Automation

Mingyu Wang, Bin Yu*, Wenjian Lu, Zhi Wang, Kefeng Gao, Cheng Wen, Xu Lu, Cong Tian
Xidian University
byu@xidian.edu.cn

Abstract

Linux kernel device drivers are tightly coupled with hardware, making them difficult to execute and test without physical devices. This heavily limits automated code analysis and vulnerability discovery. While manual modeling is unscalable, Large Language Models (LLMs) offer a new approach to scale virtual device construction across the Linux driver ecosystem. In this paper, we present DevGen[1], an LLM-powered tool that generates QEMU-based virtual devices directly from Linux driver source code. DevGen combines static analysis to gather necessary context, guides the LLM through step-by-step prompting, and uses an automated self-correction loop driven by compilation and execution feedback. To further reduce errors, similar fixes are retrieved from a library of common modeling failures and incorporated into the repair prompt, which supports more targeted corrections in later iterations. The generated devices finally integrate with QEMU and Syzkaller, enabling driver fuzzing without physical hardware. DevGen is evaluated on 50 PCI/PCIe drivers from Linux 6.18 using three mainstream LLMs, and successfully generates usable models for 44 drivers. In these drivers, 24% of them achieve significant improvements in fuzzing coverage, and 7 previously unknown crashes are triggered with 1 CVE assigned. These results demonstrate the practical capability of LLMs to automate complex, system-level code generation tasks.

1 Introduction

The security and stability of operating system kernels are fundamental to the reliability of modern computing systems (Xu et al., 2025; Hu et al., 2025; Yang et al., 2025). Device drivers are particularly important because they directly interact with hardware and implement protocols for specific devices (Kadav and Swift, 2012; Song et al., 2019; Bai et al., 2019, 2021). In practice, many driver execution paths depend on runtime feedback provided by devices (e.g., register values, interrupts, and responses required by particular protocols). Without access to real hardware or interactive devices, numerous paths remain inaccessible (Shameli-Sendi, 2021), posing potential threats to kernel reliability.

Although fuzzing has proven effective for discovering kernel vulnerabilities (e.g., Syzkaller) (Google Inc.; Mallissery and Wu, 2023; Chen et al., 2024; Yan et al., 2025; Fleischer et al., 2025), its effectiveness on device drivers is often constrained by hardware dependencies (Xu et al., 2025; Ma et al., 2022). Real hardware are expensive, difficult to provision, and hard to scale for large-scale driver testing (Peng and Payer, 2020; Song et al., 2019), whereas mainstream emulators provide far fewer device models than the kernel needs. For instance, although the Linux kernel supports over 13,000 PCI devices, popular emulators implement fewer than 130 models (Xu et al., 2025). As a result, drivers often stall during probing or initialization due to missing device feedback, preventing fuzzers from reaching deeper execution paths (Wu et al., 2023). Prior work has explored virtual device construction for driver testing (Ma et al., 2022; Peng and Payer, 2020; Song et al., 2019; Corina et al., 2017; Hetzelt et al., 2021; Wu et al., 2023), but existing approaches still face limitations in automation and scalability: PrIntFuzz (Ma et al., 2022) often produces overly simplified models, USBFuzz (Peng and Payer, 2020) is limited to the USB bus, and VIA (Hetzelt et al., 2021) relies on manually written configuration files that tightly couple device models to target drivers.

With the rapid advancement of Large Language Models (LLMs) in program understanding and code generation, new opportunities have arisen for kernel driver testing (Jiang et al., 2024; Qiu et al., 2025; Tian and Chen, 2025; Deng et al.,

* Corresponding author.
[1] Installable package: GitHub link. Demo video: Google Drive link.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 242–252

2023). However, most existing LLM-based efforts improve test input generation rather than reducing hardware dependence in driver fuzzing. For example, KernelGPT (Yang et al., 2025), SyzForge (Tang et al., 2025), and SyzGPT (Zhang et al., 2025) improve Syzkaller (Google Inc.) through system call refinement, dependency mining, and seed generation, respectively, while ECG (Zhang et al., 2024) extracts input specifications to generate structured test cases. These approaches improve fuzzing inputs, but do not directly address the lack of executable device-side interactions.

In this paper, we present DevGen, an LLM-powered framework for automatically generating virtual device models for kernel driver fuzzing. Instead of fully emulating complex hardware behavior, DevGen synthesizes a minimal interaction state machine in QEMU that provides the exact behaviors required for driver probing and initialization, thereby enabling deeper driver execution during fuzzing. Starting from the driver source code, DevGen leverages LLVM-based static analysis to extract critical hardware-software boundaries, including register definitions and state dependencies. It then utilizes structured prompts and staged templates to guide LLMs in generating the virtual device code. To ensure robustness, the generated models are iteratively refined via an autonomous repair loop driven by compilation and runtime feedback. Furthermore, to provide reusable guidance for LLM-based correction, an active historical case library is constructed by summarizing common modeling failures and successful fixes. The successfully generated device models are finally deployed in QEMU and integrated with Syzkaller to support end-to-end driver fuzzing.

