PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary

Dokyung Song*, Felicitas Hetzelt†, Dipanjan Das‡, Chad Spensky‡, Yeoul Na*, Stijn Volckaert*§, Giovanni Vigna‡, Christopher Kruegel‡, Jean-Pierre Seifert†, Michael Franz*

*Department of Computer Science, University of California, Irvine
†Security in Telecommunications, Technische Universität Berlin
‡Department of Computer Science, University of California, Santa Barbara
§Department of Computer Science, KU Leuven

***Abstract*—The OS kernel is an attractive target for remote attackers. If compromised, the kernel gives adversaries full system access, including the ability to install rootkits, extract sensitive information, and perform other malicious actions, all while evading detection. Most of the kernel's attack surface is situated along the system call boundary. Ongoing kernel protection efforts have focused primarily on securing this boundary; several capable analysis and fuzzing frameworks have been developed for this purpose.**

However, there are additional paths to kernel compromise that do not involve system calls, as demonstrated by several recent exploits. For example, by compromising the firmware of a peripheral device such as a Wi-Fi chipset and subsequently sending malicious inputs from the Wi-Fi chipset to the Wi-Fi driver, adversaries have been able to gain control over the kernel without invoking a single system call. Unfortunately, there are currently no practical probing and fuzzing frameworks that can help developers find and fix such vulnerabilities occurring along the hardware-OS boundary.

We present PERISCOPE, a Linux kernel based probing framework that enables fine-grained analysis of device-driver interactions. PERISCOPE hooks into the kernel's page fault handling mechanism to either passively monitor and log traffic between device drivers and their corresponding hardware, or mutate the data stream on-the-fly using a fuzzing component, PERIFUZZ, thus mimicking an active adversarial attack. PERIFUZZ accurately models the capabilities of an attacker on peripheral devices, to expose different classes of bugs including, but not limited to, memory corruption bugs and double-fetch bugs. To demonstrate the risk that peripheral devices pose, as well as the value of our framework, we have evaluated PERIFUZZ on the Wi-Fi drivers of two popular chipset vendors, where we discovered 15 unique vulnerabilities, 9 of which were previously unknown.

I. INTRODUCTION

Modern electronics often include subsystems manufactured by a variety of different vendors. For example, in a modern cellphone, besides the main application processor running a smartphone operating system such as Android, one might find a number of *peripheral devices* such as a touchscreen display, camera modules, and chipsets supporting various networking protocols (cellular, Wi-Fi, Bluetooth, NFC, etc.). Peripheral devices by different manufacturers have different inner workings, which are often proprietary. *Device drivers* bridge the gap between stable and well-documented operating system interfaces on one side and peripheral devices on the other, and make the devices available to the rest of the system.

Device drivers are privileged kernel components that execute along two different trust boundaries of the system. One of these boundaries is the system call interface, which exposes kernel-space drivers to user-space adversaries. The *hardware-OS interface* should also be considered a trust boundary, however, since it exposes drivers to potentially compromised peripheral hardware. These peripherals should not be trusted, because they may provide a remote attack vector (e.g., network devices may receive malicious packets over the air), and they typically lack basic defense mechanisms. Consequently, peripheral devices have frequently fallen victim to *remote* exploitation [16], [21], [23], [28], [36], [77]. Thus, a device driver must robustly enforce the hardware-OS boundary, but programming errors do occur. Several recently published attacks demonstrated that peripheral compromise can be turned into full system compromise (i.e., remote kernel code execution) by coaxing a compromised device into generating specific outputs, which in turn trigger a vulnerability when processed as an input in a device driver [22], [24].

The trust boundary that separates peripheral subsystems from kernel drivers is therefore of great interest to security researchers. In this paper, we present PERISCOPE, which to our knowledge is the first generic framework that facilitates the exploration of this boundary. PERISCOPE focuses on two popular device-driver interaction mechanisms: *memory-mapped I/O* (MMIO) and *direct memory access* (DMA). The key idea is to monitor MMIO or DMA mappings set up by the driver, and then dynamically *trap* the driver's accesses to such memory regions. PERISCOPE allows developers to register hooks that it calls upon each trapped access, thereby enabling them to conduct a fine-grained analysis of device-driver interactions. For example, one can implement hooks that record and/or mutate device-driver interactions in support of reverse engineering, record-and-replay, fuzzing, etc.

To demonstrate the risk that peripheral devices pose, as

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23176
www.ndss-symposium.org

well as to showcase versatility of the PERISCOPE framework, we created PERIFUZZ, a driver fuzzer that simulates attacks originating in untrusted, compromised peripherals. PERIFUZZ traps the driver's *read accesses* to MMIO and DMA mappings, and fuzzes the values being read by the driver. With a compromised device, these values should be considered to be under an attacker's control; the attacker can freely modify these values at any time, even in between the driver's reads. If the driver reads the same memory location multiple times (i.e., overlapping fetches [79]) while the data can still be modified by the device, double-fetch bugs may be present [45], [68]. PERIFUZZ accurately models this adversarial capability by fuzzing not only the values being read from different memory locations, but also ones being read from the same location multiple times. PERIFUZZ also tracks and logs all overlapping fetches and warns about ones that occurred before a driver crash to help identify potential double-fetch bugs.

Existing work on analyzing device-driver interactions typically runs the entire system including device drivers in a controlled environment [32], [44], [47], [49], [54], [60], [62], [66], such as QEMU [20] or S2E [33]. Enabling analysis in such an environment often requires developer efforts tailored to specific drivers or devices, e.g., implementing a virtual device or annotating driver code to keep symbolic execution tractable. In contrast, PERISCOPE uses a page fault based *in-kernel monitoring* mechanism, which works with all devices and drivers in their existing testing environment. As long as the kernel gets recompiled with our framework, PERISCOPE and PERIFUZZ can analyze device-driver interactions with relative ease, regardless of whether the underlying device is virtual or physical, and regardless of the type of the device. Extending our framework is also straightforward; for example, PERIFUZZ accepts any user-space fuzzer, e.g., AFL, as a plug-in, which significantly reduces the engineering effort required to implement proven fuzzing strategies [25], [26], [30], [31], [37].

We validated our system by running experiments on the software stacks shipping with the Google Pixel 2 and the Samsung Galaxy S6, two popular smartphones on the market at the time of development. To simulate remote attacks that would occur *over the air* in a real-world scenario, we focused on the Wi-Fi drivers of these phones in evaluating our framework. The Google Pixel 2 and Samsung Galaxy S6 are equipped with Qualcomm and Broadcom chipsets, respectively. These two are arguably the most popular Wi-Fi chipset manufacturers at the time of our experiments. In our experiments, our system identified 15 unique vulnerabilities in two device drivers, out of which 9 vulnerabilities were previously unknown, and 8 new CVEs were assigned. We have reported the discovered vulnerabilities to the respective vendors and are working with them on fixing these vulnerabilities. We hope that our tool will aid developers in hardening the hardware-OS boundary, leading to better software security.

In summary, this paper makes the following contributions:

- **A probing framework:** We introduce PERISCOPE, a generic probing framework that can inspect the interactions between a driver and its corresponding device. PERISCOPE provides the means to analyze the hardware-OS boundary, and to build more specialized analysis tools.

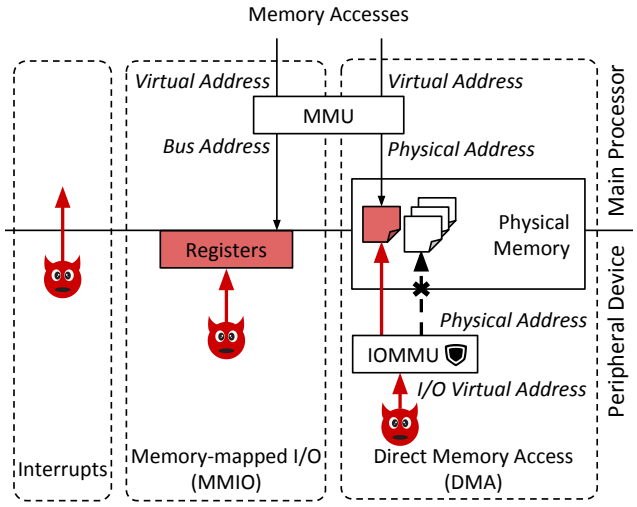


Fig. 1. Hardware-OS interaction mechanisms

- **A fuzzing framework:** We extended PERISCOPE to build PERIFUZZ, a vulnerability discovery tool tailored to detect driver vulnerabilities occurring along the hardware-OS boundary. The tool demonstrates the power of the PERISCOPE framework, and it systematizes the exploration of the hardware-OS boundary.
- **An overlapping fetch fuzzer:** PERIFUZZ fuzzes overlapping fetches in addition to non-overlapping fetches, and warns about overlapping fetches that occurred before a driver crash. A warning observed before a driver crash may indicate the presence of double-fetch bugs.
- **Discovered vulnerabilities:** As part of our evaluation, we discovered previously known and unknown vulnerabilities in the Wi-Fi drivers of two of the most prominent vendors in the market. We responsibly disclosed relevant details to the corresponding vendors.
- **An open-source tool:** We open-sourced our tool[1], in order to facilitate further research exploration of the hardware-OS boundary.

