Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators

Wenjia Zhao*
Xi'an Jiaotong University
University of Minnesota

Kangjie Lu
University of Minnesota

Qiushi Wu
University of Minnesota

Yong Qi
Xi'an Jiaotong University

***Abstract*—Device drivers are security-critical. In monolithic kernels like Linux, there are hundreds of thousands of drivers which run in the same privilege as the core kernel. Consequently, a bug in a driver can compromise the whole system. More critically, drivers are particularly buggy. First, drivers receive complex and untrusted inputs from not only the user space but also the hardware. Second, the driver code can be developed by less-experienced third parties, and is less tested because running a driver requires the corresponding hardware device or the emulator. Therefore, existing studies show that drivers tend to have a higher bug density and have become a major security threat. Existing testing techniques have to focus the fuzzing on a limited number of drivers that have the corresponding devices or the emulators, thus cannot scale.**

In this paper, we propose a device-free driver fuzzing system, DR. FUZZ, that does not require hardware devices to fuzz-test drivers. The core of DR. FUZZ is a semantic-informed mechanism that efficiently generates inputs to properly construct relevant data structures to pass the "validation chain" in driving initialization, which enables subsequent device-free driver fuzzing. The elimination of the needs for the hardware devices and the emulators removes the bottleneck in driver testing. The semantic-informed mechanism incorporates multiple new techniques to make device-free driver fuzzing practical: inferring valid input values for passing the validation chain in initialization, inferring the temporal usage order of input bytes to minimize mutation space, and employing error states as a feedback to direct the fuzzing going through the validation chain. Moreover, the semantic-informed mechanism is generic; we can also instruct it to generate semi-malformed inputs for a higher code coverage. We evaluate DR. FUZZ on 214 Linux drivers. With an only 24-hour time budget, DR. FUZZ can successfully initialize and enable most of the drivers without the corresponding devices, whereas existing fuzzers like syzkaller cannot succeed in any case. DR. FUZZ also significantly outperforms existing driver fuzzers that are even equipped with the device or emulator in other aspects: it increases the code coverage by 70% and the throughput by 18%. With DR. FUZZ, we also find 46 new bugs in these Linux drivers.

I. INTRODUCTION

In monolithic kernels like the Linux kernel, 70% of the kernel code is device drivers [20]. The kernel serves as a hardware resource manager; its device drivers are responsible for identifying and managing the specific devices. The drivers are security-critical but buggy. They run in the same privilege level as the core kernel does. If a driver is compromised, the attacker can basically control the whole system. However, drivers are much buggier than other components in the kernel, with a bug density three to seven times as the rest of the kernel [8].

Drivers are particularly vulnerable for several reasons. First, studies have shown that the development of device drivers is error-prone [40] [31]. The developers need to take care of many aspects, including OS interfaces and data structures, compilers, and integrated circuits. Second, the driver code is less tested because testing a driver requires the corresponding hardware device or at least an emulator. Moreover, the driver often accepts complicated and special inputs (from both the user space and the hardware) that are hard to cover during testing. Finally, compared with other parts of the kernel, some drivers are no longer actively maintained or supported, and thus tend to be more vulnerable [35].

In addition to being vulnerable, having a larger attack surface is another unique problem with drivers, as the untrusted inputs can be from not only the user space but also the hardware. With the development of programmable devices, a security threat aggravates: specific or generalized malicious hardware devices have become common, such as Thunderclap [29], FaceDancer [14], ThunderSpy [41], and BadUSB [23]. Therefore, securing the safety of drivers from hardware has become a top priority. Recently, a lot of works have begun to focus on vulnerabilities that are triggerable or even exploitable from external malicious hardware [47] [2].

Recent advances have turned to fuzzing to test drivers. Fuzzers use invalid, unexpected, or random data as inputs to a driver to trigger different paths at runtime, and they use sanitizers like KASAN [22], KMSAN [15], and UBSAN [42] to monitor the abnormal behaviors to find bugs. For example, DIFUZE [11] identifies 36 vulnerabilities in the Linux-kernel drivers through a set of `ioctl` interfaces. PeriScope [44] detects bugs in device drivers by intercepting driver accesses to communication channels based on page faults generated by mmio/dma. It also discovered 15 unique vulnerabilities. Agamotto [45] proposes lightweight virtual-machine checkpointing as a new primitive that enables high-throughput kernel driver fuzzing. These works show that fuzzing can be an effective approach to finding vulnerabilities in the drivers.

Existing driver fuzzers however still suffer from an inherent

* The work was done at the University of Minnesota.

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23345
www.ndss-symposium.org

limitation—*requiring the hardware device or an emulator*. The kernel supports many devices, e.g., there are more than 13,000 PCI devices alone [34]. Testing drivers with the corresponding devices or emulators have clear shortcomings. If it uses the hardware to support the driver fuzzing, both the hardware cost and the time cost for operating the hardware can be very high. If it uses an emulator, such as QEMU [6], it cannot scale: existing emulators only provide emulation for a limited number of devices. For example, there are less than 130 PCI devices in QEMU according to our study. Meanwhile, extensive manual efforts are required to build the emulators for the unsupported devices. Although protocol reverse engineering techniques [7, 12, 25] can help by automatically extracting the format specification so as to assist the emulation, they typically target a specific application due to the complexity. More importantly, hardware devices or emulators may generate too well-formed inputs that cannot broadly trigger vulnerabilities that can only be caused by malformed inputs [29] [14] [41] [23].

In this paper, we propose a novel *device-free* driver fuzzer, DR. FUZZ, that addresses the limitations of existing driver fuzzers. Through a characterization study of drivers, we observe that they follow the Linux kernel device model (LKDM), and the running of a driver requires a successful initialization of the related data structures. More importantly, the initialization process is essentially *validation chains* (code paths leading to successful initialization) that read, check, and sometimes use a number of inputs from the devices. Therefore, passing the validation chains implies a successful initialization, which will enable the driver and subsequent normal fuzzing. Based on this observation, we propose to automatically create "driver initializers" that properly construct the device-related data structures to pass the validation chains. The core is a semantic-informed mechanism that infers various classes of semantic information to efficiently generate valid inputs for succeeding the validation. DR. FUZZ's approach is fully automated and thus can scale; also, it does not require any hardware supports. Developers, maintainers, and users can readily use DR. FUZZ for testing drivers. The elimination of the needs for hardware devices and emulators removes a bottleneck in driver fuzzing.

Automatically creating such "initializers" to pass validation chains without the devices is challenging because the extremely complex device-related data structures and diverse I/O device addressing incur a huge input space. To address these challenges, we propose three new techniques to make device-free driver fuzzing practical.

(1) *Byte-level, field-sensitive value inference and mapping*. We identify the I/O-dependant fields and build an I/O-dependence graph through a field-sensitive analysis. Based on this graph, we infer the candidate values for the fields of related data structures involved in the validation chains, through a byte-level analysis. Further, we develop additional techniques to map the fields to the input bytes at specific addresses.
(2) *Byte-priority inference based on temporals*. The driver often reads a chunk of data, e.g., 8 bytes or even more. Mutating the whole input would not be practical due to the huge search space. We observe that the validation chain is naturally temporal, so the byte usage follows a clear temporal pattern. We thus propose to infer the priority of each byte in inputs based on the temporals. By focusing the mutation on only one or a few bytes each time, we dramatically reduce the mutation space.
(3) *Error states as fuzzing feedback*. Given an input, it is important to know whether it triggers a normal execution or erroneous (or even the specific error), so as to guide the fuzzer to make progress in the validation chain. This technique exploits the rich error-handling information in drivers and dynamically collects the error information as the fuzzing feedback. We combine this error-state feedback together with the code coverage to guide the fuzzer.

The semantic-informed mechanism is generic. In fact, in addition to device-free driver fuzzing, we can also re-purpose it for increasing the code coverage of driver fuzzing. Our intuition is that a high-coverage driver fuzzer requires well-formed inputs to reach deep paths but also malformed inputs to trigger broad paths. As such, we propose to instruct our semantic-informed mechanism to generate *semi-malformed* inputs. The inferred semantics offer rich information, including expected valid inputs and execution states (e.g., normal execution or erroneous execution). Therefore, we also reuse the semantic-informed mechanism as a semi-malformed input generator to improve the code coverage and throughput of driver fuzzing.

We have implemented a prototype for DR. FUZZ and evaluated its functionality, effectiveness, and performance. We evaluate DR. FUZZ on 214 Linux drivers, and the results are impressive. With a only 24-hour time budget, DR. FUZZ can successfully run 149 of them without the corresponding devices or emulators, whereas existing driver fuzzers cannot succeed in any case. We further show that when allocated with more time, DR. FUZZ can initialize more drivers. DR. FUZZ even outperforms existing fuzzers equipped with hardware devices in coverage and throughput. Compared to syzkaller, our evaluation shows that DR. FUZZ increases the code coverage by 70% and the throughput by 18%. Interestingly, when we enable the semi-malformed input generator, i.e., breadth first feedback, DR. FUZZ can even improve the coverage over syzkaller by 200%. At last, we also apply DR. FUZZ to find new bugs. With DR. FUZZ, we find 46 new bugs in the Linux drivers.