We implement a prototype of DevGen and evaluate it on 50 randomly selected PCI/PCIe drivers (Boo and Lee, 2025) from Linux 6.18 using three mainstream LLMs. Overall, DevGen successfully generates 44 usable virtual devices, with per-LLM modeling success rates ranging from 74% to 78%. With the generated devices, 24% of the evaluated drivers show notable coverage improvements, and fuzzing triggers 7 previously unknown crashes, with 1 CVE assigned. These results demonstrate that DevGen is a practical and scalable tool for hardware-constrained driver testing. Specifically, the features of our proposed DevGen include:

• **Novel LLM-powered framework for virtual device modeling.** We present DevGen, an end-to-end automated tool that generates QEMU-based virtual device models directly from Linux driver source code. By synthesizing minimal interaction state machines, it significantly reduces the reliance on physical hardware for kernel driver fuzzing.

• **Automated generation and iterative self-repair.** DevGen provides an integrated workflow from driver code to executable virtual device models, combining static context extraction, staged LLM-driven generation, syntax checking, feedback-driven repair, and historical case reuse. This pipeline achieves a high degree of automation (up to 68% fully autonomous success) and requires only minor, targeted manual adjustments for the remaining successful cases.

• **Prompt-driven usability and extensibility.** DevGen adopts a modular design that decouples static context extraction from LLM generation. This architecture allows the framework to easily generalize to other driver subsystems, such as USB, platform, or I2C, requiring only minor adaptations to the static analysis targets and the injection of corresponding bus-specific knowledge into the prompt templates. Furthermore, its seamless integration with QEMU and Syzkaller ensures immediate practicality for end-to-end driver fuzzing workflows in real-world scenarios.

• **Practical effectiveness and scalability.** Evaluated on 50 randomly selected PCI/PCIe drivers from Linux 6.18, DevGen successfully synthesizes usable virtual devices for 44 drivers. It demonstrates practical utility by achieving significant fuzzing coverage improvements on 24% of these drivers and uncovering 7 previously unknown crashes (with 1 CVE assigned), showcasing its potential to scale across the vast and complex Linux driver ecosystem.

2 Design of DevGen

Aiming to overcome the hardware dependency limit in kernel driver fuzzing, DevGen provides an automated framework to synthesize QEMU-based virtual device models. Rather than attempting a prohibitively complex full-hardware emulation, DevGen adopts a pragmatic approach: it models the *minimal interaction state machine* required to satisfy the driver's initialization sequences and early I/O interactions (Chen et al., 2025). The successful establishment of this minimal interaction is explicitly confirmed once the target driver fully binds to the virtual device, which is externally verified when the system reports `Kernel driver in use`. As

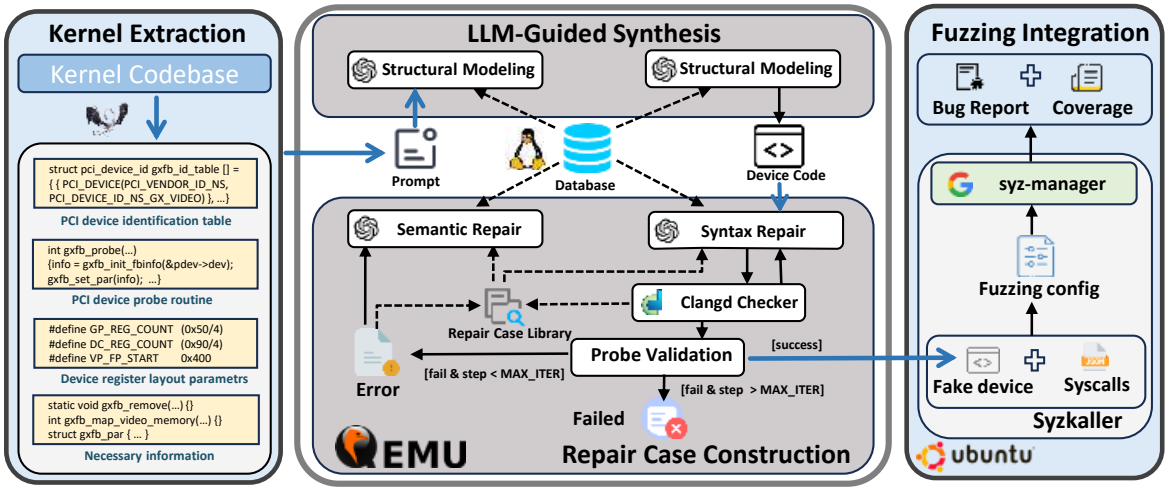

Figure 1: The framework of DevGen

illustrated in Figure 1, the workflow is orchestrated through four tightly coupled stages:

- **Kernel-Aware Static Context Extraction:** Instead of relying on naive source code parsing, this module performs targeted static analysis on the driver source to extract critical hardware-software boundaries. It identifies initialization control flows, PCI configuration layouts, and specific register access patterns. Concurrently, a structured index of the kernel source tree is built to provide precise domain context for the LLM.
- **Staged LLM-Guided Synthesis:** Using the extracted context, DevGen generates the virtual device components in a staged manner via structured prompt templates. This module orchestrates the generation of register definitions, interrupt handlers, and the core device state machine, iteratively aligning the QEMU device model with the expected behavior of the target Linux driver.
- **Repair Case Library Construction:** This module builds a library of common error patterns observed during generation and execution validation. Each case records the specific error, root cause, successful fix, and the precise edit location in the generated code. During subsequent repair iterations, relevant historical cases are dynamically selected from the library to serve as targeted, in-context examples, effectively guiding the LLM toward accurate corrections without relying on external retrieval frameworks.
- **Deployment and Fuzzing Integration:** The refined, successfully compiled device models are dynamically integrated into QEMU and paired with Syzkaller. This module facilitates the automated generation of customized system call sequences, executing hardware-free fuzzing campaigns while continuously monitoring code coverage and detecting kernel panics or crashes.