II. BACKGROUND

In this section, we provide the technical background necessary to understand how peripheral devices interact with the OS. We also discuss isolation mechanisms that allow the OS to protect itself against misbehaving peripherals, as well as tools to analyze hardware-OS interactions.

A. Hardware-OS Interaction

Figure 1 illustrates the various ways in which devices can interact with the OS and the device driver. Although we assume that the device driver runs on a Linux system with an ARMv8-A/AArch64 CPU, the following discussion generally applies to other platforms as well.

[1] https://github.com/securesystemslab/periscope

1) Interrupts: A device can send a signal to the CPU by raising an interrupt request on one of the CPU's interrupt lines. Upon receiving an interrupt request, ARMv8-A CPUs first mask the interrupt line so that another interrupt request cannot be raised on the same line while the first request is being handled. Then, the CPU transfers control to the interrupt handler registered by the OS for that interrupt line. Interrupt handlers can be configured at any time, though the OS typically configures them at boot time.

Processing Interrupts: To maximize the responsiveness and concurrency of the system, the OS attempts to defer interrupt processing so that the interrupt handler can return control to the CPU as soon as possible. Typically, interrupt handlers only process interrupts in full if they were caused by time-sensitive events or by events that require immediate attention. All other events are processed at a later time, outside of the interrupt context. This mechanism is referred to as top-half and bottom-half interrupt processing in Linux lingo.

In Linux, after performing minimal amount of work in the hardware interrupt context (`hardirq`), the device driver schedules the work to be run in either software interrupt context (`softirq`), kernel worker threads, or the device driver's own kernel threads, based on its priority. For higher priority work, a device driver can register its own `tasklet`, a deferred action to be executed under the software interrupt context, which also ensures serialized execution. Lower priority work can further be deferred either to kernel worker threads (using the workqueue API) or to the device driver's own kernel threads.

2) Memory-Mapped I/O: Analogous to peripherals using interrupts to signal the OS and the device driver, the CPU uses memory-mapped I/O (MMIO) to signal peripherals. MMIO maps a range of kernel-space virtual addresses to the hardware registers of peripheral devices. This allows the CPU to use normal memory access instructions (as opposed to special I/O instructions) to communicate with the peripheral device. The CPU observes such memory accesses and redirects them to the corresponding hardware. In Linux, device drivers call `ioremap` to establish an MMIO mapping, and `iounmap` to remove it.

3) Direct Memory Access: Direct memory access (DMA) allows peripheral devices to access physical memory directly. Typically, the device transfers data using DMA, and then signals the CPU using an interrupt. There are two kinds of DMA buffers: coherent and streaming.

Coherent DMA buffers (also known as consistent DMA buffers) are usually allocated and mapped only once at the time of driver initialization. Writes to coherent DMA buffers are usually uncached, so that values written by either the peripheral processor or the CPU are immediately visible to the other side.

Streaming DMA buffers are backed by the CPU's cache, and have an explicit owner. They can either be owned by the CPU itself, or by one of the peripheral processors. Certain kernel-space memory buffers can be "mapped" as streaming DMA buffers. However, once a streaming DMA buffer is mapped, the peripheral devices automatically acquires ownership over it, and the kernel can no longer write to the buffer. Unmapping a streaming DMA buffer revokes its ownership from the peripheral device, and allows the CPU to access the buffer's contents. Streaming DMA buffers are typically short-lived, and are often used for a single data transfer operation.

B. Input/Output Memory Management Unit

Since DMA allows peripherals to access physical memory directly, its use can be detrimental to the overall stability of the system if a peripheral device misbehaves. Modern systems therefore deploy an input output memory management unit (IOMMU) (also known as system memory management unit, or SMMU, on the ARMv8-A/AArch64 architecture) to limit which regions of the physical memory each device can access. Similar to the CPU's memory management unit (MMU), the IOMMU translates device-visible virtual addresses (i.e., I/O addresses) to physical addresses. The IOMMU uses translation tables, which are configured by the OS prior to initiating a DMA transfer. Device-initiated accesses that fall outside of the translation table range will trigger faults that are visible to the OS.

C. Analyzing Hardware-OS Interaction

Vulnerabilities in device drivers can lead to a compromise of the entire system, since many of these drivers run in kernel space. To detect these vulnerabilities, driver developers can resort to dynamic analysis tools that monitor the driver's behavior and report potentially harmful actions. Doing this ideally requires insight into the communication between the driver and the device, as this communication can provide the context necessary to find the underlying cause of a vulnerability. Analyzing device-driver communication requires (i) an instance of the device, whether physical or virtual, and (ii) a monitoring mechanism to observe and/or influence device-driver communication. Existing approaches can therefore be classified based on where and how they observe (and possibly influence) device-driver interactions.

Device Adaptation: To exercise direct control over the data sent from the hardware to the driver, an analyst can adapt the firmware of real devices to include such capabilities. This can be done by reverse engineering the firmware and reflashing a modified one [64], or by using custom hardware that supports reprogramming of devices [1]. However, these frameworks are typically tailored to specific devices, and given the heterogeneity of peripheral devices, their applicability is limited. For example, Nexmon only works for some Broadcom Wi-Fi devices [64], and Facedancer11, a custom Universal Serial Bus (USB) device, can only analyze USB device drivers [1].

Virtual Machine Monitor: A driver can be tested in conjunction with virtual devices running in a virtual environment such as QEMU [20]. The virtual machine monitor observes the behavior of its guest machines and can easily support instrumentation of the hardware-OS interface. Previous work uses existing implementations of virtual devices for testing the corresponding drivers [44], [66]. For many devices, however, an implementation of a virtual device does not exist. In this case, developers must manually implement a virtual version of their devices to interact with the device driver they wish to analyze [47]. Several frameworks alleviate the need for virtual devices by relaying I/O to real devices [71], [80], but these frameworks generally require a non-trivial porting effort for each driver and device, and/or do not support DMA.

Symbolic Execution: S2E augments QEMU with selective symbolic execution [33]. Several tools leverage S2E to analyze the interactions between OS kernel and hardware by selectively converting hardware-provided values into symbolic values [32], [49], [60], [62]. However, symbolic execution in general is prohibitively slow due to the path explosion and constraint solving problem. Moreover, symbolic execution itself does not reveal vulnerabilities, but rather generates a set of constraints that must be analyzed by separate checkers. Writing such a checker is not trivial. Most of the checkers supported by SymDrive, for example, target stateless bugs such as kernel API misuses, but ignore memory corruption bugs [62].

III. PERISCOPE DESIGN

We designed PERISCOPE as a dynamic analysis framework that can be used to examine bi-directional communication between devices and their drivers over MMIO and DMA. Contrary to earlier work on analyzing device-driver communication on the device side, we analyze this communication on the driver side, by intercepting the driver's accesses to communication channels. PERISCOPE does this by hooking into the kernel's page fault handling mechanism. This design choice makes our framework driver-agnostic; PERISCOPE can analyze drivers with relative ease, regardless of whether the underlying device is virtual or real, and regardless of the type of the peripheral device.

At a high level, PERISCOPE works as follows. First, PERISCOPE automatically detects when the target device driver creates a MMIO or DMA memory mapping, and registers it. Then, the analyst selects the registered mappings that he/she wishes to monitor. PERISCOPE marks the pages backing these monitored mappings as not present in the kernel page tables. Any CPU access to those marked pages therefore triggers a page fault, even though the data on these pages is present in physical memory.

