# DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing

Yilun Wu, *Stony Brook University*
Tong Zhang, *Samsung Semiconductor*
Changhee Jung, *Purdue University*
Dongyoon Lee, *Stony Brook University*

***Abstract*—The security of device drivers is critical for the entire operating system's reliability. Yet, it remains very challenging to validate if a device driver can properly handle potentially malicious input from a hardware device. Unfortunately, existing symbolic execution-based solutions often do not scale, while fuzzing solutions require real devices or manual device models, leaving many device drivers under-tested and insecure.**

**This paper presents DEVFUZZ, a new model-guided device driver fuzzing framework that does not require a physical device. DEVFUZZ uses symbolic execution to automatically generate the probe model that can guide a fuzzer to properly initialize a device driver under test. DEVFUZZ also leverages both static and dynamic program analyses to construct MMIO, PIO, and DMA device models to improve the effectiveness of fuzzing further. DEVFUZZ successfully tested 191 device drivers of various bus types (PCI, USB, RapidIO, I2C) from different operating systems (Linux, FreeBSD, and Windows) and detected 72 bugs, 41 of which have been patched and merged into the mainstream.**

## 1. Introduction

The security of device drivers is crucial for the entire operating system (OS) security. Modern OSes often include a large number of device drivers to provide compatibility with various hardware devices: *e.g.,* in Ubuntu Linux 20.04, device drivers (under the `driver` directory) consist of 13M lines of code occupying 64.8% of the entire Linux source code. Such a large codebase undoubtedly makes device drivers a large attack surface, and thus they have become the major sources of OS security vulnerabilities. In fact, the CVE reports of device drivers account for 27-54% of the total Linux kernel CVE reports for the last 5 years.

As device drivers run in kernel space (in many OSes), handling the input from peripheral devices to drivers without proper sanitization has been the root cause of many real-world system security issues. For instance, ThunderClap [1] demonstrated how a malicious PCI device can read the system memory including user-sensitive data and credentials. The PS3 jailbreak [2] targeted a vulnerability in the PS3 hypervisor's USB stack. The PS4 hack [3] enabled control flow hijacking via a malicious PCIe device. The Xbox DVD hack [4] exploited a vulnerable verification process in the DVD driver. The iPhone hack [5] allowed malicious attackers to compromise wireless chips and steal data.

Unfortunately, existing solutions provide limited support for testing device drivers, leaving many drivers under-tested and insecure. For instance, some prior solutions [6], [7], [8], [9] rely on symbolic execution that renders input from a device symbolic and enable device driver testing for closed-source systems without a real device. However, symbolic execution often does not scale to large software [10].

Alternatively, fuzzing [11] has shown to be more effective in testing real-world software in practice. However, fuzzing device drivers remains critically challenging for two reasons. First, it is hard for a fuzzer to pass a complex dynamic probing phase. Many modern bus architectures (*e.g.,* PCI, USB) support hot plugging a device. An OS scans a bus, recognizes a device, binds a corresponding driver, and runs the device driver's custom probing logic. The device should follow certain probing protocols: *e.g.,* providing Vendor and Product IDs, and configuring the device's address in the system memory map. Dynamic probing is considered successful if a kernel binds a device and a matching driver, and the driver completes additional custom probing logic. Unguided random inputs often fail in this dynamic probing, leading to premature testing. Second, even after successful probing (post-probing), it is difficult to scalably explore the huge input space of various I/O interfaces such as Memory-Mapped I/O (MMIO), Port I/O (PIO), Direct Memory Access (DMA), and Interrupt (IRQ).

Unfortunately, prior fuzzing solutions insufficiently address the above challenges. For probing, some [12], [13], [14] require real devices to set up device drivers under test, limiting testing for various device drivers. Others [15], [16] require user-provided device models to test device drivers without real devices. However, manually developing device models are error-prone and not scalable. Besides, they all share additional limitations: they are designed for one particular OS, and support only limited bus types.

This paper presents DEVFUZZ, a new model-guided device driver fuzzing solution. DEVFUZZ allows users to test a device driver without an actual device. DEVFUZZ supports testing a variety of device drivers of different bus types (*e.g.,* PCI, USB, RapidIO, I2C); with different I/O interfaces (*e.g.,* MMIO, PIO, DMA, and IRQ); and from different OSes (*e.g.,* Linux, FreeBSD, and Windows).

The key idea is to automatically generate three device models to facilitate device driver fuzzing in the absence of a physical device. DEVFUZZ first employs symbolic execution to generate (1) *Probe Model* that allows a device driver to satisfy the probing path constraints. Symbolic probing execution is regarded to be successful—after some explorations—if the driver is bounded to the device (model) by the kernel. Furthermore, DEVFUZZ leverages static and

dynamic program analyses of device drivers to construct (2) *MMIO/PIO Model* and (3) *DMA Model* that embody MMIO, PIO, DMA addresses and their value range constraints that affect device driver's control flows. They guide DEVFUZZ to explore the device driver's different program paths during fuzzing. As the three models are intended to represent device properties, the models generated from one OS could be reused to test another OS's device drivers.

DEVFUZZ combines the three models with a general-purpose fuzzer AFL [17], and performs fuzzing by selecting an input from either device models or AFL with a configurable probability (50% by default). The combination allows DEVFUZZ to take advantage of both the models' device awareness and the fuzzer's feedback-directed random mutation. Together, DEVFUZZ fuzzes the target device driver during both the probing and post-probing phases. For probing phase fuzzing, DEVFUZZ repeatedly plugs and unplugs the target device driver (module), triggers the probing functions multiple times, and feeds a mixed input from the *Probe Model* and AFL. For post-probing phase fuzzing, DEVFUZZ first uses *Probe Model* (with no mutation) to pass the probing logic and successfully bind the driver to the device (model). Then, it continuously runs command-line tools and test programs—that trigger the driver's other (non-probing) I/O functions—feeding a mixed input from *MMIO/PIO Model*, *DMA Model*, and AFL.

We evaluated DEVFUZZ using device drivers of different bus types (PCI, USB, RapidIO, I2C) across three OSes (Linux, FreeBSD, Windows). For Linux kernel 5.15, DEVFUZZ managed to test 150 drivers (108 PCI, 31 USB, 1 RapidIO, and 10 I2C types). With symbolic execution, DEVFUZZ successfully probed 112 devices and generated their device models (75%). We offer detailed analysis on the failed cases and the quality of the generated probe models. DEVFUZZ detected 63 bugs, 39 of which have been patched into the mainline Linux kernel. The proposed three device models increase the code coverage for both probing-phase and post-probing-phase fuzzing. The case study with 17 network device drivers demonstrates that DEVFUZZ achieves higher code coverage than PrIntFuzz [18] and Drifuzz [19]. Besides, DEVFUZZ's models obtained similar code coverage to manually designed models, *e.g.,* QEMU's virtual devices.

Finally, to demonstrate device model re-usability across different OSes, we also tested 25 FreeBSD-12.2 PCI drivers and 16 Windows-10 PCI drivers by reusing the device models derived from Linux. For FreeBSD, DEVFUZZ successfully probed 14 PCI devices (56%) and found eight bugs. So far two patches have been merged. For Windows, DEVFUZZ probed eight PCI devices (50%) and detected one bug.

## 2. Background

### 2.1. Device and Driver Interactions

There are two forms of Input/Output (I/O) interfaces between devices and drivers: Memory-Mapped I/O (MMIO) and Port I/O (PIO). Besides, they may share data using Direct Memory Access (DMA). Devices can also notify drivers via an asynchronous Interrupt Request (IRQ).

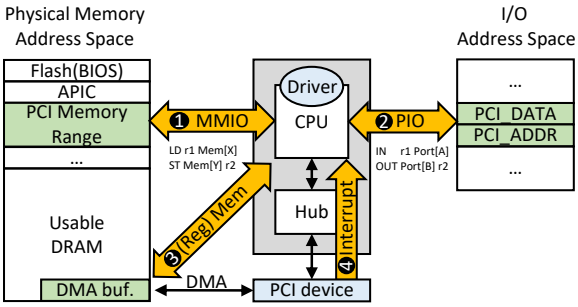


Figure 1: Four different I/O interfaces (orange arrows) between a device driver and a PCI device.

**(1) MMIO:** Devices are mapped to the "physical memory address space" for MMIO so that their device drivers can access them using memory instructions (*e.g.,* MOV) and operate on their states using logical instructions (*e.g.,* AND, OR, TEST). The Memory Management Unit (MMU) hardware redirects any access destined to such an MMIO region to the corresponding device. Figure 1 ❶ illustrates that the PCI memory range—using Base Address Registers, simply BARs (omitted in the figure)—is mapped to a part of the physical memory address space, next to the BIOS flash and Advanced Programmable Interrupt Controller (APIC) regions. For instance, in Intel x86, the PCIEXBAR register holds the base address of the PCI memory range.

**(2) PIO:** Peripheral devices can also be mapped to a separate address space called "I/O address space". Device drivers use I/O instructions (*e.g.,* IN, OUT, INS, OUTS) with the ring 0 privilege to access the I/O address space. These PIO accesses are handled by the I/O Controller Hub (ICH). Figure 1 ❷ shows that PCI configuration registers are mapped to I/O Addresses PCI_DATA and PCI_ADDR (in addition to MMIO-mapped PCI memory range).

**(3) DMA:** Direct memory access (DMA) allows devices to transfer a large chunk of data from/to main system memory without involving a CPU. DMA may be supported through a centralized DMA controller (*e.g.,* Intel 8257 [20]) or by a custom non-standard DMA controller in a device (called bus mastering). DMA is often initiated by performing some MMIO/PIO writes. Figure 1 shows that a DMA buffer is allocated in DRAM. Device drivers can access the DMA buffer using regular memory operations (Figure 1 ❸).

**(4) IRQ:** A device itself can also signal a CPU (Figure 1 ❹) to inform about a certain event (*e.g.,* transfer completion, new data arrival) in form of an interrupt request (IRQ), triggering the interrupt service routine of the device driver. In general, IRQ is asserted and routed through IO-APIC. Alternatively, PCI defines two in-band mechanisms called Message Signalled Interrupts (MSI/MSI-X).

### 2.2. Device Enumeration and Probing

Device drivers recognize devices based on the pre-populated (static) device information or detect them at run time.

**(1) Static enumeration:** Some device drivers simply recognize devices using pre-defined information. For those devices that are hardwired or whose locations do not change in the system memory map, developers can easily identify

device locations beforehand and embed the device information such as memory addresses in the driver. While such drivers are commonly found in embedded systems, general-purpose operating systems (e.g., Linux and FreeBSD) and boot loaders (U-Boot [21]) also support static device enumeration via a Device Tree Blob (DTB) file [22], which statically describes the hardware configuration and topology.

**(2) Dynamic probing:** Many modern bus architectures (*e.g.,* PCI/PCIe, USB, ISA) allow users to plug in new devices. An operating system is responsible for scanning the bus, figuring out which device is connected to the bus, and updating the bus topology at run time. Such a dynamic probing process may involve complex logic such as selecting a device at a specific address, reading device Vendor and Product IDs, (re)configuring the device's address in the system memory map, performing device initialization, enabling device IRQ, and so on.