2.1 Kernel-Aware Static Context Extraction

This module bridges the semantic gap between Linux kernel driver code and QEMU device models by extracting essential hardware-software boundaries to support LLM-guided synthesis. Operating on a kernel configured with `allmodconfig`, we apply LLVM-based static analysis to extract driver structures and hardware interaction patterns. The extracted results are organized into a structured summary that serves as phase-specific inputs to our prompts, providing explicit constraints on IDs, BAR usage, and register accesses.

To synthesize the *minimal interaction state machine*, the extraction specifically targets the information required during device probing and initialization. Using PCI/PCIe drivers as a running example (Boo and Lee, 2025), this includes device matching rules and callback functions defined in `struct pci_driver`, vendor and device identifiers in `pci_device_id`, as well as probe logic for resource allocation, register mapping, and state setup. We also extract macros and constants that describe register layouts.

Crucially, the module constructs a structured index of the kernel source tree, which serves as an external knowledge base during modeling and repair. When the LLM requires extended context (e.g.,

deeply nested helper functions or external macro definitions outside the current snippet), DevGen dynamically retrieves the relevant code locations from this index. This localized retrieval minimizes LLM hallucinations caused by missing context and enables DevGen to capture key driver interaction constraints directly from the source code, without running the driver or requiring physical hardware.

2.2 Staged LLM-Guided Synthesis

This module employs an LLM to generate virtual device models that interact correctly with a target Linux kernel driver. Rather than attempting prohibitively complex full-hardware emulation, DevGen adopts a pragmatic approach: it synthesizes a *minimal interaction state machine*. This ensures the target driver can successfully probe and initialize while significantly reducing the modeling overhead. To mitigate unpredictable LLM outputs and ensure syntactic validity, the generation process is strictly anchored to predefined QEMU device templates and modeling rules. The workflow operates in three progressive stages: structural modeling, behavioral modeling, and autonomous feedback repair. The output of each stage serves as input to the next to systematically produce an executable QEMU device model. The detailed prompt framework is shown in Appendix A.

We define a basic device template that is incrementally completed across these stages (the skeleton is provided in Appendix B). During the structural modeling stage, the LLM generates the device skeleton and static settings, including the PCI configuration space layout, Base Address Register (BAR) types, and address space declarations. In the behavioral modeling stage, it implements the dynamic device responses (e.g., MMIO and PIO handlers) necessary to satisfy the driver's initialization sequence. The workflow transitions to the autonomous feedback repair stage when the initial modeling rules are met or the iteration limit is reached. Here, executability is used as the refinement signal: the generated C code is validated using Clangd (LLVM Project), and compiler diagnostics coupled with runtime execution logs are fed back to the LLM to drive targeted, localized fixes.

To avoid unexpected changes during repair, modifications are strictly restricted to designated template placeholders. To optimally guide the LLM, stage-specific prompts are enriched with driver code snippets, static analysis results, and essential few-shot examples, while any missing context is dynamically retrieved from the indexed source tree. The iterative synthesis concludes when the target driver successfully binds to the virtual device (i.e., driver initialization succeeds) or the iteration limit is reached.

2.3 Repair Case Library Construction

To improve the robustness of the iterative generation process, a repair case library is constructed to capture common error patterns observed during compilation and execution validation. We categorize these failures into 9 distinct error types, whose detailed descriptions are listed in Appendix C.

When a generated device model fails to compile, load in QEMU, or complete driver initialization, the module documents the error signature (e.g., compiler diagnostics and runtime logs), the root cause, and the successful repair action. To facilitate LLM comprehension, each case also records the precise edit location and concise rationale explaining why the fix resolves the issue. This structured format ensures the historical data can be directly reused as high-quality prompt demonstrations.

During subsequent repair iterations, this library functions as a dynamic reference knowledge base. Given compiler or runtime feedback from a failing model, relevant historical cases are dynamically selected and their corresponding repair patterns are incorporated into the prompt. By providing the LLM with these targeted, in-context examples, the system effectively guides the model to identify faulty regions and apply precise corrections. Reusing this historical validation knowledge significantly reduces repetitive trial-and-errors, mitigates LLM hallucinations, and improves the success rate of virtual device modeling.

2.4 Deployment and Fuzzing Integration

This final module seamlessly integrates the synthesized virtual devices into a standard kernel fuzzing workflow, empowering the fuzzer to penetrate deep execution paths that traditionally mandate physical hardware. Once the virtual device is compiled and dynamically loaded into QEMU, the target driver can successfully complete its probing sequence, initialize the *minimal interaction state machine*, and expose its I/O interfaces.