When a kernel page fault occurs, PERISCOPE first marks the faulting page as present in the page table (❶ in Figure 2). Then, it determines if the faulting address is part of any of the monitored regions (❷). If it is not, PERISCOPE re-executes the faulting instruction (❺), which will now execute without problems. Afterwards, PERISCOPE marks the page as not present again (❼), and resumes the normal execution of the faulting code.

If the faulting address does belong to a monitored region, PERISCOPE invokes a pre-instruction hook function registered by the user of the framework, passing information about the faulting instruction (❹). Then, PERISCOPE re-executes the faulting instruction (❺). Finally, PERISCOPE invokes the post-instruction hook registered by the driver (❻), marks the faulting page as not present again (❼), and resumes the execution of the faulting code.

A. Memory Access Monitoring

Tracking Allocations: PERISCOPE hooks the kernel APIs used to allocate and deallocate DMA and MMIO regions[2]. We use these hooks to maintain a list of all DMA/MMIO allocation contexts and their active mappings. PERISCOPE assigns an identifier to every context in which a mapping is allocated, and presents the list of all allocation contexts as well as their active mappings to privileged user-space programs through the `debugfs` file system.

[2]Establishing DMA and MMIO mappings is a highly platform-dependent process, so device drivers are obliged to use the official kernel APIs to do so.

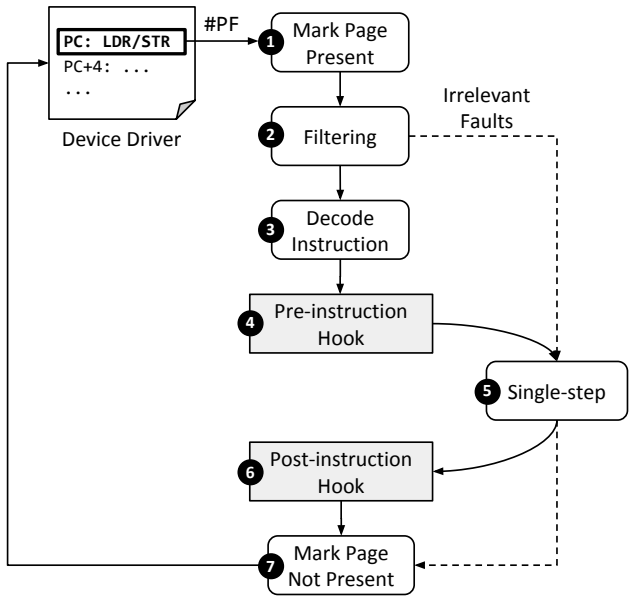


Fig. 2. PERISCOPE fault handling

Enabling Monitoring: PERISCOPE exposes a privileged user-space API that enables monitoring of DMA/MMIO regions on a per-allocation-context basis. Once monitoring is enabled for a specific allocation context, PERISCOPE will ensure that accesses to all current and future regions allocated in that context trigger page faults.

Clearing Page Presence: PERISCOPE marks all pages containing monitored regions as not present in the kernel's page tables to force accesses to such pages to trigger page faults. One complication that can arise here is that modern architectures, including x86-64 and AArch64, can support multiple page sizes within the same page table. On AArch64 platforms, a single page table entry can serve physical memory regions of 4KB, 16KB, or 64KB, for example. If a single (large) page table entry serves both a monitored and a non-monitored region, then we split that entry prior to marking the region as not present. We do this to avoid unnecessary page faults for non-monitored regions. Note that, even after splitting page table entries, PERISCOPE cannot rule out spurious page faults completely, as some devices support DMA/MMIO regions that are smaller than the smallest page size supported by the CPU.

Trapping Page Faults: PERISCOPE hooks the kernel's default kernel page fault handler to monitor page faults. Inside the hook function, we first check if the fault originated from a page that contains one of the monitored regions. If the fault originated from some other page, we immediately return from the hook function with an error code and defer the fault handling to the default page fault handler. If the fault did originate from a page containing a registered buffer, PERISCOPE marks that page as present (❶), and then checks

if the faulting address falls within a monitored region (❷). If the faulting address is outside a monitored region, we simply single-step the faulting instruction (❺), mark the faulting page as not present again (❼), and resume normal execution of the faulting code. If the faulting address does fall within a monitored region, however, we proceed to the instruction decoding step (❸).

Instruction Decoding: In order to accurately monitor and (potentially) manipulate the communication between the hardware/firmware and the device driver, we need to extract the source register, the destination register and the access width of the faulting instruction (❸ in Figure 2). We implemented a simple AArch64 instruction decoder, which provides this information for all load and store instructions. PERISCOPE carries this information along the rest of its fault handling pipeline.

Pre-instruction Hook: After decoding the instruction, PERISCOPE calls the pre-instruction hook that the user of our framework can register (❹). We pass the address of the faulting instruction, the memory region type (MMIO or DMA coherent/streaming), the instruction type (load or store), the destination/source register, and the access register width to this hook function. The pre-instruction hook function can return two values: a default value and a skip-single-step value. If the function returns the latter, PERISCOPE proceeds immediately to step ❻. Otherwise, PERISCOPE proceeds to step ❺.

PERISCOPE provides a default pre-instruction hook which logs all memory stores before the value in the source register is stored to memory. We maintain this log in a kernel-space circular buffer that can later be read from the file system using `tracefs`.

Single-stepping: When execution returns from the pre-instruction hook, and the hook function did not return the skip-single-step value, we re-execute the faulting instruction, which can now access the page without faulting. We use the processor's single-stepping support to ensure that only the faulting instruction executes, but none of its successors do (❺).

Post-instruction Hook: When PERISCOPE regains control after single-stepping, it first clears the page present flag for the faulting page again so that future accesses to the faulting page once again trigger a page fault. Then, it calls the post-instruction handler, which, similarly to the pre-instruction handler, has a default implementation that can be overridden through our API (❻). The default handler logs all memory loads by examining and logging the value that is now stored in the destination register.

IV. PERIFUZZ DESIGN

We built PERIFUZZ as a client module for PERISCOPE. PERIFUZZ can generate and provide inputs for device drivers. The goal of our fuzzer is to uncover vulnerabilities that could potentially be exploited by a compromised peripheral device.

A. Threat Model

Peripheral Compromise: We assume that the attacker can compromise a peripheral, which, in turn, can send arbitrary data to its device driver. Compromising a peripheral device is

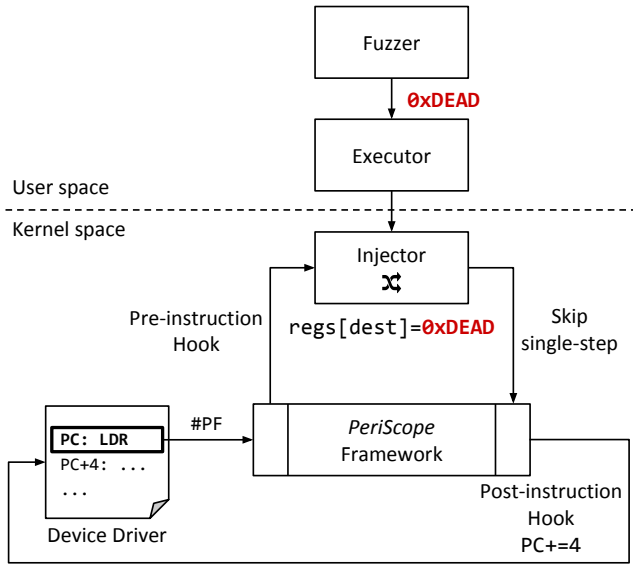


Fig. 3. PERIFUZZ overview

feasible because such devices rarely deploy hardware protection mechanisms or software mitigations. As a result, silent memory corruptions occur frequently [56], which significantly lowers the bar to mount an attack. That peripherals can turn malicious after being attacked was demonstrated by successful *remote* compromises of several network devices such as ethernet adapters [36], GSM baseband processors [28], [77], and Wi-Fi processors [16], [21], [23].

IOMMU/SMMU Protection: For many years, a strict hardware-OS security boundary existed in theory, but it was not enforced in practice. Most device drivers *trusted* that the peripheral was benign, and gave the device access to the entire physical memory (provided that the device was DMA-capable), thus opening the door to DMA-based attacks and rootkits [17], [69]. This situation has changed for the better with the now widespread deployment of IOMMU units (or SMMU for AArch64). IOMMUs can prevent the device from accessing physical memory regions that were not explicitly mapped by the MMU, and they prevent peripherals from accessing streaming DMA buffers while these are mapped for CPU access. The latter restriction can be imposed by invalidating IOMMU mappings, or by copying the contents of a streaming DMA buffer to a temporary buffer (which the peripheral cannot access) before the CPU uses them [52], [53]. We assume that such an IOMMU is in place, and that is being used correctly.

