PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation

Zheyu Ma*
Tsinghua University, BNRist
Beijing, China
mzy20@mails.tsinghua.edu.cn

Bodong Zhao*
Tsinghua University, BNRist
Beijing, China
zbd17@mails.tsinghua.edu.cn

Letu Ren†
Tsinghua University, BNRist
Beijing, China
rlt18@mails.tsinghua.edu.cn

Zheming Li*
Tsinghua University, BNRist
Beijing, China
lizm20@mails.tsinghua.edu.cn

Siqi Ma
The University of New South Wales
Canberra, Australia
siqi.ma@adfa.edu.au

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Chao Zhang*‡§
Tsinghua University, BNRist
Beijing, China
chaoz@tsinghua.edu.cn

ABSTRACT

Linux drivers share the same address space and privilege with the core of the kernel but have a much larger code base and attack surface. The Linux drivers are not well tested and have weaker security guarantees than the kernel. Missing support from hardware devices, existing fuzzing solutions fail to cover a large portion of the driver code, e.g., the initialization code and interrupt handlers.

In this paper, we present PrIntFuzz, an efficient and universal fuzzing framework that can test the overlooked driver code, including the PRobing code and INTerrupt handlers. PrIntFuzz first extracts knowledge from the driver through inter-procedural field-sensitive, path-sensitive, and flow-sensitive static analysis. Then it utilizes the information to build a flexible and efficient simulator, which supports device probing, hardware interrupts emulation and device I/O interception. Lastly, PrIntFuzz applies a multi-dimension fuzzing strategy to explore the overlooked code.

We have developed a prototype of PrIntFuzz and successfully simulated 311 virtual PCI (Peripheral Component Interconnect) devices, 472 virtual I2C (Inter-Integrated Circuit) devices, 169 virtual USB (Universal Serial Bus) devices, and found 150 bugs in the corresponding device drivers. We have submitted patches for these bugs to the Linux kernel community, and 59 patches have been merged so far. In a control experiment of Linux 5.10-rc6, PrIntFuzz found 99 bugs, while the state-of-the-art fuzzer only found 50. PrIntFuzz covers 11,968 basic blocks on the latest Linux kernel, while the state-of-the-art fuzzer Syzkaller only covers 2,353 basic blocks.

*Institute for Network Science and Cyberspace, Tsinghua University
†Department of Computer Science and Technology, Tsinghua University
‡Zhongguancun Lab
§Corresponding author

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534226

CCS CONCEPTS

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

KEYWORDS

Fuzz, Device Driver, Interrupt

ACM Reference Format:
Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534226

1 INTRODUCTION

Due to the importance and increasing complexity of Linux, its security has drawn growing attention from academia and industry. According to the statistics of CVE [1], 5 of the top 6 products with the most vulnerabilities are all Linux distributions or variants. Among these vulnerabilities, drivers attribute the most. For example, Jeff [40] pointed out that 85% of the vulnerabilities in Android reside in drivers, and Alireza [37] showed that most of the vulnerabilities in Linux originate from drivers. Since drivers share the same address space and privilege with the core of the kernel, they become attractive targets for attackers. The attackers usually exploit driver vulnerabilities via system calls issued from user space or interrupts and malformed data issued by rogue devices or hypervisors to gain kernel privilege for launching further attacks.

Given the severity of driver vulnerabilities, various solutions have been proposed to discover them [3, 8, 21, 32, 38, 42]. As one of the most promising solutions, fuzzing can test target drivers

with the support of hardware devices. Unfortunately, there are only a limited number of real devices available for testing in practice, making fuzzers relying on real devices (e.g., [38]) unable to cover most driver codes. Recent studies proposed fuzzers relying on virtual devices (e.g., [32]), which are much more scalable and practical. However, these approaches encounter three challenges in developing virtual devices to interact with target drivers.

First, the diversity of virtual devices supported by fuzzers is limited, making many drivers unable to run. Existing solutions all rely on manual efforts to craft a limited number of virtual devices. For instance, USBFuzz [32] simulates a virtual USB (Universal Serial Bus) device, and VIA [11] simulates a handful of PCI devices based on manually provided configuration files. An automated solution to generate virtual devices is demanded to increase the diversity of virtual devices at a large scale.

Second, the functionality of simulated virtual devices is limited, making the code coverage of corresponding drivers low. According to the survey [14], the significant parts of driver code are initialization and cleanup, request handling and interrupts, configuration, power management, `ioctl`, and error handling, which account for 36%, 23.3%, 15%, 7.4%, 6.2%, and 5% respectively. Existing solutions only simulate a small number of functionalities of devices. For instance, USBFuzz [32] only focuses on plugging and unplugging operations of virtual USB devices. A fuzzer should comprehensively simulate the functionalities of virtual devices to test the code of device drivers, including initialization code and interrupt handlers.

Third, the attack surface of drivers is not thoroughly explored. Note that device drivers may accept inputs from different sources, including data requested by drivers via `copy_from_user`, data fed by users via system call arguments, and data fed by devices via MMIO (Memory-mapped I/O), PMIO (Port-mapped I/O), or DMA (Direct Memory Access). Advanced adversaries may launch attacks from any of these input sources. PeriScope [38] is the first solution that hooks DMA and MMIO operations to inject data to drivers. VIA [11] relies on the manually provided harness to coordinate. A smart fuzzer should coordinate different data injection sources.

In this paper, we design and develop `PrIntFuzz`, an efficient and universal driver fuzzing framework, to address the challenges mentioned above and test the majority of driver code, including initialization (PRobing) code and INTerrupt handlers.

First, `PrIntFuzz` utilizes static analysis to extract device information required by drivers and design a scheme to *automatically generate virtual devices*, enabling the fuzzer to test various drivers.

Second, `PrIntFuzz` develops an automated solution to *simulate major functionalities of virtual devices*, enabling the fuzzer to cover the majority code of target drivers. Specifically, `PrIntFuzz` injects software faults during driver initialization and virtual hardware interrupt signals during driver testing. Therefore, it enables the fuzzer to explore the initialization code and interrupt handlers of drivers, respectively, which account for the overlooked code in drivers.

Last, `PrIntFuzz` proposes a *coordinated multi-dimension fuzzing scheme*, which enables the fuzzer to thoroughly explore the attack surface of drivers. Specifically, `PrIntFuzz` orchestrates user interactions (i.e., system calls), device functionalities (e.g., interrupt raising, fault injection), and device data injections of different sources in a specific order. `PrIntFuzz` encapsulates them in unified system call sequences to generate high-quality test cases.

We have developed a prototype of `PrIntFuzz` and evaluated it on the latest version of the Linux kernel (5.18-rc1). It found 649 PCI drivers in the Linux kernel and successfully simulated 311 PCI devices. `PrIntFuzz` also successfully simulated 472 I2C devices and 169 USB devices. Compared with state-of-the-art (SOTA) fuzzers like Syzkaller, `PrIntFuzz` promotes code coverage, i.e., from 2,353 to 11,968 basic blocks. In addition to the code coverage, `PrIntFuzz` also outperforms existing fuzzers in bug findings. Specifically, `PrIntFuzz` has found 150 bugs in Linux kernel drivers, including 112 bugs in PCI drivers, 20 bugs in I2C drivers, and 18 in USB drivers. In a control experiment, `PrIntFuzz` found 99 bugs in the Linux kernel 5.10-rc6, while the state-of-the-art fuzzer VIA [11] found 50 bugs. We followed the responsible disclosure process and have submitted patches for some of the found bugs to the Linux kernel community. So far, 59 patches have been confirmed and merged by maintainers.

In summary, we make the following contributions to this paper:

- We design an automated solution to generate virtual devices on a large scale, which relies on device information automatically extracted by a static program analysis, enabling fuzzers to test a wide range of drivers.
- We develop an automated solution to simulate major functionalities of virtual devices, i.e., injecting software faults during device initialization and raising interrupts during driver testing, enabling fuzzers to explore more code of drivers.
- We propose a novel multi-dimension fuzzing scheme, which orchestrates user interactions, device functionalities, and device data injections in a comprehensive and coordinated way to explore the attack surface of drivers thoroughly.
- We simulated 311 PCI devices, 472 I2C devices, and 169 USB devices on the latest Linux kernel and found 150 new bugs in Linux device drivers. We will open-source our solution[1].