For example, an operating system can start probing a PCI device by (1) writing the "Bus", "Device", "Function" ID to the PCI configuration register at PIO address PCI_ADDR and (2) then reading the response from the PCI data register at PCI_DATA as shown in Figure 1. If the device is present on the bus, the PCI_DATA register is set to a non-zero value. Then, the operating system reads the device's Vendor ID (VID) and Product ID (PID), finds a device driver potentially interested in the device, and calls the device driver's probing function. Finally, the probing function performs device-specific configuration and initialization processes.

### 2.3. Device Driver Security Vulnerabilities

Improper handling of input from peripheral devices to device drivers has led to real-world security incidents. ThunderClap [1] demonstrated that the inadequate use of the Input-Output Memory Management Units (IOMMUs) could allow malicious DMA-enabled peripherals to extract private data and hijack kernel control flow. The PS3 (IBM CELL BE processor with IOMMU [27]) Jailbreak [2] exploits a vulnerability in the PS3 hypervisor's USB stack. The PS4 (AMD APU with IOMMU) hack [3] launches stack-based control flow hijacking via a malicious PCIe device. The Xbox (IBM PowerPC with IOMMU [28]) DVD hack [4] leverages the vulnerable verification process in the DVD driver. Recently, a vulnerability in Apple's iPhone (ARM processor with SMMU [29]) drivers [5] enables a malicious user to compromise wireless chips and steal data. Unfortunately, things get worse as these attacks do not require complex or expensive hardware. For example, the initial version of PS3 jailbreak hardware [2] was sold as a thumb drive employing a 5 US dollar AVR microcontroller. The open-source version has also been shown to run on even a simple calculator, i.e., Texas Instruments TI-84 [30]. The bar to acquire and deploy such a hardware exploit is surprisingly low.

### 2.4. Threat Model

We presume that an attacker has physical access to a victim machine to which a malicious device can be connected. The malicious device may or may not be properly probed/recognized by the victim driver (OS). The malicious device (an attacker) may send arbitrary data to the victim device drivers via MMIO, PIO, DMA, and IRQ interfaces, exploiting software vulnerabilities (*e.g.,* buffer overflows) in the driver (OS). Our threat model is aligned with the above real-world exploits (§2.3) and prior device driver testing works [14], [15], [16], [19], [18], [26]. Side channels are not considered.

## 3. Related Work and Motivation

This section discusses existing device driver testing tools and their limitations, motivating new solutions.

### 3.1. Symbolic Execution

One class of existing tools leverages symbolic execution. SymDrive [6] is based on S2E [31] that uses QEMU [32] for device emulation and KLEE [33] for symbolic execution. SymDrive does not require an actual device because input from a device can be marked as symbolic. However, it requires manual annotation on each tested driver to specify and "symbolize" device inputs. Moreover, SymDrive runs very slowly and suffers from the well-known path explosion problem. POTUS [7] targets USB devices specifically, is built upon S2E, and uses a similar mechanism to mark USB data as a symbolic value. Therefore, POTUS shares the same drawback as SymDrive. DDT [8] is yet another driver symbolic testing tool, but it does not support PCI Express and USB devices. CABFuzz [9] leverages concolic (a hybrid of concrete and symbolic) execution and focuses on testing loop/array boundary conditions to reduce testing scope. CABFuzz's heuristic is limited to one kind of bug, i.e., improper boundary checking. On the other hand, Drifuzz [19] combines concolic execution with fuzzing, so it supports testing device drivers without devices. However, using concolic execution for fuzzing the post-probing phase may be very expensive due to frequent MMIO and DMA accesses, and IRQ requests.

### 3.2. Fuzzing

Various solutions have been developed for general kernel fuzzing [34], [35], [36], [37], [38], [39]. However, there are two unique challenges in fuzzing device drivers: (1) dynamic probing and (2) various I/O interfaces. Table 1 summarizes limited support from prior device driver fuzzing solutions.

Some solutions do not support dynamic probing. To be fair, they do not need to. P2IM [23] and DICE [24] aim to test (small) firmware of embedded systems and microcontrollers in which the complete set of devices is statically known and often hardwired into the system. Ex-vivo [25] targets Android device drivers and relies on Linux kernel's (static) device tree—specific to mobile devices—to load the drivers. These solutions are not readily applicable to general OS (*e.g.,* Linux, FreeBSD) drivers, especially for devices that require dynamic probing such as PCI and USB.

Other solutions assume that a real device is available, and thus cannot be applied to test a device driver without a real device. With an actual device, device probing is not an issue; there is no need for modeling or emulation. Charm [12] allows users to run mobile device drivers on a QEMU-based virtual machine on a workstation, facilitating dynamic analyses including fuzzing. DIFUZE [13] is specifically designed to fuzz the ioctl interfaces of Linux drivers.

TABLE 1: Comparison to prior device driver fuzzing tools.

| Tool | Target OS | Dynamic Probing Support? | Dynamic Probing How? | Supported Bus Types PCI/PCIe | USB | I2C | Others | Fuzzing I/O Interfaces MMIO | PIO | DMA | IRQ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| P2IM [23] | Firmware | ✗ | | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| DICE [24] | Firmware | ✗ | | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Ex-vivo [25] | Android | ✗ | | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Charm [12] | Android | ✓ | real device | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| DIFUZE [13] | Android/Linux | ✓ | real device | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Periscope [14] | Linux | ✓ | real device | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| USBFuzz [15] | Any OS | ✓ | manual model | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| FuzzUSB [16] | Linux | ✓ | manual model | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| DrFuzz [26] | Linux | ✓ | static analysis + fuzzing | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| PrIntFuzz [18] | Linux | ✓ | static analysis + fuzzing | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Drifuzz [19] | Linux | ✓ | concolic execution | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| DEVFUZZ | Any OS | ✓ | symbolic exec. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Periscope [14] modifies the page fault mechanism in Linux to intercept and mutate data read by a device driver.

Even though some other recent works target testing USB device drivers without actual USB devices, they still require manually-designed device models instead. While Syzkaller [34] is originally designed for fuzzing system calls, it later supports USB driver testing. However, it requires a stub USB device that forwards USB I/O to its fuzzing logic. That is, Syzkaller cannot conduct the fuzzing without the actual device. USBFuzz [15] and FuzzUSB [16] are built upon QEMU and target USB devices. FuzzUSB leverages more advanced static analysis to construct state machine to guide the USB device fuzzing.

PrIntFuzz [18] and DrFuzz [26] use static-analysis-guided fuzzing for dynamic probing. As it is ineffective to blindly feed random input for probing, they employ static analysis to identify critical fields and relevant ranges of data to focus on. Such guided fuzzing increases the probing success rate. Yet, both are not optimized for testing the post-probing phase with various I/O interfaces.

Given all this, there is a compelling demand for practical device driver testing tools, that are scalable, device-free, model-driven, automated, and capable of testing representative bus types, to cover as many drivers as possible with minimal user intervention. In §4, we propose DEVFUZZ specifically designed with this motivation in mind.

### 3.3. Other Related Work

Some solutions focus on fuzzing performance issues: *e.g.,* Agamotto [39] adds lightweight checkpoint support for fuzzing efficiency. There are static program analysis tools as well [40], [41], [42], [43].

Though it is not a device driver testing tool, it is also worth noting that ProXRay [44] (a firmware analysis tool) is related to DEVFUZZ in that both use symbolic execution to learn protocol (probing) models. DEVFUZZ currently does not perform actively-guided symbolic execution. However, DEVFUZZ may adopt ProXRay's approach that prioritizes the exploration of paths that refer to protocol fields (or that read PIO/MMIO registers in DEVFUZZ's terms).

## 4. Design of DEVFUZZ

Figure 2 illustrates DEVFUZZ's model-based device driver fuzzing framework. DEVFUZZ first takes as input the Linux kernel and a device driver to be tested, and symbolically executes the device probing logic. When successfully probed, DEVFUZZ generates *Probe Model* (§4.1) based on the values concretized during the symbolic execution. Then, DEVFUZZ performs dynamic and static analyses on Linux and a target device driver to generate *MMIO/PIO Model* (§4.2) and *DMA Model* (§4.3). As the three models resemble real hardware devices, DEVFUZZ can (re)use them for fuzz-testing any OS's device drivers (§4.4) and report bugs/crashes detected.

### 4.1. Probe Model Generation

DEVFUZZ leverages symbolic execution to complete a device-specific probing phase and to construct *Probe Model* with the concretized values. The resulting model enables the target device to be properly probed (without a real device), i.e., the device driver gets ready for post-probing fuzzing (§4.4) where the probe model serves as a basis for structure-aware fuzzing [13]. The probe model can also be used in combination with a fuzzer (*e.g.,* AFL) to fuzz the probing phase itself.

Figure 3 illustrates DEVFUZZ's probe model generation. DEVFUZZ is built on S2E [31] that uses QEMU [32] for device emulation and KLEE [33] for symbolic execution. In particular, DEVFUZZ guides symbolic execution by directing the KLEE engine to stop exploring a program path and to try alternative ones when the probing apparently fails with an error message (*e.g.,* `<module> ... failed.`). Another thing that makes DEVFUZZ different from prior symbolic execution based tools (*e.g.,* SymDrive [6], POTUS [7], DDT [8], CABFuzz [9]) is that DEVFUZZ does not require guest OS or QEMU virtual device modification. Rather, DEVFUZZ enables symbolic device probing without manual annotation by piggybacking on the x86-specific device probing protocol in which the PIO region is set up at the pre-determined location first and the MMIO regions are specified by PIO writes. During symbolic execution, DEVFUZZ traps (intercepts) PIO writes to automatically symbolize PIO/MMIO regions on demand.

Suppose we test a PCI device driver. DEVFUZZ first triggers a bus scan operation to initiate device probing (step ❶), which we explain in detail later. DEVFUZZ then intercepts the updates to those PIO regions that are pre-determined for each bus type: *e.g.,* for x86 architecture, 0xCF8 and 0xCFC are reserved for PCI devices (step ❷). From the intercepted

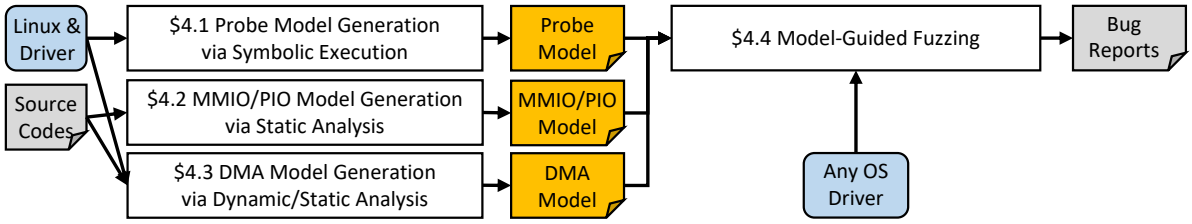


Figure 2: An overview of DEVFUZZ. DEVFUZZ employs symbolic execution and dynamic/static program analyses to generate device models. DEVFUZZ uses the models to fuzz-test any OS's device drivers and reports detected bugs/crashes.

PIO writes, DEVFUZZ identifies additional MMIO/PIO regions that the device requests to allocate dynamically (step ❸) and automatically marks those regions symbolic (step ❹). Next, DEVFUZZ performs symbolic probing in which the driver reads symbolic PIO/MMIO inputs (step ❺).