Summary: In our model, the *attacker* can (i) compromise a peripheral such as a Wi-Fi chipset over the air by abusing an existing bug in the peripheral's firmware, (ii) *exercise control* over the compromised peripheral to send arbitrary data to the device driver, and, (iii) not access the main physical memory, except for memory regions used for communicating with the device driver.

B. Design Overview

PERIFUZZ is composed of a number of components, as illustrated in Figure 3. Our design is fully modular, so each

component can be swapped out for an alternative implementation that exposes the same interface.

Fuzzer: We use a fuzzer that runs in user space. This component is responsible for generating inputs for the device driver and processing execution feedback. Our modular design allows us to use any fuzzer capable of fuzzing user-space programs. We currently use AFL as our fuzzer, as was done in several previous works that focus on fuzzing kernel subsystems [42], [57], [65].

Executor: The executor is a user-space-resident bridge between the fuzzer (or any input provider) and the injector. The executor takes an input file as an argument, and sends the file content to the injector via a shared memory region mapped into both the executor's and the injector's address spaces. The executor then notifies the injector that the input is ready for injection, and periodically checks if the provided input has been consumed. PERIFUZZ launches an instance of the executor for every input the fuzzer generates. The executor is also used to reproduce a crash by providing the last input observed before the crash.

Injector: The injector is a kernel-space module that interfaces with our PERISCOPE framework. The injector registers a pre-instruction hook with PERISCOPE, which allows the injector to monitor and manipulate all data the device driver receives from the device. At every page fault, the injector first checks if fuzzing is currently enabled, and if there is a fuzzer/executor-provided input that has not been consumed yet. If both conditions are met, the injector overwrites the destination register with the input generated by the fuzzer.

Note that PERIFUZZ manipulates only the values device drivers *read* from MMIO and DMA mappings, but not the values they write. PERIFUZZ, in other words, models compromised devices, but not compromised drivers.

C. Fuzzer Input Consumption

We treat each fuzzer-generated input as a serialized sequence of memory accesses. In other words, our injector always consumes and injects the first non-consumed inputs found in the input buffer shared between the executor and injector. This fuzzer input consumption model allows for *overlapping fetch fuzzing* as it automatically provides different values for multiple accesses to the same offsets within a target mapping (i.e., overlapping fetches [79]). Providing different values for overlapping fetches enables us to find double-fetch bugs, if triggering such bugs leads to visible side-effects such as a driver crash. Our fuzzer also keeps track of the values returned for overlapping fetches, and can output this information when a driver crashes, thereby helping us to narrow down the cause of the crash. In fact, the double-fetch bugs we identified using PERIFUZZ would not have been found without this information (see Section VI).

Since we assume that the attacker cannot access streaming DMA buffers while they are mapped for CPU access (see Section IV-A), we take extra care not to enable overlapping fetch fuzzing for streaming DMA buffers. To this end, we maintain a history of read accesses, and consult this history to determine if a new access overlaps with any previous access. If they overlap, we return the same values returned for the previous access, and do not consume any bytes from the fuzzer input. Algorithm 1 shows how we pick values to inject for each driver read from an MMIO or DMA mapping.

Algorithm 1 Fuzzer Input Consumption at Each Driver Read

```
1: global variables                ▷ Initialized when switching fuzzer input
2:     Input ← [...]
3:     InputOffset ← 0
4:     PrevReads ← {}
5:     OverlappingFetches ← {}
6: end global variables
7: function FUZZDRIVERREAD(Address, Width, Type)
8:     Value ← Input[range(InputOffset, Width)]
9:     for all Prev in PrevReads do
10:        Overlap ← Prev.range ∩ range(Address, Width)
11:        if Overlap is not empty then
12:            if Type is DMA Streaming then
13:                Value[Overlap] ← Prev.value(Overlap)
14:            else
15:                OverlappingFetches ← OverlappingFetches ∪ {(Overlap, Value)}
16:            end if
17:        end if
18:    end for
19:    InputOffset ← InputOffset + Width
20:    PrevReads ← PrevReads ∪ {(Address, Width, Value)}
21:    return Value
22: end function
```

An additional benefit of our fuzzer input consumption model is that it helps to keep the input size small, because we only have to generate fuzzer input bytes for read accesses that actually happen and not for the entire fuzzed buffer, which may contain bytes that are never read.

D. Register Value Injection

PERISCOPE provides the destination register and the access width when it calls into PERIFUZZ's pre-instruction hook handler. The fuzzer input is consumed for that exact access width, and then injected into the destination register. Our pre-instruction hook function returns the skip-single-step value to PERISCOPE (see Section III-A), as we have emulated the faulting load instruction by writing a fuzzed value into its destination register. Our post-instruction hook function increments the program counter, so the execution of the driver resumes from the instruction that follows the fuzzed instruction.

E. Fuzzing Loop

Each iteration of the fuzzing loop consumes a single fuzzer-generated input. We align each iteration of the fuzzing loop to the software interrupt handler, i.e., `do_softirq`. We do not insert hooks into the hardware interrupt handler, since work is barely done in the hardware interrupt context. The two hooks inserted before and after the software interrupt handler demarcate a single iteration of the fuzzing loop, in which PERIFUZZ consecutively consumes bytes in a single fuzzer input. This design decision allows us to remain device-agnostic, but device driver developers could provide an alternative device-specific definition of an iteration by inserting those two hooks in their drivers. Several low priority tasks are often deferred to the device driver's own kernel threads, and the fuzzing loop can be aligned to the task processing loop inside those threads.

TABLE I. LoC MODIFIED IN THE LINUX KERNEL CODE AND THE PERISCOPE FRAMEWORK ITSELF

Description	LoC
Linux DMA and MMIO allocation/deallocation APIs	92
Linux kernel page fault and debug exception handlers	46
PERISCOPE framework	3843

TABLE II. PERIFUZZ IMPLEMENTATION LoC

Component		LoC
Injector	Kernel-space	441
KCOV (modification)	Kernel-space	176
Executor	User-space	338
Python manager and utility scripts	Host	924

F. Interfacing with AFL

We use AFL [81], a well-known coverage-guided fuzzer, as PERIFUZZ's fuzzing front-end. This is in line with previous work on fuzzing various kernel subsystems [42], [57], [65]. To fully leverage AFL's coverage-guidance, we added kernel coverage and seed generation support in PERIFUZZ.

Coverage-guidance: We modified and used KCOV to provide coverage feedback while executing inputs [74]. Existing implementations of KCOV were developed for fuzzing system calls and only collect coverage information for code paths reachable from system calls. To enable device driver fuzzing, we extended KCOV with support for collecting coverage information for code paths reachable from interrupt handlers. We also applied a patch to force KCOV to collect edge coverage information rather than basic block coverage information [29]. To collect coverage along the execution of the device driver, it is first compiled with coverage instrumentation. This instrumentation informs KCOV of hit basic blocks, which KCOV records in terms of edge coverage. The executor component retrieves the coverage feedback from kernel, once the input has been consumed. Then the executor copies this coverage information to a memory region shared with the parent AFL fuzzer process, after which we signal KCOV to clear the coverage buffer for the next fuzzing iteration.

Automated Seed Generation: Starting with valid test cases rather than fully random inputs improves the fuzzing efficiency, as this lowers the number of input mutations required to discover new paths. To collect an initial seed of valid test cases, we use our PERISCOPE framework to log all accesses to a user-selected set of buffers. We provide an access log parser that automatically turns a sequence of accesses into a seed file according to our fuzzing input consumption model (see Section IV-C). That said, this step is optional; one could start from any arbitrary seed, or craft test cases on their own.

V. IMPLEMENTATION

A. PERISCOPE

We based our implementation of PERISCOPE on Linux kernel 4.4 for AArch64. Our framework is, for the most part, a standalone component that can be ported to other versions of the Linux kernel and even to vendor-modified custom kernels with relative ease. The kernel changes required for PERISCOPE are relatively small compared to the framework implementation itself as shown in Table I.