2 BACKGROUND AND RELATED WORK

2.1 Threat Model of Drivers

As Figure 1 shows, the attack surface of a driver consists of user-space programs and the peripherals (hardware devices), affecting the driver's control flow and data flow.

[1] https://github.com/vul337/PrIntFuzz

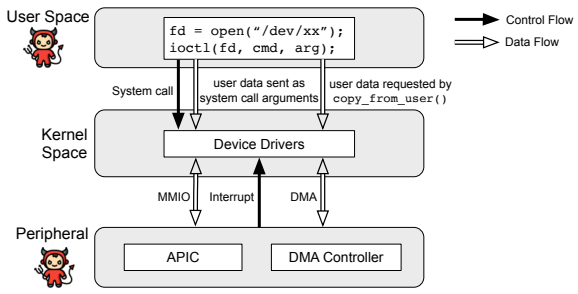

Figure 1: Attack surfaces of device drivers

Table 1: Comparison of different Linux driver fuzzing solutions.

Solutions	Platform	Device Diversity	Device Functionality	Data Injection	Coordinated Testing
Syzkaller [12]	Real/Virtual	Real devices	main	User Data	No
PeriScope [38]	Real	Network cards	Interrupt	MMIO/DMA	No
Charm [42]	Real	Real devices	main	User Data	No
USBFuzz [32]	Virtual	A generic USB device	Initialization	MMIO/DMA	No
EASIER [34]	Virtual	None	main	User Data	No
BSOD [23]	Virtual	A virtualized replay device	main	User Data	No
VIA [11]	Virtual	Manual generation of small-scale	Initialization/Interrupt/main	MMIO/DMA	Partial[1]
PrIntFuzz	Virtual	Large-scale automated generation	Initialization/Interrupt/main/Error Handling	User Data/MMIO/DMA	Yes

[1] Heavily depend on the quality of manually-provided fuzzing harness.

2.1.1 Threats from User Space. User space programs can interact with the driver through system calls (e.g., `ioctl`) and send data via system call arguments, affecting both the drivers' control and data flow. On the other hand, drivers sometimes need to request data from user space, e.g., by invoking functions such as `copy_from_user`. Hence, a malicious user could send unexpected data or system call requests to drivers, manipulating the drivers' control and data flow, then launch attacks.

2.1.2 Threats from Peripherals. Attacks from peripherals are becoming a real threat as cloud services become popular. A malicious attacker could take complete control of the hypervisor through a vulnerability in the cloud service vendor's hypervisor and, in turn, could take control of the devices in the hypervisor and use them to carry out attacks on other guests in the cloud.

Peripherals may interfere with the control flow of drivers by raising interrupts. If the driver does not disable the interrupts, the interrupts may preempt the processor when the driver is running.

Peripherals also have abundant data exchange with drivers, e.g., MMIO, PMIO, or DMA. A malicious peripheral may not abide by the agreement with the driver and send malicious data to the driver, resulting in undefined behavior of drivers [24, 32]. For example, some programmable USB devices (such as FaceDancer [9]) may exploit vulnerabilities in the USB device driver to launch attacks. Alternatively, devices with MMIO or DMA functions may also send malicious data, as shown by previous Linux kernel commits [19, 20].

2.2 Finding Bugs in Linux Drivers via Fuzzing

According to the study [37], the driver subsystem is one of the most vulnerable subsystems in the kernel. So researchers have proposed many solutions to test drivers, including symbolic execution [18, 31, 35], static analysis [3, 43, 48], and fuzzing [8, 15, 17, 26, 34, 38]. Fuzzing has become one of the most popular methods to find bugs due to its efficiency and scalability. Table 1 lists the state-of-the-art fuzzing solutions for Linux device drivers.

The most popular kernel fuzzing solution is Syzkaller [12], which tests the Linux kernel via system calls such as `open`, `close`, `read`, `write`, and others. The fuzzing engine of Syzkaller will generate system call sequences and mutate them according to the predefined system call templates which describe arguments of system calls. It requires much manual effort to write system call templates for drivers due to their complexity.

PeriScope [38] mainly focuses on MMIO and DMA interfaces of drivers and presents a dynamic analysis framework to monitor the interaction between the driver and the hardware. V-Shuttle [29] also focuses on device-driver interaction. It solves the problem of nested DMA structures and performs semantics-aware hypervisor fuzzing.

USBFuzz [32] also pays attention to the hardware attack surface. USBFuzz uses an emulated USB device to feed random data to the USB drivers and applies coverage-guided fuzzing. In each fuzzing iteration, USBFuzz continuously performs plugging and unplugging operations on the virtual device to discover bugs in the initialization code. Though USBFuzz can test the USB subsystem effectively, the bugs discovered by USBFuzz are mainly about the USB descriptor.

Charm [42] runs drivers in a virtual machine for scalability and enables them to interact with the hardware device. However, Charm requires lots of manual work to port the driver and could not fully port the driver's functionality.

EASIER [34] observes that many bugs in drivers are superficially dependent on both the hardware and the software, so it is no need to simulate the whole device to perform fuzzing. It performs a dynamic analysis framework that addresses the dependencies present in the driver to load drivers successfully and then fuzz them.

BSOD [23] utilizes an emulated device to replay the driver traces recorded with real attached devices and could fuzz binary-only drivers in virtual machines.

VIA [11] assumes that the hypervisor is malicious and presents the analysis of the driver interface under a new threat model. However, VIA relies heavily on manually written harnesses and virtual device configuration files, so it is greatly limited in scalability. In addition, VIA relies on the LKL [33], which causes it not to keep track of the latest changes to the kernel.

In addition to the above-mentioned direct fuzzing methods, other research aims to assist fuzzing in variant ways. DIFUZE [7] proposed an automatic method to extract system call interfaces from the source code of the Linux device drivers and use them to fuzz drivers. Agamotto [39] presents an effective checkpoint management policy to skip code snippets repeatedly tested by fuzzers, improving the performance of device driver fuzzing.

2.3 Finding Bugs in the Kernel via Fuzzing

The kernel is an attractive target for attackers and security researchers, so they have developed many tools to fuzz it [5, 10, 12, 18, 27, 36, 41, 46].

SyzVegas [46] uses multi-armed-bandit algorithms to adapt the task selection and seed selection algorithms in Syzkaller, improving the performance of Syzkaller. HEALER [41] utilizes system call relation learning to improve the effectiveness of fuzzing. kAFL [36] uses Intel-PT to collect coverage from the processor and performs coverage-guided fuzzing. HFL [18] combines fuzzing with symbolic execution for hybrid kernel fuzzing. IMF [10] discovers deep kernel

bugs by leveraging inferred dependence model between kernel API functions. MoonShine [27] performs static analysis to detect dependencies between system calls and collects system call traces from real-world programs for distilling seeds. Digtool [30] designs a hypervisor to catch kernel behaviors dynamically and leverages logs to discover kernel vulnerabilities. NTFuzz [5] utilizes static binary analysis to infer the system call types and performs type-aware fuzzing on Windows.

However, these tools are designed for general kernel fuzzing, so they cannot fuzz device drivers effectively.

2.4 Finding Bugs in Linux Drivers via Static Analysis or Symbolic Execution

Besides discovering vulnerabilities in drivers via fuzzing, security researchers also propose some useful static analysis tools [2–4, 21, 22, 28, 48] and symbolic execution tools [35].

SADA [3] recognizes both unsafe streaming DMA access and coherent DMA access in device drivers. DCUAF [2] takes a local-global analysis to extract function pairs that may be executed concurrently and performs a lockset analysis to detect concurrent use-after-free bugs. DSAC [4] uses a heuristics-based method to extract functions that may be sleeping at runtime to detect sleep-in-atomic-context bugs. HERO [48] finds disordered error handling bugs by pairing error-handling functions based on their unique structures. EECATCH [28] can detect exaggerated error handling bugs, which are wrong error handling consequences worse than the error itself. CRIX [21] is designed to detect missing-check bugs in drivers. DR.CHECKER [22] performs a soundy static analysis to find bugs with pointer analysis and taint analysis. However, static analysis has its shortcomings. First, manual analysis is needed to determine whether bugs reported by the tool are actual bugs. Second, static analysis is usually heuristics-based, so the manual effort is also needed to formulate rules for a specific type of bug.

