SyzParam: Incorporating Runtime Parameters into Kernel Driver Fuzzing

Yue Sun
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
sunyue163@mails.ucas.ac.cn

Yan Kang
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
kangyan@ict.ac.cn

Chenggang Wu*
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
wucg@ict.ac.cn

Kangjie Lu
University of Minnesota
Minneapolis, USA
kjlu@umn.edu

Jiming Wang
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
wangjiming21@mails.ucas.ac.cn

Xingwei Li
Information Engineering University
Zhengzhou, China
xrivendell7@outlook.com

Yuhao Hu
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
huyuhao19f@ict.ac.cn

Jikai Ren
SKLP, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
renjikai22@mails.ucas.ac.cn

Yuanming Lai
SKLP, Institute of Computing Technology, CAS
Beijing, China
laiyuanming19b@ict.ac.cn

Mengyao Xie
SKLP, Institute of Computing Technology, CAS
Beijing, China
xiemengyao@ict.ac.cn

Zhe Wang
SKLP, Institute of Computing Technology, CAS
Beijing, China
wangzhe12@ict.ac.cn

Abstract

Under the monolithic architecture of the Linux kernel, all its components operate within the same address space. Notably, device drivers constitute over half of the kernel codebase yet are particularly prone to bugs. Therefore, exploring vulnerabilities in drivers is critical for ensuring kernel security. Extensive research has been done to fuzz kernel drivers through system calls and hardware interrupts. Through a comprehensive study of the Linux Kernel Device Model, we identified that the execution of device drivers is also influenced by runtime parameters, including device attributes and kernel module parameters. Our analysis reveals that large portions of the uncovered code are masked by these parameters, which are exposed to the userspace through a specialized virtual file system known as `sysfs`. Furthermore, adjacent devices interconnected within the same device tree also impact drivers' behavior.

This paper introduces a novel fuzzing framework, SyzParam, which incorporates runtime parameters into the fuzzing process. Achieving this objective requires addressing several key challenges, including valid value extraction, inter-device relation construction, and fuzz engine integration. By inspecting the data structures and functions associated with the LKDM, our tool can extract runtime parameters across various drivers through static analysis. Additionally, SyzParam collects inter-device relations and identifies associations between runtime parameters and drivers. Furthermore, SyzParam proposes a novel mutation strategy, which leverages these relations and prioritizes parameter modification during related driver execution. Our evaluation demonstrates that SyzParam outperforms existing fuzzing works in driver code coverage and bug-detection capabilities. To date, we have identified 30 unique bugs in the latest kernel upstreams, including 10 CVEs.

CCS Concepts

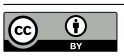
• **Security and privacy → Operating systems security**.

Keywords

Kernel Fuzzing, Operating System Security, Vulnerability Detection

ACM Reference Format:
Yue Sun, Yan Kang, Chenggang Wu, Kangjie Lu, Jiming Wang, Xingwei Li, Yuhao Hu, Jikai Ren, Yuanming Lai, Mengyao Xie, and Zhe Wang. 2025. SyzParam: Incorporating Runtime Parameters into Kernel Driver Fuzzing. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744838

*Corresponding author.



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744838

1 Introduction

As one of the most popular operating systems, Linux runs on servers and mobile devices worldwide. Due to its monolithic architecture, all drivers run in the same address space, which makes kernel security very vulnerable—a flaw in the driver could be exploited to compromise the whole kernel. Researchers have proposed various techniques to identify vulnerabilities in device drivers [1, 3, 24, 29, 42]. According to statistics, over half of the kernel code comes from device drivers [19], and 27%-54% of Linux kernel CVE reports in the recent 5 years are related to drivers according to previous research [38]. Therefore, discovering vulnerabilities in drivers is crucial to improving kernel security.

Fuzzing is one of the most popular methods to find vulnerabilities in software because of its effectiveness and scalability. In 2017, Google announced the release of Syzkaller [10], which has been used to fuzz kernels with different configurations. By the time of writing, Syzkaller has uncovered more than 7,500 vulnerabilities across different kernel versions. Recently, researchers have made numerous attempts to explore more vulnerabilities in Linux drivers, which can be categorized as hardware emulation and syscall enhancement. For hardware emulation, some literature has simulated absent hardware devices to bind and test drivers [38, 45], while other studies have expanded the scope by introducing hardware interrupts as additional input space [13, 25, 33]. For syscall enhancement, some researchers aim to automatically generate syscall descriptions for drivers utilizing kernel source code [6, 11, 35], and others focus on relations between syscalls [2, 37, 41]. Additionally, some research integrates domain-specific knowledge into fuzzing [20, 28, 44]. The initiatives above have all produced noteworthy outcomes, revealing kernel driver vulnerabilities from various angles.

In addition to the input spaces discussed in previous works, we identify an additional factor that can significantly influence driver code execution. Under the current driver model, the Linux Kernel Device Model (LKDM), drivers expose runtime parameters to users through a specialized virtual file system, `sysfs`. These parameters can be broadly categorized into 1)*kernel module parameters* associated with drivers and 2)*device attributes* linked to device instances. Through static analysis, we quantified the prevalence and impact of these parameters. Our findings show that the kernel contains over 3,000 types of runtime parameters, influencing more than 8,000 conditional statements (e.g., `if` and `switch`), directly affecting over 50,000 basic blocks, and indirectly influencing over 100,000 basic blocks through function calls. Compared to the basic blocks affected by syscall arguments, these parameters contribute 29.5% - 33.3% to code coverage. However, existing kernel driver fuzzing approaches overlook the role of runtime parameters in driver code execution, failing to uncover these functionalities.

To gain a deeper understanding of runtime parameters, we carefully inspected historical bugs associated with runtime parameters. We observed that these parameters not only impact the execution of the driver to which they belong but can also impact other device drivers. By delving into the underlying data structures, we found that all devices are organized through the device tree, attaching to a parent device or bus during registration. Consequently, modifications to one device will affect others through the device tree. Additionally, we identified that the timing of parameter modifications plays a critical role in triggering these bugs.

Given the findings above, incorporating runtime parameters in driver fuzzing is non-trivial. It faces three challenges: (1) *How to pair runtime parameters and their corresponding values to pass input validation?* Although parameters can be modified by writing files under the `sysfs`, the kernel has strict logic to check and filter invalid inputs. Generating random values to write files will not change the value of the targeted parameters, thus failing to explore the new functionalities of drivers. (2) *How to determine relations between runtime parameters and drivers?* Modifying the parameters alone cannot trigger new functionalities, but should be combined with the execution of the driver. However, there are thousands kinds of parameters. Randomly combining them with existing syscall descriptions will exponentially increase the search space, reducing the fuzzing efficiency significantly. (3) *How to integrate the runtime parameters into the fuzz engine and cooperate with other components in fuzzing?* Runtime parameter modification will have sustainable effects on the kernel driver execution, which will affect components in the fuzzing loop, leading to poor reproducibility.

To address the aforementioned challenges, we propose a two-stage driver fuzzing framework, SyzParam, which integrates runtime parameters into the driver fuzzing. Specifically, we design three targeted tasks to address these challenges systematically: *First*, to enable the fuzzing engine to modify parameters dynamically, we implement a value collection task that identifies parameter filenames and determines values that can successfully pass input validation; *Second*, to avoid the inefficiencies caused by random combinations of parameter modifications and driver executions, we perform a relation identification task. This task narrows the testing scope by focusing on inter-device connections and the relation between parameters and drivers; *Third*, to ensure compatibility with functionalities in the fuzz engine, we propose new mutation strategies and leverage existing features to seamlessly integrate parameters into the fuzzing engine, maintaining orthogonality with other components in the fuzzing process.

We implemented SyzParam and evaluated it on Linux v6.7. Overall, SyzParam can recover 1,243 types of device attributes, and 694 unique kernel module parameters. Compared with the fuzzing performance with Syzkaller, SyzDescribe, and Syzgen++ on 8 widely used device drivers on different buses, SyzParam achieves the highest edge coverage on 7 out of 8 drivers, with 32.57% improvement in edge coverage compared to Syzkaller. Further study combines our work with existing work, and achieves the highest code coverage among all, proving that our coverage improvement is complementary to existing works. We also did an ablation study on vulnerability discovery capabilities and achieved a 34.5% improvement in the number of unique crashes in a 120-hour trial in four rounds. SyzParam discovered 30 previously unknown bugs in total, 20 of which have been confirmed by developers. Among all vulnerabilities, 21 were discovered because of modifying runtime parameters, consisting of 70% of newly discovered bugs.