Tracking Allocations: PERISCOPE hooks the generic kernel APIs used to allocate/deallocate MMIO and DMA regions to maintain a list of allocation contexts. We insert these hooks into the `dma_alloc_coherent` and `dma_free_coherent` functions to track coherent DMA mappings, into the `dma_unmap_page` function[3] and `dma_map_page` to track streaming DMA mappings, and into `ioremap` and `iounmap` to track MMIO mappings.

PERISCOPE assigns a context identifier to every MMIO and DMA allocation context. This context identifier is the XOR-sum of all call site addresses that are on the call stack at allocation time. We mask out the upper bits of all call site addresses to ensure that context identifiers remain the same across reboots on devices that enable kernel address space layout randomization (KASLR).

Monitoring Interface: PERISCOPE provides a user-space interface by exposing `debugfs` and `tracefs` file system entries. Through this interface, a user can list all allocation contexts and their active mappings, enable or disable monitoring, and read the circular buffer where PERISCOPE logs all accesses to the monitored mappings.

As streaming DMA buffer allocations can happen in interrupt contexts, we use a non-blocking spinlock to protect access to data structures such as the list of monitored mappings. When accessing these data structures from an interruptible code path, we additionally disable interrupts to prevent interrupt handlers from deadlocking while trying to access the same structures.

B. PERIFUZZ

We built PERIFUZZ as a client for PERISCOPE. Table II summarizes the code we added or changed for PERIFUZZ.

Kernel-User Interface: The injector registers a device node that exposes device-specific `mmap` and `ioctl` system calls to the user-space executor. The executor can therefore create a shared memory mapping via `mmap` to the `debugfs` file exported by the injector module. Through this interface, the executor passes the fuzzer input to the injector running in the kernel space. The `ioctl` handler of the injector module allows the executor (i) to enable and disable fuzzing, and (ii) to poll the consumption status of a fuzzer input it provided. Similarly, KCOV provides the coverage feedback by exporting another `debugfs` file such that the executor can read the feedback by `mmap`ing the exported `debugfs` file.

Persisting Fuzzer Files: Many fuzzers including AFL store meta-information about fuzzing and input corpus in the file system. However, these files might not persist if the kernel crashes before the data is committed to the disk. To avoid this, we ensure that all the fuzzer files are made persistent, by modifying AFL to call `fsync` after all file writes. Persisting all files allows us (i) to investigate crashes using the last crashing input and (ii) to resume fuzzing with the existing corpus stored in the file system.

[3] `dma_unmap_page` unmaps a streaming DMA mapping from the peripheral processor. Doing so transfers ownership of the mapping to the device driver.

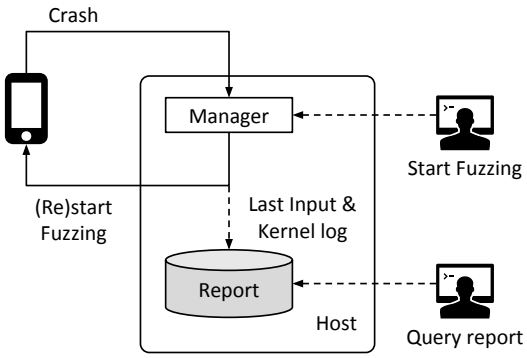


Fig. 4. Continuous fuzzing with PERIFUZZ

TABLE III. TARGET SMARTPHONES

	Google Pixel 2	Samsung Galaxy S6
Model Name	walleye	SM-G920F
Released	October, 2017	April, 2015
SoC	Snapdragon 835	Exynos 7420
Kernel Version	4.4	3.10
Wi-Fi Device Driver	`qcacld-3.0`	`bcmdhd4358`
Wi-Fi IOMMU Protection	Yes	No

Fuzzing Manager: The fuzzing procedure is completely automated through Python scripts that run on a host separate from the target device. The continuous fuzzing loop is driven by a Python program, as illustrated in Figure 4. The manager process runs in a loop in which it (i) polls the status of the fuzzing process, (ii) starts/restarts fuzzing if required, (iii) detects device reboots, (iv) downloads the kernel log and the last input generated before the crash after a reboot, and (v) examines the last kernel log to identify the issue that led to the crash.[4] The manager stores the reports and the last crashing inputs for investigation and bug reporting.

VI. EVALUATION

We evaluated PERISCOPE and PERIFUZZ by monitoring and fuzzing the communication between two popular Wi-Fi chipsets and their device drivers used in several Android smartphones.

A. Target Drivers

We chose Wi-Fi drivers as our evaluation target because they present a large attack surface, as evidenced by a recent series of fully remote exploits [16], [23]. Smartphones frequently connect to potentially untrusted Wi-Fi access points, and Wi-Fi drivers and peripherals implement vendor-specific, complex internal device-driver interaction protocols (e.g., for offloading tasks) that rely heavily on DMA-based communication.

The Wi-Fi peripheral chipset market for smartphones is dominated by two major vendors: Broadcom and Qualcomm. We tested two popular Android-based smartphones that each have a Wi-Fi chipset from one of these vendors, as shown in Table III. We tested the Google Pixel 2, with Android 8.0.0 Oreo[5] and Qualcomm's `qcacld-3.0` Wi-Fi driver. We also tested the Samsung Galaxy S6, on which we installed LineageOS 14.1 and Broadcom's `bcmdhd4358` Wi-Fi driver. LineageOS 14.1 is a popular custom Android distribution that includes the exact same Broadcom driver as the official Android version for the Galaxy S6.

Note that although the Samsung Galaxy S6 has an IOMMU, it is not being used to protect the physical memory from rogue Wi-Fi peripherals. Regardless, we did conduct our experiments under the assumption that IOMMU protection *is* in place. Newer versions of the Samsung Galaxy phones do enable IOMMU protection for Wi-Fi peripherals.

[4]We used Syzkaller's report package to parse the kernel log.

[5]`android-8.0.0_r0.28`

TABLE IV. THE NUMBER OF MMIO AND DMA ALLOCATION CONTEXTS THAT CREATE ATTACKER-ACCESSIBLE MAPPINGS

Driver	MMIO	DMA Coherent	DMA Streaming
`qcacld-3.0`	1	9	5
`bcmdhd4358`	4	11	29

TABLE V. THE NUMBER OF BASIC BLOCKS EXECUTED UNDER WEB BROWSING TRAFFIC PER KERNEL CONTROL PATH. A BASIC BLOCK COULD RUN IN INTERRUPT CONTEXT (**IRQ**), KERNEL THREAD OR WORKER CONTEXT (**KERNEL THREAD**), OR OTHERS (**OTHERS**). SOME BASIC BLOCKS CAN BE REACHED IN SEVERAL CONTEXTS.

Driver	IRQ	Kernel Thread	Others	Hit / Instrumented
`qcacld-3.0`	1633 (36.9%)	2902 (65.6%)	672 (15.2%)	4427/81637
`bcmdhd4358`	743 (68.9%)	284 (26.3%)	301 (27.9%)	1078/23404

B. Target Attack Surface

The code paths that are reachable from peripheral devices vary depending on the internal state of the driver (e.g., is the driver connected, not connected, scanning for networks, etc). In our evaluation, we assume that the driver has reached a steady state where it has established a stable connection with a network. We consider only the code paths reachable in this state as part of the attack surface. We analyzed this attack surface by counting (i) the number of allocation contexts that create attacker-accessible MMIO and DMA mappings and (ii) the number of driver code paths that are executed while the user is browsing the web.

Table IV summarizes the MMIO and DMA allocation contexts in both device drivers, which create mappings that can be accessed by the attacker while the user is browsing the web. MMIO and DMA coherent mappings were established during the driver initialization, and were still mapped to both the device and the driver by the time the user browses the web; DMA streaming mappings were destroyed after their use, but regularly get recreated and mapped to the device while browsing the web. Thus, an attacker on a compromised Wi-Fi chipset can easily access these mappings, and write malicious values in them to trigger and exploit vulnerabilities in the driver.

We then analyzed the code paths that get exercised under web browsing traffic, and classified these paths based on the context in which they are executed: interrupt context, kernel thread context, and other contexts (e.g., system call context). Table V shows the results. Of all the basic blocks executed under web browsing traffic, 36.9% and 68.9% run in interrupt context for the `qcacld-3.0` and `bcmdhd4358` drivers,

TABLE VI. ALLOCATION CONTEXTS SELECTED FOR FUZZING. DC STANDS FOR DMA COHERENT, DS FOR DMA STREAMING, AND MM FOR MEMORY-MAPPED I/O.