To orchestrate the fuzzing campaign, DevGen leverages Syzkaller's official system call descriptions to construct tailored syscall sequences. This construction is strictly anchored to the driver's source path within the kernel tree and the newly

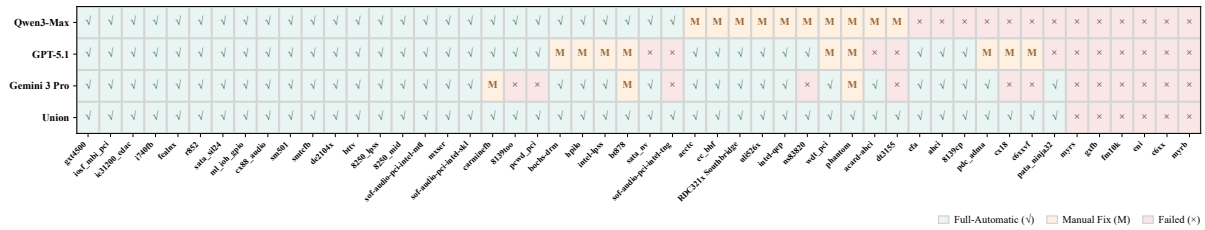

Figure 2: Device modeling results across different LLMs

instantiated `/dev` nodes. Fuzzing is then performed using either the `syzbot` configuration or a configuration generated for the target driver. Because a driver can be fuzzed only when it is compiled into the kernel, this driver configuration is generated by parsing the corresponding `Makefile` and `Kconfig` files to enable the required kernel options.

For a rigorous evaluation, the kernel build and base system call sets are kept strictly identical, isolating the generated virtual device as the sole independent variable. During the fuzzing process, KCOV is enabled to track precise code coverage, while KASAN (The Linux Kernel Organization) is deployed to catch memory corruption bugs. The resulting coverage metrics and crash reports are utilized to quantitatively evaluate whether the virtual devices successfully unlock hidden execution paths and trigger previously unreachable vulnerabilities. Furthermore, crashes and other abnormal events are also collected together with reproducer programs for later analysis and validation.

3 Preliminary Evaluation

Our evaluation is conducted on a personal computer with 8 physical cores (Intel(R) Core(TM) i7-1165G7), 32 GB RAM, and Ubuntu 22.04 LTS. We randomly select 50 PCI/PCIe device drivers from Linux 6.18. During device modeling, we evaluate three mainstream LLMs (Qwen3-Max, GPT-5.1, and Gemini 3 Pro) with a temperature of 0.1 and at most three iterations per modeling stage. During fuzzing, Syzkaller (Google Inc.) runs with four QEMU instances, each using two CPU cores. We use both the `syzbot` configuration (Syzbot) and a customized kernel configuration to compile the kernel for evaluation.

RQ1: How effective are LLMs at generating virtual device models that successfully interact with target drivers?

Under the `allyesconfig` configuration, `DevGen` performs static analysis on the kernel source code, extracts essential device-driver interaction information, and automatically constructs virtual device models for the 50 selected drivers. Figure 2 summarizes the generation results. The symbols ✓ and × indicate whether the generated device interacts successfully with the driver autonomously, while `M` denotes success achieved after a minor manual fix.

Without manual intervention, Gemini 3 Pro achieves the highest autonomous success rate (68%), compared with 54% for Qwen3-Max and 60% for GPT-5.1. After manual fixes, the success rates become comparable: GPT-5.1 reaches 78% (39/50), while Qwen3-Max and Gemini 3 Pro both reach 74% (37/50). Crucially, the `Union` row in Figure 2 illustrates that different LLMs have complementary capabilities. This pushes the overall success rate to an impressive 88% (44 out of 50 drivers), firmly demonstrating the robustness of this collaborative modeling approach.

We classify an intervention as a minor manual fix (`M`) if it is lightweight and needs fewer than three lines of code changes. These fixes address simple configuration mismatches instead of logical flaws. A primary cause is an incorrect `Vendor/Device ID` assignment when a target driver supports multiple distinct hardware identifiers.

Listing 1 provides an example. Here, the LLM synthesized the correct device state machine but chose a generic `Vendor ID (0x104c)` that failed the driver constraints. An analyst checked the `pci_device_id` table and updated the macro to `0x14dc`—a trivial one-line adjustment. Since these minor fixes do not alter the underlying hardware interaction logic, we classify them as successes.

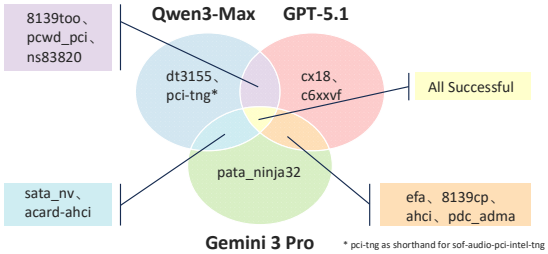

Figure 3: Overlap of device models generated by LLMs

Model	Overall Token Usage			Stage-wise Overhead: Tokens (Calls)				Total Calls
	Total	Prompt	Completion	Step 1	Step 2	Step 3	Step 4	
Gemini 3	159,471	101,336	58,134	67,948 (2.28)	52,693 (1.44)	25,326 (0.72)	13,504 (0.54)	4.98
GPT-5.1	170,676	127,182	43,493	62,408 (2.58)	70,467 (2.54)	28,893 (1.20)	8,907 (0.60)	6.92
Qwen 3	130,434	100,163	30,271	33,689 (1.66)	32,599 (1.28)	44,336 (1.96)	19,810 (1.00)	5.90

Table 1: Average token usage and API calls per driver modeling task across the evaluated LLMs

```
- #define PCIBASE_VENDOR_ID 0x104c
+ #define PCIBASE_VENDOR_ID 0x14dc
  #define PCIBASE_DEVICE_ID 0x000c
  ...
  static void pcibase_realize(PCIDevice *pdev,
    Error **errp) {
      ...
      pci_set_word(pci_conf + PCI_VENDOR_ID,
    PCIBASE_VENDOR_ID);
      pci_set_word(pci_conf + PCI_DEVICE_ID,
    PCIBASE_DEVICE_ID);
      ...
  }
```

Listing 1: Example of a minor manual fix to correct a mismatched Vendor ID.