Note that symbolic probing execution (step ❺) is considered successful (after some explorations) if a device driver passes the probing path constraints so that the driver can be successfully loaded and bound to the device (model) by the kernel. We implemented a stub script that enables S2E's symbolic execution to determine a successful device probing. The script enables kernel dynamic debugging and checks if a driver is bound to a device by referring to the kernel's `dmesg` output. The script also consults `/sys` file system to ensure that the driver holds a reference to the device after probing. For example, suppose we emulate a `net2272` device at PCI address `0000:00:02.0`. After loading the `net2272` driver, a successful probing is detected when `dmesg` shows `bus: 'pci': really_probe: bound device 0000:00:02.0 to driver net2272` with `/sys/bus/pci/drivers/net2272/0000:00:02.0` created. This can also be confirmed by executing command `lspci -vvv`, the output of which is supposed to show `00:02.0: Kernel driver in use: net2272`.

When successfully probed, DEVFUZZ solves the path constraints using an SMT solver, concretizes the symbolic values, and generates a probe model (step ❻). The probe model contains the addresses and the register/memory values that are necessary for the device driver to succeed in probing. For each register (*i.e.,* an MMIO/PIO address), DEVFUZZ generates a state machine that describes the value of the register upon each read; in reality, two reads from the same MMIO addresses (registers) may be not identical due to arbitrary change made by the hardware device, and therefore we model this behavior as a state machine with a counter. After all the reads are completed for each register during the probing phase, the state machine goes back to the start position for the next probing (if requested) to be successful.

DEVFUZZ triggers the initial bus scan (step ❶) in two different ways: (1) For both the devices connected on the buses with dynamic probing (*e.g.,* PIC/PCIe) and Linux guest OS, DEVFUZZ runs `echo 1 > /sys/bus/pci/rescan`. (2) For other devices on the buses that use static enumeration (*e.g.,* I2C), DEVFUZZ adds a PCI-to-I2C adapter and binds the I2C device to the I2C bus adapter address, e.g., `bmi160` driver is bound to the I2C adapter's address `0x11` using the following command: `echo bmi160 0x11 > /sys/bus/i2c/devices/i2c-1/new_device`.

**Example.** Figure 4 shows how the probing code of `pcnet32` driver works. First, the probing function `pcnet32_probe_pci` maps the PCI BAR 0 (line 4) and obtains the MMIO base address. Then, it calls `pcnet32_probe1` to perform the actual initialization. This function first reads from two registers to check the access methods of the device (line 16), which determines the number of words for each register. Here, `pcnet32_wio_read_csr`, i.e., a wrapper of MMIO functions, reads from address 0x10 and returns its value, while `pcnet32_wio_check` reads from address 0x14 and returns true if the value equals to 0x58; the code of the two methods is not shown in the figure. Then, the driver reads from address 0x10 two times to obtain a 32-bit chip version value and performs a sanity check on it (lines 23-36). If either of the above checks fails, the driver should terminate the probing process and return an error. The implication of the probing process is that the sanity checks are very strict and therefore a random fuzzing is not likely to pass them.

To address the problem, DEVFUZZ generates the probe model shown in Figure 6 (a). DEVFUZZ figures out that the important registers are located at 0x10 and 0x14 (others are omitted for simplicity). The "cnt" in the figure denotes the number of reads for the same address. When the driver invokes `pcnet32_wio_read_csr` reading from address 0x10, the state machine for address 0x10 returns 0x4 to pass the first comparison check; see line 16 of Figure 4. When the driver reads from 0x14 in `pcnet32_wio_check`, the state machine for that address returns 0x58 to pass the second check. In line 23, the driver invokes `read_csr` twice reading from 0x10 for the second and third time. Taking that into account, the state machine returns 0x3 and 0x243 which makes the `chip_version` become 0x2430003. This allows

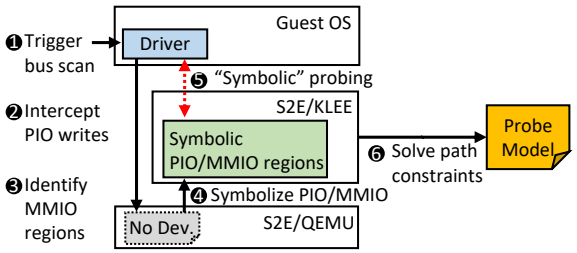


Figure 3: DEVFUZZ uses symbolic execution to generate the probe model.

```
1  int pcnet32_probe_pci(struct pci_dev *pdev,
   ↪  const struct pci_device_id *ent)
2  {
3      ...
4    ioaddr = pci_resource_start(pdev, 0);
5    ...
6    err = pcnet32_probe1(ioaddr, 1, pdev);
7    return err;
8  }
9  int pcnet32_probe1(unsigned long ioaddr, int
   ↪  shared, struct pci_dev *pdev)
10 {
11   int chip_version;
12   const struct pcnet32_access *a = NULL;
13   int ret = -ENODEV;
14   struct net_device * dev;
15
16   if (pcnet32_wio_read_csr(ioaddr, 0) == 4 &&
     ↪  pcnet32_wio_check(ioaddr)) {
17     a = &pcnet32_wio;
18   } else {
19     ...
20         goto err_release_region;
21   }
22
23   chip_version = a->read_csr(ioaddr, 88) |
     ↪  (a->read_csr(ioaddr, 89) << 16);;
24   if ((chip_version & 0xfff) != 0x003) {
25     goto err_release_region;
26   }
27   chip_version = (chip_version >> 12) & 0xffff;
28   switch (chip_version) {
29   ...
30   case 0x2430:
31     ...
32     break;
33   ...
34   default:
35     goto err_release_region;
36   }
37     ...
38   dev->base_addr = ioaddr;
39   return 0;
40 err_release_region:
41   release_region(ioaddr, PCNET32_TOTAL_SIZE);
42   return ret;
43 }
```

Figure 4: Probing codes of `pcnet32` device driver.

the check in line 24 to be passed. Finally, in line 27, the `chip_version` is right-shifted 12 times resulting in 0x2430 which matches with the case in line 30.

**Limitation.** DEVFUZZ may not be able to generate a probe model. The reason is that symbolic execution may fail to complete the probing phase if the probing logic is too complex to solve within some time budget, and/or if it requires DMA/IRQ that are rare so DEVFUZZ's (current) symbolic execution does not support. However, note that for those successfully probed, DEVFUZZ ensures that the device driver can pass its probing logic with *Probe Model* and be successfully loaded for testing; §6.3 provides analysis on both symbolic probing evaluation results and the quality of the generated probe models.

## 4.2. MMIO/PIO Model Generation

The above probe model ensures that a device driver is ready for a fuzzer (*e.g.,* AFL) to test the post-probing phase (§4.4). All the MMIO and PIO regions (but not DMA regions, §4.3) are identified and a fuzzer can feed mutated input as needed. However, the possible value space of the MMIO/PIO regions and the address range of the region could be very large. This implies that blindly feeding random data into such a large address space tends to be ineffective in exploring different program paths.

To improve the effectiveness of the post-probing fuzzing,

```
1  static irqreturn_t pcnet32_interrupt(int irq,
   ↪  void *dev_id)
2  {
3    struct pcnet32_private *lp;
4    unsigned long ioaddr;
5    u16 csr0;
6    int boguscnt = max_interrupt_work;
7    ioaddr = dev->base_addr;
8    lp = netdev_priv(dev);
9
10   csr0 = lp->a->read_csr(ioaddr, CSR0);
11   while ((csr0 & 0x8f00) && --boguscnt >= 0) {
12     if (csr0 == 0xffff)
13       break;
14     lp->a->write_csr(ioaddr, CSR0, csr0 &
       ↪  ~0x004f);
15     if (csr0 & 0x4000)
16       dev->stats.tx_errors++;
17     if (csr0 & 0x1000) {
18       dev->stats.rx_errors++;
19     }
20     if (csr0 & 0x0800) {
21       ...
22     }
23     ...
24     csr0 = lp->a->read_csr(ioaddr, CSR0);
25   }
26   return IRQ_HANDLED;
27 }
```

Figure 5: MMIO value checking in `pcnet32` driver

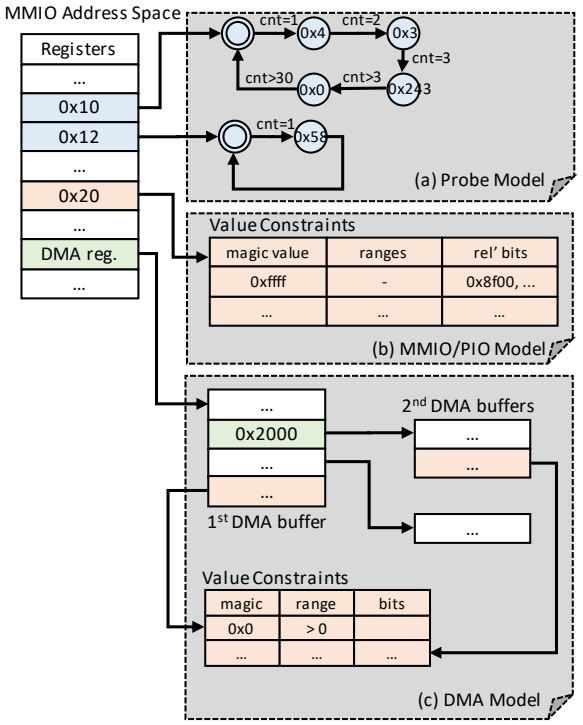


Figure 6: The generated (a) Probe model, (b) MMIO/PIO model, and (c) DMA model for `pcnet32` device driver.

DEVFUZZ performs *static value analysis* and generates *MMIO/PIO Model*, thereby guiding the fuzzing process. The goal of the static value analysis is to obtain three kinds of value constraints (for each base address/offset) that determine the control flow of device driver code: (a) magic values (*e.g.,* `0xff` for `if (p==0xff)`); (b) boundary values (*e.g.,* `0` and `10` for `if (0<p<10)`); and (c) relevant bits, (*e.g.,* 2nd and 3rd least significant bits for `if (p&0x04 || p&0x08)`). The derived constraints are later used in mutating

MMIO register values for fuzzing. If multiple constraints exist for an MMIO register, DEVFUZZ randomly applies one for each different fuzzing run.

Since the above three value constraints are based on heuristics we devised with manual code reviews of many device drivers (*e.g.,* Figure 5), DEVFUZZ does not claim completeness—though the constraints work very well in practice. It is important to note that DEVFUZZ's model purposely rules out the case where an MMIO register acts like a simple storage, returning the most recently written value in response to a read as usual. This is because, we are not interested in mimicking the benign behavior of device hardware, but in synthesizing potentially malicious hardware activities that may arbitrarily change the register value after being written, breaking a device driver's assumption. According to our empirical analysis result (§6.5), the proposed three value constraints help DEVFUZZ increase its code coverage with a significant margin.

The proposed static value analysis consists of three components: (1) base address analysis, (2) I/O wrapper function analysis, and (3) condition predicate analysis. DEVFUZZ as output produces the MMIO/PIO model, a mapping from a base address, and an offset to value constraints.

**(1) Base Address Analysis.** For modularity and encapsulation, it is common practice in Linux to maintain a struct (object) representing a device: *e.g.,* `struct net_device` for a network device. The first base address analysis analyzes device probing/initialization codes and identifies which fields of a device struct hold base addresses so that later we can create a mapping between base addresses and value constraints. To this end, DEVFUZZ performs data flow analysis to keep track of where the return values of the Linux APIs that map MMIO/PIO regions to DRAM (*e.g.,* `pci_iomap`, `pci_resource_start`) are stored.