In summary, we make the following research contributions.

- **A new study and fuzzing mechanism.** We perform a study to characterize the organization and the code semantics of device drivers. The findings indicate that device-free driver fuzzing is feasible—the essence of a successful driver initialization is to pass its validation chains. We then propose a semantic-informed mechanism to automatically create "driver initializers" that know how to properly initialize the related data structures involved in the validation chains.
- **New techniques.** We propose three new techniques to make device-free driver fuzzing practical: (1) byte-level and field-sensitive value inference which infers expected valid values in validation and maps them to I/O addresses, (2) byte-priority (temporals) inference which dramatically reduces the mutation space, as the validations chains are naturally temporal, and (3) error state as fuzzing feedback which directionally guides the fuzzing to trigger deep normal execution and broad erroneous execution.
- **Implementation and new bugs.** We further instruct the semantic-informed mechanism to generate *semi-malformed* inputs to both broadly and deeply cover driver paths. We

implement DR. FUZZ and extensively evaluate it. We will release source code and artifacts at https://github.com/secsysresearch/DRFuzz.git. DR. FUZZ can successfully run drivers without the hardware devices. DR. FUZZ even achieves a higher code coverage and throughput than existing fuzzers equipped with hardware devices. With DR. FUZZ, we also find many new bugs in Linux drivers.

The rest of this paper is structured as follows. §III describes the adversary model and the overview of DR. FUZZ. §IV describes the design of DR. FUZZ. §V shows implementation details. §VI presents the evaluation results. §VII describes the related works, and §VIII concludes the paper.

II. A CHARACTERIZATION STUDY ON DEVICE DRIVERS

This section aims to characterize device drivers in the Linux kernel and to confirm the feasibility of device-free driver fuzzing. We first show the Linux kernel device model followed by drivers and further present the dynamic workflow of device drivers. Finally, we present important features of device drivers, e.g., the validation chain, that make device-free driver fuzzing feasible.

Our study of the driver model is based on the Linux driver implementer's API guide [3]. For this study, combining with the dynamic tracing techniques, we manually analyze the subsystems under the 5 core buses, which are the most commonly used by the device drivers [20], across different kernel versions, including v3.1.1, v4.1.1, and v5.1. Table I shows a small part of the drivers we analyzed.

Bus	**PCI**		**ISA**		**SCSI**		**I2C**		**USB**	
Driver	*e1000*	*8139cp*	*tscan1*	*advansys*	*ch*	*sd*	*ipmi_ssif*	*cdce806*	*ath3k*	*bcm203x*

TABLE I: The studied device drivers under the 5 core buses.

A. The Linux Kernel Device Model

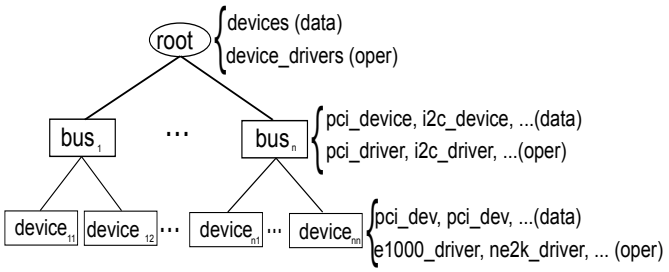


Fig. 1: The layered Linux device model. Each node contains its data structures and operations

The current driver model, Linux Kernel Device Model (LKDM) [32], is a uniform data model that describes the buses, and the devices connected to these buses. LKDM intends to augment the bus-specific drivers for devices and bridges by merging a set of data and operations into global data structures. As shown in Figure 1, based on LKDM, the device organization follows a hierarchical structure with three layers: a root, buses, and devices. Each layer has its own data structures that represent the device states and record the available operations (using function pointers). This hierarchical structure significantly simplifies the device management—the bus driver defines the interfaces that the device driver should implement, and the kernel uses these interfaces to manage the device. The device model shows that devices are *uniformly* organized and managed through a set of data structures and operations implemented in the corresponding drivers.

B. The Two-Stage Workflow of Drivers

This section summarizes the dynamic workflow of device drivers. The workflow has two stages: driver initialization and communication between the driver and device.

Stage 1: structure-centric initialization. Kernel first discovers the device by scanning the bus and matching the attached device with the corresponding driver according to device's characteristics. Bus driver builds the basic *device structures* based on the device inputs. The device driver uses this structure to build its more specific structures. Meanwhile, bus driver calls the predefined interface implemented by the device driver to configure the device, allocate necessary resources, and complete the whole driver initialization. Once the initialization is done, the kernel can then interact with the device through these device structures and corresponding operation functions in the driver.

Stage 2: communication between the driver and device. After the initialization, the kernel can check the device status and communicate with the device through driver functions. Most of the devices will export some userspace interfaces such as an inode in the file system to support its usage. User programs can operate the devices through syscall on these userspace interfaces such as `open()` and `ioctl()`. Finally, the user commands will pass to the device through the device driver. Meanwhile, regardless of the first or second stage, all driver operations will eventually be passed to the device through the low-level operations such as `in, out, readl, writel, mov`.

C. Is Device-Free Driver Fuzzing Feasible? – The Input-Validation Chains!

As we discussed in §II-B, the initialization stage is the key for successful device-free driver fuzzing. Thus, this section further investigates important features in the initialization stage to confirm if automating the initialization stage is feasible.

Fortunately, we observe that the driver initialization is essentially to pass *input-validation chains*. Through a device read, the driver obtains inputs from the device. During the initialization, the driver enforces a series of validations against the inputs, forming a validation chain. To successfully launch the driver, the inputs must pass through the whole validation chain. These validations typically target parts of the whole input (e.g., one or a few bytes) and follow the communication protocols for the driver and its corresponding device. Interestingly, such validations are pretty informative in revealing what values in which input bytes are expected as valid. This offers opportunities to precisely infer the input bytes for a successful initialization. We further present important features with the validation chains.

The temporal input usage. The bytes from the inputs are not only checked by the validations, but also used for initializing structures. Both the validation and the usage against the input bytes show an important pattern—the validations and uses occur *sequentially* with a temporal order, and each of them targets only one or a few bytes, instead of the whole input obtained from an I/O read. This finding is important because it implies that the fuzzing does not have to mutate the input as a whole, but potentially byte by byte, which would dramatically reduce the mutation space.

Hard-coded I/O address-value mappings. During the driver initialization, the bus and device drivers read the input from the device, and assign the input to some fields of the device-related structures. A device read consists of two essential elements: its address and data size; both are typically hard-coded as constants in the driver code. Recovering such constants would greatly help us build the mapping between the inputs with specific offsets from the devices and their references in the driver code.

Prevalent error handling. The driver code enforces a large number of sanity checks against inputs and internal operations. Whenever a check fails, the driver handles the error. Therefore, error handling in the drivers is extremely common. The most common way of handling errors is to pass an error code as a return value upstream. We observe that such error codes can serve as helpful feedback that indicates the current path-execution states. That is, based on the return values, the fuzzer can know if the input triggers an erroneous execution or a normal execution. This is useful because it tells if the fuzzer has generated the byte values in the inputs correctly.

Other findings. At the driver initialization, we also found that I/O ports/MMIO are commonly used during the device discovery and initialization. I/O ports use a set of I/O instructions, e.g. `IN, OUT, INS, OUTS`, to transfer the data between the system and device. I/O ports have a separate I/O address space from the physical-memory address space. They consist of 2^{16} (0x0~0xFFFFH) individually addressable 8-bit I/O ports. MMIO maps the I/O address space to the physical-memory address. The driver can use the traditional instructions, e.g. `mov`, to access the device I/O address space. Except from the I/O ports and MMIO, some drivers also call a few Direct Memory Access (DMA) APIs to setup the DMA at the initialization, such as `dma_alloc_*()/dma_map_*()`.

Example. We take `e1000`, a PCI driver for Intel gigabit network card, as an example to illustrate the aforementioned features. Figure 2 shows a part of the source code for `e1000` driver initialization. Bus driver scans the PCI bus (line 11) and reads the `vendor_id` and `device_id` (line 16). To initialize successfully, input validations (line 17) require the `vendor_id/device_id` to satisfy some requirements. According to line 17, `vendor_id/device_id` cannot be 0xFFFF or 0x0000. Meanwhile, bus driver uses them to match the device driver (line 32) by comparing whether their value is equal to one element in `e1000_pci_tbl`, which is a constant array. According to this, their value can only be one element in `e1000_pci_tbl`. Actually, the validation contains many entries, like `command` and `int_pin`. They form a chain, *the input-validation chain*.

To read `vendor_id` and `device_id`, the macros (lines 1), `bus`, and `devfn` together form their I/O address in `pci_bus_read_config_dword`. In line 16, the driver reads them from the address by I/O port instruction. As the `bus` and `devfn` are fixed after the device attached to the system, the formed addresses are hard-coded I/O addresses. Meanwhile, `vendor_id/device_id` are constant according to the input validation. These addresses and the constant values constitute a *hard-coded I/O address-value mapping*.