In this paper, we make the following contributions:

- **New findings on driver fuzzing.** We identify a long-neglected factor, runtime parameters, that can significantly impact driver code execution. Moreover, we conducted a thorough study of

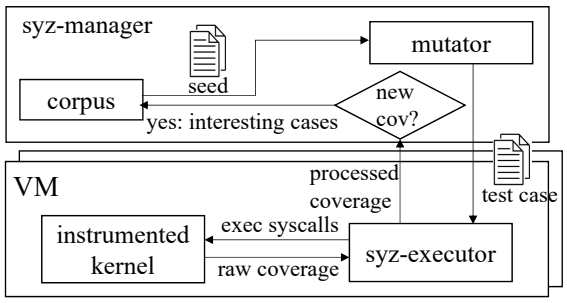


Fig. 1: The workflow of Syzkaller. The mutator in syz-manager is responsible for generating and mutating test cases, while syz-executor is responsible for executing syscalls and collecting coverage feedback.

```
1  resource fd_fb[fd]
2  // The return value of openat() is a file descriptor, marked as resource in syzlang
3  openat$fb(fd const, file ptr[in, string["/dev/fb0"]], flags flags, mode const[0]) fd_fb
4  // Different syscall types on same resource
5  write$fb(fd fd_fb, data ptr[in, array[int8]], len bytesize[data])
6  read$fb(fd fd_fb, data ptr[out, array[int8]], len bytesize[data])
7  // Same syscall with different arguments
8  ioctl$FBIOCMD1(fd fd_fb, cmd const[FBIOCMD1], arg ptr[out, array[int8, FB_SIZE]])
9  ioctl$FBIOCMD2(fd fd_fb, cmd const[FBIOCMD2], arg ptr[in, fb_cmap_user])
10 // User space structure used in syscalls
11 type fb_cmap_user {
12         start int32
13         len len[red, int32]...
14 }
```

Fig. 2: An example of *syzlang* provided by Syzkaller. *Syzlangs* are declarative descriptions of syscall interfaces to manipulate programs, which will be used to generate, mutate, execute, minimize, serialize, and deserialize programs.

the LKDM and pointed out that the inter-device relationship is the key factor to trigger the bug.

- **A novel driver fuzzing approach.** We propose a new driver fuzzing approach based on the findings above. We extend the ability of current kernel fuzzing tools by incorporating runtime parameters into driver fuzzing and utilizing the device relation tree to guide mutation strategies. We will open source our solution to facilitate future research.
- **The implementation and evaluation of SyzParam.** We implement a fuzzing framework prototype of SyzParam and evaluate it comprehensively. Compared to related works, SyzParam could achieve higher code coverage among tested drivers. We have discovered 30 previously unknown vulnerabilities, of which 21 were discovered because of modifying parameters. So far, 20 bugs are confirmed and 14 are patched into the mainline kernel, and 10 CVEs have been assigned.

2 Background

In this section, we will first introduce the workflow of current kernel fuzzers and the recent progress in kernel driver fuzzing. Then, we will illustrate the current driver model in Linux, along with the runtime parameters it provides. An example will be given to demonstrate how runtime parameters influence the driver execution through a special virtual filesystem called *sysfs*.

2.1 Linux Kernel Fuzzing

Recently, many works have explored the Linux kernel vulnerabilities using the fuzzing technique [18, 30, 39, 44]. Most were built on the popular kernel fuzzing framework Syzkaller [10]. Syzkaller is a coverage-guided kernel fuzzer that uses system calls as inputs. The workflow of Syzkaller is illustrated in Fig. 1. Syzkaller initializes multiple virtual machine (VM) instances, which are managed by a component called the *syz-manager*, and executes a program named *syz-executor* within each VM. The *syz-manager* randomly selects a seed from the corpus, mutates it using predefined strategies, and generates test cases. The *syz-executor* subsequently invokes the system calls specified in the test cases and records their code coverage after execution. When new coverage is detected, test cases that trigger the new coverage (referred to as "interesting cases") are minimized and incorporated into the corpus for future mutation.

To generate complex system call sequences capable of triggering the execution of deep kernel code, Syzkaller introduced a syscall description language with type information, referred to as *syzlang* [9]. *Syzlang* is a template that depicts the syscall types and their arguments. As shown in Fig. 2, *syzlang* supports various syscall types and argument types, including complex structure types (e.g., Lines 11-14). Given the wide variety of system calls and the fact that a single system call interface can accept different arguments and perform multiple functionalities, Syzkaller implements distinct *syzlang* descriptions with different arguments (e.g., ioctl() in Lines 8-9). To establish data dependencies between *syzlangs*, Syzkaller designates file descriptors as `resource` and establishes connections between *syzlangs* through resource dependencies.

For complex logic that cannot be achieved by a single system call, Syzkaller provides an extension called *pseudo-syscall*. These are C functions defined in the executor, which allow the executor to have code blocks to perform certain actions. They may also be used as more test-friendly wrappers for primitive syscalls. When Syzkaller generates a test case containing pseudo-syscalls, the executor invokes the predefined C functions.

2.2 Existing Fuzzing Works on Linux Drivers

Device drivers have long been and continue to be a significant source of defects and vulnerabilities in Linux kernels [31]. Defects and vulnerabilities are an inherent part of the fast-growing and evolving driver codebase. Therefore, exploring device driver vulnerabilities is crucial to improving the security of the Linux kernel.

Existing research primarily focuses on effectively triggering the execution of drivers. Most approaches utilize the system call interface, as it provides the most accessible method for unprivileged users to initiate driver execution. Some studies aim to leverage static analysis and symbolic execution to extract driver-related system calls and their parameters from the kernel and automatically generate *syzlangs* [4–6, 11, 35]. Other works propose advanced mutation strategies to achieve higher coverage with existing *syzlangs* [32, 37, 39]. Furthermore, some research highlights that interrupts can still serve as an attack surface [25]. Researchers try to simulate hardware and send arbitrary interrupts or return values to the kernel [13, 25, 33, 38], thus triggering driver execution.

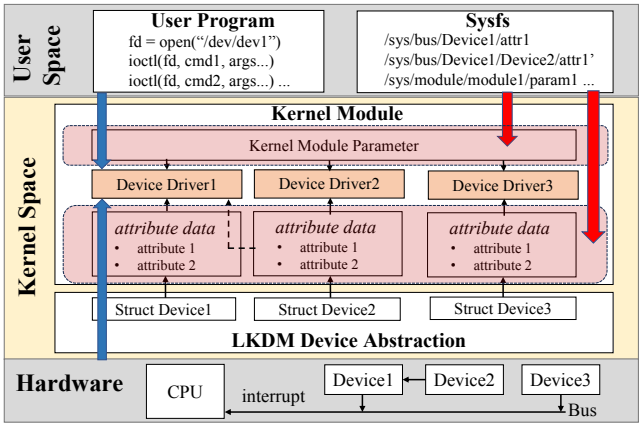


Fig. 3: An illustration of how runtime parameters affect kernel driver execution through sysfs. The red blocks represent the runtime parameters, and the orange blocks represent the device driver. Both user space and hardware can trigger code execution through blue arrows, and runtime parameters will influence the control flow through red arrows.

```
1  // Structure containing the parameter's name and callbacks
2  struct device_attribute zeroing_mode = {
3      .attr = {.name = "zeroing_mode"};
4      .show = zeroing_mode_show;          --> /sys/device/pci/scsi_disk/zeroing_mode
5      .store = zeroing_mode_store;
6  }
7  // Callback that does input validation and parameter modification
8  static ssize_t zeroing_mode_store(struct device *dev, const char *buf, ...) {
9      struct scsi_disk *sdkp = to_scsi_disk(dev);
10     int mode = match_string(zeroing_mode, buf);
11     if (mode < 0)                       } Input Validation
12         return -EINVAL;
13     sdkp->zeroing_mode = mode;
14     return count;
15 }
16 // Predefined valid inputs
17 static const char *zeroing_mode[] = {
18     [SD_ZERO_WS16_UNMAP]    = "writesame_16_unmap",
19     [SD_ZERO_WS10_UNMAP]    = "writesame_10_unmap", ...};
20 // Influenced control flow
21 static blk_status_t sd_setup_write_zeroes_cmnd(struct scsi_cmnd *cmd) {
22     struct scsi_disk *sdkp = scsi_disk(rq->q->disk);
23     switch (sdkp->zeroing_mode) {       } Control Flow Diversion
24     case SD_ZERO_WS16_UNMAP:
25         return sd_setup_write_same16_cmnd(cmd, true);
26     case SD_ZERO_WS10_UNMAP:
27         return sd_setup_write_same10_cmnd(cmd, true);
28     }...
29 }
```

Fig. 4: An example of a runtime parameter and how it influences the driver execution.

2.3 The Linux Kernel Device Model

The Linux Kernel Device Model (LKDM) provides a common, uniform data model for describing buses and their corresponding devices [26]. Specifically, it implements device management from a bus-oriented perspective through a series of device-related data structures. Among these structures, we identify a series of variables that can influence the driver execution. We call them *runtime parameters*, which can be broadly categorized into two types: *kernel module parameters* and *device attributes*.

Kernel module parameters. Kernel module parameters are global variables defined within kernel modules, which can control driver behaviors. These variables are initialized when the kernel module is installed into the kernel. Since one or more device drivers may reside within the same kernel module, modifying the kernel module parameters might influence more than one driver's execution.

Device attributes. Device attributes are driver- or bus-specific variables that record device status, and can also influence driver behaviors. It is worth pointing out that since devices are oriented in a tree-like structure physically, device attributes will not only affect the driver behavior of the current device, but also might influence the behavior of adjacent devices' drivers.

Linux employs a specialized virtual filesystem, known as `sysfs`, to expose kernel variables to user space. Due to the hierarchical nature of the bus-device structure, `sysfs` presents device structures and their relationships in the form of a file directory.

We summarize how runtime parameters influence the driver execution through `sysfs`. As depicted in Fig. 3, blue arrows indicate the inputs that trigger the code execution, including syscalls and interrupts, and red boxes represent the runtime parameters. Both kernel module parameters and device attributes are exposed to user space through `sysfs`, enabling users to configure the corresponding variables in the kernel to affect the driver's execution (through red arrows). `Sysfs` also reflects the underlying device connections. As shown in the bottom gray box of Fig. 3, `Device 2` is a sub-device of `Device 1`, which is exported to user space by `sysfs` as hierarchical directories. Kernel module parameters reside in a separate folder. Modifying the *device attributes* of `Device 2` may affect the driver execution of `Device 1` due to interconnected data structures.