**(2) I/O Wrapper analysis.** Linux provides MMIO/PIO read/write functions such as `ioread32, readl, writeb`. In practice, a device drive often creates its own custom wrapper functions (with additional checks, etc.). The second analysis aims to identify such custom I/O wrapper functions that eventually call the low-level MMIO/PIO read/write APIs with the same base address and offset parameters or with another constant offset added. DEVFUZZ uses traditional call graphs and data flow analyses for this purpose.

**(3) Condition Predicate Analysis.** The last step performs data- and control-flow dependence analysis from the code reading a base address from a device struct (the result of (1) base address analysis), to the code calling MMIO/PIO wrappers (the result of (2) I/O wrapper analysis), and finally to the code checking condition predicates. The base address and offset are extracted from the arguments passed to the MMIO/PIO read functions. The summary of dependent condition predicates forms value constraints.

Though Linux source code analysis is used to construct the MMIO/PIO model, it can be reused by other OSes as the model represents a hardware device and it only models possible values of the device input.

**Example.** Figure 5 shows an example of the `pcnet32` driver that uses device MMIO input to determine its behaviors (control flows). With (1) based address analysis (not shown), we know that the driver holds a base address of the MMIO region in `dev->base_addr`. In line 7, the driver reads the base address from the field. The (2) I/O wrapper analysis (not shown) tells that the function `read_csr` is a wrapper function for MMIO reads. The result of the read is stored in the variable `csr0`. The value is checked in the while loop and the if statements. (3) condition predicate analysis can identify two value constraints here: the magic exit value 0xffff and a set of relevant bits (to fuzz). Figure 6 (b) illustrates an example MMIO/PIO model that includes value constraints for `0x20` (CSR0 register).

```
1  static int
2  pcnet32_probe1(unsigned long ioaddr, int shared,
   ↪  struct pci_dev *pdev)
3  {
4    struct pcnet32_private *lp;
5    ...
6    /* dma_alloc_coherent returns virtual DMA
     ↪  buffer address */
7    lp->init_block = dma_alloc_coherent(... ,
     ↪  &lp->init_dma_addr, ...);
8    ...
9    /* The physical DMA address is written to the
     ↪  device */
10   a->write_csr(ioaddr, 1, (lp->init_dma_addr &
     ↪  0xffff));
11   a->write_csr(ioaddr, 2, (lp->init_dma_addr >>
     ↪  16));
12   ...
13   return ret;
14 }
15
16
17 static int pcnet32_init_ring(struct net_device
   ↪  *dev)
18 {
19   struct pcnet32_private *lp = netdev_priv(dev);
20
21   for (int i = 0; i < lp->rx_ring_size; i++) {
22     ...
23     /* Allocate secondary DMA Buffers */
24     if (lp->rx_dma_addr[i] == 0) {
25       lp->rx_dma_addr[i] = dma_map_single(...);
26       ...
27     }
28     /* Write the secondary DMA address into L1
       ↪  DMA buffer */
29     lp->rx_ring[i].base =
       ↪  cpu_to_le32(lp->rx_dma_addr[i]);
30     ...
31   }
32   ...
33   return 0;
34 }
```

Figure 7: An example of setting up two-level DMA buffers in `pcnet32` driver.

**Limitation.** The precision of DEVFUZZ's MMIO/PIO model is bounded by the precision of underlying static value analysis. DEVFUZZ does not require the model to be precise. Any imprecision would simply lead to more fuzzing/testing.

### 4.3. DMA Model Generation

DEVFUZZ supports fuzzing DMA regions that are not only requested by the centralized DMA controller (*e.g.,* Intel 8257 [20]) but also performed by bus mastering. For example, a PCI device can serve as a bus master, and a manufacturer can implement their own non-standard DMA engine. Though supporting a DMA buffer requested by the standard DMA controller is relatively simple, it is challenging to reason about all the custom DMA buffers precisely. To generate the DMA model, DEVFUZZ performs hybrid dynamic/static program analyses: (1) DMA register

analysis and (2) secondary DMA buffer analysis. Moreover, DEVFUZZ performs static (3) DMA value constraints analysis, similar to that of the MMIO/DMA model (§4.2).

**(1) DMA Register Analysis.** The goal of the first DMA register analysis is to figure out which registers (mapped in MMIO regions) hold DMA buffer addresses. In Linux, the DMA kernel API `dma_alloc_coherent` is used to allocate and map a DMA region. Thus, in theory, it is possible to design static analysis that keeps track of the return values of `dma_alloc_coherent` function calls and the assignments to MMIO regions. However, we noticed that the imprecision of static analysis caused by alias analysis often hinders us from reliably collecting DMA registers.

To address the problem, DEVFUZZ makes use of profiling with a modified Linux kernel. We leverage the convention that in Linux and most OSes, a device driver informs the device about the OS-allocated DMA addresses by writing the addresses into the MMIO (or PIO) regions. Based on the observation, we instrument Linux's `dma_alloc_coherent` function with a `vmcall` hypercall, which will be trapped by QEMU, to collect the DMA addresses at runtime. We use the probe model (§4.1) to pass the probing logic and allow the driver to set up DMA buffers during a profile run. Then, we let QEMU track all MMIO (and PIO) writes and compare the store values with the collected DMA addresses. Any register (MMIO offset) to which the DMA addresses are stored is a DMA register. Though the analysis itself depends on Linux, the locations of DMA registers are device-specific properties that can be reused by other OSes.

**(2) Secondary DMA Buffer Analysis.** DEVFUZZ supports two-level chained DMA buffers in which the first-level buffer holds the pointers to the secondary buffer that actually keeps the data. These two-level DMA buffers are commonly used in many device drivers. For instance, network drivers often allocate a ring of DMA buffers as the first level and use them to communicate the status of the associated secondary DMA buffers that contain the actual payload. The problem is that the addresses of the secondary DMA buffers are not written to the devices (*i.e.,* not to the MMIO regions). Thus, we cannot identify secondary DMA buffers from the aforementioned dynamic (1) DMA register analysis.

To address the issue, DEVFUZZ resorts to static analysis. In Linux, the DMA kernel APIs `dma_alloc_coherent` and `dma_map_single` are used to initialize the first- and second-level DMA buffers, respectively. Based on the observation, DEVFUZZ identifies first-level DMA buffers based on the struct types used with the `dma_alloc_coherent` function. Then, DEVFUZZ performs data flow analysis from the return value of the `dma_map_single` function to the assignment to the first level DMA buffers. This allows DEVFUZZ to obtain the offsets in the first-level buffers that are used to keep the secondary DMA buffer addresses.

**(3) DMA Value Constraints Analysis.** Similar to the MMIO/PIO model, DEVFUZZ performs static value analysis to analyze the constraints of the DMA contents/values. One difference is that for DMA, DEVFUZZ analyzes `dma_alloc_coherent` and `dma_map_single` functions.

**Example.** Figure 7 shows how `pcnet32` initializes its two-level DMA buffers. As a part of probing/initialization function `pcnet32_probe1`, `pcnet32` maps the first-level DMA buffer using `dma_alloc_coherent` kernel API (line 7). The DMA address is written to the device's MMIO region (lines 10-11). With dynamic (1) DMA register analysis, DEVFUZZ can trap this MMIO writes to reliably find this first-level DMA buffer address. Later, in `pcnet32_init_ring`, `pcnet32` creates the second-level DMA buffers using `dma_map_single` kernel API (lines 25) and then stores the address in the first-level DMA buffer (line 29). The static (2) secondary DMA buffer analysis tracks this data flow and obtains the offset. Figure 6 (c) visualizes a simplified DMA model of `pcnet32` in which MMIO's DMA register points to the first DMA buffer and its members point to the second-level DMA buffers. The static (3) value constraints analysis example is omitted.

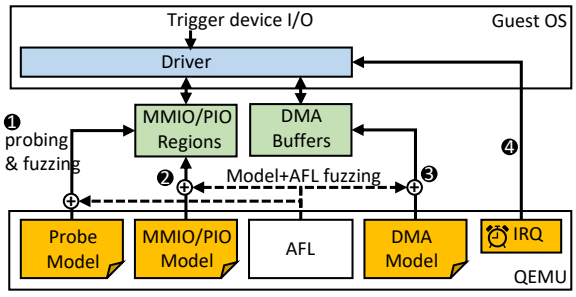


Figure 8: DEVFUZZ fuzzes MMIO, PIO, DMA data using three device models and AFL; and generates IRQ using a timer. DEVFUZZ tests both probing and post-probing phases.

**Limitation.** DEVFUZZ's profiling-based DMA register analysis may have false negatives as a profiling run does not guarantee complete code coverage. If DEVFUZZ fails to identify DMA registers, it cannot fuzz DMA buffers and thus may miss any bugs triggered by malicious DMA content. Moreover, DEVFUZZ's secondary DMA buffer and value constraints analyses share the limitations of underlying static data flow and alias analysis.

## 4.4. Model-Guided Fuzzing

Figure 8 shows how DEVFUZZ performs model-guided fuzzing, given the generated three probe, MMIO/PIO, and DMA models. Note that the three models (*e.g.,* Figure 6) hold deterministic values for the MMIO, PIO, DMA regions to pass the probing logic, or to explore diverse program paths, etc. To enable (randomized) fuzzing, DEVFUZZ mutates those regions by selecting either input from a model or a generic fuzzer with a tunable probability (❶-❸). Furthermore, DEVFUZZ uses a timer to trigger IRQs at a regular interval (❹). This allows DEVFUZZ to initiate a device driver execution (*e.g.,* an interrupt handler) that is not possible only through command-line test cases.

Next we explain the differences between DEVFUZZ's (1) probing phase and (2) post-probing phase fuzzing. To support fuzzing during probing, DEVFUZZ mutates MMIO/-PIO regions by choosing either model's or ALF's input (❶). DEVFUZZ does not use MMIO/PIO or DMA models, nor IRQ in this case. On the other hand, for post-probing fuzzing, DEVFUZZ uses the probe model as is (without

fuzzing) so that the device can be probed with no issue. Then, DEVFUZZ uses MMIO/PIO and DMA along with AFL (❷ and ❸). During fuzzing, to support two-level DMA buffers (§4.3), DEVFUZZ intercepts MMIO/PIO writes to the DMA register (provided by the DMA model) to obtain the first-level DMA buffer addresses. Then DEVFUZZ scans the first-level DMA buffers using the offsets (provided by the DMA model) to extract the second-level DMA buffer addresses. Last, DEVFUZZ enables IRQ generation (❹).

By default, DEVFUZZ uses the AFL fuzzer and mixes the model's and AFL's inputs at an equal (50-50) probability. To provide AFL with coverage information, DEVFUZZ enables Intel's Process Trace (PT) [45] for the QEMU process as soon as it is created. This allows DEVFUZZ to collect coverage information during the booting phase as well. Unlike kAFL[46], RedQueen[35], DEVFUZZ does not require modification in host or guest OSes.

# 5. Implementation

DEVFUZZ is implemented (with 7100 LoC C++) based on S2E 2.0 [31] for symbolic execution, QEMU v5.1 [32] for hardware emulation, LLVM 13.0 for program analysis, and AFL v2.57 [17] for feedback-directed fuzzing. The source codes will be open-sourced and released on publication.

DEVFUZZ currently supports four bus types: PCI, USB, RapidIO, and I2C. The three non-PCI buses are supported via PCI-to-USB, PCI-to-RapidIO, and PCI-to-I2C (bridge) adaptors, and thus symbolic probing starts from the same PIO regions. When a new bus type is desired, users need to add the adaptor similarly if there exists a PCI bridge adaptor for that bus. Otherwise, users should manually specify the bus' own PIO region (similar to that of PCI) and add any bus-specific device enumeration protocols.