Line 18-19 shows how the driver fills the `vendor/device` fields. The driver first reads the 4 bytes `l` (line 16), and then uses its lower two and higher two bytes to fill `vendor` (line 18) and `device` (line 19) respectively. When the kernel validates them (line 25), the `vendor_id` is validated first, then `device_id`, which shows a *temporal input usage*. Due to space limitations, we only use them as examples here. In fact, according to our statistics, about 15% of I/O reads contained in the `e1000` driver have such characteristics.

Also, if they are not the value the driver expects, as shown in line 17, 21, 27, the kernel will execute the error handling. The driver codes have `prevalent error handling`. When the read of the `vendor_id` is failed (line 17), the caller will return a NULL pointer to the device structure. Similarly, when the PCI device (from input) is unknown (line 20), the error code `-EIO` is returned. When the `e1000_probe` failed to execute, it also returns a non-zero value to show the reason. Such feedback helps pass the validation chain to trigger deep paths. For example, if line 17 return the NULL pointer to a `struct pci_dev` variable, we know the value of `l` is incorrect. We can change the input related to `l` and pass the validation for the next running.

```
1  #define PCI_VENDOR_ID       0x00    /* 16 bits */
2  #define PCI_DEVICE_ID       0x02
3  struct bus_type pci_bus_type = {
4      .probe      = pci_device_probe,
5  };
6  struct pci_driver e1000_driver = {
7      .id_table = e1000_pci_tbl,
8      .probe    = e1000_probe,
9  };
10 unsigned int presdue_entry(struct pci_bus *bus, int devfn){
11     struct pci_dev dev = pci_scan_device(bus, devfn);
12     if (!dev) return NULL;
13 }
14 struct pci_dev *pci_scan_device(struct pci_bus *bus, int devfn){
15     u32 l;
16     pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l);
17     if (*l == 0xffffffff || *l == 0x00000000) return NULL;
18     dev->vendor = l & 0xffff;
19     dev->device = (l >> 16) & 0xffff;
20     if (pci_read_config_byte(dev, PCI_HEADER_TYPE, &hdr_type))
21         return -EIO;
22 }
23 struct pci_device_id *
24 pci_match_device(struct pci_device_id *id, struct pci_dev *dev){
25     if((id->vendor==dev->vendor) && (id->device==dev->device))
26         return id;
27     return NULL;
28 }
29 int pci_device_probe(struct device *dev){
30     struct pci_driver *drv = to_pci_driver(dev->driver);
31     /*vendor_id, device_id are in  e1000_pci_tbl*/
32     id = pci_match_device(drv, pci_dev);
33     int rc = e1000_probe(dev, id);
34     if (!rc) return rc;
35 }
```

Fig. 2: Part of source code for PCI device initialization. `pseudo_entry` is an abstract entry point to scan the PCI devices.

Conclusion. The features discovered above confirm that the driver code actually provides very informative semantics that can be used to infer what values at which locations in the inputs are expected to pass the validation chains for a successful initialization. More importantly, the temporal usage and the rich error handling will dramatically reduce the mutation space and provide effective fuzzing feedback. As such, we conclude that device-free driver fuzzing is feasible, which motivates us to develop DR. FUZZ.

III. AN OVERVIEW OF DR. FUZZ

Design rationales. Based on the study presented in §II, to enable the device-free driver fuzzing, we only need to instruct the drivers to properly construct the corresponding data structures to pass the validation chains for a successful initialization stage. Therefore, we propose to automatically create "driver initializers" that construct the related data structures for a driver. Considering that fuzzing provides a "trial and error" process to generate desired inputs and that our ultimate goal is to enable device-free driver fuzzing, we decide to also employ fuzzing to construct the initializers by figuring out the inputs that pass the validation chains and construct the valid device-related structures. Once the initialization is successful, we enable the driver and can continue with the normal fuzzing. Two strengths of such a design decision are scalability (no path explosion, as opposed to symbolic execution) and determinism (we confirm if the initialization indeed succeeds by running it).

Specifically, we instruct our fuzzer to prepare the inputs and to inject them into the kernel of a virtual machine through its virtualization I/O stack. At this point, the main research question becomes: *how to practically and properly prepare the valid inputs expected by the validation chains*. In the rest of this section, we will first discuss the technical challenges for solving this question. Then, we will introduce our solutions against these challenges and also the framework of DR. FUZZ.

A. Technical Challenges

The core of DR. FUZZ is to employ fuzzing to generate the proper inputs to construct correct device-related structures. The proper inputs mean kernel needs to know (1) which address to read from and (2) what value should read. However, getting this input entails overcoming multiple challenges.

C1: Large input space caused by extremely complex data structures. A device driver contains numerous complex device-related data structures which usually contain many fields, and are hard to understand and analyze. For example, `struct net_device` is a data structure in `e1000` driver, and it contains 114 fields and has a large size of more than 2,000 bytes. The huge mutation space and complex data structures will easily render the traditional fuzzing impossible to figure out the correct value (oftentimes only one) for each field.

C2: Diverse and complicated I/O addressing. Intuitively, if we know the value of every field, we can build the structures by providing the expected value at I/O read. However, due to the diversity of I/O addresses, it is challenging to figure out the specific I/O address to inject these values. For example, I/O ports, as a separated I/O address space, has 2^{16} I/O addresses. Furthermore, an MMIO address can be any part of the memory address. Even worse, the I/O interaction is frequent, and some fields of the structure come from a dynamic address, which means some I/O addresses change at runtime.

B. Solution Overview

In this section, we explain how DR. FUZZ overcomes these challenges to make device-free driver fuzzing practical. Our solution is a new semantic-informed fuzzing mechanism that comes with multiple new techniques.

Overcome the first challenge. To address the large input space challenge—extremely complex data structures, we should minimize the number of relevant data structures, as well as the range of possible values of their fields. Our idea is to select the ones dependent on the I/O inputs. In particular, we identify the device-related data structures in the driver and the I/O-related fields in these data structures by building an *I/O-dependent graph* using precise static analysis. We reduce the number of these complex data structures by extracting the *critical fields* in these data structures. After we ensure the fields are of our interest, we use a byte-level and field-sensitive analysis to identify the validation chain of a field—the validation chain is very informative in revealing the expected values. By analyzing the validation chain, we collect the constraints (i.e., the dependent values) from the branch statements using field-sensitive analysis, which helps us determine the candidate value set of the fields.

In addition to fields whose candidate values can be determined, we also need to reduce the mutation space for unknown fields. Validation chains are inherently sequential, following a temporal order. That is, fields are used sequentially (concurrency is unlikely to occur during this stage). We thus propose byte-priority (temporal) inference to determine which bytes of the fields should be constructed first and more importantly, which ones can be constructed later; this byte-wise priority can help the fuzzer to significantly reduce the input mutation space. In addition, we employ error state as fuzzing feedback. We will intercept the return values to understand if the current input triggers a correct or erroneous execution, which guides the fuzzing to generate proper input more effectively.

Overcome the second challenge. After we get the candidate values and the byte-priority information, we also need to associate this information with the specific I/O address of the device, so that we can prepare the values at the right addresses. To overcome the challenge of the diverse and complicated I/O addressing, we exploit the hard-coded I/O address-value mappings and dynamic mapping analysis. Specifically, we first identify the fields whose I/O addresses are hard-coded in the driver code and map the fields to the constant arguments of the I/O ports/MMIO statements. Then, we can generate the I/O address-value mappings. This way, we know which I/O address the values should be injected to. For cases without hard-coded mappings, we employ dynamic mapping analysis. That is, we use runtime feedback to collect the order of byte usage and inform the fuzzer the priority. Details will be shown in §IV-A2.

C. The Framework and Components

Figure 3 depicts the overall design of DR. FUZZ. DR. FUZZ is mainly comprised of two parts. (1) *The semantic-*

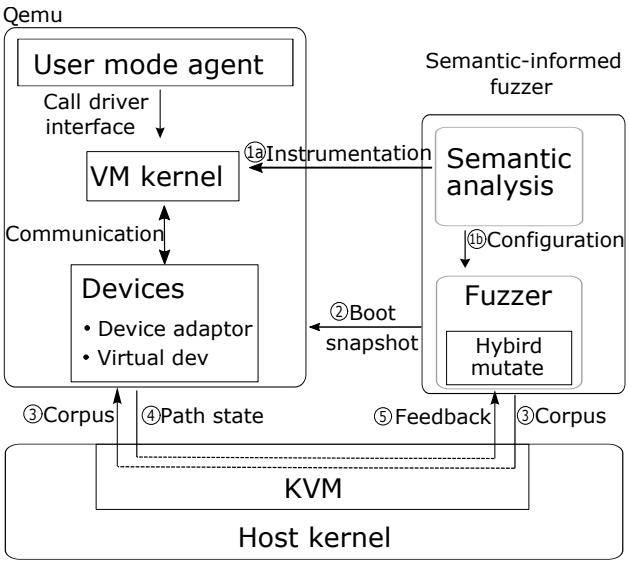


Fig. 3: The overview of DR. FUZZ.

informed mechanism. This part statically collects the semantic information, instructs the driver code, gets the feedback at runtime, and mutates the current input. (2) *The fuzzing framework for drivers*. It manages the VM running and injects the fuzzer output to the VM kernel through a device adapter. In this framework, the most important and unique design is the I/O interception and data injection. It first intercepts the target-device I/O access, and then forwards the address and size to the fuzzer. Then, the framework uses the fuzzer output as a device input to inject the data to the qemu virtual memory regions. We briefly introduce each component as follows.