SymDrive [35] removes the need for hardware via symbolic execution. However, besides the common problems of symbolic execution, SymDrive also requires manual annotation of the driver code and manual configuration.

3 DESIGN

`PrIntFuzz`, our detection system for exploring initialization code and interrupt handlers in Linux device drivers, utilizes driver information extracted by static analysis to specify the drivers and then fuzz them with multi-dimension. `PrIntFuzz` executes a three-phase workflow for driver analysis and fuzzing (see Figure 2).

First, in the *Virtual Device Modeling* phase, `PrIntFuzz` models the virtual device from two aspects. `PrIntFuzz` first extracts necessary driver information, e.g., data expected by the driver and the memory and configuration space, through inter-procedural flow-sensitive, path-sensitive, and field-sensitive static analysis, and then uses this information to create a large number of virtual devices for the drivers to prepare for fuzzing (§ 3.1.1). The other is the system call templates containing various descriptions of system calls. Fuzzer will utilize these templates to generate system call sequences for interacting with device drivers (§ 3.1.2).

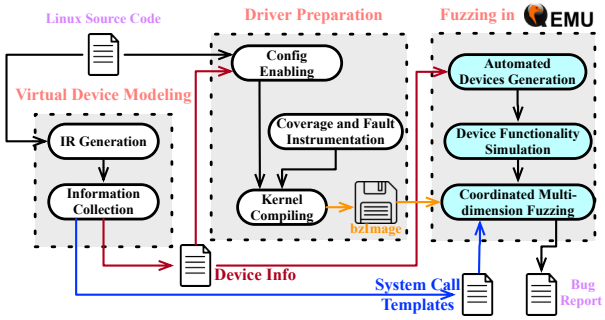

Figure 2: The framework of `PrIntFuzz`

Second, in the *Driver Preparation* phase, `PrIntFuzz` enables the device drivers by creating a specific configuration file and compiling the kernel since drivers can be tested only when compiled into the kernel (§ 3.2.1). `PrIntFuzz`'s kernel instrumentation collects code coverage and performs fault injection (§ 3.2.2).

Third, in the *Fuzzing in QEMU* phase, `PrIntFuzz` first creates virtual devices (§ 3.3.1) and then simulates some of the device functionalities (§ 3.3.2), finally conducts multi-dimension fuzzing from user space programs and the hardware (§ 3.3.3).

3.1 Virtual Device Modeling

3.1.1 Driver Information Collection. We found an initialization process before the driver is ready to interact with the user space program. The driver performs some sanity checks on the hardware device during the initialization process to determine whether it works properly. Therefore, an essential part of virtual device modeling is to let the virtual device pass the driver's checks and prepare for the normal interaction.

`PrIntFuzz` takes the PCI driver as an example of analyzing the driver since PCI drivers account for 36% of all Linux drivers [14]. However, the methods shown by `PrIntFuzz` can also be applied to drivers under other buses, which will be discussed later. `PrIntFuzz` models virtual devices in three dimensions: data space, I/O and memory space, and configuration space through inter-procedural field-sensitive, path-sensitive, and flow-sensitive analysis.

Data Space Modeling. During the initialization process, the driver will read values from the hardware registers and perform sanity checks. If the data provided by the hardware does not meet the constraints, the driver will report an error, as Listing 1 shows.

```
1  status = he_readl(he_dev, RESET_CNTL);
2  if ((status & BOARD_RST_STATUS) == 0) {
3      hprintk("reset failed\n");
4      return -EINVAL;
5  }
```

Listing 1: drivers/atm/he.c

For this type of constraint, `PrIntFuzz` first collects the functions with reading functionality provided by the Linux kernel (such as `readl`) from the official document [47]. Some drivers will implement their read functions as the wrapper of built-in functions, and `PrIntFuzz` also identifies such functions using the following rules:

- The function name contains "read".
- The number of basic blocks of the function is less than five.
- The function calls the built-in functions capable of reading.

Then PrIntFuzz tries to infer the proper value that the hardware register should be based on several patterns. The most common pattern is shown in Listing 1: the driver reads the input from the hardware into a variable, performs a bitwise operation with a mask, and finally checks the result of the operation. To solve such a constraint, PrIntFuzz performs a static data-flow analysis. PrIntFuzz traces the def-use chain of the variable that holds the read value, then determines whether its user is a bitwise operation. Then, PrIntFuzz continues to trace the result of the bitwise operation to determine whether an icmp instruction uses it. If so, it extracts the constant operand of the icmp instruction, which is the proper value that PrIntFuzz finds.

The other patterns are similar, except that other operations are made on the read value, such as comparing it directly with a constant, doing a bitwise operation with a mask after a shift operation, or comparing it after a bitwise expansion, which can be handled similarly as the above methods.

However, it is not enough to solve the constraint. PrIntFuzz also needs to determine whether the constraint should be satisfied, i.e., PrIntFuzz should be path-sensitive to determine whether satisfying the constraint will allow the probing process to continue or exit. In Listing 1, the driver will exit if the constraint is satisfied. Therefore, PrIntFuzz should make the result of status & BOARD_RST_STATUS not zero to pass the check.

It is worth mentioning that since the driver reads the hardware registers in a specific order, such analysis should also be flow-sensitive. Otherwise, the driver would read out-of-order data and fail to probe. Hence, PrIntFuzz sorts the constraint solving results in the order the driver calls the read-related functions.

I/O and Memory Space Modeling. Each PCI device can implement up to six I/O address regions consisting of memory or I/O addresses. The driver checks the region type and returns an error if it does not match, as shown in Listing 2.

```
    if (!(pci_resource_flags(dev, 0) & IORESOURCE_IO))
        return -ENODEV;
```

Listing 2: drivers/i2c/buses/i2c-amd8111.c

PrIntFuzz first identifies the relevant macros or functions (such as pci_resource_flags) to pass such a check. Then PrIntFuzz extracts the position of the region being checked (zero in this example) and the type of the resource (IORESOURCE_IO in this example). Finally, PrIntFuzz record the position of the region and the corresponding resource type in the final analysis result.

Configuration Space Modeling. PCI devices contain several configuration registers that hold configuration information about the hardware, and drivers can read or write to these registers [6].

Among these configuration registers, five standard registers are critical: vendorID, deviceID, class, subsystem vendorID, and subsystem deviceID. These registers are read by the PCI bus when scanning for devices to determine which driver this hardware should match. Correspondingly, on the driver side, the pci_device_id structure is provided to define the list of different types of devices supported by this driver [25], as shown in Listing 3.

This structure is a field of the structure pci_driver. PrIntFuzz performs a field-sensitive static analysis to identify the structure pci_driver and further extract pci_device_id from the driver.

```
    struct pci_device_id {
        __u32 vendor, device; /* Vendor and device ID or PCI_ANY_ID*/
        __u32 subvendor, subdevice; /* Subsystem ID's or PCI_ANY_ID */
        __u32 class, class_mask;  /* (class,subclass,prog-if) triplet */
        kernel_ulong_t driver_data; /* Data private to the driver */
    };
```

Listing 3: Device ID structure

In addition to the several configuration registers mentioned above, the driver may also check other non-standard registers, as shown in Listing 4.

```
    pci_read_config_dword(pdev, 0x80, &reg);
    if (reg != ADM8211_SIG1 && reg != ADM8211_SIG2) {
        printk("%s : Invalid signature (0x%x)\n", pci_name(pdev), reg);
        err = -EINVAL;
        goto err_disable_pdev;
    }
```

Listing 4: drivers/net/wireless/admtek/adm8211.c

For this type of check, PrIntFuzz first collects relevant functions (such as pci_read_config_xxx) in the driver, then solves for the register values to be checked in these functions with constraints similar to the data space constraints, and saves the address of the register and the value that this register should be.

Although PrIntFuzz analyzes each of the three aspects separately, modeling a virtual device requires a combination of information from all these three aspects.

Also, since PrIntFuzz infers the proper value using some heuristic rules, it cannot be applied to all cases.

3.1.2 System Call Templates Generation. In addition to modeling the virtual devices by extracting the necessary information of drivers, PrIntFuzz also analyzes the driver's interfaces to generate the system call templates.

PrIntFuzz amends DIFUZE [7] to make it work on the latest kernel and collect system calls from driver interfaces. In general, user-mode programs can only interact with drivers via specific system call interfaces, and these interfaces are usually stored in the structures of the program. Hence, PrIntFuzz first identifies all structures with unique strings (such as operations or ops) in their names and then extracts function pointers from the structures and matches function names with names of the system call interfaces.