We will give a brief example to illustrate how the runtime parameters get modified, and how they affect driver behaviors. In Fig. 4, "zeroing_mode" is one of the device attributes of scsi_disk device. It is exported to user space through `sysfs` and uses a set of *show/store* function callbacks. When the user modifies the file "zeroing_mode", the *store* function will check the user-provided string *buf* against predefined constants (Lines 10-12). If *buf* passes the validation, the corresponding attribute will be updated, and therefore affect the driver code execution (Lines 23-27).

In sum, besides system calls and hardware interrupts, drivers' execution can also be affected by two types of variables: kernel module parameters and device attributes. It is worth pointing out that *a driver is not only affected by the kernel module parameters to which it belongs and the device attributes of the bound devices, but also by the device attributes of other adjacent devices.*

3 Motivation

This section provides a comprehensive study of runtime parameters. *Firstly*, we conduct a preliminary experiment to quantify the number of branches affected by device attributes and kernel module parameters. We compare these with the code influenced by the system call arguments, and conclude that runtime parameters have a significant impact on control flow; *Secondly*, we demonstrate a historical bug as our motivation example and provide our insights on how to incorporate these parameters into kernel fuzzing.

3.1 The Impact of Runtime Parameters

We aim to understand the extent to which runtime parameters influence kernel code. To this end, we count the number of affected branch statements (i.e., `if` and `switch` statements) and basic blocks, as branch and basic block coverage are the most commonly used metrics in fuzzing. To make the results more persuasive, we also count the branches that can be affected by syscall arguments, which are likewise directly controllable by users.

We conducted a static taint flow analysis to quantify the amount of code influenced by runtime parameters and syscall arguments. Our experiment was performed on the upstream Linux v6.10 kernel with the all-yes-config enabled. In the following, we first describe how taint sources are identified, then present the details of the taint analysis, and finally discuss the measurement results.

Kernel Module Parameter Collection. The Linux kernel uses the "*__module_param_call*" macro to export all kernel module parameters to userspace-accessible files. We examined all invocation sites of this macro to identify the corresponding kernel module parameters. Only writable parameters were collected, as read-only parameters do not contribute to coverage during fuzzing.

Device Attribute Collection. Device attributes are created and initialized during device registration. Kernel developers use the "*struct device_attribute*" to define *show*/*store* interfaces, where the corresponding kernel variables can be accessed. We first identified all such interface structures and filtered out those lacking a *store* field. In *show* functions, device attribute values are converted to strings using *sysfs_emit()* or *printf()*-style functions (e.g., *sprintf*, *snprintf*, *scnprintf*, etc.). We use this feature to identify the associated device attribute variables reliably.

Syscall Argument Collection. To strengthen the persuasiveness of our statistics, we also analyzed syscall arguments, as they are directly controllable by users. However, system calls are eventually dispatched to device drivers through indirect function calls, which are difficult to trace statically. Inspired by [11], we examined callback functions registered during driver initialization, such as *struct file_operations* for character devices, *struct block_device_operations* for block devices, and *struct net_device_ops* for network devices. We focused on commonly used callback functions, and considered all of their arguments as user-controllable.

Taint Analysis. We perform an inter-procedural, field-sensitive taint analysis to collect basic blocks affected by the previously identified sources: module parameters, device attributes, and syscall arguments. The collection process consists of two stages. First, we perform a global taint analysis using CodeQL to trace the influence of taint sources and identify all usage points of tainted variables. This allows us to extract all relevant branch statements. Next, we collect the affected basic blocks, which include basic blocks: 1) directly governed by these branch statements, 2) within functions that are directly or indirectly invoked by those in 1), and 3) in functions that are subsequently called by functions identified in 2).

Since some user-controllable arguments are pointers to userspace memory, they are accessed in the kernel through functions like *copy_from_user()*. Furthermore, the **dst** variable may itself be a pointer, and its subfields may subsequently be accessed in later operations. To capture these behavior, we add custom flow steps that mark the destination (**dst**) argument of these functions as tainted, track pointer-field accesses and taint the field variable accordingly to ensure accurate coverage of taint propagation. By tainting variables above, we track data flows across different syscalls, enhancing their effectiveness in capturing inter-syscall dependencies.

Table 1: The number of the branches and basic blocks affected by different user-controlled data.

Statistics	Syscall Arg	Module Param	Device Attr
Source Number	12,839	1,494	1,407
Aff. If Stmt.	30,236	15,790	9,226
Aff. Switch Stmt.	1,036	46	58
Direct Aff. BB.	74,172	35,178	20,875
Direct Aff. + Called BB.	224,833	55,018	47,863
Direct Aff. + Called BB. Dedup	224,833	39,843	35,136
All Aff. BB. Dedup	340,980	56,418	44,135

With the all-yes-config, debug options like sanitizers are enabled and compiled into the kernel image. To avoid affecting our analysis due to these debug-only code paths, we introduce taint barriers to prevent taint propagation both from and to these functionalities.

Due to the prevalence of indirect calls in the kernel source code, resolving their targets is essential for accurate analysis. Inspired by TypeDive [22], we implement a type-based analysis to identify indirect call targets. Specifically, for each indirect call site, we locate the variable holding the function pointer and determine the type of its parent structure. We then traverse the source code to find all instances of that structure type and collect the possible functions assigned to the corresponding function pointer field.

We identify all branch statements directly influenced by taint sources and classify them as affected. The basic blocks that immediately follow these branches are labeled as **Direct Aff. BB**. We then include the basic blocks within functions invoked at these branches, marked as **Direct Aff + Called BB**. To provide a conservative result of how runtime parameters will contribute to code coverage, we deduplicate these blocks by removing commonly shared code across different taint sources. We report the result as **Direct Aff + Called BB. Dedup**. Lastly, we collect and deduplicate all reachable basic blocks and denote as **All Aff. BB. Dedup**.

Static Results. As shown in Table 1, device attributes and kernel module parameters affect a considerable number of `if`/`switch` statements compared to syscall arguments and directly control a wide range of basic blocks. The deduplication results indicate that while some affected basic blocks are shared among different taint sources, runtime parameters enable coverage of additional basic blocks that syscall arguments cannot directly or indirectly reach, potentially contributing an additional 29.5% to code coverage.

To this point, we conclude that there is a large amount of code that remains uncovered due to runtime parameters. Although theoretically, these parameters can be modified by existing syscall interfaces like write(), this approach is impractical for current fuzzers due to the complexity of generating valid strings for file paths. Moreover, even if files under *sysfs* are accidentally opened, writing random values is unlikely to yield meaningful results, as the parameters are governed by value-checking logic (as shown in Fig. 4). As of this writing, only 39 files have been explicitly included in Syzkaller, accounting for less than 2% of the total runtime parameters.

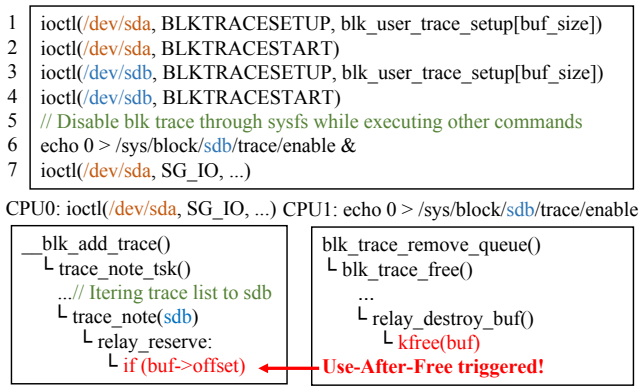


Fig. 5: PoC and call stacks related to CVE-2021-47375. The upper half is the proof-of-concept that triggers the bug, while the lower half depicts its root cause.

3.2 Motivating Example

We will first show a historical bug related to runtime parameters, analyzing the root cause of it and the prerequisites for triggering it. Based on this example, we summarize our insights on integrating these parameters into kernel fuzzing.

CVE-2021-47375. A previously discovered bug relating to runtime parameters is CVE-2021-47375, illustrated in Fig. 5. The top portion of the figure presents the proof-of-concept code for this vulnerability. Lines 1-4 enable the BLKTRACE functionality for both hard disk devices, *sda* and *sdb*, causing the kernel to create corresponding nodes in a global list. Subsequently, on Line 6, the BLKTRACE functionality for *sdb* is disabled via `sysfs`, while commands are simultaneously sent to *sda* using `ioctl`.

The bottom portion of Fig. 5 depicts the call stacks when triggering the bug. On CPU0, the `ioctl()` invokes `trace_note_tsk()`, which iterates over all nodes in the global trace list. Concurrently, CPU1 executes `blk_trace_remove_queue()`, removing *sdb* from the trace list. However, CPU0 remains unaware of this removal and attempts to access the `buf→offset` field of the now-deleted node, resulting in a Use-After-Free vulnerability.

This example inspires us to believe that runtime parameters will not only affect the current device driver it is attached to, but also drivers of all devices with underlying data structures interconnected. In this case, changing device attributes for *sdb* will result in crashing on executing commands related to *sda*. Also, parameter modification alone cannot trigger the vulnerability. It should combine with the driver execution, and the time of modification is important for triggering the bug.

