Poster: 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.**

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 [4]. 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. 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, KMSAN, and UBSAN to monitor the abnormal behaviors to find bugs. For example, DIFUZE [2] identifies 36 vulnerabilities in the Linux-kernel drivers through a set of `ioctl` interfaces. PeriScope [8] 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. 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 limitation—*requiring the hardware device or an emulator*. The kernel supports many devices, e.g., there are more than 13,000 PCI devices alone [1]. 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, 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 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 [6] [3] [7] [5].

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.

* The work was done at the University of Minnesota.

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.

REFERENCES

[1] "List of pci id's." [Online], 2018, http://pci-ids.ucw.cz/.
[2] 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*.
[3] T. Goodspeed and S. Bratus, "Facedancer usb: Exploiting the magic school bus."
[4] 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*.
[5] D. Kierznowski, "Badusb 2.0: Usb man in the middle attacks," 2016.
[6] 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*.
[7] B. Ruytenberg, "Breaking Thunderbolt Protocol Security: Vulnerability Report," 2020.
[8] 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*.

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

Wenjia Zhao[1,2], Kangjie Lu[2], Qiushi Wu[2], Yong Qi[1]

[1]Xi'an Jiaotong University and [2]University of Minnesota

Problem, Goal, and Architecture

Problem

- Device driver is critical in the Kernel
- The drivers are security-critical but buggy
- Driver fuzzing *requires the hardware device or an emulator* which is typically unavailable.

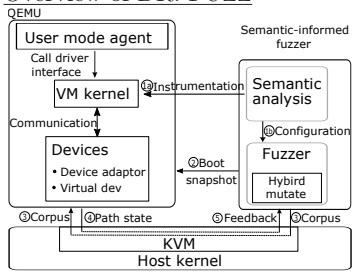
Goal

- Device-free driver fuzzing

Architecture

To implement the device-free driver fuzzing, we find the device input needs to pass the input-validation chains and build the critical structures. To pass these chains, we find some useful features, the temporal input usage, hard-coded I/O address-value mappings and prevalent error handling. Then we propose three new techniques based on these features. Finally, we design a semantic-informed mechanism and implement DR. FUZZ to support the device-free driver fuzzing.

Overview of DR. FUZZ



Technical Contributions

Semantic-informed mechanism

(I) Byte-level and field-sensitive value inference. (II) Byte-priority (temporals) inference. (III) Error state as fuzzing feedback.

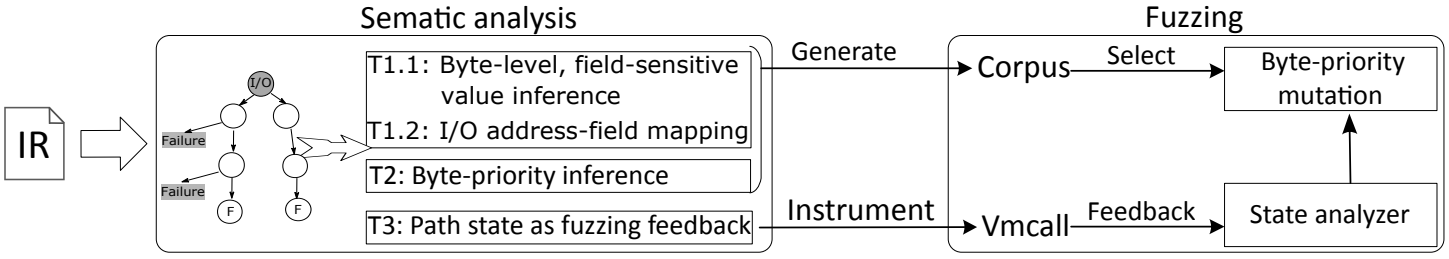
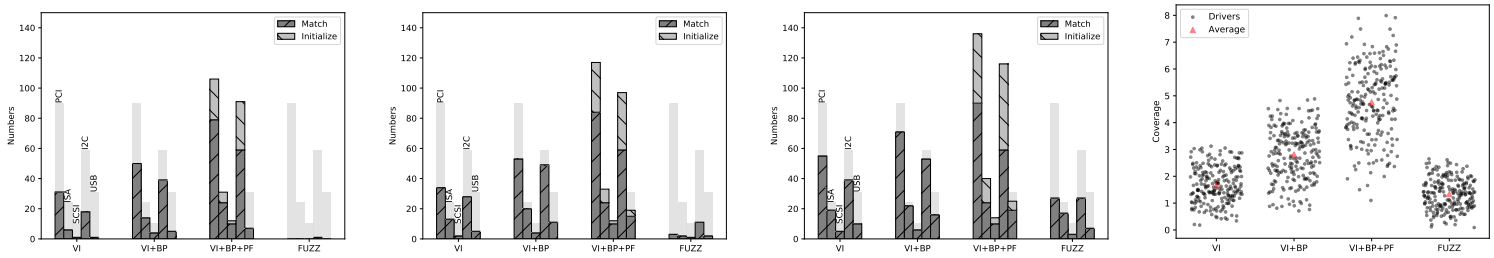


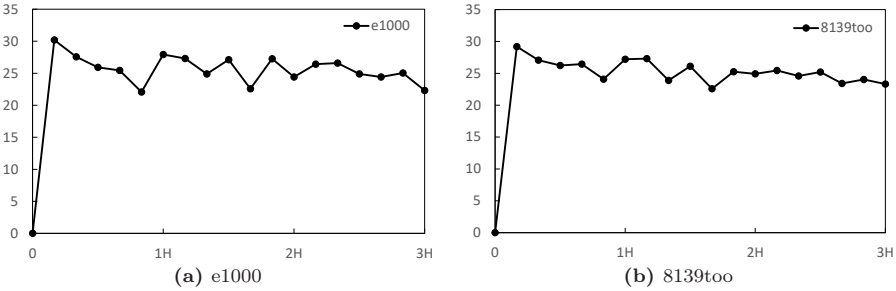
Figure 1: The semantic-informed mechanism. The IR bitcode of the kernel is the input. The semantic analysis generates the initial inputs in the corpus. Meanwhile, it instruments the code based on the analysis to collect the state feedback and pass it to the analyzer.

Results



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

Figure 2: The number of device Match/Initialization with each technique (a) (b) (c) and the coverage of using each technique after the device initialization (d). 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.



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

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

Bug Finding

Across the 214 drivers, we in total found 46 unique new memory bugs. In these bugs, DR. FUZZ detected 6 via a kernel warning or crash, and the checkers (KASAN) caught the remaining 40. These 40 bugs include slab-out-of-bounds access (8), use-after-free (13), NULL pointer dereference (19).

Conclusions

- Semantic-informed mechanism supports to the driver running without the device.
- Three new techniques to make the semantic-informed mechanism practical.
- DR. FUZZ, a new device-free driver fuzzer, successfully run drivers without the corresponding devices.