Semantic analysis. The semantic analysis extracts various useful semantic information and generates the *semantic corpus* (①a) that contains the inferred values. It also instruments the driver code with new VMCALL instructions at locations where error states can be captured (①b). These special instructions will send feedback to the fuzzer through KVM at runtime. The semantic analysis will be detailed in §IV.

Fuzzer. The fuzzer is an engine that manages the VM snapshot (②) and generates data to the device adaptor as input (③) through the virtualization I/O stack. The fuzzer also receives the error states as feedback, and performs the mutation according to the feedback. The fuzzer boots the snapshot for the next run after a mutation (②).

Device adaptor. As the device address is different on the different buses, the device adaptor is responsible for "attaching" the given fake device address to the proper bus. The hypervisor forwards all the input/output of I/O ports/MMIO/DMA to the adaptor, and the fuzzer injects the mutation data as device input and interrupts to the device driver through this adaptor (③).

Modified KVM module. The modified KVM module captures and parses the VM exits caused by executing new VMCALL instructions. These VMCALL instructions are inserted by the semantic analysis instrumentation into the driver (①a). When the driver execution triggers these VMCALL instructions, VM-exit happens, and this module extracts a parameter from register RCX passed by the VM-exit. The parameter indicates the location of the exit (④). At last, KVM model sends the VM-exit information to fuzzer to guide its mutation (⑤).

User mode agent. The agent is a program running in the userspace of the VM. When the driver is initialized, the control flow is transferred to this program. It automatically executes the pre-defined syscalls to trigger the driver-function execution.

IV. THE SEMANTIC-INFORMED MECHANISM

In this section, we present the details of the semantic-informed mechanism which is the fundamental component that makes the device-free driver fuzzing practical. The mechanism aims to infer the valid inputs needed to pass the validation chains for a successful driver initialization and to dramatically reduce the mutation space of the input. It contains two parts, namely semantic analysis and informing. Figure 4 depicts the diagram of semantic-informed mechanism which incorporates three new techniques (T1, T2, and T3) presented as follows.

A. Semantic Analysis

1) T1.1: Byte-Level and Field-Sensitive Value Inference: The goal of this technique is to infer the candidate values of inputs bytes that are used in initialization. Since the bytes will propagate to fields of relevant data structures, the technique will identify the fields in the data structures that are dependent on I/O, and then build the mapping from the fields to the input bytes. To facilitate the analysis and make it precise, we will build an I/O-dependant graph through a byte-level and field-sensitive analysis.

Identifying I/O-dependent fields and building the graph. DR. FUZZ provides the proper device input to help kernel complete the driver initialization, so we should focus on the structures and their fields that are dependent on I/O, i.e., the fields whose values come from I/O. To make the fuzzing practical, our principle is to minimize the number of I/O-dependent fields, so as to minimize the input space for fuzzing.

In order to identify I/O-dependent fields, we employ backward *field-sensitive* data flow analysis against all fields of data structures in the driver. An alternative strategy is to employ forward analysis from I/O. However, this may not be efficient, as I/O data can propagate to many other places. The goal of the backward analysis is to find the sources of a field, and if the source is device input, i.e., the destination operand of the I/O instruction, we confirm that the corresponding field is dependent on I/O. Then, we can get an I/O-dependent graph (as illustrated in the left part of Figure 5) based on the data flows. Its node represents a field variable, and its edge represents the data flow between two variables. Through the backward data flow analysis, we get a set (S) of all fields dependent on I/O.

Identifying validation chains, critical fields and their values. The goal of DR. FUZZ is to succeed in the initialization by passing the validation chain. Therefore, this step focuses on identifying the fields that are used in the validation chain, and we call such fields *critical fields* (CS). Once we identify the critical fields, we will also collect the expected valid values based on the validation chain. This process starts by recognizing the validation chain—a series of checks that either result in

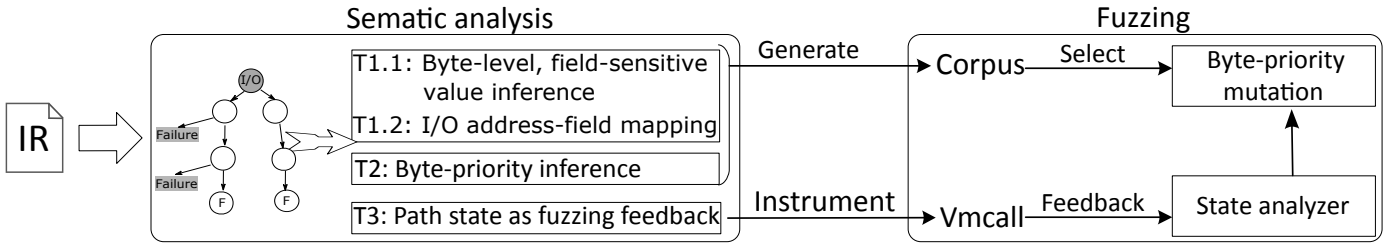


Fig. 4: The semantic-informed mechanism. The IR bitcode of the kernel is the input. The semantic analysis generates the initial inputs in the corpus. Meanwhile, it instruments the code based on the analysis to collect the state feedback and pass it to the analyzer.

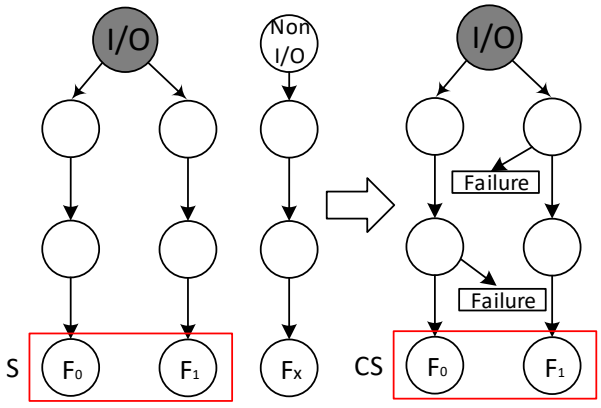


Fig. 5: The I/O-dependent graph of critical fields. IO means the I/O instruction. F is a field of structure.

a failure or continue the execution. A validation in fact has a clear pattern. More specifically, a validation is a conditional statement such as `if` that has branches, as at least one of them leads to execution failure such as returning an error (see the right part of Figure 5). Based on the pattern, we identify validations and collect the validated fields as the critical fields. Note that the identification of validation is similar to how [27] identifies security checks. Since the handling of failures during the initialization is nearly uniform [16, 50], the validation identification is precise and reliable.

After obtaining the critical fields, we need to further infer the possible values of these fields. We regard these validations as field constraints, and extract the possible values from these constraints. We start with the critical field, and use a static inter-procedural and field-sensitive data-flow analysis to collect the constraints of the field. Specifically, based on the I/O-dependence graph, we analyze and collect the branch conditions related to the critical fields in the graph. In our current implementation, we focus on constant conditions. After applying the constant folding [33], if the condition is still a non-constant variable, we discard the condition. After the collection, we divide the corresponding branch conditions into four categories: equal, not equal, greater than and less than. Finally, we provide a candidate value set that contains the possible values of the field.

2) T1.2: I/O Address-Field Mapping: **Handling hard-coded mapping.** After identifying the possible values of critical fields, we further map the fields to the I/O addresses, so that we can know which values should be assigned to which bytes in the inputs. To obtain the I/O address of a field, we exploit the I/O-dependent graph to find its source—the I/O instruction filling the field. The source operand of this instruction is the corresponding I/O address of the field. We found that many of these addresses are hard-coded. Specifically, a part of the source operand is a macro, and the bus uses this macro and the device address to form the final I/O address. As the device address on the bus is known at the start of the virtual machine, we can then get the I/O address of the field.

For example, when the kernel reads a value from a PCI device to fill one field in `pci_dev`, the I/O address of the field is generated by lines 1-3 in the following code:

```
1 #define PCI_CONF1_ADDRESS(bus, devfn, reg) \
2     (0x80000000 | ((reg & 0xF00) << 16) | (bus << 16) \
3     | (devfn << 8) | (reg & 0xFC))
4 ...
5 outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);
```

As we can specifiy the "device" address when starting the VM, the value of `bus` and `devfn` (line 1) is known. We can use this macro to build the final I/O address of that field and further generate the I/O address-field mapping.