# 6. Evaluation

In this section, we start by describing our evaluation methodology (§6.1) and report the experimental results. First, we evaluate how well DEVFUZZ can find bugs in Linux device drivers (§6.2). Second, we evaluate the effectiveness of DEVFUZZ's symbolic probing execution (§6.3). Third, we compare DEVFUZZ with two prior works (§6.4). Fourth, we report how much each model of DEVFUZZ improves fuzzing code coverage and conduct a sensitivity study on model use probability (§6.5). Fifth, we evaluate the quality of DEVFUZZ's models and compare them with expert-derived (manual) models (§6.6).

Last, we evaluate how well DEVFUZZ can reuse Linux-based models for fuzzing FreeBSD (§6.7) and Windows (§6.8) device drivers.

## 6.1. Evaluation Methodology

We evaluated DEVFUZZ using device drivers of different bus types (PCI, USB, RapidIO, I2C) and various OSes: Linux Kernel v5.11, FreeBSD release 12.2, and Windows 10, running on S2E 2.0 [31] and QEMU v5.1 [32]. Our testing machine has four Intel(R) Xeon(R) Gold 5218 CPUs (2.30GHz, 16 cores each) and 128GB of DDR4 memory. The host OS is Ubuntu 20.04.3 LTS with Linux v5.4.0. For Linux, we randomly selected and tested 108 PCI, 31 USB, 1 RapidIO, and 10 I2C device drivers (150 drivers in total).

TABLE 2: Bugs detected and reported by DEVFUZZ.

| Type | Tested | Probed | Bugs | Patched |
|---|---|---|---|---|
| Linux-PCI | 108 | 83 | 54 | 35 |
| Linux-USB | 31 | 19 | 6 | 3 |
| Linux-RapidIO | 1 | 1 | 2 | 0 |
| Linux-I2C | 10 | 10 | 1 | 1 |
| FreeBSD-PCI | 25 | 14 | 8 | 2 |
| Windows-PCI | 16 | 8 | 1 | 0 |

For FreeBSD and Windows, we searched for the device drivers in which we can find the equivalent ones in Linux. We found 25 FreeBSD and 16 Windows PCI drivers in total. We use the device models generated from Linux to test the FreeBSD and Windows drivers. We manually analyzed all the bug reports, wrote patches (whenever possible), and submitted them for review.

Given device models, for probing phase fuzzing, we repeatedly plug and unplug the target device driver (module) and fuzz its probing functions. For post-probing phase fuzzing, we need test cases to trigger device accesses. Besides timer-based IRQs, we used the following tools to test device drivers from userspace. For network devices (*e.g.,* Ethernet, CAN, ATM, ISDN, RapidIO, IEEE1394 FireWire), we used net-tools [47], can-utils [48], RapidIO_RRMAP [49], inarpd, sethdlc [50], and nosy-dump [51]. For framebuffer devices, we used Xorg [52]. For drivers that create a device file under `/dev/`, we used cat, dd [53], trinity [54], and ltp [55]. Following the best practices for fuzzing evaluation [56], we run fuzzing experiments 3 times for 24 hours each.

## 6.2. Finding Bugs in Linux Device Drivers

We first focus on our experiments with Linux device drivers. Table 2 (Linux- rows) summarizes basic test results. Among 150 tested drivers (including all PCI, USB, RapidIO, and I2C drivers), DEVFUZZ successfully probed 112 devices (about 75%). Later in §6.3, we discuss the cases DEVFUZZ fail to use symbolic execution to complete probing in detail.

**Fuzzing performance.** We measured the DEVFUZZ's fuzzing speed by inspecting the AFL's stat files. DEVFUZZ performs 1.15 executions per second on average, where each execution contains about 100 MMIO/PIO accesses of the devices. Note that given the models, DEVFUZZ performs QEMU/KVM-based full-speed fuzzing without any symbolic (or concolic) execution.

**Detected bugs.** In total, the proposed model-guided fuzzing detected 63 bugs/crashes. At the time of this submission, we submitted 39 patches, all of which have been merged into the mainline Linux kernel. We also received one CVE assignment, CVE-2022-0487[57], for one USB bug.

Table 3 (IDs 1-63) reports the detailed information of each detected bug: bus type, source file, function name, bug description, if the bug is patched, and bug location. For the bug location, "Probe" means the bug is detected in a probing logic, "IRQ" in an interrupt handler, and "Others" for other locations. For Linux, there were 27 Probe, 3 IRQ, and 31 Others cases. The results demonstrate that DEVFUZZ can effectively detect many bugs in Linux device drivers of various bus types. In Appendix A and Appendix B,

TABLE 3: Bugs Reported By DEVFUZZ.

| | Type-# | File | Function | Description | Patched? | Loc. |
|---|---|---|---|---|---|---|
| 1 | Linux-PCI | drivers/media/pci/bt8xx/bt878.c | bt878_irq | null pointer dereference | Y | IRQ |
| 2 | Linux-PCI | drivers/misc/cardreader/alcor_pci.c | alcor_pci_find_cap_offset | null pointer dereference | Y | Probe |
| 3 | Linux-PCI | drivers/gpu/drm/radeon/radeon_object.c | radeon_bo_evict_vram | null pointer dereference | Y | Probe |
| 4 | Linux-PCI | drivers/crypto/qat/qat_c3xxxvf/adf_drv.c | adf_iov_putmsg | user memory access | Y | Probe |
| 5 | Linux-PCI | drivers/crypto/qat/qat_common/adf_vf_isr.c | adf_vf_isr_resource_alloc | double irq free | Y | Probe |
| 6 | Linux-PCI | drivers/usb/gadget/udc/amd5536udc_pci.c | pci_write_config_word | null pointer dereference | Y | Probe |
| 7 | Linux-PCI | drivers/staging/comedi/drivers/das800.c | das800_attach | request irq format error, /proc/irq file not created | Y | Probe |
| 8 | Linux-PCI | sound/pci/rme9652/rme9652.c | snd_rme9652_free | disable already disabled device | Y | Probe |
| 9 | Linux-PCI | sound/pci/rme9652/hdspm.c | snd_hdspm_free | disable already disable device | Y | Probe |
| 10 | Linux-PCI | sound/pci/rme9652/hdsp.c | snd_hdsp_card_free | disable already disabled device | Y | Probe |
| 11 | Linux-PCI | drivers/staging/comedi/drivers/cb_pcidas64.c | auto_attach | request irq format error, /proc/irq file not created | Y | Probe |
| 12 | Linux-PCI | drivers/staging/comedi/drivers/cb_pcidas.c | cb_pcidas_auto_attach | request irq format error, /proc/irq file not created | Y | Probe |
| 13 | Linux-PCI | drivers/net/can/c_can/c_can.c | c_can_pm_runtime_enable | unbalanced power management reference counter | Y | Probe |
| 14 | Linux-PCI | drivers/net/can/c_can/c_can_pci.c | pci_iounmap | use after free | Y | Other |
| 15 | Linux-PCI | drivers/net/arcnet/com20020-pci.c | com20020pci_probe | null pointer dereference | Y | Probe |
| 16 | Linux-PCI | drivers/isdn/hardware/mISDN/mISDNipac.c | WriteISAC | null pointer dereference | Y | Probe |
| 17 | Linux-PCI | drivers/gpu/drm/drm_fb_helper.c | drm_client_buffer_vunmap | null pointer dereference | Y | Other |
| 18 | Linux-PCI | drivers/atm/idt77105.c | zatm_start/stop | null pointer dereference | Y | Other |
| 19 | Linux-PCI | drivers/atm/uPD98402.c | zatm_start | null pointer dereference | Y | Probe |
| 20 | Linux-PCI | drivers/atm/lanai.c | lanai_dev_open | unable to handle kernel page fault | Y | Probe |
| 21 | Linux-PCI | drivers/atm/eni.c | suni_stop | null pointer dereference | Y | Probe |
| 22 | Linux-PCI | drivers/gpu/drm/ast/ast_drv.c | driver_detach | memory leak | Y | Other |
| 23 | Linux-PCI | drivers/gpu/drm/qxl/qxl_display.c | qxl_destroy_monitors_object | user memory access | Y | Probe |
| 24 | Linux-PCI | drivers/net/wan/lmc/lmc_main.c | lmc_init_one | format string issue | Y | Probe |
| 25 | Linux-PCI | drivers/net/wan/lmc/lmc_main.c | lmc_init_one | null pointer dereference | Y | Probe |
| 26 | Linux-PCI | drivers/nvme/host/hwmon.c | nvme_hwmon_get_smart_log | null pointer dereference | Y | Other |
| 27 | Linux-PCI | drivers/nvme/host/pci.c | nvme_timeout | null pointer dereference | Y | Other |
| 28 | Linux-PCI | drivers/nvme/host/pci.c | nvme_reset_work | IRQ double free | Y | Other |
| 29 | Linux-PCI | drivers/atm/eni.c | eni_ioctl | null pointer dereference | Y | Other |
| 30 | Linux-PCI | drivers/atm/firestream.c | top_off_fp | use after free | Y | Other |
| 31 | Linux-PCI | drivers/atm/he.c | he_service_rbrq | null pointer dereference | N | Other |
| 32 | Linux-PCI | drivers/atm/idt77252.c | idt77252_interrupt | null pointer dereference | N | Other |
| 33 | Linux-PCI | drivers/atm/lanai.c | atm_dev_deregister | unable to handle page fault | N | Other |
| 34 | Linux-PCI | drivers/atm/nicstar.c | ns_init_card_error | invalid opcode | N | Probe |
| 35 | Linux-PCI | drivers/atm/zatm.c | zatm_open | rcu: INFO: rcu_sched detected stalls on CPUs/tasks | N | Probe |
| 36 | Linux-PCI | drivers/infiniband/hw/hfil/pcie.c | ioremap | kernel panic - Fatal exception in interrupt | N | IRQ |
| 37 | Linux-PCI | drivers/net/can/c_can/c_can.c | c_can_start_xmit | kernel panic - Fatal exception in interrupt | N | IRQ |
| 38 | Linux-PCI | drivers/net/can/sja1000/sja1000.c | can_put_echo_skb | use after free | N | Other |
| 39 | Linux-PCI | drivers/net/can/sja1000/sja1000.c | sock_efree | use after free | N | Other |
| 40 | Linux-PCI | net/can/af_can.c | sock_efree | use after free | N | Other |
| 41 | Linux-PCI | drivers/net/wan/lmc/lmc_main.c | lmc_mii_readreg | null pointer dereference | N | Other |
| 42 | Linux-PCI | drivers/scsi/arcmsr/arcmsr_hba.c | arcmsr_abort | unable to handle page fault | N | Other |
| 43 | Linux-PCI | drivers/net/ethernet/cadence/macb_pci.c | macb_remove | use after free | Y | Other |
| 44 | Linux-PCI | sound/pci/vx222/vx222.c | snd_vx222_create | null pointer dereference | Y | Probe |
| 45 | Linux-PCI | drivers/net/can/c_can/c_can_ethtool.c | c_can_get_drvinfo | unable to handle page fault | Y | Other |
| 46 | Linux-PCI | drivers/scsi/dc395x.c | dc395x_init_one | null pointer dereference | Y | Probe |
| 47 | Linux-PCI | drivers/net/ethernet/smsc/epic100.c | epic_remove_one | use after free | Y | Other |
| 48 | Linux-PCI | drivers/net/ethernet/smsc/epic100.c | epic_rx | buffer overflow | N | Other |
| 49 | Linux-PCI | drivers/net/ethernet/marvell/sky2.c | sky2_mac_intr | null pointer dereference | N | Other |
| 50 | Linux-PCI | drivers/net/ethernet/realtek/r8169_main.c | rtl_rx | out of bound read | N | Other |
| 51 | Linux-PCI | drivers/net/ethernet/realtek/8139cp.c | cp_rx_poll | skb over panic | N | Other |
| 52 | Linux-PCI | drivers/net/vmxnet3/vmxnet3_drv.c | vmxnet3_rq_rx_complete | BUG_ON statement | N | Other |
| 53 | Linux-PCI | drivers/net/ethernet/intel/e1000/e1000_main.c | e1000_clean_rx_irq | skb over panic | N | Other |
| 54 | Linux-PCI | drivers/net/ethernet/intel/e1000e/netdev.c | e1000_clean_rx_irq | skb over panic | N | Other |
| 55 | Linux-USB | drivers/memstick/host/rtsx_usb_ms.c | memstick_free_host | use after free | Y | Other |
| 56 | Linux-USB | drivers/memstick/host/rtsx_usb_ms.c | rtsx_usb_ms_drv_remove | use after free | Y | Other |
| 57 | Linux-USB | drivers/net/wireless/marvell/mwifiex/main.c | mwifiex_fw_dpc | divide error | N | Probe |
| 58 | Linux-USB | drivers/net/wireless/marvell/libertas/if_usb.c | usb_tx_block | URB submitted while active | N | Other |
| 59 | Linux-USB | kernel/dma/swiotlb.c | swiotlb_tbl_sync_single | Attempt for buffer overflow. | N | Other |
| 60 | Linux-USB | drivers/net/usb/hso.c | hso_mux_serial_read | kernel panic | N | Other |
| 61 | Linux-RapidIO | drivers/rapidio/rio-scan.c | rio_enum_mport | kernel WARN and create /sys entry error | N | Probe |
| 62 | Linux-RapidIO | drivers/rapidio/rio-scan.c | rio_scan_alloc_net | memory leak | N | Other |
| 63 | Linux-I2C | drivers/iio/imu/bmi160/bmi160_core.c | bmi160_chip_init | Kernel WARN regulator not disabled | Y | Probe |
| 64 | FreeBSD-PCI | sys/modules/drm2/i915kms.ko | make_dev_sv | kernel panic, duplicate /dev/agpgart | N | Probe |
| 65 | FreeBSD-PCI | sys/modules/sound/driver/hda/snd_hda.c | snd_hda | crash and non-responsive console | N | Other |
| 66 | FreeBSD-PCI | sys/dev/arcmsr/arcmsr.c | arcmsr_drain_donequeue | device can control data pointer | N | IRQ |
| 67 | FreeBSD-PCI | sys/dev/stge/if_stge.c | stge_detach | null pointer dereference | Y | Other |
| 68 | FreeBSD-PCI | sys/dev/twa/tw_cl_intr.c | tw_cl_create_event | stack overflow | N | Probe |
| 69 | FreeBSD-PCI | sys/dev/my/if_my.c | my_attach | kernel panic | N | Probe |
| 70 | FreeBSD-PCI | sys/dev/tl/if_tl.c | tl_start | panic during transmit. deadlock. | N | Other |
| 71 | FreeBSD-PCI | sys/dev/axgbe/if_axgbe_pci.c | axgbe_if_attach_pre | resource leak | Y | Probe |
| 72 | Windows10-PCI | AMDXgbe.sys | - | PAGE FAULT IN NONPAGED AREA | N | Other |