Figure 3 visualizes the overall modeling outcomes under this inclusive counting rule: 44 drivers successfully complete initialization with generated virtual devices, 30 of which succeed consistently across all three LLMs (as illustrated by the central intersection in Figure 3), while 6 remain unsuccessful. These remaining 6 hard failures (12%) stem from deep firmware dependencies, strict timing constraints, and a large volume of required interaction code, making targeted repair difficult. Overall, the results firmly validate that DevGen can effectively generate usable virtual devices to support hardware-free kernel driver fuzzing.

Beyond modeling effectiveness, we also assess the resource efficiency of this synthesis process. Table 1 quantifies the average inference cost—measured in token volume and API calls—per driver modeling task. Rather than uniformly scaling, each LLM exhibits a distinct resource footprint. Gemini 3 achieves the highest interaction efficiency, requiring the fewest API calls (4.98). Conversely, GPT-5.1 incurs the highest total token overhead, driven primarily by the dense context demanded during the behavioral modeling stage (Step 2). Notably, Qwen 3 maintains a highly optimized output profile, consuming the fewest completion tokens (30,271) across all tasks.

RQ2: How well do the LLM-generated devices improve code coverage during fuzzing?

To assess whether DevGen's virtual devices unlock deeper driver code paths, we measure Syzkaller basic block coverage before and after device integration. Fuzzing relies on official Syzkaller descriptions (Syzlang). We utilize the default `syzbot` configuration whenever the virtual device successfully initializes under it; otherwise, we generate a custom kernel configuration by parsing the driver's `Makefile` and `Kconfig` to enable necessary dependencies. As a strict baseline, we fuzz the identical kernel build and syscall set without any virtual devices. We conduct three independent one-hour runs per driver, recording the average coverage achieved during the stable phase.

Table 2 highlights representative coverage shifts, where ✓ and × denote successful and failed interactions, respectively, and △ indicates successful initialization with negligible coverage gain. The `Syz+Device` column reports average coverage exclusively across successful (✓) runs. We categorize these results into three groups. The top group (soft blue) demonstrates substantial improvements, confirming the virtual devices unblock previously unreachable paths. The middle group (soft orange) successfully initializes and exposes `/dev` nodes, yielding moderate gains. This plateau is expected: because DevGen intentionally models a *minimal interaction state machine* rather than comprehensive hardware behavior, deep device-specific logic remains unexercised. Finally, the bottom group (light gray) represents drivers with zero baseline coverage; integrating our models effectively bypasses the initial probe-failure roadblock, enabling fundamental fuzzing exploration. Full evaluation results are available on GitHub.

Overall, these findings confirm that LLM-synthesized virtual devices effectively mitigate hardware dependencies, enabling fuzzers to achieve significantly deeper execution states.

RQ3: How effective are the LLM-modeled virtual devices at discovering real-world driver vulnerabilities?

In total, fuzzing with the successfully modeled virtual devices triggered 7 previously unknown crashes. To rigorously ensure these are genuine

Driver	LLM			Syz	Syz+Device
	Qwen	GPT-5.1	Gemini		
gxt4500	✓	✓	✓	3665	7945
smtcfb	✓	✓	✓	3925	8392
i740fb	✓	✓	✓	3694	8528
carminefb	△	△	✓	3907	9220
intel-lpss	✓	△	△	5544	10135
bt878	✓	✓	△	3442	5275
bttv	✓	✓	✓	3655	5415
intel-qep	✓	✓	✓	5980	6423
dt3155	✓	×	×	3877	4515
phantom	✓	✓	✓	0	2276
wdt_pci	✓	✓	✓	0	2576
pcwd_pci	✓	✓	×	0	2720

Table 2: Coverage changes with generated virtual devices

Crash Signature	Device	LLM			Status
		Qwen	GPT-5.1	Gemini	
WARNING in drm_gem_release	bochs-drm	✓	✓	×	CVE-2026-23149
WARNING in idr_alloc	intel-qep	×	✓	✓	CVE-2026-23149
KASAN: UAF read in adf_dev_up	c6xxvf	✓	×	×	Confirmed
general protection fault in h5_recv	8250_lpss	✓	×	✓	Confirmed
INFO: rcu detected stall in sys_mmap	pci-tng	✓	×	×	New
INFO: rcu detected stall in do_idle	wdt_pci	×	×	✓	New
INFO: task hung in i2c_smbus_xfer	bttv	✓	✓	✓	New

Table 3: New crashes detected by fuzzing with DevGen devices

kernel vulnerabilities rather than artifacts of incorrect device emulation, we manually triaged each crash using minimized reproducers. Crucially, a vulnerability is classified as a valid bug only after we submit patches or bug reports to the official Linux kernel maintainers and receive their explicit confirmation[2]. Currently, two confirmed bugs have been patched upstream and share a CVE identifier.

Table 3 summarizes these discoveries. The Device column lists the virtual device required to expose the crash, while the ✓ and × symbols indicate whether a device generated by a specific LLM successfully triggered it. The Status column reports the current validation state: Confirmed means kernel maintainers have officially acknowledged the bug, whereas New indicates it is currently under official review. Notably, although the first two crashes share CVE-2026-23149 and both manifest as warnings, they follow different execution paths with distinct call stacks (both were ultimately fixed by the same patch). Furthermore, the crash-triggering capability varies across LLMs, reflecting subtle differences in the fidelity and behavior of the synthesized state machines.