Driver	Alloc. Context	Alloc. Type	Alloc. Size	Used For
qcacld-3.0	**QC1**	DC	8200	DMA buffer mgmt.
	QC2	DC	4	DMA buffer mgmt.
	QC3	DS	2112	FW-Driver message
	QC4	DS	2112	FW-Driver message
bcmdhd4358	**BC1**	DC	8192	FW-Driver RX info
	BC2	DC	16384	FW-Driver TX info
	BC3	DC	1536	FW-Driver ctrl. info
	BC4	MM	4194304	Ring ctrl. info

respectively. Some of the code that executes in interrupt context may not be reachable from any system calls through legal control-flow paths, and therefore may not be fuzzed by system call fuzzers.

C. Target Mappings

We investigated how each of the active mappings are used by their respective drivers, and enabled fuzzing for DMA/MMIO regions that are accessed frequently, and that are used for low-level communication between the driver and the device firmware (e.g., for shared ring buffer management). We used PERISCOPE to determine which regions the driver accesses frequently, and we manually investigated the driver's code to determine the purpose of each region.

For `qcacld-3.0`, we enabled fuzzing for two allocation contexts for DMA coherent buffers and two contexts for DMA streaming buffers. For `bcmdhd4358`, we enabled fuzzing for three allocation contexts for DMA coherent buffers and one allocation context for an MMIO buffer. Table VI summarizes the allocation contexts for which we enable fuzzing; all the mappings allocated in those contexts are fuzzed.

D. Fuzzer Seed Generation

We used PERISCOPE's default tracing facilities to generate initial seed input files. For each selected allocation context, we first recorded all allocations of, and all read accesses to the memory mappings while generating web browsing traffic for five minutes. We then parsed the allocation/access log to generate unique seed input files. Finally, we used AFL's corpus minimization tool to minimize the input files. This tool replays each input file to collect coverage information and uses that information to exclude redundant files.

E. Vulnerabilities Discovered

Table VII summarizes the vulnerabilities we discovered using our fuzzer. Each entry in the table is a unique vulnerability at a distinct source code location.

Disclosure: We responsibly disclosed these vulnerabilities to the respective vendors. During this process, we were informed by Qualcomm that some of the bugs had recently been reported by external researchers or internal auditors. We marked these bugs as "Known". All the remaining bugs were previously unknown, and have been confirmed by the respective vendors. We included CVE numbers assigned to the bugs we reported. Also, we included the vendor-specific, internal severity ratings for these bugs if communicated by the respective vendors during the disclosure process.

Error Type and Impact: Vulnerabilities found by PERIFUZZ fall into four categories: buffer overflows, address leaks, reachable assertions, and null-pointer dereferences. We mark buffer overflows and address leaks as potentially exploitable, and reachable assertions and null-pointer dereferences as vulnerabilities that can cause a denial-of-service (DoS) attack by triggering device reboots.

Double-fetch Bugs: We did not attempt to find double-fetch bugs in streaming DMA buffers, since we operated under the assumption that an IOMMU preventing such bugs is in place (see Section IV-A). That said, we did identify several double-fetch bugs in code that accesses coherent DMA buffers. These bugs can potentially be exploited, even when the system deploys an IOMMU. We discuss these bugs in detail in Section VI-G.

F. Case Study I: Design Bug in `qcacld-3.0`

One of the vulnerabilities we found in `qcacld-3.0` is in code that dereferences a firmware-provided pointer. PERIFUZZ fuzzed the pointer value as it was read by the device driver. The driver then dereferenced the fuzzed pointer and crashed the kernel. An analysis of this vulnerability revealed that it is in fact a design issue. The pointer was originally provided by the driver to the device. Line 11 in Listing 1 turns a kernel virtual address, which points to a kernel memory region allocated at Line 4, into a 64-bit integer called `cookie`. The driver sends this `cookie` value to the device, thereby effectively leaking a kernel address.

```
1  A_STATUS ol_txrx_fw_stats_get(...)
2  {
3    ...
4    non_volatile_req =
     ↪  qdf_mem_malloc(sizeof(*non_volatile_req));
5    if (!non_volatile_req)
6      return A_NO_MEMORY;
7
8    ...
9
10   /* use the non-volatile request object's
     ↪  address as the cookie */
11   cookie =
     ↪  ol_txrx_stats_ptr_to_u64(non_volatile_req);
12
13   ...
14 }
```

Listing 1: Kernel address leak in `qcacld-3.0`

An attacker that controls the peripheral processor can infer the kernel memory layout based on the cookie values passed by the driver. This address leak can facilitate exploitation of memory corruption vulnerabilities even if the kernel uses randomization-based mitigations such as KASLR. This bug can be fixed by passing a randomly generated cookie value rather than a pointer to the device.

G. Case Study II: Double-fetch Bugs in `bcmdhd4358`

The `bcmdhd4358` driver contains several double-fetch bugs that allow an adversarial Wi-Fi chip to bypass an integrity check in the driver. Listing 2 shows how the driver accesses

TABLE VII. UNIQUE DEVICE DRIVER VULNERABILITIES FOUND BY PERIFUZZ

Alloc. Context	Alloc. Type	Error Type	Analysis	Double-fetch	Status (Severity)	Impact
QC2	DC	Buffer Overflow	Unexpected RX queue index		CVE-2018-11902 (High)	Likely Exploitable
QC3	DS	Null-pointer Deref.	Unexpected message type		Confirmed (Low)[a]	DoS
QC3	DS	Buffer Overflow	Unexpected peer id		Known	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected number of flows		Known	Likely Exploitable
QC3	DS	Address Leak/Buffer Ovf.	Unexpected FW-provided pointer		CVE-2018-11947 (Med)[b]	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected TX descriptor id		Known	Likely Exploitable
QC4	DS	Reachable Assertion	Unexpected endpoint id		Known (Med)	DoS
QC4	DS	Reachable Assertion	Duplicate message		Known (Med)	DoS
QC4	DS	Reachable Assertion	Unexpected payload length		Known (Med)	DoS
BC1	DC	Buffer Overflow	Unexpected interface id	✓	CVE-2018-14852, SVE-2018-11784 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in create rsp.	✓	CVE-2018-14856, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in delete rsp.	✓	CVE-2018-14854, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in flush rsp.	✓	CVE-2018-14855, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Null-pointer Deref.	Uninitialized flow ring state		CVE-2018-14853, SVE-2018-11783 (Low)	DoS
BC4	MM	Buffer Overflow	Unexpected flow ring pointer		CVE-2018-14745, SVE-2018-12029 (Low)	Likely Exploitable

[a]Qualcomm confirmed the vulnerability but they do not assign CVEs for low-severity ones.
[b]CVE assigned for the address leak.

a coherent DMA buffer that holds meta-information about network data. At Line 4 and Line 5, the driver verifies the integrity of the data in the buffer by calculating and checking an XOR checksum. The driver then repeatedly accesses this coherent DMA buffer again. The problem here is that the device, if compromised, could modify the data between the point of the initial integrity check, and the subsequent accesses by the driver.

```
1  static uint8 BCMFASTPATH
   ↪  dhd_prot_d2h_sync_xorcsum(dhd_pub_t *dhd,
   ↪  msgbuf_ring_t *ring, volatile cmn_msg_hdr_t
   ↪  *msg, int msglen)
2  {
3    ...
4    prot_checksum = bcm_compute_xor32((volatile
     ↪  uint32 *)msg, num_words);
5    if (prot_checksum == 0U) { /* checksum is OK */
6      if (msg->epoch == ring_seqnum) {
7        ring->seqnum++; /* next expected sequence
         ↪  number */
8        goto dma_completed;
9      }
10   }
11   ...
12 }
```

Listing 2: Initial fetch and integrity check in `bcmdhd4358`

PERIFUZZ was able to trigger multiple vulnerabilities by modifying the data read from this buffer *after* the integrity check was completed. We show one buffer overflow vulnerability in Listing 3, which was triggered by fuzzing the `ifidx` value used at Line 4. The overlapping fetch that occurred before this buffer overflow is a double-fetch bug, because the overlapping fetch can invalidate a previously passed buffer integrity check. Thus, in addition to safeguarding the array access with a bounds check, the driver should copy the contents of the coherent DMA buffers to a location that cannot be accessed by the peripheral device, before checking the integrity of the data in the buffer. Subsequent uses of device-provided data should also read from the copy of the data, rather than the DMA buffer itself.

```
1  void dhd_rx_frame(dhd_pub_t *dhdp, int ifidx,
   ↪  void *pktbuf, int numpkt, uint8 chan)
2  {
3    ...
4    ifp = dhd->iflist[ifidx];
5    if (ifp == NULL) {
6      DHD_ERROR(("%s: ifp is NULL. drop packet\n",
7                 __FUNCTION__));
8      PKTFREE(dhdp->osh, pktbuf, FALSE);
9      continue;
10   }
11   ...
12 }
```