TABLE 4: Symbolic Execution Time and Path Explored

| Item | Average | Median | 99th Percentile |
|---|---|---|---|
| Number of paths explored | 1637 | 100 | 25282 |
| Execution time (seconds) | 419 | 42 | 12400 |

we discuss two detected bugs (IDs 1 and 48) in detail to illustrate the benefits of the proposed MMIO/PIO/DMA models and IRQ generation.

### 6.3. Symbolic Probing Execution

This section focuses on evaluating the probe model generation (symbolic execution) component of DEVFUZZ. As reported earlier, we tested 150 Linux drivers and probed 112 (about 75%). Table 4 reports the average, median, and 99th percentile of the number of paths explored and execution time. Recall that DEVFUZZ directs symbolic execution to stop exploring a program path once the probing fails on that path, based on an error message (See §4.1). For those successfully probed drivers, the result shows that most cases were completed quickly, yet there were long tails. The 99th percentile data shows that DEVFUZZ explored 25K paths for more than 3 hours. We also confirmed that in all cases, the generated probe model (after a successful symbolic probing) enables the corresponding device driver to pass the dynamic probing logic and can be successfully loaded for fuzzing.

On the other hand, Table 2 shows that DEVFUZZ fail to probe 38 devices (26 PCI 12 USB devices). DEVFUZZ successfully probed all the RapidIO and I2C devices. We manually reviewed those 38 cases where DEVFUZZ could

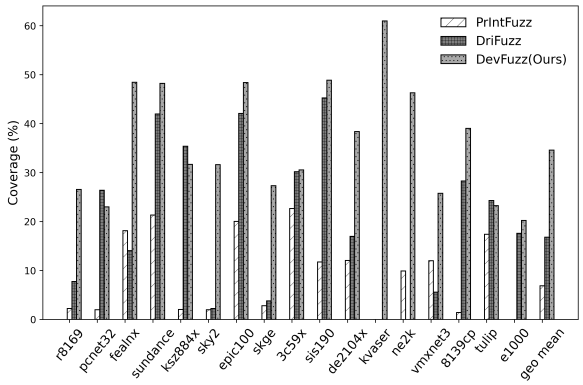


Figure 9: Code coverage comparison with network device drivers: PrIntFuzz vs. Drifuzz vs. DevFuzz (this work).

not generate probe models. There were three root causes:

**(1) Complex Probing Logic** 25/38 (64%) cases fall under this category. Though grouped together, under the hood there are two slightly different reasons. First, DEVFUZZ cannot generate a device probe model simply due to the complex probing logic. For example, we observed DEVFUZZ cannot generate results for `realtek` wireless driver and `e100` ethernet driver within an 8-hour symbolic execution budget. The reason is that the probe function involves a complex checksum check that symbolic execution can't solve. The second case is different. The driver requires other resources not modeled by the current implementation of DEVFUZZ. For instance, `AMDGPU` driver checks the VGA BIOS content, and `nVIDIA` driver checks the ACPI table. DEVFUZZ does not run symbolic execution for those device memory regions.

**(2) IRQ** 10/38 (28%) drivers wait for the device to trigger an IRQ during probing. DEVFUZZ's symbolic probing does not generate an IRQ. This issue may be addressed by adding an IRQ detection and triggering mechanism in the symbolic execution engine.

**(3) DMA** 3/38 (8%) cases require DMA, but DEVFUZZ does not support DMA during symbolic execution. Similar to the above IRQ, the problem may be resolved with additional DMA support in the symbolic execution engine.

## 6.4. Comparing DEVFUZZ with Prior Work

This section aims to compare DEVFUZZ with two prior device driver fuzzing solutions PrIntFuzz [18] and Drifuzz [19] in terms of probing success rate and code coverage. For the experiments, we randomly selected 17 network devices for which we can use the common, relatively complex test case. Figure 9 shows the code coverage results.

### 6.4.1. DEVFUZZ vs. PrIntFuzz

PrIntFuzz [18] uses static analysis to generate a sequence of MMIO values to help pass the driver's probing logic and perform fault injection fuzzing for the probing code. We found that PrIntFuzz could not probe 9 out of 17 network devices (`e1000` and `kvaser` are not supported) due to the unsoundness of its static analysis. On the other hand, DevFuzz's symbolic execution could probe all the 17 network devices at the cost of symbolic execution time. When tested with all the 150 Linux device drivers, we found that PrIntFuzz was able to probe 61 devices (40.6%), which is smaller than DEVFUZZ's symbolic probing (75%, See §6.3). Note that PrIntFuzz also reports a 43.3% success probing rate for network drivers in their paper.

Furthermore, PrIntFuzz is mainly designed for probing-phase fuzzing and cannot generate a syscall template for fuzzing the post-probing phase for many drivers. The reason is that PrIntFuzz uses DIFUZE's [13] static analysis to identify syscall interfaces and relevant syscall parameters for drivers, yet the static analysis fails to generate those test cases for the post-probing phase. After probing, PrIntFuzz blindly relies on syzkaller [34] to fuzz the syscall parameters, MMIO spaces, and DMA buffers. As a result, PrIntFuzz leads to the least code coverage (6.94% on geometric mean).

### 6.4.2. DEVFUZZ vs. Drifuzz

Drifuzz [19] relies on concolic execution. We tested Drifuzz with 17 network devices by running golden seed generation and 24-hour fuzzing. As Drifuzz only supports Ethernet and USB devices, it does not support one CAN device `kvaser` and fails to boot with the `ne2k` device. Drifuzz successfully probed 12/17 devices. Unfortunately, Drifuzz does not report the successful probing rate in their paper. The reason for failure cases is that their concolic execution falls back to "forced execution" to pass certain branch conditions. This may lead to infeasible cases, resulting in probing failure. Their search algorithm is based on simple heuristics and cannot guarantee the input with the highest score pass the probing conditions.

Drifuzz's post-probing testing combines concolic execution and fuzzing. Its concolic execution may pass more path constraints, compared with DevFuzz's static analysis-generated model for some cases, leading to higher code coverages than DevFuzz (*e.g.,* `pcnet32`, `ksz884z` in Figure 9). However, the concolic execution makes fuzzing speed very slow, compared to DevFuzz. We observed that DevFuzz is 10x-20x faster. Moreover, Drifuzz leads to lower code coverages than DevFuzz for the remaining device drivers (16.8% vs 34.6% on geometric mean). One reason is that a driver execution is inherently non-deterministic due to the concurrency between driver functions and interrupt handlers. As a result, the constraints solved from one execution often were not able to be used in the next execution as two executions diverge, reducing the effectiveness of a concolic execution.

## 6.5. Effectiveness of DEVFUZZ Models

For fuzzing, DEVFUZZ mixes the model's and AFL's inputs (by default with 50-50% probability) with the goal to leverage both the modeled constraints and feedback-directed mutation. In this section, we first evaluate how much each model of DEVFUZZ improves fuzzing code coverage (on top of what AFL can offer) and then conduct a sensitivity study on different model use probabilities.

### 6.5.1. Models' Impacts on Code Coverage

We first evaluate how each DEVFUZZ model improves fuzzing code coverage for the same set of 17 network devices. For probing-phase fuzzing (not shown for space), when pure AFL is used, the code coverage was 6.88% on

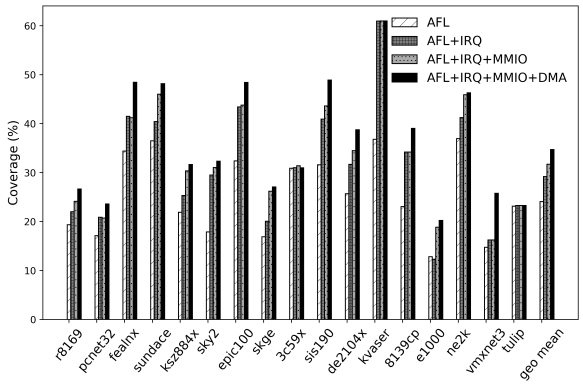


Figure 10: Code coverage comparison: Pure AFL vs. Model-based (adding one each) fuzzing.