3.2 Driver Preparation

3.2.1 Config Enabling. The drivers should be enabled in the configuration of the kernel before fuzzing. Although allyesconfig and menuconfig are provided to configure all the drivers or manually select drivers, they cannot automatically configure a large number of specific drivers.

PrIntFuzz takes the following two steps to enable the drivers in the kernel configuration. First, PrIntFuzz recursively searches the files under the directory where the driver code is located to obtain Makefile, which contains the config name of the driver. Second, PrIntFuzz uses kconfiglib [44] to parse Kconfig and find other kernel options that the driver depends on. Then, PrIntFuzz enables them together in the kernel configuration file to enable the driver.

3.2.2 Coverage and Fault Instrumentation. Inspired by FIZZER[13], PrIntFuzz implements a fault injection method to inject software fault in the driver initialization code. Fault injection aims to test

the error handling code of the driver's initialization code, which is rarely tested in the normal execution.

The first step of injecting software fault is to recognize the possible error sites of the driver, whose value will affect the return value of the initialization function. PrIntFuzz considers the functions that satisfy the following three conditions as potential error sites.

- The type of function's return value is a pointer or an integer.
- The return value of the function is checked in a conditional statement.
- The function is a declaration rather than a definition. That is, PrIntFuzz only does the fault injection on the API function of the kernel.

After recognizing the error sites, PrIntFuzz replaces the original function with a wrapper function. This wrapper function checks whether the current function should be injected with a fault dynamically, i.e., returning a value indicating an error.

After instrumenting the initialization code, PrIntFuzz uses a user agent to test the corresponding driver, which is done by repeatedly loading/unloading the driver with different fault positions.

3.3 Fuzzing in QEMU

3.3.1 Automated Virtual Devices Generation. Based on the driver information collected in § 3.1.1, PrIntFuzz performs automated device generation for subsequent matching with drivers and further testing of drivers. All virtual devices are generated according to a basic template, as shown in Listing 5. PrIntFuzz fills in the critical information previously extracted from the driver to the place reserved of the device template (Capital letters in Listing 5).

```
void init(PCIDevice *pdev)
{
    pdev->vendor_id = VENDOR_ID;
    pdev->device_id = DEVICE_ID;
    ...
    pdev->buf = PCI_DATA;
    pci_set_config(pdev, PCI_CNFIG_POS, PCI_CONFIG_VALUE);
    pci_register_bar(pdev, PCI_REGION_POS, PCI_REGION_TYPE);
    ...
}
```

Listing 5: Basic device template

Although such a generated virtual device cannot fully simulate a real peripheral, it already has most functionalities to interact with the driver. First, it can be matched with the driver, and the user space program can interact with the driver normally after the initialization. Second, the virtual device can also actively send interrupt signals to the driver and trigger the interrupt handlers. Third, the simulation of MMIO and DMA (§ 3.3.2) can inject the fuzzer-generated data into the corresponding driver.

3.3.2 Device Functionality Simulation. Since PrIntFuzz does not need to fully simulate the behavior of the virtual devices but only needs to ensure that they work normally, PrIntFuzz minimizes the functions of the virtual devices and retains the essential functions such as handling MMIO, DMA, and interrupts.

The most straightforward way of injecting data into the MMIO region used by virtual devices is to fill the whole MMIO region with data generated by the fuzzer. However, such an approach will make a large proportion of data useless since drivers usually access needed data instead of the whole mapped memory.

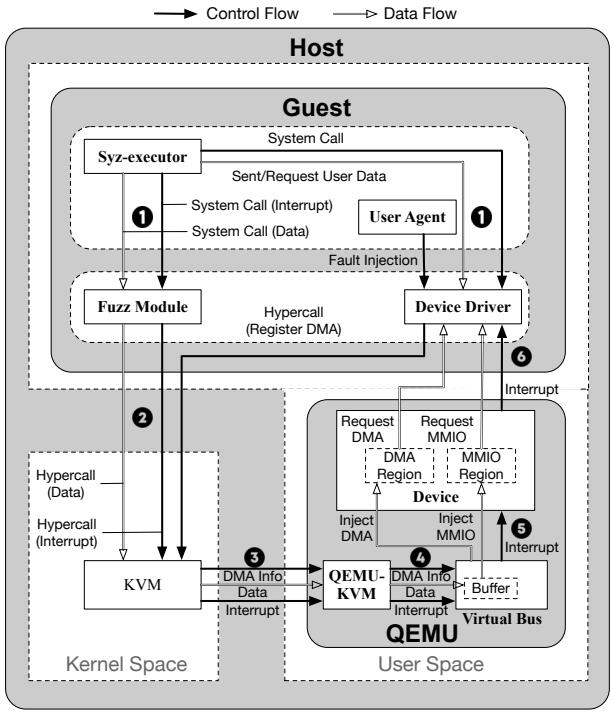

Figure 3: Multi-dimension fuzzing

Thus, PrIntFuzz presents a sequential I/O model to feed data to virtual devices. Regardless of which register the driver starts reading from, the virtual device will always return the first byte of the buffer and then move the buffer pointer to the next byte until the driver finishes reading. More, the virtual device ignores all write operations from the driver. Since the DMA region is usually contiguous, PrIntFuzz reserves a special memory region in each device to store the DMA data.

Although PrIntFuzz can successfully simulate most virtual devices' functionalities, such virtual devices cannot handle all cases, such as network congestion.

3.3.3 Coordinated Multi-dimension Fuzzing. After generating the virtual devices, PrIntFuzz conducts multi-dimension fuzzing, as shown in Figure 3.

PrIntFuzz uses Syzkaller to generate system call sequences based on the descriptions restored in the system call templates. Then the executor interacts directly with the driver via standard system calls and the fuzz module using two special system calls containing interrupt signals and fuzzer-generated input (❶). Next, the fuzz module in the guest kernel space raises hypercalls and sends specific data (i.e., interrupt signals and fuzzer-generated data) to KVM (❷). After handling the hypercalls from the guest, the KVM quits to QEMU-KVM (❸), which is responsible for sending the interrupt signal to the virtual bus or the generated data to the buffer of the virtual bus, respectively (❹). Referring to PCI virtual device names, the virtual bus traverses the virtual devices and matches their name of them. If a device name is matched, the virtual bus then triggers the corresponding interrupt line of the device and injects fuzzer-generated data to the DMA region or MMIO region of the virtual device (❺). Consequently, the interrupt

handler of the driver will be executed after the interrupt is triggered (❻). PrIntFuzz collects the drivers' coverage and feeds it back to the fuzzer for the next iteration.

Fault Injection of the Initialization Code. Referring to the instrumented code, PrIntFuzz uses a user agent to inject software faults to the initialization code. In particular, the user agent repeatedly loads/unloads the drivers, injects fault into different function calls, and executes different error handling codes each time.

Fuzzer-generated data injection. To simulate the attack surface of the hardware input, PrIntFuzz monitors the interactions between the device and the driver, and injects data into the driver as the devices sent it. PrIntFuzz mainly focuses on two types of hardware interfaces: MMIO and DMA.

The fuzzer first generates the system calls for data injection. Then, PrIntFuzz modifies the generation and mutation algorithm of the system call sequence, which inserts system calls of data injection before each normal system call. Such a system call sequence ensures that the driver can read the latest generated data during each system call. After generating data, the fuzzer sends the generated data to the fuzz module in the guest kernel via system calls, and the fuzz module initiates hypercalls to forward the data to KVM. Then the data are processed separately based on the type.

For MMIO interfaces, PrIntFuzz copies the fuzzer-generated data into the MMIO region of each device when the driver reads data from the device. When processing DMA interfaces, PrIntFuzz dynamically identifies the mapping addresses. When a driver registers a DMA region, the registration function initiates a hypercall to send DMA registration information to KVM, which forwards it to QEMU. The QEMU further fills in the information to the corresponding DMA entry of the device. The QEMU fills in the corresponding DMA region of the device according to the registration information when receiving the fuzzer-generated data.

Interrupt injection. VMM (Virtual Machine Monitor) could control the virtual machine's behavior. PrIntFuzz could utilize this feature to inject the interrupt signal to virtual devices.