The above recovered I/O address is the base address used in the I/O read. Oftentimes, a field is only a part of the read data. That is, the corresponding input bytes have an offset into the base address. Therefore, we further precisely infer the offset of the input bytes for the field. As Figure 2 shows, `dev->vendor` and `dev->device` come from I/O read, namely `1`. Their offsets relative to the I/O address of `1` are 0 and 2, respectively. To obtain the byte offset, after collecting the I/O instructions that use the previous candidate set, we search backwardly along the data flow of the source operand of the I/O instruction. We identify the bitmask statement and find the byte corresponding to the non-zero bit in bitmask. In the same way, for the value obtained after shifting, we only need to obtain the bitmask when shifting. This can be seen as the offset of an I/O access. Then, we use this information to update I/O address-value mapping. In this way, we can achieve the byte-level address mapping for fields.

Handling dynamic mapping. In addition to fields that have hard-coded I/O addresses, there are other fields whose corresponding I/O addresses are dynamically generated. In this case, static analysis alone cannot figure out the I/O address. To address the problem, we propose a dynamic address mapping technique. The basic idea is that if DR. FUZZ does not prepare the right values at the right I/O address, there must be an error state (i.e., a failure) which will be collected as feedback in DR. FUZZ. We design the error state information to contain

a special parameter that will be fed back to the fuzzer. This parameter includes the critical field information that caused the initialization failure. To achieve this, we first number the critical field, and then use code instrumentation to write the number into the relevant register during path state feedback to complete the parameter transfer. More details about the error-state feedback will be presented in §IV-B1.

3) T2: Byte-Priority Inference: The value inference effectively narrows down the possible values of inputs. It however still has two limitations. First, its results may not be precise, so the fuzzer still has to test them. Second, parts of inputs cannot be inferred due to the lack of semantic information. In these cases, the fuzzer has to mutate the inputs in a huge space the guess the valid values at specific addresses. To address the limitations, we propose priority-based byte mutation based on the fact that the validation chains are inherently sequential; the fuzzer will not mutate bytes that are not used yet in the current runs, but focus on only the ones that are being used. After all, as described in §II-C, during initialization, the byte usage of the input obtained from an I/O read follows a clear temporal order. We can extract the temporal semantics of the input bytes based on the validation chain and byte usage.

The temporal validation-chain pattern. We find that the validation-chain follows a specific temporal pattern. First, the code execution starts with a device read. The read gets a value (V) from the I/O address ($Addr_V$). Second, the value (V) will then be used as a source operand and a bitmask for bit operation to generate a new intermediate value ($I = V\&bitmask$). Third, this value (I) will have validation(s) before the value (V) is used. Forth, the value is used. In this execution pattern, the value (V) is read in once, but it is referenced subsequently. Therefore, when fuzzer provides device input for current address at next running, it should mutate the part covered by bitmask first, then validation, uses, and so forth.

Extracting temporals. To extract the byte-priority semantics, we need to match the fields to the temporal pattern. We check the nodes in the I/O-dependence graph, and identify the relevant operations against the nodes, including bitmask, validation, and usage. Since the I/O-dependence graph is constructed in a *flow-sensitive* manner, we can easily infer the temporal order of the operations. That is, we know the temporal order of fields used in the driver. After that, we reuse the I/O address-field mapping constructed in §IV-A2 to finally recover the priority of bytes in inputs. The inferred byte priority serves as a guidance instead of a guarantee. Therefore, we will still employ the error states at runtime to fix incorrect inferences. For example, if the fuzzing focuses the mutation on a late-used byte, it would not make progress. In this case, we will deprioritize the byte during fuzzing.

B. Semantic Informing

1) T3: Path State as Fuzzing Feedback: Existing fuzzers commonly use code-coverage as a feedback. This kind of feedback would not work well during the initialization of the device. The main reason is that our major goal in this stage is to pass the validations instead of to increase code coverage. We thus propose to use error states as a new feedback. As mentioned in §II-C, error handling is particularly popular in the driver code. The error states will provide very informative feedback regarding the states of the current execution.

As we need the kernel to initialize the driver successfully, we should prioritize normal execution states. If the driver initialization failed (i.e., with an error state), we need to re-run the driver with a new input. We classify the state to two types, normal state or error state. Normal state means the initialization code path can still be triggered by input, while error state means the driver has entered into a failure, and the initialization cannot be done if the fuzzer continues along the direction.

Error locations. The error-state feedback can provide the following useful information for the fuzzer. First, through this feedback, the fuzzer can know that the driver initialization has failed as soon as possible, which can make the fuzzer run the next time earlier. Second, the fuzzer can more accurately know which input caused the initialization failure, which can make the fuzzer more effective when selecting the next input. To generate this feedback, we need to identify the driver execution states. We found that when the driver execution failed, the most common way is to pass an error code as a return value to its upstream. Except that, there are functions in the driver that return a pointer to the device structure. When this type of function returns a null value, it means that the device's structure has failed to be obtained, and the null will be returned to the caller. The caller will change the return value to non-zero and continue to pass it to the upstream. Since the initialization code of drivers nicely follows the uniform error handling, we use the non-zero return values of the driver function as a flag to indicate the error state of the driver.

Collecting error states. To feed back this state information, we first collect the locations of all driver functions that can return errors. These marked locations are used to send the feedback to fuzzer. Then, we insert a stub before these marked locations. This stub is a small piece of code with a special vmcall. At runtime, when the driver enters the error state, the driver code will call the vmcall instruction which causes the system exit. DR. FUZZ notifies this behavior to the fuzzer. The special vmcall takes a parameter that contains useful information for the state analyzer. This parameter can indicate whether the input meets the expected value since the last mutation, and the state analyzer uses this parameter to hint the byte-priority mutation, so as to improve the mutation for the next input.

2) New Mutation Strategy: Code coverage is a very useful feedback to guiding the fuzzing, where the input triggering new code paths is used to produce more inputs. It is worth noting that having a successful initialization requires to not only trigger the paths (control flow) but also to prepare valid data structures (data flow). Therefore, code coverage alone is not sufficient in DR. FUZZ.

To complement it, our state-based feedback offers a more direct and effective way for guiding the fuzzing because it tells if the fuzzing is passing validations required by a successful initialization. To use two feedback mechanisms together, we employ a new mutation strategy that knows how to incorporate the two feedbacks. In particular, in the process of driver initialization, we prioritize the error-state feedback over the code-coverage feedback because it is more relevant to the initialization progress. When there is progress according to the error-state feedback, DR. FUZZ will deprioritize the progress

in code coverage; however, when the fuzzers does not make progresses in the error-state feedback, DR. FUZZ then chooses to use the code coverage as feedback. This way, DR. FUZZ combines both feedback, but with a preference for the error-state feedback.

V. IMPLEMENTATION

We implement DR. FUZZ based on LLVM, Syzkaller, and QEMU. Specifically, we use LLVM to perform the semantic analysis and instrumentation. We implement the semantic-informed mechanism based on the Syzkaller. The implementation complexity is shown in Table II. In this section, we present important implementation details.

Technique	LoC	Base Framework
Semantic analysis and instrumentation	3,308	LLVM
QEMU and KVM	642	
Fuzzer	1,132	Syzkaller
Others	682	-

TABLE II: The implementation complexity of DR. FUZZ in Linux. Our implementation involves modifying the source code of the syzkaller and instrumenting kernel code using LLVM [10].

Device adaptor. We implement the adaptor as a module in QEMU. The adaptor registers some I/O handlers to the VM exit handler of the QEMU; these handlers parse the exit events caused by accessing the IO ports/MMIO/DMA address of our adaptor. Every read/write operation from the guest OS's kernel is dispatched to the registered functions provided by the adaptor. Meanwhile, the adaptor creates multiple MemoryRegions for passing the fuzzer output to the target I/O address. Adaptor reads the fuzzer-generated data through file socket and writes the value to the MemoryRegions.

User mode agent. Its implementation is specific to the device; that is, it provides corresponding fuzzing logic according to the functions of the specific device. Therefore, its fuzzing logic should be provided by the driver tester. The implementation of the agent can determine whether it can trigger some of the functions provided by the driver. As the specific implementation of the agent is not our focus, so we reuse the syscall generator of syzkaller to generate the user mode agent automatically.

Semantic analysis. To perform the semantic analysis, we first configure the kernel to enable the device drivers and generates the IR files of the driver code. These IR files will be analyzed by our LLVM pass. The LLVM pass outputs the semantic information to a corpus file. The fuzzer generates the proper values as the device output according to this file. When doing the semantic analysis, our pass will insert the pre-implemented `hypercall` function to the target locations. `hypercall` is a piece of assembly code that finally executes the `vmcall` instruction. For each device driver, the semantic analysis will generate the instrumented version through the LLVM pass.

Currently, the most representative Linux fuzzer tool, syzkaller, uses `kcov` [49] to collect coverage information. `kcov` exposes kernel code coverage information in a form suitable for coverage-guided fuzzing. Coverage data of a running kernel is exported via the "kcov" debugfs file. However, the driver code is enabled at the kernel startup; the "kcov" debugfs is not ready to read at that time, so it is not suitable for our driver fuzzing at the initialization stage. To overcome this problem, we use Intel PT to trace the execution at the startup process, which is similar to kAFL [43]. When we use the user mode agent to test the driver, we switch back to the kcov as it can capture precise coverage of a single system call.