Listing 3: Buffer overflow in `bcmdhd4358`

H. Case Study III: New Bug in `qcacld-3.0`

Listing 4 shows a null-pointer deference bug we discovered in the `qcacld-3.0` driver. The pointer to the `netbufs_ring` array dereferenced at Line 9 is null, unless the driver is configured to explicitly allocate this array. The driver configuration used by the Google Pixel 2 did not contain the entry necessary to allocate the array. Although the driver never executes the vulnerable code under normal conditions, we found that the vulnerable line *is* reachable through legal control flow paths.

```
1  static inline qdf_nbuf_t
   ↪  htt_rx_netbuf_pop(htt_pdev_handle pdev)
2  {
3    int idx;
4    qdf_nbuf_t msdu;
5
6    HTT_ASSERT1(htt_rx_ring_elems(pdev) != 0);
7
8    idx = pdev->rx_ring.sw_rd_idx.msdu_payld;
9    msdu = pdev->rx_ring.buf.netbufs_ring[idx];
10   ...
11 }
```

Listing 4: Null-pointer dereference in `qcacld-3.0`

It is difficult to detect this bug statically, as it requires a whole-program analysis of the device driver to determine if the `netbufs_ring` pointer is initialized whenever the vulnerable line can execute. PERIFUZZ consistently triggered the bug,

TABLE VIII. TIME CONSUMED BY PERISCOPE'S PAGE FAULT HANDLER (MEASURED IN µ SECONDS)

	Mean	Minimum	Maximum
Tracing Only	117.6	99.8	194.5
Tracing + Fuzzing	227.8	182.7	379.7

however. This vulnerability discovery therefore bolsters the argument that fuzzing can complement manual auditing and static analysis.

I. Performance Analysis

1) Page Fault: PERISCOPE incurs run-time overhead as it triggers a page fault for every instruction that accesses the monitored set of DMA/MMIO regions. We quantified this overhead by measuring the number of clock cycles spent inside PERISCOPE's page fault handler. We read the AArch64 counter-timer virtual count register `CNTVCT_EL0` when entering the handler and when exiting from the handler, and calculated the difference between the counter values, divided by the counter-timer frequency counter `CNTFRQ_EL0`. To minimize interference, we disabled hardware interrupts while executing our page fault handler. We also disabled dynamic frequency and voltage scaling.

We tested the page fault handler under two configurations. In one configuration, PERISCOPE calls the default pre- and post-instruction hooks that only trace and log memory accesses. In the other configuration, we registered PERIFUZZ's instruction hooks to enable DMA/MMIO fuzzing. Table VIII shows the mean, minimum, and maximum values over samples of 500 page fault handler invocations for each configuration.

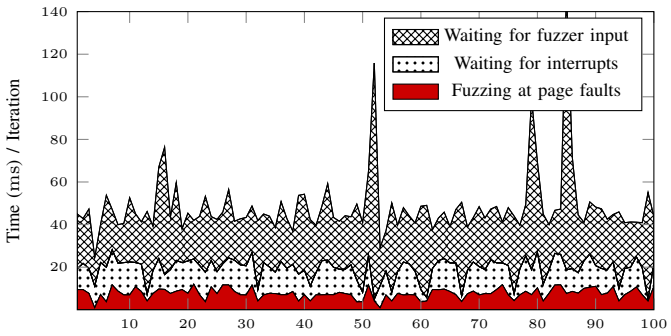
Note that we deliberately trade performance for deterministic, precise monitoring of device-driver interactions, by trapping every single access to a set of monitored mappings. In fact, this design allowed us to temporally distinguish accesses to the same memory locations, which was essential to find the double-fetch bugs. The drivers still function correctly, albeit more slowly, when executed under our system, making it possible to examine device-driver interactions dynamically and enabling PERIFUZZ to fuzz it.

2) Fuzzing: PERIFUZZ builds on PERISCOPE and has additional components that interact with each other, which incur additional costs. The primary contributors to this additional cost are: (i) waiting for the peripheral to signal the driver, (ii) waiting for a software interrupt to be scheduled by the Linux scheduling subsystem, (iii) interactions with the user-space fuzzer, which involve at least two user-kernel mode switches (i.e., one for delivering fuzzer inputs and the other for polling and retrieving feedback), and (iv) other system activities.

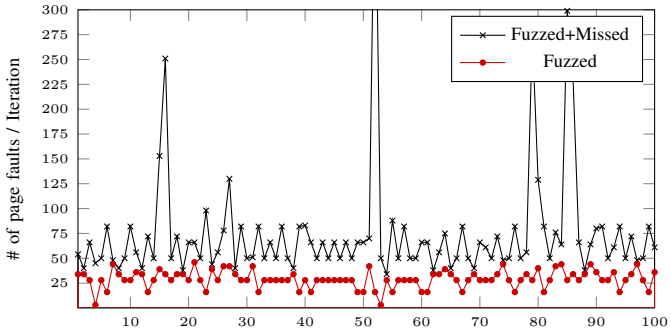
Peak Throughput: We measured the overall fuzzing throughput to quantify the overhead incorporating all interactions between the PERIFUZZ components. We only report the peak throughput in Table IX, since crashes and device driver lockups heavily impact the average fuzzing throughput (see Section VII-A). The inverse of the peak fuzzing throughput is a conservative lower bound for the execution time required to process a single fuzzer-generated input. Although we did not optimize PERIFUZZ for throughput, we believe that these numbers are still in a range that makes PERIFUZZ practical for dynamic analysis.

TABLE IX. PEAK FUZZING THROUGHPUT FOR EACH FUZZED ALLOCATION CONTEXT

Driver	Alloc. Context	Peak Throughput (# of test inputs/sec)
`qcacld-3.0`	**QC1**	23.67
	QC2	15.64
	QC3	18.77
	QC4	7.63
`bcmdhd4358`	**BC1**	9.90
	BC2	14.28
	BC3	10.49
	BC4	15.92



(a) The execution time of three phases



(b) The number of fuzzed and missed page faults

Fig. 5. Fuzzing overhead breakdown

Overhead Breakdown: To illustrate how the fuzzing throughput can be optimized, we present a breakdown of the fuzzing overhead. We divide each iteration of the fuzzing loop into three phases: (i) waiting for fuzzer input to be made available to our kernel module, (ii) waiting for the device to raise an interrupt and for the driver to start processing it, and (iii) fuzzing the data read from monitored I/O mappings upon page faults. Once the driver has finished processing the interrupt, the next iteration begins. We measured the execution time of each phase in each iteration. To evaluate the impact of page faults on the fuzzing performance, we also counted the number of page faults triggered during each iteration.

We performed the experiment while fuzzing the buffer having the highest peak throughput (**QC1**). Figure 5a shows our measurements of per-phase execution time in a stacked manner, over 100 consecutive iterations of the fuzzing loop. 60% of the total execution time is spent on waiting for the next fuzzer input to be available. This delay is primarily caused by a large number of missed page faults, as hinted by Figure 5b.

The current implementation of PERIFUZZ can miss page faults, when they are triggered while PERIFUZZ is preparing for the next input. This delay can be reduced by disabling page faults until the next input is ready. The delay caused by waiting for relevant interrupts, which accounts for 24.2% of the total execution time, can be reduced by forcing hardware to raise relevant interrupts more frequently.

The actual fuzzing at each page fault still takes 15.8% of the total execution time. One way to reduce this overhead is to trigger page faults only at first access to a monitored mapping within each iteration. At first access, the underlying page can be overwritten with the fuzzer input and then made present, so that subsequent accesses to the page within the same iteration do not trigger extra page faults. This would come, however, at the cost of precision, because it loses precise access tracing capability, effectively disabling overlapping fetch fuzzing as well as detection of potential double-fetch bugs.

VII. DISCUSSION

A. Limitations

We discuss problems that limit both the effectiveness and efficiency of PERIFUZZ. These are well-known problems that also affect other kernel fuzzers, such as system call fuzzers.