Case Study: CVE-2026-23149 triggered by the `intel-qep` virtual device.

Malicious user-space requests can reach the GEM handle management path through the DRM ioctl interface. As shown in Listing 2, the function drm_gem_change_handle_ioctl() allows a process to rename an existing GEM handle. However, the kernel does not validate the value provided by the user in args->new_handle before passing it to idr_alloc(): it uses args->new_handle as the start index and args->new_handle+1 as the end index of the allocation range.

```
int drm_gem_change_handle_ioctl(struct drm_device *dev, void
      *data, struct drm_file *file_priv){
    struct drm_gem_change_handle *args = data;
    struct drm_gem_object *obj;
    obj = drm_gem_object_lookup(file_priv, args->handle);...
    ret = idr_alloc(&file_priv->object_idr, obj,
                    args->new_handle, args->new_handle+1,
                    GFP_NOWAIT);...
}
```

Listing 2: Integer overflow vulnerability in drm_gem_change_handle_ioctl()

If an attacker sets new_handle to INT_MAX, the computation of args->new_handle+1 overflows, turning the intended range into an invalid interval [INT_MAX, INT_MIN). This violates assumptions inside the idr allocator, triggers the warning, and may lead to inconsistent resource management.

In this case, DevGen generates a virtual intel-qep device that matches the target DRM driver, allowing the driver to complete initialization and expose the GEM handle path. With this prerequisite satisfied, fuzzing can issue abnormal ioctl requests that exercise idr_alloc(), showing how generated virtual devices enable deeper driver testing without physical hardware and help uncover subtle resource management bugs.

4 Conclusion and Future Work

This paper presents DevGen, an LLM-powered tool that reduces hardware dependence in kernel driver fuzzing through the automated generation of virtual device models. By targeting the interfaces required for driver probing and early interactions, and iteratively refining the generated models using compilation and runtime feedback, DevGen enables practical fuzzing without physical hardware. Our evaluation shows its effectiveness and scalability for driver testing under strict hardware constraints.

In future work, we plan to enhance our modeling specifications as formal constraints to guide LLM generation, covering a broader range of complex hardware behaviors. Additionally, we will leverage the accumulated cases to explore fine-tuning open-source LLMs to further reduce hallucinations.

[2]The communication logs with the kernel maintainers can be transparently verified via the Linux kernel mailing list archive: https://lore.kernel.org/all/?q=wangzhi.

References

Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *Proceedings of the 28th USENIX Annual Technical Conference (USENIX ATC)*, pages 255–268.

Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, pages 1629–1645.

Kyungwook Boo and Byoungyoung Lee. 2025. Finding Device Driver Bugs with Fuzzing PCIe Configuration Input. *IEEE Access*, pages 150216 – 150227.

Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2025. Seal: Towards Diverse Specification Inference for Linux Interfaces from Security Patches. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*, pages 1246–1262.

Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP)*, pages 4661–4677.

Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 24th ACM SIGSAC conference on Computer and Communications Security (CCS)*, pages 2123–2138.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 423–435.

Marius Fleischer, Harrison Green, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2025. SyzGrapher: Resource-Centric Graph-Based Kernel Fuzzing. *International Symposium on Research in Attacks, Intrusions and Defense (RAID)*.

Google Inc. Syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*, pages 273–284.

Kun Hu, Qicai Chen, Zilong Lu, Wenzhuo Zhang, Bihuan Chen, You Lu, Haowen Jiang, Bingkun Sun, Xin Peng, and Wenyun Zhao. 2025. A Survey of Fuzzing Open-Source Operating Systems. *arXiv preprint arXiv:2502.13163*.

Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and 1 others. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE-Companion)*, pages 492–496.

Asim Kadav and Michael M Swift. 2012. Understanding Modern Device Drivers. *ACM SIGPLAN Notices*, 47(4):87–98.

LLVM Project. What is clangd? https://clangd.llvm.org/.

Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 404–416.

Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the Fuzzing Methods: A Comprehensive Survey. *ACM Computing Surveys*, 56(3):71:1–71:38.

Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 2559–2575.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, and 1 others. 2025. LoCoBench-Agent: An Interactive Benchmark for LLM Agents in Long-Context Software Engineering. *arXiv preprint arXiv:2511.13998*.

Alireza Shameli-Sendi. 2021. Understanding Linux Kernel Vulnerabilities. *Journal of Computer Virology and Hacking Techniques*, 17(4):265–278.

Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the 26th Annual Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15.

Syzbot. Syzbot. https://syzkaller.appspot.com/upstream/.

Syzlang. Syscall_descriptions_syntax. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

ZhiZhuo Tang, Jian Lin, Weiyu Dong, Hang Ma, and Tieming Liu. 2025. SyzForge: An Automated System Call Specification Generation Process for Efficient Kernel Fuzzing. In *Proceedings of the 22nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 118–139.

The Linux Kernel Organization. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.13/dev-tools/kasan.html. [n.d.].

Zhao Tian and Junjie Chen. 2025. Aligning Requirement for Large Language Model's Code Generation. *arXiv preprint arXiv:2509.01313*.

Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP)*, pages 3246–3261.