In PrIntFuzz, the interrupt signal is wrapped as a system call. The fuzzer generates the interrupt signal based on the corresponding system call template and normally schedules the interrupt system call like any other system call. Then the fuzz module will forward the interrupt signal sent by the fuzzer to the host's KVM module. After KVM in the host forwards the signal to QEMU, PrIntFuzz traverses the devices attached to the PCI bus and searches the name on this list. If any of the devices match, the virtual bus will inject an interrupt signal to the virtual device regardless of whether the virtual device has a traditional interrupt or MSI (Message Signaled Interrupts) interrupt.

4 IMPLEMENTATION

4.1 IR Generation

In order to analyze the driver comprehensively, PrIntFuzz operates on an intermediate representation, i.e., LLVM IR, of the driver code. Specifically, PrIntFuzz uses scripts to hook the kernel complication process to generate LLVM bitcode files. For drivers that have multiple source files, PrIntFuzz hooks the linking process to generate a final bitcode file instead of multiple individual files.

```
+static int curr_pos = 0;
+module_param(fault_pos, int, -1)
+
+int foo_f1() {
+    curr_pos += 1;
+    if (fault_pos == curr_pos) {
+        return -1;
+    } else {
+        return f1();
+    }
+}
+
+void *foo_f2() {
+   curr_pos += 1;
+   if (fault_pos == curr_pos) {
+       return NULL;
+   } else {
+       return f2();
+   }
+}
+
int xxx_probe() {
-   if (f1()) {
+   if (foo_f1()) {
        return -EINVAL;
    }

-   if (f2() == NULL) {
+   if (foo_f2() == NULL) {
        return -ENOMEM;
    }
}
```

Listing 6: Fault instrumentation

Further, PrIntFuzz adds the `-O0` flag to the IR generation process to turn off compiler optimization for the accuracy of the analysis.

4.2 Coverage and Fault Instrumentation

In order to perform fault injection on the initialization code of the driver, PrIntFuzz should instrument the initialization code especially. PrIntFuzz uses the previously proposed heuristic rules to identify potential error sites and then replaces these functions with the `foo` function, as Listing 6 shows. For the different cases where the return value of the error site is an integer or a pointer, PrIntFuzz replaces them with functions that have the same original type of return value.

In the `foo` function, `fault_pos` is a module parameter that can be assigned when loading the driver and is used to indicate the function in which fault injection is performed. `curr_pos` is a global variable that is increased by one each time the `foo` function is executed. Fault injection is performed only when `curr_pos` and `fault_pos` are equal. Otherwise, the original function is executed.

With such instrumentation, it is possible to control the location of the fault injection when loading the driver. The user agent specifies a different `fault_pos` in each iteration, allowing the driver to fail at a different function. Therefore, `fault_pos` will be reset after each iteration.

Furthermore, PrIntFuzz enables KCOV instrumentation for coverage feedback and KASAN [16] for bug detection.

4.3 Coordinated Multi-dimension Fuzzing

PrIntFuzz implements and modifies the following components to perform coordinated multi-dimension fuzzing.

Kernel Module. PrIntFuzz implements a kernel module to send hypercalls to KVM. The hypercall function prototype is *static inline long kvm_hypercall1(unsigned int nr, unsigned long p1).* The first

Table 2: The experiment setup for comparison of bugs and code coverage with Syzkaller

No.	**(D)**Default Devices	**(E)**Extra Devices	**(I)**Interrupt
1	139	0	×
2	139	0	✓
3	139	281	×
4	139	281	✓

parameter of this function is a magic number that indicates the function of the hypercall. The second parameter is the value that PrIntFuzz wants to pass to the KVM.

KVM. The KVM module is responsible for handling the hypercall from the virtual machine. PrIntFuzz implements two hypercalls. One is for injecting interrupts to virtual devices. The other is for injecting data into virtual devices. PrIntFuzz also adds two magic numbers, as mentioned above. After the virtual machine's system call enters the KVM of the host machine, the KVM will actively exit and hand it over to QEMU for processing. QEMU will decide whether to inject interrupts or data according to the corresponding magic number.

KCOV. Syzkaller specially developed the KCOV [45] mechanism for collecting the coverage in the kernel. However, KCOV only supports collecting coverage from the softirq. It does not support collecting coverage from the hardware interrupt handler, nor does it support collecting the coverage in the initialization code. To collect the coverage of the interrupt handler, PrIntFuzz modifies the KCOV to support collecting the coverage from the hardware interrupt. PrIntFuzz uses hardware features of Intel-PT to collect coverage during the initialization process.

5 EVALUATION

To assess the performance of PrIntFuzz, we evaluated PrIntFuzz by answering the following five research questions (RQs):

- **RQ1: How well does PrIntFuzz simulate devices?**
- **RQ2: How good is PrIntFuzz at finding bugs?**
- **RQ3: How good is PrIntFuzz at exploring code?**
- **RQ4: How scalable is PrIntFuzz?**
- **RQ5: What typical bugs were found by PrIntFuzz?**

We performed our evaluation on a personal computer with 16 physical cores and the CPU model Intel(R) Core(TM) i9-12900KS, along with 128GB of RAM. The kernel's KVM module has been patched to support the hypercalls PrIntFuzz added. The OS is Ubuntu 20.04.4 LTS with Linux kernel version 5.4.0.

5.1 Experiment Setup

All experiments compared with Syzkaller in the evaluation are performed in the following four groups, as shown in Table 2. We group the experiments from two perspectives, one is whether to use extra simulated devices, and the other is whether to enable interrupt injection.

The virtual devices used in the experiments can be divided into two categories: QEMU default devices and the extra devices simulated by PrIntFuzz. This comparison allows us to analyze the impact of the increase of simulated devices on the results. In the experiments, for the sake of the efficiency of the experiments, we only enable the system call interfaces identified by PrIntFuzz in fuzzing instead of all of the system calls supported by Syzkaller. We use **D** for Default Devices (i.e., the original combination of Syzkaller and QEMU), **E** for Extra devices, and **I** for Interrupt in the results.

To avoid interference between drivers, we only fuzz one device driver at a time. Since the interfaces of a single driver are limited, we fuzz the driver for only one hour, which can also ensure the adequacy of fuzzing a single driver.

Table 3: The total number of PCI drivers and the number of PrIntFuzz simulated virtual devices

Type	Total	PrIntFuzz	Success Rate
net	164	71	43.3%
others*	147	62	42.2%
sound	58	17	29.3%
ata	56	53	94.6%
scsi	47	31	66.0%
media	34	16	47.1%
video	29	17	58.6%
i2c	17	9	52.9%
misc	16	9	56.3%
usb	14	8	57.1%
edac	12	5	41.7%
atm	11	3	27.3%
tty	11	10	90.9%
Total	616	311	50.5%

* All other types with less than 10 drivers.

5.2 Virtual Device Modeling

5.2.1 Matching with Drivers on the Latest Kernel. On the Linux kernel 5.18-rc1, PrIntFuzz extracted 649 device IDs corresponding to the PCI bus. PrIntFuzz has successfully simulated 311 PCI devices, with a success rate of 50.5%[2]. The results are shown in Table 3.

Different types of drivers vary greatly. For example, the high success rate of device matching for tty and ata means that the corresponding drivers are less checked or checked more easily during initialization. In contrast, drivers of sound or atm have a higher failure rate, indicating that such drivers are complex and the existing method is challenging to succeed.

We manually verified some of these drivers and broadly grouped the reasons for the failure to match with drivers into the following categories.

- **Complex Data Dependency:** Though PrIntFuzz uses static data flow analysis to infer the possible values of the registers, it cannot be applied to particularly complex cases. Some constraints are very complex, and there are even checks between constraints. PrIntFuzz cannot infer the values in such cases, resulting in some drivers failing to match.
- **Dynamic Configuration Requirements:** PrIntFuzz only handles the case where the configuration registers are constants. If the argument of the functions that can read configuration registers is a variable, PrIntFuzz cannot infer the corresponding position of the configuration.
- **Additional Devices Requirements:** Some drivers detect the presence of another specific PCI device in the system during initialization and return a failure if this device does not exist. PrIntFuzz does not currently handle such dependencies between devices.

[2]Some drivers crash during the probing process, and we could not determine whether the matching is successful, so we do not count this part of the drivers in the total.