Conclusion. Based on the statistics above and the motivating example, we conclude that runtime parameters are critical in kernel fuzzing. Existing fuzzing tools have not effectively utilized these interfaces, failing to uncover latent bugs. By analyzing historical vulnerabilities, incorporating runtime parameters with syscalls that invoke drivers will expose more vulnerabilities. We highlight that the relationships between devices and the timing of modifications are essential factors in triggering kernel bugs.

4 Design

As mentioned, to test the kernel driver in-depth, we need to integrate runtime parameters, as well as the relationship among them, into fuzzing. However, incorporating the runtime parameters into the fuzz engine is non-trivial. It still faces three technical challenges, as listed below respectively:

- **C-1:** How to identify runtime parameters and their corresponding values to pass input validation? Current interfaces can open and write files under the filesystem, but the kernel has strict check logic to prevent arbitrary values from being written to parameters. Randomly generated values may fail to modify the parameter, thereby hindering the execution of deep driver code.
- **C-2:** How to determine relations between runtime parameters and drivers? The kernel encompasses thousands of associated parameters, and there will be exponential combinations if there is no guidance, decreasing fuzzing efficiency.
- **C-3:** How to integrate the runtime parameters into the fuzz engine and cooperate with existing features? Setting runtime parameters will affect not only the current test case but also the following cases until the system reboots. This will result in unreproducible coverages for the following test cases, affecting other components in the fuzz engine.

In the following subsections, we will first propose an overview of our design, and explain how the design addresses the above three challenges. Then we will give out the details of each component, which resolves the challenges respectively.

4.1 Overview

To address the aforementioned challenges, we propose a new fuzzing framework. The overall framework of SyzParam is depicted in Fig. 6, comprising two primary stages: the *offline information collection stage* and the *online fuzzing stage*.

In the first stage, SyzParam conducts the **value collection task** to overcome the **C-1**, collecting the filename and valid values from the source code to pass the input validation, thus modifying parameters successfully during the next stage. These values are subsequently used in *syzlang* generation. To resolve the **C-2**, SyzParam performs the **relation identification task** to identify the two kinds of relationships, *inter-device* and *parameter-driver* relationships, and constructs a device relation tree. These relations will be used to guide mutators and narrow the search space.

In the second stage, SyzParam conducts the **fuzz engine integration task** to incorporate runtime parameters into fuzzing. We extend the fuzz engine and leverage relationships to prioritize the newly generated *syzlangs*, facilitating more in-depth testing of relevant drivers. By utilizing existing interfaces in Syzkaller, we propose new mutation strategies that determine the parameter before driver execution, therefore solving the **C-3**.

4.2 Value Collection

We need the filenames and valid inputs to change the value of the runtime parameters successfully. The filenames are predefined strings that are registered in global variables in the kernel source code. By identifying the definition points, we can recover the filename from the variable. To obtain valid values for parameters,

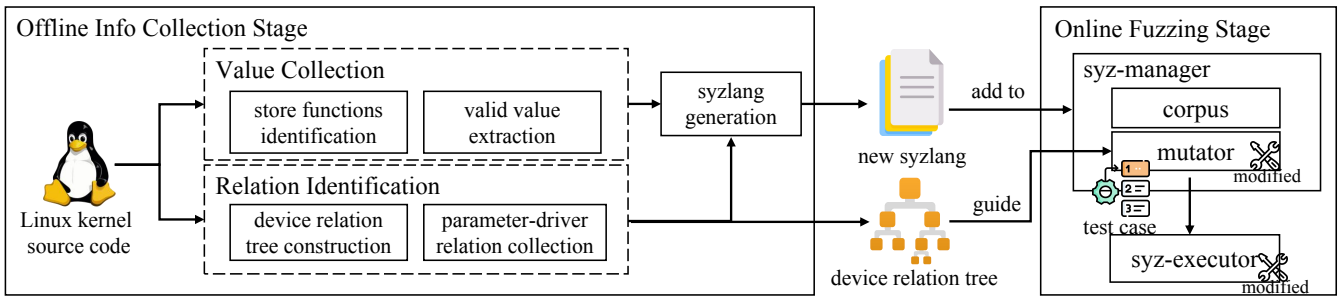


Fig. 6: The framework overview of SyzParam.

SyzParam first identifies store functions responsible for updating the parameter values, then extracts valid values according to the validation logic embedded within the store functions.

As demonstrated in previous sections, the *struct device_attribute* encapsulates the device attribute's name (which is also the filename in *sysfs*) and associated *store* function. Based on the design pattern of runtime parameters, we degrade the value collection task into two steps: store function identification and valid values extraction.

4.2.1 Store function identification. The key to gathering a filename-function pair is to identify globally defined *struct device_attribute* variables. Although in most cases these variables are initialized by the `__ATTR()` macro provided by the Linux kernel, some device drivers define their device attribute structures. Since parameters can only be exported to the user space through *struct attribute*, we manually inspect all the kernel structures that contain *struct attribute* in the kernel source code (56 results in total) and summarize the characteristics of all kinds of attribute variables:

- First, they are defined in the kernel as a series of global variables initialized when the driver is loaded.
- Second, the show or store functions must be provided. Otherwise, it cannot be accessed through the file system.

For those driver-specific device attribute structures, at least one subfield matches the above rules. Thus, we iterate over all global variables in the kernel and gather structure type variables. Following the above rules, we check every subfield recursively. Since read-only parameters will have limited influence on drivers' control flow, we omit all device attributes without writable permission.

Algorithm. Building on the understanding mentioned above, we propose the following algorithm, with the corresponding pseudo-code presented in Algorithm 1. The `for` loop in Lines 20-22 iterates over all global variables for analysis, calling the procedure "GetStoreFunc()" to verify each variable. In the procedure, Line 8 verifies whether the input variable is structure type. Subsequently, the *matchDeviceAttr()* function is called to determine whether the variable includes the attribute structure along with pointers to the associated *show* and *store* functions (Lines 1–6). If this condition is satisfied, the algorithm traverses all elements of the variable to locate the *store* function, pairs it with the attribute's name, and appends the result to the list *store_functions* (Lines 9–11). It is important to note that the file name, the device attribute's name, and the variable's name are not always identical. Therefore, we specifically extract the file name string as defined within the *struct device_attribute*. For structures that do not meet the criteria, the algorithm recursively iterates through all sub-fields to identify the target structure, as depicted in Lines 13–15.

Algorithm 1 Store Functions Identification

Input: LLVM bitcode of Linux kernel
Output: *store_functions*, the list of pairs between store functions and attribute names

```
1:  function matchDeviceAttr(var)
2:      if hasStructAttr(var) & hasStoreFuncPtr(var) then
3:          return True
4:      end if
5:      return False
6:  end function
7:  procedure getStoreFunc(var)
8:      if isStructTy(var) then
9:          if matchDeviceAttr(var) then
10:             pair ← getNameFuncPair(var)
11:             store_functions.push(pair)
12:         else
13:             for each, elem ∈ var do
14:                 struct_elem ← getInitializer(elem)
15:                 getStoreFunc(struct_elem)
16:             end for
17:         end if
18:     end if
19: end procedure
20: for each, gv ∈ GlobalVariables do
21:     getStoreFunc(gv)
22: end for
```

4.2.2 Valid Values Extraction. We construct a static taint analysis pass targeting all the identified *store* functions to extract the variable types and the corresponding range of valid values. By marking the user-provided `buf` as the taint source and analyzing the sink points along the control flow, we identify the functions responsible for validation. For string-type parameters, the check logic resides in string comparison functions (e.g., `match_string()`). For parameters of other types, the kernel invokes helper functions to convert the original string(e.g., `kstrtouint()`). In such cases, we track the

converted value and its sink points instead of the original user-provided string. Subsequently, we determine the range of values that can successfully pass the validation logic.

Since kernel module parameters do not have input validation, we collect all parameters and their relative types by identifying their definition points. Linux uses the macro `module_param()` to define kernel module parameters. The macro will finally be expanded to a structure containing parameter names and types, which are sufficient for us to construct *syzlang*.

4.3 Relation Identification

After identifying the parameter file names, their corresponding store functions, and their valid values, the next objective is to build up the relationship between parameters and related drivers. Based on our understanding of parameter-related vulnerabilities, the relationship contains two perspectives: *Inter-device relationship.* Since devices are organized by device tree through underlying data structures, the device's attributes will affect topologically connected devices; *Parameter-driver relationship.* Modification of the parameters alone is insufficient to trigger bugs. It should combine with the related driver code execution. The following sections will introduce our strategies for collecting the relations mentioned above.

4.3.1 Device Relation Tree Construction. The *sysfs* interface exports devices in a hierarchical structure, enabling the construction of a device relation tree through folder traversal. Specifically, a sub-device appears as a sub-folder within the parent device's directory in the *sysfs*, with the root directory corresponding to the bus to which the device is attached. The device relation tree is constructed with the bus as the root node.

4.3.2 Parameter-Driver Relation Collection. To trigger the execution of related drivers after modifying parameters, it is necessary to establish a mapping between parameter files and their corresponding drivers. In Linux, parameter files reside in the device's folder under */sys*, and the user can invoke drive execution by opening and operating files under the */dev* directory. Consequently, for all devices located in the */dev* directory, the relationship between their parameters and the drivers they bond to can be established by matching their file names with the directory names under */sys*.