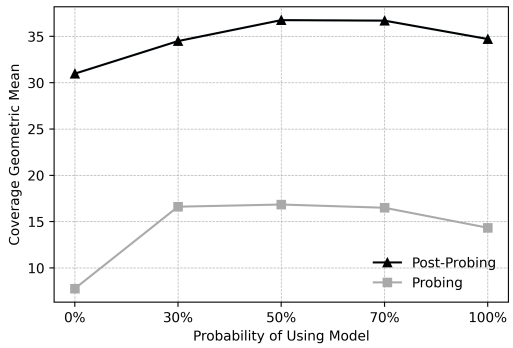


Figure 11: Code coverage (geometric mean) with varying model use probabilities.

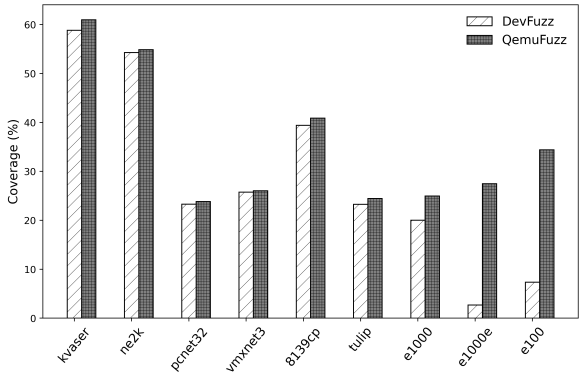


Figure 12: Code coverage comparison: DevFuzz (auto-generated models) vs. QemuFuzz (expert-derived models)

the geometric mean. When DEVFUZZ's probe model is used in combination (with the default 50% probability), the code coverage was increased to 16.84%. Many drivers follow a very strict probing protocol with special values and/or a specific temporal order of values read from the device. Thus, it is hard for a pure AFL to pass such complex probing logic.

Figure 10 shows the code coverage for post-probing phase fuzzing. For this experiment, we incrementally add DEVFUZZ's IRQ, MMIO/PIO model, and DMA model atop a pure AFL. The `kvaser` driver has no DMA support and is relatively simple, and thus the MMIO/DMA models had a minor impact. Overall, the figure shows that each model gradually increases the code coverage.

TABLE 5: DEVFUZZ model completeness analysis, reporting the number of modeled registers over the number of total registers (counted via manual code reviews)

| Driver | MMIO&PIO Regs | DMA Regs | Total |
|---|---|---|---|
| 3c59x | 8/9 | 1/1 | 9/10 |
| ksz884x | 4/5 | 1/1 | 5/6 |
| pcnet32 | 3/5 | 3/3 | 6/8 |
| skge | 8/10 | 2/2 | 10/12 |
| kvaser | 7/7 | 0/0 | 7/7 |
| ne2k | 4/5 | 0/0 | 4/5 |
| 8139cp | 6/6 | 2/2 | 8/8 |
| tulip | 6/7 | 1/1 | 7/8 |
| vmxnet3 | 3/5 | 4/9 | 7/14 |
| sundance | 9/10 | 1/1 | 10/11 |

#### 6.5.2. Sensitivity Study on Fuzzing Probability

DEVFUZZ selects either model-generated input or AFL-generated one at some probability (by default 50%). We studied how configurable probability affects code coverage. Figure 11 shows the geometric mean of code coverages across all the 17 tested drivers while varying the probability from 0% (AFL only) to 100% (Model only). For both probing and post-probing fuzzing, using some combination of the two achieved higher code coverage than those using only one technique. Based on the result, we set the default probability to 50%. The main limitation of using AFL only (0%) is that the fuzzing input space is too huge to explore without any constraints. Using model only (100%) is not effective either as it may give up the opportunity to explore other paths such as error handling code paths.

### 6.6. Quality of DEVFUZZ Models

This section focuses on evaluating the completeness and correctness of the generated models. As an oracle does not exist, we propose two proxy evaluation methods: one based on the number of modeled registers, and another based on expert-derived manual models.

#### 6.6.1. Model Completeness Analysis

We first evaluate the quality of DEVFUZZ's models by counting how many registers to which a device driver refers are indeed (properly) included in DEVFUZZ's automatically generated models. Any missing register implies incompleteness. Note that we count the total number of registers from the device driver's perspective as counting the number of actual registers in hardware requires hardware firmware analysis. We believe this is a good proxy to evaluate the quality of DEVFUZZ's models as the purpose of modeling is to explore more program paths inside device drivers. As the study requires manual code review, we performed the analysis on 10 random drivers.

Table 5 summarizes the result of the modeled vs. total MMIO, PIO, DMA registers analysis. In total, DEVFUZZ's models include 62-100% of registers that are read by a device driver: *i.e.,* there were some missing registers (false negative models) due to the limitations in static analyses discussed in §4.2. We did not encounter any false positive registers (that appear in a model but are not accessed by a device driver). Though DEVFUZZ does not guarantee

completeness, our empirical study in §6.5 and §6.6.2 shows that DEVFUZZ models are good enough to effectively improve the fuzzing code coverage.

#### 6.6.2. Comparison to Expert-derived Models

Furthermore, we attempt to evaluate the quality of DEVFUZZ's models by comparing DEVFUZZ's models with "expert-derived" manual models. For this experiment, we regard a QEMU's virtual device as an oracle expert-derived model. QEMU as a whole-system emulator includes a set of virtual devices, designed and implemented by (third-party) experts, intended to mimic actual hardware behaviors. Based on this observation, we built QEMUFuzz that combines a QEMU's virtual device (as a model) with the AFL fuzzer. Like DEVFUZZ, QEMUFuzz fuzzes a target device driver by choosing input from either a QEMU virtual device or the AFL fuzzer. This allows us to make an apple-to-apple fuzzer-level comparison and evaluate the impacts of (precise) models on the end-to-end fuzzing code coverage.

Figure 12 shows the code coverage comparison between DEVFUZZ and QEMUFuzz for 9 network drivers, whose virtual devices are available in QEMU. As expected, QEMUFuzz's virtual device (oracle) passes the dynamic probing and supports post-probing-phase fuzzing for all device drivers. However, DEVFUZZ could not complete symbolic probing for two drivers `e1000e` and `e100`, leading to low code coverage. Upon investigation, the root cause was found to be unmodeled resources during symbolic execution (similar to `AMDGPU` and `nVIDIA` cases in §6.3). On the other hand, for the remaining 6 drivers, DEVFUZZ was able to probe devices and achieve marginally smaller code coverages.

We found four bugs/crashes during this experiment. There were two false negatives in `e1000e` and `vmxnet3`: QEMUFuzz detected a bug but DEVFUZZ could not. The `e1000e` case appears as DEVFUZZ could not succeed in probing as mentioned earlier. The other case in `vmxnet3` (that DEVFUZZ could probe) was due to missing DMA registers (See Table 5). The other two bugs in `8139cp` and `e1000` (which DEVFUZZ could probe) were found by both fuzzers. The above two code coverage and false negative experiments demonstrate that DEVFUZZ could generate device models that are reasonably good enough for fuzzing, yet symbolic probing and static analysis could be a bottleneck.

### 6.7. Finding Bugs in FreeBSD Device Drivers

Next, we tested FreeBSD device drivers. The goal of this experiment is to demonstrate that a device model learned from one OS can be transferred to test device drivers of other OSes. To this end, we first find out common devices supported by both Linux and FreeBSD. We then use Linux-based device models to fuzz-test FreeBSD device drivers.

It is worth noting that we found it not trivial to find common device drivers. We used the device VID/PID matching method to find the common devices supported by both OSes. For Linux, we used `modinfo` command. For example, running `modinfo arcmsr.ko` returns `pci:v000017D3d00001110sv*sd*bc*sc*i*`, which implies that the device is a PCI device, the VID is `0x17d3`, and the PID is `0x1110`. However, to the best of our knowledge, we were not able to find such an easy-to-use command-line tool. Instead, we reviewed the driver codes and found that FreeBSD PCI devices use `pci_get_vendor(dev)` and `pci_get_devid(dev)` APIs to get the VID and PID, respectively. We wrote a static analysis tool based on this heuristic to extract supported PCI devices. Finally, we found 25 PCI devices that are supported by both Linux and FreeBSD.

TABLE 6: Test FreeBSD with DEVFUZZ

| BSD Driver | Linux Driver | Model Size (LoC) | Bug? |
|---|---|---|---|
| i915kms.ko | i915.ko | 69 | Y |
| arcmsr.ko | arcmsr.ko | 280 | Y |
| snd_hda.ko | snd_intel8x0.ko | 66 | Y |
| iwlwifi.ko | iwlwifi.ko | 72 | N |
| mga.ko | mgag200.ko | 53 | N |
| if_stge.ko | dl2k.ko | 55 | Y |
| twa.ko | 3w-9xxx.ko | 87 | Y |
| atapci.ko | pata_pdc202xx_old.ko | 155 | Y |
| if_tx.ko | epic100.ko | 109 | N |
| isci.ko | isci.ko | 220 | N |
| sge.ko | sis190.ko | 179 | N |
| if_tl.ko | tlan.ko | 64 | Y |
| if_axp.ko | amd_xgbe.ko | 32 | Y |
| age.ko | atl1.ko | 158 | N |

TABLE 7: Test Windows 10 with DEVFUZZ

| Windows Driver | Linux Driver | Model Size (LoC) | Bug? |
|---|---|---|---|
| storahci.sys | acard_ahci.ko | 137 | N |
| amdacpbus.sys | acp3x.ko | 64 | N |
| AMDXgbe.sys | amd_xgbe.ko | 32 | Y |
| arcmsr.sys | arcmsr.ko | 280 | N |
| L160x64.sys | atl1.ko | 159 | N |
| L260x64.sys | atl2.ko | 133 | N |
| usbehci.sys | ich9_ehci.ko | 62 | N |
| nvme.sys | nvme.ko | 88 | N |

Among them, DEVFUZZ was able to probe 14 (56%) device drivers. Table 6 lists all the probed FreeBSD drivers, corresponding Linux drivers, model size, and whether a bug is detected in the FreeBSD driver. DEVFUZZ deteceted 8 bugs in total; 2 have been patched. Table 2 lists the detailed information. In , we describe one representative bug case.

### 6.8. Finding Bugs in Windows Device Drivers

We used a similar method to test 16 Windows device drivers where 8 could be probed. Table 7 lists all the probed/tested Windows drivers along with Linux drivers from which device models were created. DEVFUZZ found 1 crash in Windows `AMDXgbe` driver during fuzz-testing. Since Windows drivers are closed-source blobs, we did not analyze the root cause.

## 7. Conclusion

This paper presents DEVFUZZ, a model-based, automatic, device-less device driver fuzz testing tool. DEVFUZZ supports various device drivers and bus types including but not limited to PCI, USB, RapidIO, and I2C. The hybrid use of symbolic execution and static/dynamic program analyses allows DEVFUZZ to automatically fuzz-test device drivers without actual devices as well as to efficiently and effectively detect many bugs in various operating systems such as Linux, FreeBSD, and Windows.