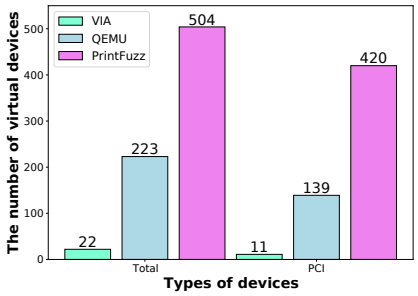

Figure 4: Comparison with QEMU and VIA of the number of simulated devices, including the total number of devices and the number of PCI devices

- **Firmware Requirements:** During the probing process, drivers may need to load the firmware, and some of the firmware is hard to obtain.

Although PrIntFuzz mainly focuses on drivers under the PCI bus, extending the method to other drivers (e.g., USB drivers or I2C drivers) is not hard. We will discuss the scalability of PrIntFuzz in Section 5.5.

5.2.2 Comparison with QEMU and VIA [11]. We compare the number of devices that PrIntFuzz can simulate with QEMU and VIA, as shown in Figure 4. The number of virtual devices supported by PrIntFuzz has increased considerably from the original QEMU. On the other hand, VIA does not simulate devices through QEMU but with the help of LKL [33] and requires handwritten device configurations.

As shown in Figure 4, VIA can simulate 22 devices, including 11 PCI devices, QEMU can simulate 223 devices, including 139 PCI devices, and PrIntFuzz supports another 281 (=420-139=504-223) PCI devices based on QEMU. VIA could be extended to support more virtual devices but requires much manual effort to write configuration templates. On the other hand, PrIntFuzz supports 2X more PCI devices than QEMU.

Moreover, PrIntFuzz could simulate 311 (=281+30) PCI devices in total based on QEMU. In fact, PrIntFuzz could also simulate some of the 139 PCI devices supported by QEMU but does not take all of them into consideration due to two reasons. First, PrIntFuzz only tries to simulate devices for drivers that exist in the Linux kernel, while QEMU has some virtual devices for specific usage. Second, PrIntFuzz only simulates one virtual device for one driver, while QEMU may provide multiple virtual devices for one driver. For instance, QEMU provides multiple variants of NIC (Network interface controller) devices, all of which correspond to the same driver but with different parameters.

5.3 Bug Finding in PCI Drivers

5.3.1 Overview. At the time of writing, PrIntFuzz has found 112 unique bugs of PCI drivers in the Linux kernel that were not found by other fuzzers, and 50 of them are patched so far.

As Table 4 shows, PrIntFuzz has found many different bugs across Linux PCI drivers, including null-pointer-dereference, use-after-free, out-of-bounds and double free, and other types of bugs.

Table 4: Bugs found by PrIntFuzz in PCI drivers

Type	Location			Number
	Main	Init	Interrupt	
Null-Pointer-Dereference	1	13	9	23
Divide-by-Zero	15	2	0	17
Use-After-Free	1	10	0	11
Page Fault	2	1	2	5
Out-of-Bounds	1	1	1	3
Wild-Memory-Access	1	0	0	1
Schedule-while-Atomic	0	0	1	1
Sleep-in-Atomic	0	1	0	1
Deadlock	1	0	0	1
Double Free	0	1	0	1
Warning	2	38	2	42
Assertion Failure	2	4	0	6
Total	26	71	15	112

The null-pointer-dereference bugs occur in the interrupt handler and initialization code. The interrupt handler often handles external events according to the state of the driver. Besides, the interrupt handler checks the state of the driver by reading from a specific register from the device. So if the device provides an invalid status to the driver, the driver may access variables that are not initialized, which causes a null-pointer-dereference bug.

Many divide-by-zero bugs are mainly located in the frame buffer drivers. These drivers do not check the user input carefully, and when the user does not provide a specific item of a structure, a divide-by-zero bug can occur.

The use-after-free bug often occurs in a device driver's initialization and cleanup function. Some drivers do not handle the order of releasing resources correctly or release resources that are being used, which can cause a use-after-free bug.

Warnings found by PrIntFuzz mainly exist in the initialization code of the device driver. The initialization functions return 0 for success and a negative value for failure. However, some drivers return positive values when encountering errors, which may confuse the kernel. The kernel will treat the positive value as success, thus assuming the device has matched with the driver, although the truth is the opposite. There are also drivers that do not properly release the requested resources, resulting in warnings.

Other types of individual bugs found by PrIntFuzz will not be discussed here, and we will give a detailed analysis of some bugs in the case study (§ 5.6).

The bugs found by PrIntFuzz are derived from repeated experiments, and each of them is reproducible. All the bugs mentioned in the paper have been reproduced and verified manually.

5.3.2 The Distribution of Found Bugs. We also evaluate the ability of different methods to find bugs, as shown in Figure 5. The figure shows that the initialization code has the most bugs and the fault injection method helps discover the bugs. Furthermore, the bugs found in the interrupt handler and other parts of the driver also prove that the multi-dimension fuzzing proposed by PrIntFuzz is effective. The figure also shows that some bugs need to be triggered by both the initialization process and interrupt handlers.

5.3.3 Comparison with Syzkaller and VIA. To compare the ability of PrIntFuzz with Syzkaller and VIA to discover bugs, we performed

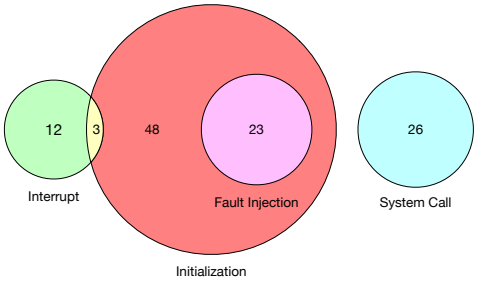

Figure 5: The distribution of bugs found in the PCI drivers

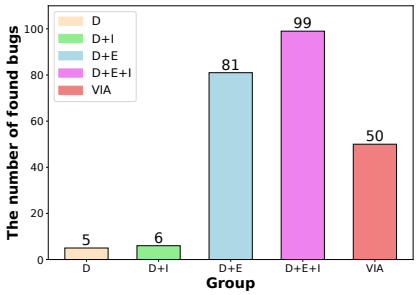

Figure 6: Comparison of found bugs on the Linux kernel 5.10-rc6 with different configurations

three trials of the experiments on Linux 5.10-rc6 [3]. The experiment setup is the same as described in section 5.1, and the results are shown in Figure 6.

As we can see from Figure 6, PrIntFuzz discovered 94 more bugs than Syzkaller and 49 more bugs than VIA [4]. The scenario of injecting interrupts is valid even if no extra emulated devices are used, and some bugs are discovered in drivers of QEMU default devices. After using extra emulated devices, the ability of bug discovery has improved a lot.

5.4 Code Coverage

In addition to the ability to find bugs, we evaluated PrIntFuzz's improvement in code coverage compared with Syzkaller on the Linux kernel 5.18-rc1.

The experiments are still set up as described in section 5.1, and the average results of the three trials are shown in Figure 7. We only instrumented the kernel's drivers and sound directories in this experiment. This experiment wants to show more about the improvement of PrIntFuzz for initialization code and interrupt handler coverage, so we need to avoid counting extraneous code, such as coverage in drivers/core, which is often covered in the usual Syzkaller fuzzing. Specifically, we added a baseline for each group of the experiments, which uses the Syzkaller's default system call templates for fuzzing without adding additional templates or virtual devices. By adding such a baseline for each group of experiments, we can exclude the interference of some generic code.

[3]Since the latest version of Linux is not net supported by the LKL on which VIA relies, we can only compare the bug findings on the latest version supported by VIA.
[4]The number of bugs found by VIA comes from the original paper.

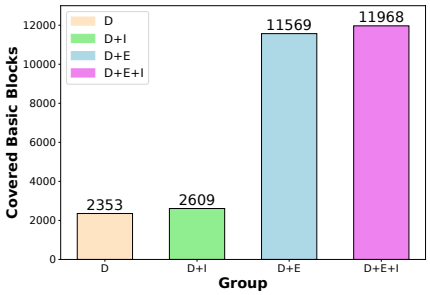

Figure 7: Comparison of basic block coverage on the Linux kernel 5.18-rc1 with different configurations

This method presents us with a clearer picture of the coverage improvement of PrIntFuzz on related code.