Jiacheng Xu, Shihao Jiang, He Sun, Qinying Wang, Mingming Zhang, Xiang Li, Kaiwen Shen, Charles Zhang, Shouling Ji, Peng Cheng, and 1 others. 2025. A Survey of Operating System Kernel Fuzzing. *ACM Transactions on Software Engineering and Methodology*.

Qian Yan, Minhuan Huang, Huayang Cao, and Shuaibing Lu. 2025. SoK: From Systematization to Best Practices in Fuzz Driver Generation. In *Proceedings of the 30th Australasian Conference on Information Security and Privacy (ACISP)*, pages 348–368.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, pages 560–573.

Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):4238–4249.

Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. 2025. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):848–870.

A Prompt Framework Details

This appendix details the prompt design underlying our LLM-powered device modeling workflow, which comprises a fixed system role, global generation constraints, and a four-phase procedure. Figure 4 illustrates the prompt framework driving DevGen. The workflow is inherently iterative: each phase propagates the generated context from the previous step and integrates it with external execution feedback, such as compiler diagnostics or runtime logs. To ensure consistency and facilitate automated parsing across iterations, all phases strictly enforce structured generation, requiring the LLM to produce JSON-only outputs.

System: You are an expert in Linux PCI drivers and QEMU device modeling (**QEMU 8.2.10 only**). Generate a compilable QEMU PCI device C file strictly from the provided driver source, and reply with **JSON only**.

Global rules: Generate the device model strictly from the provided Linux driver source (and any given register documentation); do not introduce behavior that is not explicitly supported by the inputs. Always select the first entry in `pci_device_id` and never modify `TYPE_PCIBASE_DEVICE`. Modify only the designated placeholder blocks and do not change non-placeholder template code. If required driver-specific symbols are missing, list them in `needed_sources`. Output **JSON only**.

User (Phase 1: Structure) **Input:** driver headers/macros/PCI IDs + `PCI_TEMPLATE`. **Target:** fill placeholders using only the provided data (no behavior).	**Assistant** `{"code":"<structure-only C file>", "needed_sources":[...]}`
User (Phase 2: Implementation) **Input:** Phase-1 code + driver implementation (`Relevant_Source`). **Target:** implement only behaviors exercised by the driver, keeping the Phase-1 structure unchanged.	**Assistant** `{"code":"<full functional C file>", "needed_sources":[...]}`
User (Phase 3: Repair) **Input:** `QEMU_CODE` + `SYNTAX_ERROR_MESSAGE` + driver snippets. **Target:** fix compile errors with minimal edits; keep runtime behavior unchanged.	**Assistant** `{"code":"<compilable file or empty>", "needed_sources":[...]}`
User (Phase 4: Debug & Update) **Input:** `QEMU_CODE` + `ERR_MSG` + driver snippets + repair cases. **Target:** apply minimal fixes to reach `"Kernel driver in use"` based on logs and driver visible behavior.	**Assistant** `{"code":"<updated file or empty>", "needed_sources":[...], "changes":[...]}`

Figure 4: Prompt framework used by DevGen

Specifically, the modeling workflow is divided into four sequential phases:

- **Phase 1: Skeleton Construction.** DevGen builds a basic device skeleton by filling a QEMU template with static information extracted from the driver. It also removes any unnecessary code blocks to keep the LLM focused.
- **Phase 2: Functional Implementation.** The LLM turns this skeleton into a working device by generating the specific hardware behaviors (such as MMIO or PIO responses) that the driver needs during initialization.
- **Phase 3: Syntax and Compile-time Repair.** This validation step fixes syntax and compile-time errors, such as missing headers, incorrect types, or wrong QEMU API usage.
- **Phase 4: Debug and Update.** This step uses runtime logs to fix interaction mismatches between the driver and the device, iteratively refining the code until the driver successfully binds.

During the repair steps (Phases 3 and 4), DevGen retrieves relevant examples from the case library and includes them in the prompt, helping the LLM learn from past successful fixes.

B QEMU PCI Template Overview

Listing 3 illustrates the core structure of our QEMU PCI template. The template defines a fixed device skeleton, including type declarations, state definitions, BAR registrations, and hooks, while exposing placeholder regions denoted by `#placeholder#`. These placeholders cover static driver information and behavior logic, such as identifiers, register fields, MMIO/PIO handlers, initialization code, interrupts, and reset. Optional components, such as interrupts or DMA, can be removed when they are not required by the target driver.

```
#include "hw/pci/pci.h" ...
  #HeadFile#

#define TYPE_PCIBASE_DEVICE "DRIVER_NAME_pci"
  #Related_Config_Info#

struct PCIBaseState {
  PCIDevice parent_obj; MemoryRegion bar_regions[6];
  BARInfo bar_info[6]; int num_bars;
  bool has_msi, has_msix;
  #Reg_Stru#; #Status_Stru#;
  #Interrupt_Stru#; #DMA_Info_Stru#;
};

pcibase_register_bar(...){ ... }

pcibase_mmio_read(...){ #MMIO_Read_Func# ... }
pcibase_mmio_write(...){ #MMIO_Write_Func# ... }
pcibase_pio_read(...){ #PIO_Read_Func# ... }
pcibase_pio_write(...){ #PIO_Write_Func# ... }

pcibase_realize(...){
  pci_set_word(...,#VENDOR_ID#);
  pci_set_word(...,#DEVICE_ID#);
  #Field_Init_Real# ...
  #MSI_OR_MSIX_INIT#; #DMA_Func# ... }
pcibase_reset(...){ #Reset_Func# ... }
...
```

Listing 3: Core structure of the QEMU PCI template

During the staged generation and repair process, DevGen selectively targets these placeholders. Specifically, it (i) populates the placeholders with static information extracted from the driver, (ii) generates the minimal interaction behaviors necessary for driver initialization, and (iii) iteratively refines these regions based on compiler diagnostics and runtime logs. Throughout this workflow, the fixed non-placeholder code remains strictly unchanged to ensure structural integrity.