Based on the files and values collected in the previous steps, we adopt the methodology outlined in existing work [11] to generate *syzlangs*, facilitating the modification of runtime parameters. To minimize redundancy caused by multiple devices of the same type (e.g., `usb0` and `usb1` sharing identical parameter types), we merge parameters of the same type into one *syzlang*.

4.4 Fuzz Engine Integration

We must not only enable parameter modification but also ensure that the collected relationships are integrated into the fuzzing engine. Simply adding syzlang descriptions for parameter modifications can lead to reproducibility issues in code coverage, as modified parameters may affect the execution of subsequent test cases until the system is rebooted. However, current fuzzing mechanisms are unaware of this effect. For example, if test case A modifies a parameter and test case B is subsequently influenced by it and triggers new coverage, the fuzzer will verify the new coverage across other VMs to determine stability. Since the parameter modification does not occur on those VMs, the fuzzer may incorrectly discard test case B, leading to decreased fuzzing efficiency. This problem is further exacerbated by the lack of explicit dependencies between parameters under the */sys* directory and device files under the */dev* directory, making it even harder for the fuzzer to infer the relationship between parameter modifications and driver execution. Our method binds parameter modifications directly to test cases, ensuring that new code coverage resulting from parameter changes can be reliably reproduced.

Algorithm 2 Pseudo Code for syz_mod_dev()

Input: param_path, param_val, dev_path, rand_int, flags
Output: **fd**, the file handler of opened device

```
1:  function SYZ_MOD_DEV(param_path, param_val, dev_path, rand_int, flags)
2:      if hasWildcard(dev_path) then
3:          dev ← replacWildcard(dev_path, rand_int)
4:          param ← replaceWildcard(param_path, rand_int)
5:      end if
6:      sysfd ← open(param)
7:      write(sysfd, param_val, len(param_val))
8:      fd ← open(dev, flags)
9:      return fd
10: end function
11: function REPLACEWILDCARD(file_path, rand_int)
12:     strncpy(buf, file_path)
13:     tmp ← rand_int
14:     for hasWildcard(buf) do
15:         id ← tmp%10 +′ 0′
16:         replaceOne(buf, id)
17:         tmp ← tmp/10
18:     end for
19:     return buf
20: end function
```

Therefore, two primary components must be modified accordingly: The executor should include additional primitives that bind parameters to related driver executions, ensuring that new coverage resulting from parameter modifications can be reliably reproduced across different VMs; The mutator should leverage these relations to group related parameters and schedule their execution timing to more effectively trigger vulnerabilities. In the following sections, we present our design for each component.

4.4.1 Execution Engine Modification. To establish the association between drivers and parameters within Syzkaller, we employ the pseudo-syscall mechanism, which integrates parameter modifications with syscalls that invoke the corresponding drivers. As discussed above, pseudo-syscalls' design adheres to two key principles: First, pseudo-syscalls must be universal, as a single driver may have numerous runtime parameters. Creating unique pseudo-syscalls for each parameter would demand substantial computational resources to record and update *syzlang* relations. Second, pseudo-syscalls should establish dependencies with other *syzlangs* associated with the same driver to maintain coherence.

We implement pseudo-syscalls within the execution engine and register them alongside existing syscalls to ensure seamless integration. To satisfy the first principle, we generate pseudo-syscalls at the device level, combining the runtime parameters of a device and its sub-devices into a single pseudo-syscall. To fulfill the second principle, our pseudo-syscall establishes resource dependencies between existing syzlangs and our pseudo-syscalls.

Additionally, many buses initialize device names using a naming convention such as "device type - device number." To avoid redundantly generating homogeneous syzlang definitions for each device, we use the wildcard "#" to represent any device of the same type, allowing a random device instance to be selected during fuzzing.

Our pseudo-syscall is defined as shown in Algorithm 2. Its primary function is to modify parameters, open the associated device file in the */dev* directory, and return its file descriptor. The arguments include the path to the parameter file and device file, the modified value for the target parameter, a random seed, and the open permissions for the device file. As discussed above, the algorithm first checks whether the input device file contains a wildcard. If the wildcard "#" is found, the function *replaceWildcard* is called to resolve the file path to a specific device, determined by rand_int (Lines 2-4). Specifically, the wildcard is replaced by a character range from '0' to '9', and the fixed file path is returned as a string (Lines 11-19). The replacement process is identical for both device and parameter files, ensuring that the parameter is correctly related to the device. The parameter file is then opened and modified with the given value, and the pseudo-syscall retrieves the file descriptor for the device file as the return value (Lines 6-9).

4.4.2 Mutation Strategy Integration. Based on our understanding of runtime parameters, we design a mutation strategy targeting these parameters and integrate it into the fuzzing process. The proposed mutation strategy determines: 1) which parameters to modify, and 2) when to modify their values.

As demonstrated in the motivating example (Fig. 5), both the relationships between devices and the timing of parameter modifications are crucial for uncovering kernel vulnerabilities. Therefore, for the first question, we prioritize modifying the parameters of topologically related devices. Given a seed program that modifies parameters, we insert additional system calls to modify the parameters of its parent device. During fuzzing, parameter values are randomly selected from valid values gathered, utilizing the original algorithm employed by Syzkaller for mutating syscall arguments.

For the second question, in addition to modifying parameters before driver execution using the pseudo-syscalls described above, we initiate asynchronous threads that modify parameters while executing other system calls targeting the corresponding device driver. Specifically, for a given test case, we add system calls that concurrently modify the parameters of topologically related devices, or concurrently modify the same parameters to different values to trigger data races. The asynchronous threads continue executing until the current test case completes or until the predefined maximum number of iterations is reached.

To this end, we incorporate fuzzing with the runtime parameters, exposing long-neglected code functionalities. By integrating new *syzlangs* and proposing new mutation strategies, we extend Syzkaller's ability to test kernel drivers.

Table 2: The statistics of the lines-of-code of each component.

Component	LoC	Language	Implemanted on
Attribute value collection	1598	C++	LLVM, CRIX
Syzlang generation	396	C++	SyzDescribe
Fuzz engine implementation	252	C / golang	Syzkaller
Other Scripts	391	C / Codeql	-

5 Implementation

We implemented a prototype of SyzParam, and the overall lines of code of our implementation are presented in Table 2. As shown in the table, the static analysis of kernel global variables and functions is based on LLVM[8] and CRIX[23]. The implementation of syzlang generation is based on SyzDescribe[11]. The incorporation of the fuzzing system is based on Syzkaller[10]. We will illustrate the implementation details of each component below.

Value Collection. We develop a static analysis tool to examine kernel source code and extract predefined values. Using customized LLVM module passes, we track definition points of global variables of *struct device_attribute* type. Then, we locate the attribute value verification logic within the corresponding attribute store functions. We determine whether the validation involves string comparisons or string conversion utilities by performing a static taint analysis and matching sink points of user-provided strings.

Relations Identification. We develop scripts to traverse the */sys* directory within the virtual machine, with each subdirectory representing a device. We record their hierarchical relationships using a depth-first approach, catalog all parameter files, and map them to the previously identified structures and corresponding value ranges. For parameter-driver relationship, we collect device files within the */dev* directory and associate them with corresponding device folders in the *sysfs* based on matching file names.

Syzlang Generation. We construct new syzlangs for each parameter type in this module. Due to different device instances sharing the same parameter type, we generate one syzlang for all relevant instances that have the parameter type to minimize the number of newly generated syzlangs. For kernel module parameters, we iterate over all files under */sys/modules* directory and match them with parameters collected before to determine their types.

Execution Engine Modification. We utilize the pseudo-syscall interface provided by Syzkaller to ensure maximum compatibility. We integrate the C functions representing the newly defined pseudo-syscalls into Syzkaller's executor module. By leveraging the established mapping relationship between the */dev* and */sys* directories, we statically generate parameter combinations for the pseudo-syscall. This approach ensures consistency between the files while optimizing fuzzing efficiency.

Mutation Integration. We enhance Syzkaller's existing mutator by introducing a novel mutation strategy. Specifically, when mutating a seed containing newly defined syzlangs, we integrate a complementary syzlang that modifies the parameters of the other devices on the same device tree and execute it concurrently, trying to trigger the race condition. This approach facilitates a comprehensive exploration of device interactions during fuzzing.

6 Evaluation

In this section, we evaluate SyzParam by comparing it with related approaches and presenting comprehensive experiment results. Specifically, we aim to address the following questions:

- How many device attributes can be extracted during the offline information collection stage? (§6.2)
- Can the incorporation of runtime parameters enhance the code coverage of kernel drivers during fuzzing? (§6.3)
- Does leveraging relationships between devices contribute to the identification of vulnerabilities? (§6.4)
- Can new vulnerabilities, previously undetectable from the perspective of the driver model, be identified? (§6.5)

6.1 Experiment Setup

All experiments were conducted on a server equipped with two Intel Xeon Gold 6430 CPUs and 256GB of RAM, running Ubuntu 22.04 LTS as the platform system. The target Linux kernel version was v6.7. Our fuzzing framework was developed based on Syzkaller[10], with Clang-14 utilized as the compiler to generate LLVM IR. Both the offline and online stages employed the same Linux kernel configuration, which is also used by the upstream kernel in Syzbot. These kernel compilation options, provided by Google engineers, reflect their understanding of testing the Linux kernel with QEMU. The kernel configuration option `CONFIG_SYSFS` is enabled by default. To ensure comprehensive coverage, we manually enabled all `sysfs`-related configuration options (ten in total).