VM snapshots. In the fuzzing process, we use two VM snapshots. The first one is in the device initialization phase. In order to improve throughput, we skip some kernel-related initialization codes. We create a snapshot at the beginning of the actual bus scan. When the fuzzer receives an error feedback, we restart execution from the snapshot. The second snapshot is after the device driver is initialized; that is, after the relevant probe function is successfully executed, we create a snapshot of the actual point of the user mode agent, so that the userspace program can be executed more quickly without re-executing the kernel. After the successful execution of `probe()`, the VM will continue to the boot, and then the agent configured in `rc.local` will be executed. This agent will call a VMCALL instruction through a new syscall to suspend the VM and create the second snapshot. After that, the fuzzer will use this snapshot to perform the driver fuzzing, and the agent executes the testing functions.

VI. EVALUATION

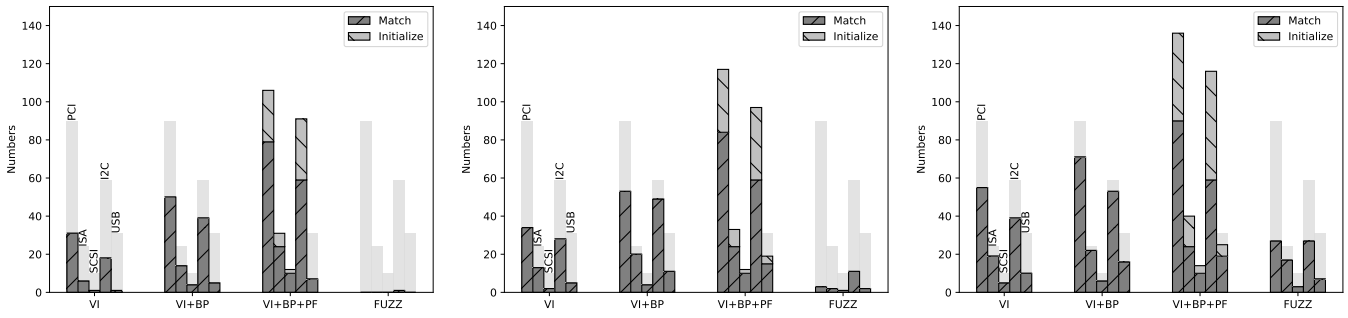
In this section, we extensively evaluate DR. FUZZ from the following perspectives: the effectiveness for driver initialization, bug findings, and performance. Our experiments were performed on a Dell workstation, with an Intel Xeon W-2133 CPU and 32G RAM, running an Ubuntu 16.04 Server, and LLVM v9.0.

Methodology. We choose the device drivers that use 5 common buses (PCI, ISA, SCSI, I2C, USB) and use DR. FUZZ to fuzz the drivers without the corresponding devices. In linux-v5.9, there are more than 2,000 drivers attached to these 5 buses. Considering each driver may support a series of devices, it may contain lots of devices. It is thus hard to choose which one is used more than the other for each bus type. Because of the trial-and-error characteristics of fuzzing itself, it takes a relatively long time to successfully initialize the device driver. Considering the large number of drivers, we randomly choose 10% of device drivers of each bus, then use DR. FUZZ to test whether the drivers can be discovered and enabled without the hardware.

DR. FUZZ is implemented based on syzkaller, which is a state-of-the-art fuzzing system. While vanilla syzkaller does not support device-free driver fuzzing, after DR. FUZZ integrates the semantic-informed mechanism into syzkaller through the device adaptor, it can support device-free driver fuzzing. For comparison purpose, we create a new syzkaller (namely syzkaller-dev) by adding the device adaptor (V) to the vanilla syzkaller. Syzkaller-dev tries to complete the driver initialization by injecting random data. We will compare DR. FUZZ with syzkaller-dev to confirm its effectiveness.

A. The effectiveness for driver initialization

We first evaluate the effectiveness of DR. FUZZ for the driver initialization without the devices. Note that driver initialization



(a) One hour **(b)** Three hours **(c)** Twenty-four hours

Fig. 6: The number of device Match/Initialization with each technique. VI denotes only enabling byte-level value inference. BP denotes enabling byte-priority inference. PF denotes enabling path state feedback. FUZZ denotes syzkaller-dev, which is a the traditional fuzzing based only on code coverage.

Devices Driver	#Match/#Initialize/#Devs	Average time
PCI drivers	90/46/90	401s
ISA drivers	24/16/24	382s
SCSI drivers	10/4/10	388s
I2C drivers	59/56/59	126s
USB interface drivers	31/27/31	661s

TABLE III: The number of device drivers initialized by DR. FUZZ. `Match` denotes the number of drivers DR. FUZZ can match. `Initialize` denotes the number of device drivers DR. FUZZ can initialize. The difference between these two is whether the probe function can return successfully. `Average time` denotes the average time for matching.

also includes driver matching. The difference between the two is that initialization additionally includes the execution of the probe function. Thus, initialization is more complicated than matching, and we separately evaluate the effectiveness for these two steps.

We allocate a time budget of 24 hours for each driver. Table III shows the results. DR. FUZZ successfully matched all of these drivers. That is, DR. FUZZ successfully triggers the execution of the `probe` function for all drivers. DR. FUZZ further successfully initialized 149 drivers, with a success rate of almost 70%. The drivers under I2C has a higher success rate. This is mainly because the protocol of I2C bus is simpler than others, with less structures. We analyzed some drivers that DR. FUZZ cannot initialize, by checking the implementation of their `probe` and the fuzzing input logs. We find the main reason to be that the number of critical structures is too large, so that the budgeted fuzzing time is not enough. To further confirm our cause analysis, we run the failed drivers again with 3 hours more. Not surprisingly, DR. FUZZ is able to successfully initialize 6 more drivers. This indicates that, once we allocate more time budget for DR. FUZZ, more device drivers will be successfully initialized.

Meanwhile, the average time for matching is short. As Table III shows, the shortest cost is the I2C driver is 126s. This is because our static analysis successfully identifies the valid values for the fields. PCI drivers and USB drivers both need more time to initialize because their structures are more complex. For example, `e1000` obtains 204 bytes from inputs, and DR. FUZZ identifies the values for 86 bytes. The remaining 118 bytes have to be guessed through fuzzing at runtime. Since the time for each driver to complete the initialization is very different, the average calculation time is not statistically significant, so we only show the shortest time of driver initialization. Among the tested drivers, driver `aat2870`, only costs 6 minutes to be initialized successfully, which costs the shortest. The more complex PCI device driver still has a large number of drivers that failed to complete the initialization after more than 24 hours.

Comparison to syzkaller-dev in initialization. To show the effectiveness of the semantic-informed mechanism, we compare DR. FUZZ with the syzkaller-dev. We count the number of drivers initialized by syzkalle-dev. As shown in Figure 6, almost no driver can be matched after running for 1 hour. Even after running for 24 hours, only a few I2C drivers can be matched; however, syzkaller-dev can still not initialize any of them. This result confirms the effectiveness of the semantic-informed mechanisms in DR. FUZZ.

The contributions of each technique. In order to analyze the contribution of each technique for device initialization, we break down our evaluation results. For the same device drivers, we enable the combination of byte-level value inference, byte-priority inference, and error state feedback to complete the driver initialization. We run each device driver for up to one hour, three hours, and 24 hours. Figure 6 shows the results with each technique.

Using value inference, running for 1 hour, 57 drivers were successfully matched, and no driver was successfully initialized. When using value inference and byte priority both, 112 drivers can be successfully matched. The rate is increased by 2X. However, they both still cannot initialize any drivers. This is mainly because the main I/O access methods of match and initialization are different, which leads to the reduction of hard-code address, which further makes byte-priority unable to work. With the introduction of path state feedback, 68 drivers can be successfully initialized. Path state not only allows the fuzzer to generate more appropriate values, but also provides byte-priority information when part of the value is mutation. Finally, after enabling all of the three techniques, DR. FUZZ can successfully initialize 149 of 214 device drivers, within 24 hours.

It can be seen from the analysis that value inference can only complete the initialization of part of the driver. While the value inference is effective in matching, it is not effective in completing the whole initialization—the value inference alone cannot succeed in initialization. With the help of byte-priority technique, we matched more device drivers. However, without the help of the state feedback, it is difficult for byte-priority alone to produce great results, and the best results can only be produced when all three are enabled.

B. Bug Finding

Although LKDM is uniformly followed after Linux-v2.6. We chose Linux v5.9-rc8 as the target as it was the latest version as of the evaluation. To show the ability of DR. FUZZ in finding new bugs, we run DR. FUZZ for each device driver for 24 hours after the device driver is enabled. Overall, we run four instances for about four weeks with our desktop.

Across the 214 drivers, we in total found 46 unique new memory bugs. The vulnerabilities are summarized in Table IV. In these bugs, DR. FUZZ detected 6 via a kernel warning or crash, and the checkers (KASAN [22]) caught the remaining 40. These 40 bugs include slab-out-of-bounds access (8), use-after-free (13), NULL pointer dereference (19). All these bugs are critical, as they will result in memory corruption or kernel crash. For examples, `pluto2` and `ens1371` both are the slab-out-of-bounds access bugs which can be used to compromise the system. We have submitted these findings to developers through patches and Bugzilla [1]; 27 have been confirmed and applied by Linux maintainers, while the remaining 19 are pending.