C Repair Case Library

Table 4 presents a summary of the repair cases. The Error Cause column describes each error type, while the Fix Solution column summarizes the corresponding fix. In total, there are 9 categories and 15 repair cases.

Error Type	Count	Error Cause	Fix Solution
already_exists	1	Device registration name conflict causes an error.	Rename the device type definition macro and remove unused macro definitions.
assertion_failed	2	Assertion failure occurs because the PCI BAR memory size is not a power of two.	Round the size up to the nearest power of two.
invalid_signature	1	Driver probe fails due to reading an invalid signature.	Write the correct signature value at the specified offset in PCI configuration space.
ioremap_failed	3	Driver probe fails because the BAR configuration did not comply with hardware specifications.	Adjust the number, index, and size of the BARs.
no_MMIO_resource	1	Failure is caused by missing MMIO resources and incorrect BAR configuration.	Configure BARs to support both PIO and MMIO, implementing PIO read/write handlers, and correcting register offsets.
no_such_device	2	Driver binding fails due to a device ID mismatch.	Change the Device ID to a value supported by the driver.
not_found	3	Driver loading fails due to incorrect PCI vendor ID, device ID, or subsystem ID configuration.	Correct the relevant ID values.
table & pba overlap	1	MSI-X table and PBA offset conflict.	Use a valid BAR and properly align offsets.
already_registered	1	Conflict is caused by repeated device type registration checks.	Remove redundant validation logic to prevent duplicate registration errors.

Table 4: Main content of the repair case library

Listing 4 presents an error repair case for a device modeled based on the AHCI driver. Each repair case is represented as a JSON file containing five key-value pairs:

- **`err_info`:** includes error information, such as messages generated during QEMU startup or logs obtained from `dmesg`.
- **`pre_code`:** refers to the buggy code snippet before any modifications are made.
- **`post_code`:** denotes the corrected code snippet after the modification is applied.
- **`modified_location`:** provides a summary of the specific code modifications
- **`modify_details`:** describes the complete repair process, including problem identification, root cause analysis, and the implemented fix.

The following illustrates the repair process for a device model generated from the AHCI driver. During diagnostic execution, the AHCI driver probe fails and returns error code `-12`. By tracing the return paths associated with this error in the AHCI driver source, the failure can be localized to the call to `pcim_iomap()` (Listing 5). In this call, the parameter `ahci_pci_bar` is set to `AHCI_PCI_BAR_STANDARD`, whose enumerated value is 5, indicating that the driver attempts to access PCI BAR 5.

Further inspection of the QEMU device implementation (Listing 6) shows that, although the code intends to configure BAR 5, it incorrectly uses 5 as the index of `bar_info`. Since `num_bars` is set to 1, QEMU reads only `bar_info[0]` in the BAR initialization and registration loop. As a result, the configuration stored in `bar_info[5]` is ignored, and BAR 5 is never registered. The correct fix is to change the index of `bar_info` from 5 to 0. This allows QEMU to read the BAR configuration during initialization and register BAR 5 correctly, thereby resolving the driver loading failure. In `DevGen`, such constraints are fed back to the repair stage as prompt context, so that only the relevant placeholders are revised. The fix pattern can then be recorded in the repair library for similar failures.

Other repair cases follow a similar process: using error logs to localize faults in the driver source, identifying the corresponding device functions, and then adjusting the QEMU emulation accordingly.

```
"err_info": "ahci 0000:00:04.0: probe with driver ahci
             failed with error -12",
"pre_code": "s->bar_info[5].index = 5;
             s->bar_info[5].type = BAR_TYPE_MMIO;
             s->bar_info[5].size = 4096;
             s->bar_info[5].name = "ahci-mmio";",
"post_code": "s->bar_info[0].index = 5;
              s->bar_info[0].type = BAR_TYPE_MMIO;
              s->bar_info[0].size = 4096;
              s->bar_info[0].name = "ahci-mmio";",
"modified_location": "change bar_info[5] to bar_info[0]",
"modify_details": "1. Failure Symptom: When executing the
                   diagnostic command...
                   2. Troubleshooting Process: ..."
```

Listing 4: AHCI BAR configuration repair case

```
static int ahci_init_one(struct pci_dev *pdev, const struct
      pci_device_id *ent) { ...
    int ahci_pci_bar = AHCI_PCI_BAR_STANDARD;
    ...
    hpriv->mmio = pcim_iomap(pdev, ahci_pci_bar, 0);
    if (!hpriv->mmio)
        return -ENOMEM; ...
}
```

Listing 5: AHCI driver source code

```
/* Set up BARs */
s->bar_info[5].index = 5;
s->bar_info[5].type = BAR_TYPE_MMIO;
s->bar_info[5].size = 4096;
s->bar_info[5].name = "ahci-mmio";
s->num_bars = 1;
...
/* register BARs */
for (int i = 0; i < s->num_bars; i++) {
    pcibase_register_bar(pdev, s, &s->bar_info[i], errp);
}
```

Listing 6: QEMU device code generated based on the AHCI driver