For comparative evaluation, we selected state-of-the-art kernel fuzzing tools, including Syzgen++[4] and Syzdescribe[11]. Syzgen++ extracts syzlangs from kernel image files using symbolic execution and dynamic debugging techniques, aiming to improve code coverage by analyzing relationships between syzlangs. Syzdescribe applies static analysis techniques to identify system call-related code in the kernel source, and generate new syzlangs for fuzzing. Since Syzgen++ was initially tested on the Linux kernel v5.15, we adapted it for compatibility with the target kernel version 6.7.

6.2 Results of Static Analysis

Our tool analyzed all device attribute variables in the target kernel version, collecting their valid inputs respectively. In total, 1,243 device attributes were identified. Among these, 358 attributes accept unsigned integers, 229 accept signed integers, 107 accept boolean values, 60 accept strings, and 80 accept formatted string inputs. Additionally, 46 device attributes were found to execute regardless of user-provided values. However, the input legality could not be determined for 363 functions due to the following primary reasons:

- **Value propagation and indirect calls.** Store functions propagate the *buf* value to other functions via indirect calls. Due to the inherent limitations of static analysis, identifying the target functions of these indirect calls was not feasible.
- **Complex parameter processing logic.** Some intricate input validation logic (e.g. customized state machines) currently exceeds the analytical capabilities of our method, preventing the determination of feasible values for these device attributes.

As for kernel module parameters, our tool finds 694 unique parameters. The number of runtime parameters is less than the data collected in previous sections, with the reason we use the fuzzing config here instead of the all-yes-config. Also, the all-yes-config will include runtime parameters for different architectures, which will never appear in a single test run.

We then gathered the mappings of all files and directories within the */sys* directory in the VM at the offline profiling stage and found a total of 17,081 writable files. We extracted the filenames of these attribute files and matched them with the name field in the collected device attribute structures. Subsequently, we associated these files with their respective device attribute store functions, recording 819 types of files that have their corresponding functions. We noticed that 424 store functions do not pair with files under the */sys* directory. This is because although the driver defined the parameters interface, there is no corresponding device bond to the driver, thus failing to create parameter file instances.

For runtime parameters that cannot be distinguished, our current approach uses random strings to populate the buf parameter when generating syzlang. To enhance the accuracy of our analysis in the future, we plan to integrate techniques such as symbolic execution to gather more precise values of device attributes.

6.3 The Code Coverage

In this subsection, we compare our work with related work to demonstrate how incorporating runtime parameters can enhance code coverage during kernel driver fuzzing. We selected eight drivers on different buses from the list of drivers supported by Syzgen++ and Syzdescribe. These drivers also supported runtime parameters that can be modified.

Experiment setup. To ensure fairness, we modified our tools to produce only syzlang for tuning parameters that the original Syzkaller can directly parse. Two primary considerations drove this decision: 1) All other related works in the experiments utilized the unmodified Syzkaller; 2) Our modified mutator established relationships between devices, and it would introduce coverage from other device drivers, which is not fair for comparison. Hence, we provided syzlangs generated by our tool(marked as SyzParam-), Syzgen++, and Syzdescribe to the same version of Syzkaller, and turned on only the syzlang of the corresponding driver. During the test environment setup, we identified some errors in the device file names and paths within the syzlangs generated by Syzdescribe and manually corrected them. For each driver, we deployed two VMs with two vCPUs per VM, executing each test for 24 hours, totaling five runs. The experiment results and corresponding confidence p-values are presented in Table 3.

Edge coverage results. The results show that SyzParam achieves the highest edge coverage among all related works in 7 out of 8 drivers. SyzParam outcompetes original Syzkaller in all 8 drivers and obtains an average coverage improvement of 32.57%. According to the Mann-Whitney U Test, the results for most drivers are significant, with a p-value lower than 0.05.

However, for drivers such as *i2c* and *seq*, the p-value exceeds the threshold of 0.05 compared to SyzGen++. To further validate that our coverage improvements are not merely a subset of those

Table 3: The edge coverage results for selected drivers.

Drivers	#Avg. Edge Coverage					p-value		
	Syzkaller	SyzDescribe	SyzGen++	SyzParam-	↑Comparing to Syzkaller	Syzkaller	SyzDescribe	SyzGen++
rtc	3,424	3,537	3,517	**5,337**	55.89%	0.0060	0.0040	0.0040
usb	11,301	N/A	11,638	**16,523**	46.21%	0.0040	N/A	0.0040
loop	6,348	7,708	7,120	**9,244**	45.63%	0.0040	0.016	0.0040
msr	1,883	2,100	2,474	**2,529**	34.28%	0.0040	0.0040	0.0040
i2c	3,481	3,297	4,309	**4,386**	26.02%	0.0040	0.0040	0.21
sr	7,301	6,954	N/A	**8,653**	18.52%	0.0040	0.0040	N/A
mouse	2,958	3,690	N/A	**3,482**	17.73%	0.0040	0.0040	N/A
seq	3,610	3,690	**4,244**	4,198	16.29%	0.0040	0.0278	0.85

Table 4: Comparison for SyzParam, Syzgen++ and combined.

Drivers	Syzgen++	SyzParam-	Combined	↑ in Edge Cov.
rtc	3,517	5,337	5,560	58.09%
usb	11,638	16,523	17,671	51.84%
loop	7,120	9,244	9,227	29.59%
msr	2,474	2,529	3,158	27.65%
i2c	4,309	4,386	4,742	10.05%
seq	4,244	4,198	4,759	12.13%

achieved in previous research, we conducted an additional experiment combining our syzlangs with those of SyzGen++, with the same experiment setup. The edge coverage results, presented in Table 4, demonstrate that the combined syzlangs outperform SyzGen++ in terms of edge coverage across all six drivers supported by both approaches. These findings indicate that our method not only enhances the performance of existing techniques but also achieves higher coverage by exploring new code paths rather than retesting the same portions of the codebase.

Given the result that the coverage improvement is not evenly distributed among drivers, we conclude that this is because the functionalities of different buses vary significantly, leading to variations in the number and types of parameters they possess. For block device drivers like `loop`, the runtime parameters can control the IO status, including poll rate and sector bytes. However, in drivers such as `i2c`, runtime parameters only have limited control, including power management and device notification.

6.4 The Number of Vulnerabilities

To illustrate the contribution of the syzlang we generated and the mutation strategy we proposed to vulnerability discovery, we did an ablation study on these components. We choose the number of unique crashes as the indicator of vulnerability discovery ability. The overall results are shown in Fig. 7. The original Syzkaller is used as the baseline (marked as "baseline"), compared to a version that adds newly generated syzlangs alone (marked as "syzlang") and another version that combines syzlangs with modifications to the fuzzing engine's mutation strategy (marked as "syzlang+mutation"). Each fuzzer uses 8 QEMU VMs with 2 vCPUs per VM, conducting an experiment lasting 120 hours (5 × 24h), totaling four rounds of experiments. We count the number of UNIQUE crashes as well as the discovery time and calculate the average results and the

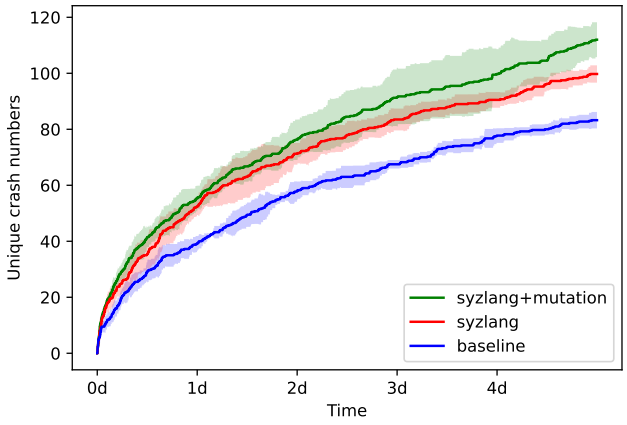


Fig. 7: Number of unique crashes over time. Each setup is tested in four rounds, with each round lasting for 120h.

confidence interval of 0.90. Unique crashes are distinguished by their crash title, including their crash types and the function that triggers them. To mitigate any impact from Syzkaller's inherent issues, we ignore crash types beginning with "SYZFATAL".

The results demonstrate that the incorporation of syzlang for modifying runtime parameters facilitates the discovery of previously unknown vulnerabilities. Additionally, leveraging device relationships to guide fuzzing mutations enhances the efficiency of vulnerability discovery. Over 120 hours, the two optimizations that we propose improve the number of vulnerabilities found significantly. Adding syzlang alone increases the number of unique crash types by 19.8% compared to baseline, while adding syzlang and mutation strategy gains 34.5% improvement in average.

6.5 Bug Finding

To demonstrate our tool's ability to discover new vulnerabilities, we conducted a long-term experiment. As shown in Table 5, we identified a total of 30 previously undiscovered vulnerabilities. We reported these vulnerabilities to the Linux community and actively cooperated with them. Of these, 20 have been confirmed, 14 have been fixed, and 10 CVEs have been assigned to date.

We analyzed the root causes of the discovered vulnerabilities. As shown in the table, 21 out of the 30 bugs were found through the modification of runtime parameters. A detailed analysis reveals

Table 5: Previous-unknown bugs found by our tool.