Bug Type	New Bugs	Crash/Checker
slab-out-of-bounds access	8	KASAN
use-after-free access	13	KASAN
general protection	6	Crash
NULL-pointer dereference	19	KASAN

TABLE IV: Summary of new bugs found

C. Performance

To further show the performance of DR. FUZZ, we analyze the execution logs. We aim to show the code coverage and the throughput of the fuzzer after introducing new feedback.

Code coverage. We collect the accumulated code coverage in the driver-related code. We use different ways of collecting coverage for system boot and driver loading. The code coverage after testing 214 drivers have reached 5%, compared to the 3% without DR. FUZZ; that is, DR. FUZZ increases the coverage by 66.7%. The reason for the small coverage is that the coverage is compared to the entire system code. Because the driver codes we tested account for a very small proportion of the entire kernel code, the overall coverage is not high even after the driver is initialized. To show the coverage changes more clearly, we analyze parts of the drivers. We randomly select some drivers with different scales of code lines to investigate the details. The results are shown in Table V.

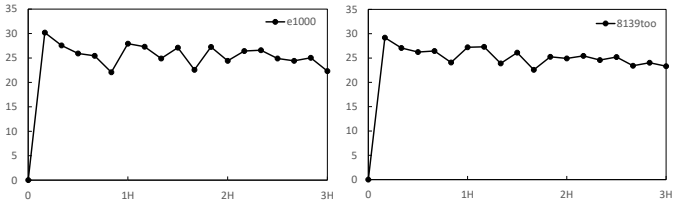
For the selected drivers, the average coverage of driver code is less than 3% after running syzkaller for 24 hours. However, with DR. FUZZ, the average coverage of driver code is more than 5% after running for only 24 hours. `aat2870` has the highest coverage as the I2C protocol is simpler than other bus drivers. It has 16 functions. The number of executed functions is 8. The function coverage is 50%. As the unexecuted functions include complicated ones, the code coverage is only 18%.

Driver	Locs	Funs	Coverage
aat2870	541	8	5.8%
pluto2	621	8	3.9%
8139too	1,826	23	1.2%
maestro	1,897	12	1.6%
dl2k	2,121	18	0.9%
pcnet32	2,544	21	1.4%
be2net	3,542	24	1.1%
forcedeth	5,624	24	1.1%
et131x	8,455	32	1.5%
e1000	14,861	43	0.8%

TABLE V: Code coverage. Locs describes the total lines of the driver code. Funcs is the number of the executed functions.

Indeed, the overall coverage is not high. It is worth noting that in this experiment, we only target system calls and fuzz for only 24 hours. According to the comparison (Table V) in HFL [24], syzkaller has already outperformed many other kernel fuzzers. Even in their much higher-performance testing environment, syzkaller achieves only a 7.9% of code coverage after 50 hours, and HFL is only slightly better than syzkaller, with a 33% increase. Therefore, DR. FUZZ's achievement in increasing coverage is actually significant.

Fuzzing Throughput. Figure 7 shows the execution speed of DR. FUZZ in a sampled period of 3 hours. The figure shows that DR. FUZZ achieves a fuzzing throughput ranging from 21–33 exec/sec, much lower than the fuzzers for userspace program where fuzzer often achieve up to thousands of executions per second. However, driver fuzzing is generally slow. In fact, DR. FUZZ outperforms existing driver fuzzers in throughput. To compare with the syzkaller, the average throughput is actually increased from 24.2/s to 28.7/s, i.e., by 18%. The main reason is that the feedback of error states execution effectively accelerates the fuzzing process.



(a) e1000 **(b)** 8139too

Fig. 7: DR. FUZZ fuzzing throughput (execs/second) measured every 10 minutes for 3 hours

D. The semi-malformed input generation

As we mentioned in §I, the semantic-informed mechanism is generic. After a successful initialization, we can also use our semantic-informed mechanism to generate semi-malformed inputs to improve the code coverage and throughput of driver

fuzzing. To show its effectiveness, we further collect the coverage after the kernel initializes the driver successfully with 24 hours. In this experiment, we instead employ our state feedback to prioritize error states instead of normal states, i.e., breadth first, as opposed to depth first in initialization. Figure 8 shows the coverage.

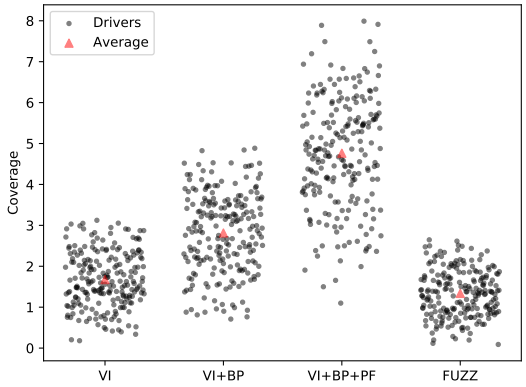


Fig. 8: The coverage of using each technique after the device initialization. VI denotes only enabling byte-level value inference. BP denotes enabling byte-priority inference. PF denotes enabling path state feedback. FUZZ denotes the baseline—coverage-only fuzzing.

From the figure, we can see that compared to the original syzkaller, the coverage of all drivers increased. Among them, the improvement is the most significant when all of the three techniques are used. The average code coverage has increased from 1.6% to about 4.8%, by 200% or 3X. When we only use byte-level value inference, the coverage of DR. FUZZ is closer to syzkaller, and the average coverage has increased from 1.6% to 1.8%. After the device is initialized, the I/O operations are mainly composed of a large number of MMIO and DMA operations, and these operations rarely contain fixed candidate values, so the effect of value inference is reduced. And state feedback still provides a good fuzzing guidance. This is mainly due to the large number of error states before and after initialization, so it can guide fuzzing to generate malformed data which makes the fuzzer execute broader paths.

E. Compared to Related Tools

We compare DR. FUZZ against other driver testing/bug finding tools to demonstrate its importance and uniqueness.

Fuzzer	Device free	Multiple buses	Semantic	Coverage guide	New feedback
USBFuzz	√	×	×	√	×
Syzkaller	×	×	×	√	×
DR. FUZZ	√	√	√	√	√

TABLE VI: Comparison of Virtual-Device Fuzzers

We compare DR. FUZZ with syzkaller and USBFuzz. USBFuzz is a portable, flexible, and modular framework for fuzz-testing USB drivers. At its core, USBFuzz uses a software-emulated USB device to provide random device data to drivers (when they perform IO operations). Syzkaller, a state-of-the-art kernel fuzzer, is an unsupervised coverage-guided kernel fuzzer. We reused its `sysgen` as our user mode agent. Syzkaller supports fuzzing the Linux kernel USB subsystem.

We compare these three tools from multiple aspects, and the results are shown in Table 6. Although syzkaller is currently the most widely used kernel fuzzing tool, there are still many problems with this tool in driver fuzzing. Though syzkaller currently provides support for USB hardware, it is still unable to test other bus device drivers. USBFuzz solves the USB device driver testing problem very well, but as it implements an emulated USB device in the QEMU, it cannot be used to test other device drivers such as PCI devices. We both implement fake device to attach to the target system through the Qemu. However, USBFuzz only focuses on the USB bus drivers. More importantly, there is a large amount of interesting information in the device driver code, and USBFuzz does not make full use of the semantics to help the fuzzing. DR. FUZZ is generic and can support multiple buses and provide the byte-priority of input mutation. It also provides new feedback using error states.

VII. RELATED WORK

Device emulation. Thunderclap [29] develops a peripheral device emulation platform that utilizes a CPU on the FPGA to implement a full software model of an arbitrary peripheral device. It needs hardware (FPGA) and depends on the qemu device simulation code. POTUS [37] is a bug-detection system that is built on the S2E framework and can automatically identify vulnerabilities in USB device drivers for Linux. It presents a new method for simulating a virtual USB device based on the QEMU, allowing people to test arbitrary client drivers. MARSS [36] provides a simulation framework for x86 full system, which can simulate various IO devices. However, these simulation both needs manual analysis and only supports limited devices. They are inefficient and time-consuming to be used for Linux drivers testing.

Code analysis on driver. Dr. CHECKER [28] is a bug-finding tool for Linux kernel drivers based on program analysis techniques. Microsoft's Static Driver Verifier (SDV) [48] identifies API-misuse using static data-flow analysis. Linux Driver Verification (LDV) [5] is a tool based on BLAST which offers precise pointer analysis. The liberal use of pointers in the kernel makes the result not precise. Charm [46] is a system that can dynamically analyze device drivers of mobile systems. Charm can execute device drivers in a virtual machine of workstations based on remote device driver execution. These tools both have a high false positive rate, whose results need to be confirmed manually. Agamotto [45] is a dynamic-analysis system around lightweight VM checkpointing primitives and can fuzz USB and PCI drivers of Linux. However, it cannot do the fuzzing for the specific device driver. SADA [4] is a static-analysis approach to automatically detect unsafe DMA accesses in device drivers. It cannot support the device driver running to provide the specific input to trigger the bugs.