## References

[1] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "Thunderclap: Exploring

vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[2] "Playstation 3 jailbreak," https://en.wikipedia.org/wiki/PlayStation_3_Jailbreak.

[3] "The first ps4 kernel exploit: Adieu," https://fail0verflow.com/blog/2017/ps4-namedobj-exploit/.

[4] "Xbox exploits," https://xboxdevwiki.net/Exploits#Savegames.

[5] "An ios zero-click radio proximity exploit odyssey," https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

[6] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 279–292.

[7] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "POTUS: Probing Off-The-Shelf USB drivers with symbolic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[8] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing Closed-Source binary device drivers with DDT," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[9] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical concolic testing techniques for COTS operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 689–701.

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[12] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 291–307.

[13] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[14] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[15] H. Peng and M. Payer, "USBFuzz: A framework for fuzzing USB drivers by device emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2559–2575.

[16] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks."

[17] "American fuzzy lop - a security-oriented fuzzer," https://github.com/google/AFL.

[18] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 404–416. [Online]. Available: https://doi.org/10.1145/3533767.3534226

[19] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1275–1290. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/shen-zekun

[20] "Intel 8257 dma controller," https://en.wikipedia.org/wiki/Intel_8257.

[21] "U-boot," https://en.wikipedia.org/wiki/Das_U-Boot.

[22] "Linux and the device tree," https://www.kernel.org/doc/html/latest/devicetree/usage-model.html.

[23] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[24] A. Mera, B. Feng, L. Lu, and E. Kirda, "Dice: Automatic emulation of dma input channels for dynamic firmware analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1938–1954.

[25] I. Pustogarov, Q. Wu, and D. Lie, "Ex-vivo dynamic analysis framework for android device drivers," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1088–1105.

[26] W. Zhao, K. Lu, Q. Wu, and Y. Qi, "Semantic-informed driver fuzzing without both the hardware devices and the emulators," in *Network and Distributed Systems Security (NDSS) Symposium*, April 2022.

[27] "Ibm cell be iommu," https://github.com/torvalds/linux/blob/master/arch/powerpc/platforms/cell/iommu.c.

[28] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, "The price of safety: Evaluating iommu performance," in *The Ottawa Linux Symposium*, 2007, pp. 9–20.

[29] "Arm smmu version 1 and 2," https://wiki.osdev.org/ARM_SMMU_versions_1_and_2.

[30] "Playstation 3 jailbreak on ti84 calculator," https://hackaday.com/2010/09/10/playstation-3-exploit-using-a-ti84-calculator/.

[31] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–278. [Online]. Available: https://doi.org/10.1145/1950365.1950396

[32] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[33] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[34] "Syzkaller is an unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller.

[35] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[36] Z.-M. Jiang, J.-J. Bai, J. Lawall, and S.-M. Hu, "Fuzzing error handling code in device drivers based on software fault injection," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 128–138.

[37] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[38] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[39] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2541–2557.

[40] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[41] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static detection of unsafe DMA accesses in device drivers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1629–1645.

[42] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with Two-Dimensional consistency checking," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2471–2488.

[43] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2041–2058.

[44] F. Fowze, D. Tian, G. Hernandez, K. Butler, and T. Yavuz, "Proxray: Protocol model learning and guided firmware analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1907–1928, 2021.

[45] "Intel process tracing," https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[46] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[47] "Net tools," https://github.com/ecki/net-tools.

[48] "Can utils," https://github.com/linux-can/can-utils.

[49] "Rapidio remote memory access platform software," https://github.com/RapidIO/RapidIO_RRMAP.

[50] "Generic hdlc layer," https://mirrors.edge.kernel.org/pub/linux/utils/net/hdlc/.

[51] "Linux kernel," https://github.com/torvalds/linux.

[52] "X server," https://github.com/freedesktop/xorg-xserver.

[53] "Coreutils - gnu core utilities," https://www.gnu.org/software/coreutils/.

[54] "Trinity: Linux system call fuzzer," https://github.com/kernelslacker/trinity.

[55] "Linux test project," https://github.com/linux-test-project/ltp.

[56] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[57] "CVE-2022-0487," https://nvd.nist.gov/vuln/detail/CVE-2022-0487.

[58] "media: bt878: do not schedule tasklet when it is not setup," https://github.com/torvalds/linux/commit/a3a54bf.

# Appendix

## 1. BUG 1: An IRQ Handler Bug in `bt878`

Figure 13 shows a null pointer deference bug found in the IRQ handler of `bt878` driver. The driver reads `stat` and `mask` from the device's MMIO registers in lines 4-5. It then checks whether the device actually raised the IRQ at line 6. The PCI system allows multiple devices to share the same IRQ line; when the host receives an IRQ request, it needs to check which device actually raised the IRQ. Lines 4-6 check whether BT878 has raised an IRQ or not. In line 8, it checks some flags in the `BT878_AINT_STAT` register. If the check succeeds, it will schedule a tasklet (`&bt->tasklet`) in line 10. The tasklet (one of the bottom half logics), allows a non-critical part of IRQ handling to be postponed to improve system responsiveness. In the bottom half, the tasklet calls a callback function (`bt->tasklet.callback`)

```
1  // bt878.c, bt878.ko
2  static irqreturn_t bt878_irq(int irq, void
   ↪  *dev_id)
3  {
4    stat = btread(BT878_AINT_STAT);
5    mask = btread(BT878_AINT_MASK);
6    if (!(astat = (stat & mask)))
7      return IRQ_NONE;
8    if (astat & BT878_ARISCI) {
9      bt->finished_block = (stat & BT878_ARISCS)
       ↪  >> 28;
10 -    tasklet_schedule(&bt->tasklet);
11 +    if (bt->tasklet.callback)
12 +      tasklet_schedule(&bt->tasklet);
13     break;
14   }
15 }
16 // dvb-bt8xx.c, dvb-bt8xx.ko
17 static int dvb_bt8xx_load_card(struct
   ↪  dvb_bt8xx_card *card, u32 type)
18 {
19   tasklet_setup(&card->bt->tasklet,
     ↪  dvb_bt8xx_task);
20 }
```

Figure 13: A null pointer deference bug in Linux `bt878` driver. Fixed in: a3a54bf [58]

to process the rest of the IRQ handler. However, the problem is that the `bt->tasklet.callback` is initialized in a separate kernel module `dvb-bt8xx.ko`, line 19. The `bt878` kernel module can be loaded independently, while the `bt->tasklet.callback` remains null. In this case, when the tasklet is scheduled to run, it will cause a null pointer dereference.

It is worth noting that during post-probing fuzzing, our MMIO model could set the bits that can pass the checks at lines 6 and 8, constructing a proper context where the tasklet is scheduled. Together with timer-based IRQ injection, our MMIO model cooperatively helped the bug detection.

## 2. Bug 48: A DMA Bug in `epic100`

Figure 14 shows a buffer overflow bug detected in the network device `epic100` using our DMA model-guided fuzzing. The driver reads from the secondary DMA buffer when the `DescOwn` bit in the `rxstatus` field of the first level DMA buffer is not set. In line 11, the packet length is calculated using the upper 32 bit of the status read from the DMA buffer. The buffer overflow bug is caused by an arithmetic overflow of the `pkt_len` variable. When the `pkt_len` is -1, the length checking in line 13 is bypassed. Since the variable `rx_copybreak` is 0 by default, the if condition in line 17 is satisfied and its true branch is executed. In line 20, the driver tries to copy the `pkt_len` size of data to the skb buffer. However, the function `skb_copy_to_linear_data` treats `pkt_len` as a 64-bit unsigned integer. This will cause 65535 bytes of data to be copied into the skb buffer, which only has the size of one page, triggering a buffer overflow.

Though it is a severe security vulnerability that can be fixed by changing the type of `pkt_len` to `unsigned short`, triggering the bug is non-trivial. First, without knowing the DMA buffer location, the DMA buffer will never be fuzzed. Second, reaching the bug needs to combine two constraints together: the flag check in line 6 and the `pkt_len` check in line 17. Thanks to our DMA model, this bug is revealed just in a few minutes during fuzzing.

```
1  // epic100.c
2  static int epic_rx(struct net_device *dev, int
   ↪  budget)
3  {
4    ...
5    /* Read RX buffer status from DMA Buffer */
6    while ((ep->rx_ring[entry].rxstatus & DescOwn)
   ↪  == 0) {
7      int status = ep->rx_ring[entry].rxstatus;
8      if (status & 0x2006) {
9        /* Omitted */
10     } else {
11 -     short pkt_len = (status >> 16) - 4;
12 +     unsigned pkt_len = (status >> 16) - 4;
13       if (pkt_len > PKT_BUF_SZ - 4) {
14         pkt_len = 1514;
15       }
16
17       if (pkt_len < rx_copybreak &&
18           (skb = netdev_alloc_skb(dev,
             ↪  pkt_len+2)) != NULL) {
19         ...
20         skb_copy_to_linear_data(skb,
           ↪  ep->rx_skbuff[entry]->data,
           ↪  pkt_len);
21         skb_put(skb, pkt_len);
22       } else {
23         /* Omitted */
24       }
25       netif_receive_skb(skb);
26     }
27     work_done++;
28     entry = (++ep->cur_rx) % RX_RING_SIZE;
29   }
30
31   return work_done;
32 }
```

Figure 14: A buffer overflow bug in `epic100`.

```
1  static void arcmsr_drain_donequeue(struct
   ↪  AdapterControlBlock *acb, u_int32_t
   ↪  flag_srb, u_int16_t error) {
2    srb = (struct CommandControlBlock
     ↪  *)(acb->vir2phy_offset+(flag_srb<< 5));
3    if((srb->acb != acb) || (srb->srb_state !=
     ↪  ARCMSR_SRB_START)) {
4      if(srb->srb_state == ARCMSR_SRB_TIMEOUT) {
5        arcmsr_free_srb(srb);
6        return;
7      }
8      printf("...srb_state=0x\%x...", ...
       ↪  ,srb->srb_state,...);
9      return;
10   }
11   arcmsr_report_srb_state(acb, srb, error);
12 }
13 static void arcmsr_hba_postqueue_isr(struct
   ↪  AdapterControlBlock *acb)
14 {
15   u_int32_t flag_srb;
16   u_int16_t error;
17   ...
18   while((flag_srb =
     ↪  CHIP_REG_READ32(HBA_MessageUnit,
19     0, outbound_queueport)) != 0xFFFFFFFF) {
20     ...
21     arcmsr_drain_donequeue(acb, flag_srb,
       ↪  error);
22   }  /*drain reply FIFO*/
23 }
```

Figure 15: A kernel information disclosure bug in `arcmsr`.

## 3. Bug 56: An IRQ Handler Bug in `arcmsr`

Figure 15 shows a code snip from FreeBSD's `arcmsr` driver. The function `arcmsr_hba_postqueue_isr` can be triggered by a device interrupt. In this example, the variable `flag_srb` can be controlled by the device. This variable is read from line 19 and then passed to function `arcmsr_drain_donequeue`. In this function, the variable `srb` is calculated based on this value at line 2. This means the device can change `srb` to any arbitrary value. The following statement at line 8 can cause kernel information disclosure: *e.g.,* can be used to defeat kernel address space layout randomization (ASLR). If lines 5 and 11 are reached, the kernel will overwrite the data pointed by variable `srb`. In the worst case, this means the device can fool the driver to modify important kernel data structure and it may be used to help gain root privilege for a user process.