ID	Bug Type	Error location	Status	Mod Param	Need Root Priv.
#1	KASAN slab out-of-bound / UAF	tomoyo_write_control	CVE-2024-26622	✓	✓
#2	task hung (deadlock)	hub_activate	CVE-2024-26933/26934	✓	✓
#3	divide-by-zero	wq_update_node_max_active	CVE-2024-27055	✓	✓
#4	nullptr deref	disable_store	CVE-2024-36896	✓	✗
#5	UBSAN: shift-out-of-bounds	dctcp_update_alpha	CVE-2024-37356	✓	✓
#6	divide zero	alauda_transport	CVE-2024-38619	✗	✗
#7	KASAN slab-out-of-bound	xlog_cksum	CVE-2024-39472	✗	✗
#8	KASAN out-of-bounds read	in4_pton	CVE-2024-39487	✓	✓
#9	KASAN: slab use after free read	zswap_decompress	CVE-2025-21693	✓	✗
#10	ODEBUG_STATE_WARNING	mcheck_cpu_init_timer	fixed	✓	✓
#11	UBSAN: shift-out-of-bounds	fault_around_bytes_set	fixed	✓	✓
#12	warning	sel_write_load	fixed	✓	✓
#13	UBSAN: array-index-out-of-bound	diFree	fixed	✗	✗
#14	warning	static_key_slow_inc_cpuslocked	fixed	✓	✓
#15	UBSAN: shift-out-of-bounds	sg_build_indirect	patch under revision	✓	✓
#16	warning	ax25_dev_device_down	patch under revision	✓	✓
#17	KASAN: slab use after free read	pressure_write	patch under revision	✓	✓
#18	task hung (deadlock)	rfkill_fop_write	confirmed	✓	✓
#19	warning on once	static_key_disable_cpuslocked	confirmed	✓	✓
#20	task hung	blk_mq_get_tag	confirmed	✗	✗
#21	task hung	lock_sock_nested	reported	✓	✓
#22	deadlock	ptp_clock_unregister	reported	✓	✓
#23	warning	disksize_store	reported	✓	✓
#24	warning	cfg80211_bss_update	reported	✓	✓
#25	general protection fault	udmabuf_create	reported	✓	✓
#26	KASAN: slab-out-of-bound	asus_report_fixup	reported	✗	✗
#27	warning	read_rindex_entry	reported	✗	✗
#28	warning	ext4_iomap_begin	reported	✗	✗
#29	kernel bug	folio_end_read	reported	✗	✗
#30	general protection fault	gfs2_unstuffer_folio	reported	✗	✗

that 19 of these bugs can only be triggered by modifying runtime parameters, while two can also be triggered through other syscalls or under specific kernel configurations. Since modifying files under sysfs requires root access, we label all 19 bugs that require parameter modifications as **Need Root Priv** in Table 5. Furthermore, among the 21 bugs related to runtime parameters, 7 are triggered by altering default values. For example, Bug #3 in Table 5 is discovered by changing the affinity of a device workqueue to different CPUs via sysfs, whereas by default it is set to all CPUs. If the target CPU goes offline, the workqueue bound to that CPU reduces `total_cpus`, which is then used as a divisor, leading to a divide-by-zero error. Eight bugs are exposed through concurrent behaviors, including four involving interactions between multiple devices and three involving concurrent modifications of the same parameter with different values, demonstrating that our mutation strategy effectively uncovers vulnerabilities.

For vulnerabilities that can be triggered without modifying parameters, further analysis shows that leveraging sysfs still plays a key role in uncovering them. For instance, in Bug#9 of Table 5, the issue cannot be triggered when the zswap shrinker is disabled—this happens if CONFIG_ZSWAP_SHRINKER_DEFAULT_ON is not set at compile time. These features are enabled and tested via sysfs. However, in kernels where the configuration is enabled (e.g., Ubuntu 24.04 LTS), the bug becomes triggerable by unprivileged users through syscalls. Similarly, Bug#4 can be triggered without using sysfs, but doing so via the regular syscall interface is challenging due to complex syscall dependencies between parent and child devices and the need for precise timing. In contrast, sysfs enables the fuzzer to easily identify the relevant devices and set devices to a different status, making the bug more readily discoverable.

In conclusion, introducing runtime parameters to fuzzing also helps us uncover various bug types, including use-after-free, out-of-bounds read, null pointer dereference, deadlock, etc. These bugs are security-related and can affect kernel stability.

6.6 Case Study

In this subsection, we will use a specific vulnerability uncovered by SyzParam, CVE-2024-36896, to demonstrate its ability to identify vulnerabilities. This bug is caused by the improper handling of inter-device relationships and should be triggered by concurrently accessing the runtime parameters.

USB device hierarchy. In kernel devices, USB devices are an excellent example to illustrate relationships between devices. USB

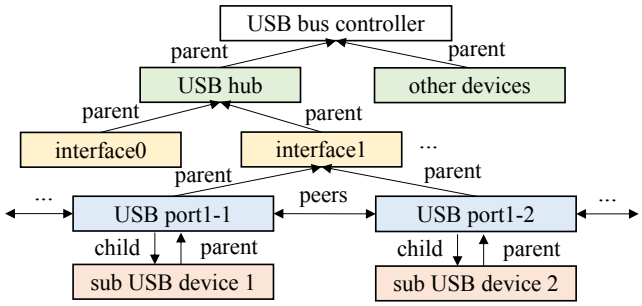


Fig. 8: Device relationships under USB bus.

devices can connect to the system through USB hubs, and a single USB device may have multiple functionalities (e.g., a webcam with both video and audio capabilities). The Linux kernel uses hierarchical structures to describe USB device organizations, such as `usb_device`, `usb_hub`, `usb_port`, and `usb_interface`. The USB driver uses *parent* pointer to organize devices and form a device tree, with the USB bus controller as the root. The relationships among these device structures are illustrated in Fig. 8.

Root cause analysis. The kernel code related to the aforementioned vulnerability is displayed in Fig. 9. When a USB hub device is removed from the system, the kernel invokes the *hub_disconnect* function (Line 2), which acquires the `device_state_lock` of the USB hub to prevent other devices from tampering with the data state (Line 7). The driver sets the maxchild number to zero and sets the data of *struct usb_interface* to NULL to avoid access (Lines 10-11). During removal, if someone attempts to access a subdevice's parameter under the hub (e.g., to access the `disable` parameter on one of the ports), the kernel invokes the store functions of that parameter. In this case, the store function `disable_store()` will try to get the port device's parent hub, and get private data from the hub(Lines 22-24). `usb_hub_to_struct_hub()` checks whether the necessary data for the hub device exists, and returns the *struct usb_hub* pointer if passed the check. The `disable_store()` function directly accesses the fields in the structure pointer without attempting to acquire the lock of the parent device or checking the return value, resulting in a null pointer dereference.

In this case, developers failed to take into account the relationship between devices. During the removal of the parent device, child devices can still access their parent device's data through `parent` pointer, without holding the parent's lock. Thus the code execution of the child device's driver is affected by its parent.

During testing, our executor attempts to concurrently access the parameters of the USB port while removing the USB hub (USB port's parent device) from the system. This vulnerability demonstrates that the relationship between devices affects the logic of data access between devices in the driver, as well as the parameter modification timing. Consequently, our mutation strategy successfully triggers a previously challenging-to-find vulnerability. Additionally, while this example illustrates how accessing device attributes can trigger the null pointer dereference through `usb_hub_to_struct_hub()`, it is important to note that this function is widely used in USB drivers and is not confined to the `sysfs` interface.

```
1  // drivers/usb/core/hub.c
2  static void hub_disconnect(struct usb_interface *intf) {
3          struct usb_hub *hub = usb_get_intfdata(intf);
4          struct usb_device *hdev = interface_to_usbdev(intf);
5          int port1;
6          mutex_lock(&usb_port_peer_mutex);
7          spin_lock_irq(&device_state_lock);
8          port1 = hdev->maxchild;
9          hdev->maxchild = 0;
10         usb_set_intfdata(intf, NULL);
11         spin_unlock_irq(&device_state_lock);
12         ...
13 }
14 struct usb_hub *usb_hub_to_struct_hub(struct usb_device *hdev) {
15         if (!hdev || !hdev->actconfig || !hdev->maxchild)
16                 return NULL;
17         return usb_get_intfdata(hdev->actconfig->interface[0]);
18 }
19 // drivers/usb/core/port.c
20 static ssize_t disable_store(struct device *dev, struct device_attribute *attr,
21                              const char *buf, size_t count) {
22         struct usb_port *port_dev = to_usb_port(dev);
23         struct usb_device *hdev = to_usb_device(dev->parent->parent);
24         struct usb_hub *hub = usb_hub_to_struct_hub(hdev);
25         struct usb_interface *intf = to_usb_interface(hub->intfdev);
26         ...
27 }
```

Fig. 9: Kernel source code of USB hub and USB port. The concurrent access triggers a null pointer dereference.

7 Discussion

In the following section, we will first discuss the severity of the bugs we discovered, then discuss the limitations of the preliminary measurements and our approach respectively, and finally point out future research directions.