SymDrive [40] is a symbolic execution system that can test drivers of the Linux kernel and FreeBSD through symbolize the devices and corresponding input without accessing the devices. SymDrive proposes favor-success path-selection algorithm that also exploits the function's return value. However, its aggressive path pruning may terminate paths that lead to bugs. In addition,

as a symbolic execution–based system, SymDrive is limited in scalability and performance. The I/O address is very large in the Linux kernel, and symbolizing the large input space would be impractical; also, symbolic execution often leads to path explosion. In particular, SymDrive tested only 21 Linux drivers. Mousse [26] is a system that analyzes programs that cannot be virtualized, such as OS services managing I/O devices. FirmUSB [18] is a domain informed firmware analysis tool, which uses domain knowledge of USB protocol to test firmware images through symbolic execution. Considering the challenges in modeled OS services and complicated driver code, the works are still hard to support all drivers.

Driver Fuzzing. USBFuzz [38] is a flexible and modular tool for fuzz-testing USB drivers. It uses a software-emulated USB device to generate random device inputs and further test USB drivers in the Linux kernel, macOS, etc. It only focuses on the USB drivers and does not exploit semantic information. DIFUZE [11] is an interface-aware fuzzing, which can effectively test Linux kernel drivers through a set of `ioctl` interfaces. DIFUZE focuses on the userspace-kernel boundary, while our work focuses on the hardware-OS boundary. PeriScope [44] is a generic probing framework that can detect bugs in device drivers, which addresses the analysis needs of two types of peripheral interface mechanisms, MMIO and DMA. Wdev-Fuzzer [30] is a fuzzer that can analyze device drivers of communication protocols, such as Wi-Fi device driver. Sylvester Keil et al. [21] presented a fuzzing system for 802.11 devices by moving the target system into a simulated environment and replacing the complex communication with high-level inter-process communication. They both do not use any semantic information to improve the fuzzing.

Firmware re-hosting. HALucinator [9] can re-host firmware with the virtual device. It provides replacements for hardware abstraction layers (HAL) functions to support the firmware execution. However, HAL is now deprecated on most Linux distributions. P2IM [13] generates processor-peripheral interface models to enable hardware independent and scalable firmware testing. It only focuses on microcontrollers that is simple than the general processor. Jetset [19] uses directed symbolic execution to info the values that need to be read from devices to progress toward the goal address. It does not use the semantics of the device drivers. Pretender [17] records the low-level hardware interactions and creates the modeled peripherals to provide full-system emulation of the embedded firmware. It requires recording the interactions first, which is almost impossible when doing the driver fuzzing without the device. EASIER [39] proposes a dynamic ex-vivo device driver analysis for Android phones. It utilizes the evasion kernel to load and initialize drivers by copying the device tree from host kernel or generating the device tree entry from random content. Its evaluation shows it cannot build the proper struct to initialize the driver without the host device tree. However, DR. FUZZ automatically create "driver initializers" to construct the proper structure.

VIII. CONCLUSION

In this paper, we presented DR. FUZZ, a new device-free driver fuzzer. DR. FUZZ eliminates a bottleneck in existing driver fuzzing. At the core of DR. FUZZ is the new semantic-informed mechanism. The mechanism is based on two insights. First, based on our characteristics study of the device drivers, we find that the device drivers enforce validation chains for initialization, so properly constructing the data structures to pass the validation chains will allow us to enable the driver without the devices. Second, we find that the driver code contains rich and useful semantic information that can help use figure out expected valid inputs from the devices. We proposed three new techniques to make the semantic-informed mechanism practical: the byte-level and field-sensitive value inference, byte-priority inference, and error state as feedback. We implemented DR. FUZZ and thoroughly evaluated its effectiveness and performance. DR. FUZZ successfully run drivers without the corresponding devices. Evaluation results show that DR. FUZZ even outperforms existing driver fuzzers equipped with hardware—increasing the code coverage by 70% and the throughput by 18%. With DR. FUZZ, we also find 46 new bugs in the Linux drivers.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their feedback and suggestions. Wenjia Zhao was supported in part by China Scholarship Council. Kangjie Lu and Qiushi Wu were supported in part by NSF awards CNS-1815621, CNS-1931208 and CNS-2045478. Yong Qi was supported in part by the Blockchian Core Technology Strategic Research Program under Grant 2020KJ010801. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] "Bugzilla," http://www.bugzilla.org/.

[2] "Usbguard." [Online]. Available: https://usbguard.github.io/

[3] "The linux driver implementer's api guide," 2020, https://www.kernel.org/doc/html/v4.11/driver-api/index.html.

[4] J. Bai, T. Li, K. Lu, and S. Hu, "Static detection of unsafe DMA accesses in device drivers," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 1629–1645. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bai

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, Y. Berbers and W. Zwaenepoel, Eds. ACM, 2006, pp. 73–85. [Online]. Available: https://doi.org/10.1145/1217935.1217943

[6] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 41–46. [Online]. Available: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html

[7] J. Caballero, H. Yin, Z. Liang, and D. X. Song, "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications*

Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 317–329. [Online]. Available: https://doi.org/10.1145/1315245.1315286

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.

[9] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[10] T. L. compiler infrastructure. [Online], 2019, https://llvm.org/.

[11] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[12] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz, "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, P. Ning, P. F. Syverson, and S. Jha, Eds. ACM, 2008, pp. 391–402. [Online]. Available: https://doi.org/10.1145/1455770.1455820

[13] 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 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[14] T. Goodspeed and S. Bratus, "Facedancer usb: Exploiting the magic school bus," in *Proceedings of the REcon 2012 Conference*, 2012.

[15] Google, "Kernelmemorysanitizer, a detector of uses of uninitialized memory in the linux kernel," 2018, https://github.com/google/kmsan.

[16] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "Eio: Error handling is occasionally correct." in *FAST*, vol. 8, 2008, pp. 1–16.

[17] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[18] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

[19] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 321–338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson

[20] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds. ACM, 2012, pp. 87–98. [Online]. Available: https://doi.org/10.1145/2150976.2150987

[21] S. Keil and C. Kolbitsch, "Stateful fuzzing of wireless device drivers in an emulated environment," *Black Hat Japan*, 2007.

[22] L. kernel document, "The kernel address sanitizer (kasan)," 2018, https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html.

[23] D. Kierznowski, "Badusb 2.0: Usb man in the middle attacks," *Retrieved from RoyalHolloway: https://www. royalholloway. ac. uk/isg/documents/pdf/technicalreports/2016/rhul-isg-2016-7-david-kierznowski. pdf*, 2016.

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

[25] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. [Online]. Available: https://www.ndss-symposium.org/ndss2008/automatic-protocol-format-reverse-engineering-through-context-aware-monitored-execution/

[26] Y. Liu, H. Hung, and A. A. Sani, "Mousse: a system for selective symbolic execution of programs with untamed environments," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds. ACM, 2020, pp. 34:1–34:15. [Online]. Available: https://doi.org/10.1145/3342195.3387556

[27] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1769–1786.

[28] 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 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1007–1024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[29] 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 *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/thunderclap-exploring-vulnerabilities-in-operating-system-iommu-protection-via-dma-from-untrustworthy-peripherals/

[30] M. Mendonça and N. Neves, "Fuzzing wi-fi drivers to locate security vulnerabilities," in *2008 Seventh European Dependable Computing Conference*. IEEE, 2008, pp. 110–119.

[31] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[32] P. Mochel, "The linux kernel device model," in *Ottawa Linux Symposium*, 2002, p. 368.

[33] S. Muchnick *et al.*, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[34] L. of PCI ID's. [Online], 2018, /usr/share/misc/pci.ids.

[35] R. on Component That is Not Updateable. [Online], 2020, https://cwe.mitre.org/data/definitions/1329.html.

[36] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 1050–1055.

[37] 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.

[38] 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.

[39] 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 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1088–1105. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00094

[40] 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 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds. USENIX Association, 2012, pp. 279–292. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann

[41] B. Ruytenberg, "Breaking Thunderbolt Protocol Security: Vulnerability Report," 2020, public version. [Online]. Available: https://thunderspy.io/assets/docs/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf

[42] A. Ryabinin, "Ubsan: run-time undefined behavior sanity checker," *Retrieved April*, vol. 10, p. 2020, 2014.

[43] 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.

[44] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/periscope-an-effective-probing-and-fuzzing-framework-for-the-hardware-os-boundary/

[45] 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.

[46] 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 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 291–307. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[47] J. D. Tian, A. Bates, and K. R. B. Butler, "Defending against malicious USB firmware with goodusb," in *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*. ACM, 2015, pp. 261–270. [Online]. Available: https://doi.org/10.1145/2818000.2818040

[48] M. D. Verifier., May 2010., http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[49] D. Vyukov., "kernel: add kcov code coverage," https://lwn.net/Articles/671640/.

[50] 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.