From the first two groups of experiments, we could learn that the combination of Syzkaller and QEMU can reach very little code even with the addition of interrupt injection. After adding the extra devices simulated by PrIntFuzz, the coverage achieves a significant improvement, showing that PrIntFuzz has a wide range of tests and can effectively assist Syzkaller in improving the coverage. Since PrIntFuzz extends the number of devices supported by QEMU, more code can be triggered by the interrupt injection, proving that the interrupt injection approach effectively improves coverage.

In addition to the coverage comparison with Syzkaller, we also compare PrIntFuzz with Agamotto [39] and VIA [11] on Linux kernel 5.10-rc6, as shown in Table 5. While PrIntFuzz simulates a wide range of drivers, this is not the case for Agamotto and VIA. Our comparison can only be made on the limited devices supported by Agamotto and VIA.

Table 5: Comparison of coverage with Agamotto and VIA

Driver	Agamotto	VIA	PrIntFuzz
8139cp	4%	11%	26%
e100	3%	10%	47%
e1000	4%	16%	26%
e1000e	2%	9%	21%
gve	5%	18%	4%
ne2k-pci	1%	1%	29%
nvme	11%	25%	9%
rocker	1%	2%	3%
sungem	6%	8%	19%
sunhme	14%	18%	18%
vmxnet3	7%	13%	28%

PrIntFuzz outperforms the other two fuzzers on most of the targets. The low coverage of PrIntFuzz on the target gve is that PrIntFuzz cannot successfully initialize this target. The gve driver requires the device to have the MSIX interrupt type, but PrIntFuzz does not currently support this. As for the nvme driver, PrIntFuzz lacks the corresponding manually written harness, resulting in the inability to cover more code.

5.5 Scalability

Although PrIntFuzz used the PCI driver as the example above, the methods provided by PrIntFuzz can still be applied to drivers

Table 6: The total number of USB drivers and I2C drivers and the number of PrIntFuzz simulated virtual devices

Type	USB	PrIntFuzz	I2C	PrIntFuzz
media	121	60	131	72
net	69	18	0	0
serial	49	34	0	0
others*	40	18	127	29
misc	22	14	16	12
input	19	13	0	0
storage	14	7	0	0
sound	12	5	100	64
iio	0	0	165	96
hwmon	0	0	146	110
input	0	0	73	7
regulator	0	0	43	27
mfd	0	0	38	12
rtc	0	0	30	27
power	0	0	26	16
Total	346	169	895	472
Success Rate	N/A	48.8%	N/A	52.7%

* All other types with less than 10 drivers.

under other buses. This subsection shows how to migrate the virtual device modeling approach to drivers under I2C and USB buses.

The modeling of virtual devices can be divided into two aspects. One is modeling the essential functions of virtual devices, such as MMIO, DMA, and interrupts. This can be directly migrated to other drivers because the interfaces for driver-device interaction are the same. The other aspect requires driver-specific analysis, which is different for drivers under different buses.

For I2C drivers, there is no need to model the memory space and configuration space as PCI drivers. Nevertheless, the modeling of data space is similar. PrIntFuzz only needs to collect some functions with reading functionalities used by the I2C driver. The solution to the constraints can reuse the previous static analysis. I2C drivers match the device by the identifier string, so PrIntFuzz needs to extract the corresponding string from the structure of I2C.

The USB device uses the USB descriptor, a more complex structure to describe the device, but it can be solved using similar techniques. Another notable feature of the USB driver is that it communicates with all USB devices through URB (USB Request Block), rather than simply using a function like `readl`. The analysis of such functions with read-like functionality can still be reused from the previous static analysis code, with only minor modifications for the characteristics of the USB drivers.

PrIntFuzz supports the modeling of I2C devices and USB devices by adding only about 200 lines of analysis code to the original code, proving that PrIntFuzz is also very scalable.

Table 6 shows the number of USB and I2C devices simulated by PrIntFuzz. In total, PrIntFuzz successfully simulates 169 out of 346 USB devices and 472 out of 895 I2C devices, showing the approach of PrIntFuzz is effective.

Based on the success of the device simulation, PrIntFuzz finds 20 and 18 bugs in the initialization code of the I2C drivers and the USB drivers, respectively. We have submitted some patches, and 9 patches have been confirmed and merged by the maintainers.

5.6 Case Study

In this subsection, we elaborate on three bugs to show the effectiveness of PrIntFuzz. All three bugs can be found only by PrIntFuzz, and they are located in different routines, the interrupt handler, initialization code, and main functions of drivers. PrIntFuzz uses different methods to trigger them, proving the effectiveness of PrIntFuzz's simulated devices and multi-dimension fuzzing.

5.6.1 Out-of-Bounds Bug in Interrupt Handlers Triggered by Interrupt and Data Injection. The bug exists in the `flexcop_pci_isr` function and can be triggered by an interrupt. The root cause of this bug is the lack of checking register values. As shown on line 2 in Listing 7, the driver reads a value from a register and uses it as an address. Since a malicious adversary may control the device, the value may be very large.

```
1   dma_addr_t cur_addr =
2       fc->read_ibi_reg(fc,dma1_008).dma_0x8.dma_cur_addr << 2;
3   u32 cur_pos = cur_addr - fc_pci->dma[0].dma_addr0;
```

Listing 7: `flex_pci_isr` in `flexcop-pci`

Function `flexcop_pci_isr` then calls `find_next_packet` and passes it with this large value via the `count` parameter, which further causes an out-of-bound read to `buf`, as shown in Listing 8.

```
1   while (pos < count) {
2       if (buf[pos] == 0x47 || (pktsize == 204 && buf[pos] == 0xB8))
3           break;
4       pos++;
5   }
```

Listing 8: Out-of-bounds bug in flexcop-pci

In order to discover this bug, the fuzzer first needs to simulate the device to interact with the driver. Then it should trigger the driver's interrupt handler and inject malicious data into the driver. Only PrIntFuzz has such a collaborative fuzzing capability.

5.6.2 Use-After-Free Bug in Initialization Code Triggered by Fault Injection. A use-after-free bug exists in the initialization code of the `igbvf` driver. In the initialization code of `igbvf`, the driver allocates memory for `adapter->rx_ring` and adds the address of `adapter->rx_ring->napi` to a doubly-linked list, as shown in Listing 9.

```
1   adapter->rx_ring = kzalloc(sizeof(struct igbvf_ring), GFP_KERNEL);
2   if (!adapter->rx_ring) {
3       kfree(adapter->tx_ring);
4       return -ENOMEM;
5   }
6   netif_napi_add(netdev, &adapter->rx_ring->napi, igbvf_poll, 64);
```

Listing 9: Memory allocation of `rx_ring` of `igbvf`

Suppose the initialization function encounters an error and goes to the error handling code labeled `err_hw_init`. The driver will free the memory of `adapter->rx_ring` and then iterate through the doubly-linked list mentioned above to free each element, which is done in `free_netdev` as shown in Listing 10. `adapter->rx_ring->napi` is visited after `adapter->rx_ring` is freed resulting in use-after-free.

```
1   err_hw_init:
2       kfree(adapter->tx_ring);
3       kfree(adapter->rx_ring);
4   err_ioremap:
5       free_netdev(netdev);
```

Listing 10: Error handling of the initialization code of `igbvf`

PrIntFuzz can find this bug because it can simulate the device to match the `igbvf` driver and inject faults in the initialization

function to trigger the error handling code. VIA cannot trigger the error handler and thus can never find the bug.

5.6.3 Use-After-Free Bug in `ioctl` *Triggered by System Call Interactions.* The `nosy` driver can match with multiple clients, and if a client invokes `ioctl` twice to add itself to the doubly-linked list, it will cause a use-after-free bug.

The root cause of this bug is that the element in the doubly linked list is reentered into the list. This bug can be fixed by adding a check before inserting a client and skipping it if already inserted, as Listing 11 shows.