Bug Severity. More than 60% of the newly discovered bugs are found because of modifying runtime parameters, which requires privileged permissions in most cases. However, we argue that the vulnerabilities we found are still harmful. Root user permissions can still be restricted under certain circumstances, and here are two common situations where attackers with root permissions need to further expand their capabilities:

- (1) Mandatory access control (MAC) mechanisms, like SELinux, impose additional restrictions on root users to prevent unauthorized access to objects. Bugs in Table 5 help attackers with root privileges circumvent these constraints. For example, attackers could exploit Bug#1 to bypass restriction rules in the MAC mechanism TOMOYO.
- (2) Namespaces isolate processes and limit their capabilities, which is particularly relevant in containerized environments such as Docker. Processes with root privileges inside Docker are still constrained by the namespace, and they could use Bug#14 to break the namespace's isolation.

Limitations and future work. The indirect call target collection in §3.1 is limited. There are two types of indirect call targets: function pointers explicitly defined in global variables, and function pointers passed through local variables. Our method collects the former type, but the collection of the latter is impractical due to the computing resources needed for additional inter-procedure dataflow analysis

on top of our taint analysis. However, the latter only occupies 6.3% indirect call targets, which we believe will not significantly affect the results. Furthermore, this preliminary validation does not affect our subsequent design and fuzzing experiment results.

To address the reproducibility issue in code coverage, our design establishes a connection between parameter modifications and driver execution by integrating parameter changes into driver execution through our mutation strategy. With the help of the coverage feedback mechanism, our method can effectively collect combinations of parameters and the drivers that are significantly influenced by them. An alternative approach would be to reset parameters after each test case, either by restoring a VM snapshot or by reverting modified parameters. While this method could return the system to a clean state for most test cases, it would significantly degrade fuzzing efficiency. Since runtime parameters are shared across fuzzing processes, all fuzzing threads must pause before resetting the parameters; otherwise, resetting in one thread could interfere with the execution of others. Therefore, we adopt the former approach to maximize fuzzing efficiency and leave the implementation of parameter resetting as future work.

During the implementation of this work, we identified limitations in extracting valid attribute values. To address these issues, we plan to enhance the accuracy of identifying indirect call target functions by integrating dynamic information collection methods, such as ftrace, in future work. Additionally, we aim to improve the precision of parameter analysis by concolic execution.

8 Related Work

Generic Kernel Fuzzing. In recent years, considerable research efforts have focused on kernel fuzzing to enhance its performance across various domains. Google's Syzkaller stands out as the most widely used kernel fuzzer, enabling developers to integrate syzlang continuously to extend its fuzzing capabilities for testing new functional modules. Recent advancements [6][35][11] use static analysis to recover the syscall interface syntax. Others leverage dynamic symbolic execution or even large language models to automatically generate syzlang from kernel source code [4][41]. Although existing approaches automatically generate syzlangs to test kernel drivers, they fail to consider runtime parameters, which can significantly influence code execution. Our framework identifies and integrates runtime parameters into fuzzing, modifying their values to trigger previously untested behaviors in kernel drivers.

Beyond syzlang generation, recent research has focused extensively on interrelationships between syzlangs [27][37]. HEALER [36] investigates whether executing two syscalls together yields new code coverage compared to executing them separately, while ACTOR [7] scrutinizes the memory reads and writes associated with each syscall and synthesizes use-after-free primitives. However, the above approaches still fail to establish the relationship between runtime parameters and device drivers, as modifying runtime parameters does not allocate memory objects, and exhaustively combining thousands of runtime parameters during fuzzing is impractical.

Given that operating systems commonly run in multi-core environments, recent research has made progress in identifying concurrency vulnerabilities within the kernel [16][43][15][18][40]. For example, Segfuzz [17] explores the data contention results under different execution orders by mutating the execution order of kernel code fragments. The approaches above improve fuzzing performance in various aspects, but they primarily focus on concurrently modifying the same memory location. However, the bugs discovered by SyzParam demonstrate that concurrent access to runtime parameters of different devices can also lead to vulnerabilities.

A recent study [12] selected different kernel configurations during fuzzing, achieving higher code coverage and discovering more bugs. In contrast, our work focuses on a single configuration and improves coverage by integrating runtime parameters into the fuzzing process. This provides a novel perspective, as the majority of the newly covered code is controlled exclusively by runtime parameters rather than kernel configuration options.

Kernel Driver Fuzzing. Due to the extensive volume of kernel driver code and the prevalence of vulnerabilities, significant research efforts have been dedicated to kernel driver fuzzing [34][14]. Some works [33][28] inject interrupts at the device level to trigger the driver's execution, while others [13][25] perform fuzzing on the driver with harnesses. To address the issue of a limited number of simulated devices in QEMU, Drfuzz [45] and DevFuzz [38] analyze device initialization processes and simulate interactions between devices and drivers. While these works show promising results in code coverage, they target one device at a time and overlook inter-device relationships. In contrast, SyzParam is aware of these relationships by traversing the device tree after booting, enabling it to discover more bugs caused by interactions between devices.

Static Analysis on Kernel Drivers. In addition to fuzzing, static analysis methods are extensively employed to identify vulnerabilities in operating systems [23][21][29][3]. DCUAF[1] discovers concurrent UAF between driver threads, while UACatcher[24] identifies a unique form of UAF vulnerability arising from a different order of resource release and resource use during device removal. Although these approaches help discover vulnerabilities in kernel drivers, they can only find certain types of vulnerabilities, and need manual efforts to avoid false positives. In contrast, our work found various kinds of vulnerabilities, with no false positives introduced.

9 Conclusion

In this paper, we introduce SyzParam, a novel kernel driver fuzzing framework that incorporates runtime parameters, including device attributes and kernel module parameters, into the fuzzing process. The core design of SyzParam is grounded in a comprehensive analysis of the driver model and its associated data structures. First, we propose an algorithm to extract parameters from the kernel source code along with their corresponding valid values to satisfy validation checks. Second, we establish multiple relationships to optimize the utilization of runtime parameters, including their associations with device files and the parameters of topologically connected devices. Third, we present a new mutation strategy that integrates newly generated syzlangs into the existing fuzzing framework alongside pre-existing syzlangs. This mutation strategy encompasses both syzlang relationships and execution timing considerations. We evaluated SyzParam on the latest upstream kernel, uncovering 30 new bugs, with 14 being fixed upstream, and received 10 CVE assignments so far.

Acknowledgments

We want to thank our anonymous reviewers for their valuable comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 62272442, 61902374, U1736208, the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), and the Innovation Funding of ICT, CAS under Grant No.E161040.

References

[1] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 255–268.

[2] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *Network and Distributed System Security (NDSS) Symposium*.

[3] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. 2023. Place your locks well: understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3727–3744.

[4] 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 *IEEE Symposium on Security and Privacy*.

[5] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 749–763.

[6] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[7] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. {ACTOR}:{Action-Guided} Kernel Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5003–5020.

[8] Google. 2000. The LLVM Compiler Infrastructure. https://llvm.org/.

[9] Google. 2015. Syscall description language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[10] Google. 2015. Syzkaller. https://github.com/google/syzkaller.

[11] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3262–3278.

[12] Sanan Hasanov, Stefan Nagy, and Paul Gazzillo. 2024. A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 521–533.

[13] 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*. 273–284.

[14] Sönke Huster, Matthias Hollick, and Jiska Classen. 2024. To boldly go where no fuzzer has gone before: Finding bugs in linux'wireless stacks through virtio devices. In *2024 IEEE Symposium on Security and Privacy (SP). Los Alamitos, CA, USA: IEEE Computer Society*. 24–24.

[15] Dae R Jeong, Yewon Choi, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2024. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 229–248.

[16] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[17] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2104–2121.

[18] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[19] Asim Kadav and Michael M Swift. 2012. Understanding modern device drivers. *ACM SIGPLAN Notices* 47, 4 (2012), 87–98.

[20] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2212–2229.

[21] Kangjie Lu. 2023. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1256–1270.

[22] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[23] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. 1769–1786.

[24] Lin Ma, Duoming Zhou, Hanjie Wu, Yajin Zhou, Rui Chang, Hao Xiong, Lei Wu, and Kui Ren. 2023. When Top-down Meets Bottom-up: Detecting and Exploiting Use-After-Cleanup Bugs in Linux Kernel. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2138–2154.

[25] 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*. 404–416.

[26] Patrick Mochel. 2002. The linux kernel device model. In *Ottawa Linux Symposium*. 368.

[27] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.

[28] Hui Peng and Mathias Payer. 2020. {USBFuzz}: A Framework for Fuzzing {USB} Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2559–2575.

[29] Gabriel Ryan, Abhishek Shah, Dongdong She, and Suman Jana. 2023. Precise Detection of Kernel Data Races with Probabilistic Lockset Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2086–2103.

[30] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*. 167–182.

[31] Alireza Shameli-Sendi. 2021. Understanding Linux kernel vulnerabilities. *Journal of Computer Virology and Hacking Techniques* 17, 4 (2021), 265–278.

[32] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. 2022. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *31st USENIX Security Symposium (USENIX Security 22)*. 1275–1290.

[33] 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 *NDSS*.

[34] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. 2541–2557.

[35] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. {KSG}: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 351–366.

[36] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.

[37] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. 2021. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 2741–2758.

[38] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DEVFUZZ: automatic device model-guided device driver fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3246–3261.

[39] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. 2024. MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency.

[40] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.

[41] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563* (2023).

[42] Chengfeng Ye, Yuandao Cai, and Charles Zhang. 2024. When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux. In *33st USENIX Security Symposium (USENIX Security 24)*.

[43] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing.. In *USENIX Security Symposium*. 2849–2866.

[44] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. {StateFuzz}: System {Call-Based}{State-Aware} Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3273–3289.

[45] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. 2022. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.