`PrIntFuzz` can find this bug because it can simulate the device, match it with the nosy driver, and interact with the driver via system call interfaces. Even if VIA can simulate the device by manually constructing a device configuration, VIA cannot trigger this bug that requires system calls to interact.

```
case NOSY_IOC_START:
+   ret = -EBUSY;
    spin_lock_irq(client_list_lock);
-   list_add_tail(&client->link, &client->lynx->client_list);
+   if (list_empty(&client->link)) {
+       list_add_tail(&client->link, &client->lynx->client_list);
+       ret = 0;
+   }
    spin_unlock_irq(client_list_lock);
-   return 0;
+   return ret;
```

Listing 11: Patch of the use-after-free bug in nosy

6 DISCUSSION

6.1 Future Work

6.1.1 The System Call Interfaces of Driver Fuzzing. The interfaces are essential for driver fuzzing. In `PrIntFuzz`, we amend DIFUZE to recognize the system call interfaces of device drivers. Nonetheless, DIFUZE mainly focuses on the `ioctl` system call, widely used in character drivers but not PCI drivers. Most of our fuzzing targets have interrupt handlers, but few have interfaces like `ioctl`. It is thus crucial for fuzzers to extract the interfaces of PCI drivers.

6.1.2 Simulate More Devices. Although `PrIntFuzz` has achieved good results in simulating devices under different buses, some virtual devices cannot be matched with the driver for various reasons, such as complex data dependency, firmware requirement, and others mentioned above. Automated solutions for these problems can be explored in the future.

6.2 Threats to Validity

6.2.1 Internal Validity. Threats to internal validity arise from several sources. One of the biggest threats comes from the functionality of the virtual device. `PrIntFuzz` only provides the driver with fundamental interactions such as MMIO, DMA, and interrupts. However, `PrIntFuzz` cannot provide more advanced semantic features such as handling network congestion based on these basic features. Since the semantic information that each driver interacts with the device is usually customized, it is difficult for the virtual device to simulate common advanced features for each device.

Another aspect of the threats comes from the false positives that exist with fault injection. False positives are caused because `PrIntFuzz` hooks the function's return value without actually executing the function, which leads to an inconsistent program state and may result in error states that will not be reached.

As shown in Listing 12, `PrIntFuzz` performs fault injection on function `ipack_device_init`. Since `PrIntFuzz` does not execute this function, the reference count of the structure is not initialized, and a false positive occurred in the error path.

```
static int tpci200_pci_probe() {
    ret = ipack_device_init(dev);
    if (ret < 0) {
        put_device(&dev->dev);
        return ret;
    }
    ...
}

static int ipack_device_init(struct ipack_device *dev) {
    device_initialize(&dev->dev);
    ...
}
```

Listing 12: False positive of fault injection

However, such drivers are infrequent, and `PrIntFuzz` manually excludes the effects of false positives in the evaluation. Of the 26 crashes reported using the fault injection method, only 3 were false positives after manual confirmation.

6.2.2 External Validity. The main external threat to validity is the difference in driver patterns under different buses. For example, the I2C and USB drivers differ from the PCI drivers mentioned in the evaluation. The generality of `PrIntFuzz` can reduce this difference in driver patterns. For instance, almost all drivers follow a similar pattern of matching the device by id, and the driver reads the device register value for the check. So the generality of `PrIntFuzz` can provide confidence for research.

6.3 Application of `PrIntFuzz`

The approach of `PrIntFuzz` could be applied to many other targets that require hardware support. For instance, we could extend it to drivers that do not have source code, e.g., Windows drivers and macOS drivers. We could also extend it to support testing IoT firmware, e.g., simulating peripherals of IoT devices and rehosting the firmware in an emulator (e.g., QEMU). Furthermore, we can also apply the approach of `PrIntFuzz` to other targets that require extra effort to build the testing environment. For instance, blockchains are deployed in a decentralized way, requiring multiple nodes to cooperate and causing troubles for efficient testing. We could utilize the approach of `PrIntFuzz` to simulate nodes in blockchains and conduct efficient fuzzing tests to discover potential bugs in them.

7 CONCLUSION

Existing driver fuzzing solutions overlooked a large portion of driver code due to insufficient device support. `PrIntFuzz` addressed this problem by automated virtual device simulation, which supports device initialization, hardware interrupts raising, and hardware I/O data interception. Given the support of virtual devices, `PrIntFuzz` performs multi-dimension fuzzing, combining multiple methods to influence the driver's control flow and data flow, such as system call interaction, interrupt injection, and data injection.

Evaluation results showed that this solution could enable the fuzzer to explore more driver code and find more bugs in drivers.

ACKNOWLEDGMENTS

We thank the anonymous ISSTA reviewers for their valuable comments. This work was supported in part by National Key R&D Program of China (2021YFB2701000), National Natural Science Foundation of China under Grant 61972224, Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006 and Hong Kong RGC Project (No. PolyU15223918).

REFERENCES

[1] 2021. Top 50 Products By Total Number Of "Distinct" Vulnerabilities in 2021. https://www.cvedetails.com/top-50-products.php?year=2021.
[2] 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.
[3] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe {DMA} Accesses in Device Drivers. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.
[4] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. 2018. {DSAC}: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 587–600.
[5] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 677–693.
[6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux device drivers.* " O'Reilly Media, Inc.".
[7] 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.
[8] Baojiang Cui, Yunze Ni, and Yilun Fu. 2015. ADDFuzzer: A New Fuzzing Framework of Android Device Drivers. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. IEEE, 88–91.
[9] GoodFET. 2018. Facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21/.
[10] HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2345–2358.
[11] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. VIA: Analyzing Device Interfaces of Protected Virtual Machines. *arXiv preprint arXiv:2109.10660* (2021).
[12] Google Inc. [n.d.]. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.
[13] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2019. Fuzzing error handling code in device drivers based on software fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 128–138.
[14] Asim Kadav and Michael M Swift. 2012. Understanding modern device drivers. *ACM SIGPLAN Notices* 47, 4 (2012), 87–98.
[15] Sylvester Keil and Clemens Kolbitsch. 2007. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan* (2007).
[16] Linux kernel document. [n.d.]. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.13/dev-tools/kasan.html.
[17] David Kierznowski. [n.d.]. BadUSB 2.0: Exploring USB Man-In-The-Middle Attacks. ([n. d.]).
[18] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.
[19] Linux. 2020. Patch. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/drivers/net/vmxnet3?id=de1da8bcf40564a2adada2d5d5426e05355f66e8.
[20] Linux. 2020. Patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3e1c6846b9e108740ef8a37be80314053f5dd52a.
[21] 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.
[22] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1007–1024.
[23] Dominik Maier and Fabian Toepfer. 2021. BSOD: Binary-only Scalable fuzzing Of device Drivers. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 48–61.
[24] Theo Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon Moore, and Robert Watson. 2019. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. (2019).
[25] "Grant Grundler" "Martin Mares". [n.d.]. *1. How To Write Linux PCI Drivers* (v5.16-rc1 ed.). The kernel development community.
[26] Tianshi Mu, Huabing Zhang, Jian Wang, and Huijuan Li. 2021. CoLaFUZE: Coverage-Guided and Layout-Aware Fuzzing for Android Drivers. *IEICE TRANSACTIONS on Information and Systems* 104, 11 (2021), 1902–1912.
[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] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1203–1218.
[29] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2197–2213.
[30] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 149–165.
[31] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2017. {POTUS}: Probing Off-The-Shelf {USB} Drivers with Symbolic Fault Injection. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*.
[32] 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.
[33] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*. 328–333.
[34] Ivan Pustogarov, Qian Wu, and David Lie. 2020. Ex-vivo dynamic analysis framework for Android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1088–1105.
[35] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. 2012. Symdrive: Testing drivers without devices. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 279–292.
[36] 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.
[37] Alireza Shameli-Sendi. 2021. Understanding Linux kernel vulnerabilities. *Journal of Computer Virology and Hacking Techniques* (2021), 1–14.
[38] 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*.
[39] 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.
[40] Jeff Vander Stoep. 2016. Android: protecting the kernel. https://events.static.linuxfound.org/sites/events/files/slides/Android-protectingthekernel.pdf.
[41] 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.
[42] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 291–307.
[43] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.
[44] ulfalizer. 2019. Kconfiglib. https://github.com/ulfalizer/Kconfiglib/.
[45] Dmitry Vyukov. 2018. kernel: add kcov code coverage. https://lwn.net/Articles/671640/.
[46] 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.
[47] Matthew Wilcox and Alan Cox. [n.d.]. Bus-Independent Device Accesses. https://www.kernel.org/doc/html/latest/driver-api/device-io.html.
[48] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. 2021. Understanding and Detecting Disordered Error Handling with Precise Function Pairing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.