1) System Crashes: The OS typically terminates user-space programs when they crash, and they can be restarted without much delay. Crashing a user-space program therefore has little impact on the throughput of fuzzing user-space programs. Crashes in kernel space, by contrast, cause a system reboot, which significantly lowers the throughput of any kernel fuzzer. This is particularly problematic if the fuzzer repeatedly hits shallow bugs, thereby choking the system without making meaningful progress. We circumvented this problem by disabling certain code paths that contain previously discovered shallow bugs. This does, however, somewhat reduce the effectiveness of our fuzzer as it cannot traverse the subpaths rooted at these blacklisted bugs. Note that this problem also affects other kernel fuzzers, e.g., DIFUZE and Syzkaller [35], [75].

2) Driver Internal States: Due to the significant latency involved in system restarts, whole-system fuzzers typically fuzz the system without restarting it between fuzzing iterations. This can limit the effectiveness of such fuzzers, because the internal states of the target system persist across iterations. Changing internal states can also lead to instability in the coverage-guidance, as the same input can exercise different code paths depending on the system state. This means that coverage-guidance may not be fully effective. Worse, when changes to the persisting states accumulate, the device driver may eventually lock itself up. For example, we encountered a problem where, after feeding a certain number of invalid inputs to a driver, the driver decided to disconnect from the network, reaching an error state from which the driver could not recover without a device reboot. Existing device driver checkpointing and recovery mechanisms could be adapted to alleviate the problem [46], [70], because they provide mechanisms to roll drivers back to an earlier state. Such a roll back takes significantly less time than a full system reboot.

B. Augmenting the Fuzzing Engine

Although we used mutational, feedback-guided fuzzing to mutate the data stream on the device-driver interaction path, our fuzzing framework can also benefit from other fuzzing techniques. Like DIFUZE [35], static analysis can be introduced to infer the type of an I/O buffer, which can save fuzzing cycles by respecting the target type when mutating a value. The dependencies between device-driver interaction messages can also be inferred using static and trace analysis techniques [41], [58], which can help fuzzing stateful device-driver interaction protocols. Alternatively, developers can specify the format of an I/O buffer and/or interaction protocol in a domain-specific language [10], [75]. In addition to improving the mutation of the data stream, we could use system call fuzzers such as Syzkaller that generate different user-space programs [75]. These generated programs could actively send requests to the driver and potentially to the device, which in turn can increase reachable interrupt code paths. We believe that our modular framework allows for easy integration of these techniques.

C. Combining with Dynamic Analysis

Our framework runs in a concrete execution environment; thus, existing dynamic analysis tools can be used to uncover silent bugs. For example, kernel sanitizers such as address sanitizer and undefined behavior sanitizer can complement our fuzzer [48], [63]. Memory safety bugs often silently corrupt memory without crashing the kernel. Our fuzzer, by itself, would not be able to reveal such bugs. When combined with a sanitizer, however, these bugs *would* be detected. Other dynamic analysis techniques such as dynamic taint tracking can also be adapted to detect security-critical semantic bugs such as passing security-sensitive values (e.g., kernel virtual addresses) to untrusted peripherals.

VIII. RELATED WORK

A. Protection against Peripheral Attacks

An IOMMU isolates peripherals from the main processor by limiting access to physical memory to regions configured by the OS. Markuze et al. proposed mechanisms that can achieve strong IOMMU protection at an affordable performance cost [52], [53]. Several other work proposed mechanisms that can limit functionalities exposed to potentially malicious devices [15], [72], [73]. Cinch encapsulates devices as network endpoints [15], and USBFILTER hooks USB APIs [73], to enable user-configurable, fine-grained access control. However, neither IOMMU protection nor fine-grained access control prevents exploitation of vulnerabilities found in code paths that are still reachable from the device.

The effects of vulnerabilities on these valid code paths can be mitigated by isolating device drivers from the kernel [27], [34], [38], [50]. Android, for example, switched from the kernel-space Bluetooth protocol stack [12] to a user-space Bluetooth stack [13]. The OS kernel merely acts as a data path by forwarding incoming packets to the user-space Bluetooth daemon process. This approach can mitigate vulnerabilities in the device driver because the driver cannot access kernel memory and cannot execute privileged instructions. The daemon process still runs at a higher privilege level than standard user-space processes, however, and therefore remains an attractive

target for adversaries looking to access sensitive data [11]. Additionally, this approach is currently not viable for certain types of device drivers. High-bandwidth communication devices such as Wi-Fi chips, for example, cannot afford the mode and context switching overhead incurred by user-space drivers.

B. Kernel Fuzzing

Most kernel fuzzing tools focus on the system call boundary [9], [14], [19], [35], [41], [43], [58], [59], [65], [75]. DIFUZE uses static analysis and performs type-aware fuzzing of the IOCTL interface, which can expose a substantial amount of driver functionality to user space [35]. Syzkaller, a coverage-guided fuzzer, fuzzes a broader set of system calls, based on system call description written in a domain-specific language [75]. IMF infers value-dependence and order-dependence between system call arguments by analyzing system call traces [41]. kAFL uses Intel Processor Trace as a feedback mechanism, to enable OS-independent fuzzing [65]. Digtool uses virtualization to capture and analyze the dynamic behavior of kernel execution [59].

PERIFUZZ can be augmented with techniques that facilitate type-aware fuzzing [35], [41], [58], [75], as discussed in Section VII-B. Tools based on certain hardware features can fuzz closed-source OSes [59], [65], but smartphones often do not contain or expose the necessary hardware features to the end user. For example, most smartphone OSes block access to the bootloader and to hypervisor mode, thus preventing end users from running code at the highest privilege level [61]. None of these fuzzers target DMA/MMIO-based interactions between drivers and devices, nor do they cover code paths that are not reachable from system calls (e.g., interrupt handlers).

C. Kernel Tracing

There are many general-purpose tools to monitor events in the Linux kernel. Static kernel instrumentation mechanisms such as Tracepoint allow the developer to insert so-called probes [5]. Ftrace and Kprobe are dynamic mechanisms that can be used to probe functions or individual instructions [6], [7]. eBPF, the extended version of the Berkeley Packet Filter mechanism, can attach itself to existing Kprobe and Tracepoint probes for further processing [40]. LTTng, SystemTap, Ktap and Dprobe are higher level primitives that build on the aforementioned tools [2]–[4], [55].

These tools, however, are not well suited to monitoring device-driver interactions, because they require developers to identify and instrument each device-driver interaction. These manual efforts can be alleviated by using page fault based monitoring, which Mmiotrace uses to trace MMIO-based interactions in x86 and x86-64 [8]. However, Mmiotrace does not support the DMA interface, i.e., DMA coherent and streaming buffers, and it lacks the ability to manipulate device-driver interactions. In contrast, PERISCOPE can trace both MMIO and DMA interfaces, and can be used to manipulate device-driver interactions by plugging in PERIFUZZ, enabling adversarial analysis of device drivers.

D. Kernel Static Analysis

Static analysis tools can detect various types of kernel and driver vulnerabilities [18], [39], [51], [76], [79]. Dr. Checker runs pointer and taint analyses specifically tailored to device drivers, and feeds the analysis results to various vulnerability detectors [51]. K-Miner uses an inter-procedural, context-sensitive pointer analysis to find memory corruption vulnerabilities reachable from system calls [39]. Symbolic execution can complement these static analyses to work around precision issues. Deadline [79], for example, uses static analysis to find multi-reads in the kernel, and symbolically checks whether each multi-read satisfies the constraints to be a double-fetch bug. With the help of this symbolic checking, Deadline can precisely discern double-fetch bugs from statically identified multi-reads. Generally speaking, however, techniques based on symbolic execution may not scale well due to the path explosion problem.

Static analysis techniques have traditionally been applied to the system call interface only. Although the core ideas can apply to the hardware-OS interface too, statically identifying the necessary entry points may not be as trivial as with system calls, since accesses to an I/O mapping are difficult to distinguish from other memory accesses, and interrupt processing code can run in different, unrelated contexts (e.g., software interrupt context, kernel thread context, etc.).

E. Finding Double-fetch Bugs

Double-fetch bugs are a special case of time-of-check-to-time-of-use (TOCTTOU) race conditions. They occur when privileged code fetches a value from a memory location multiple times, while less privileged code is able to change the value between the fetches [45], [68]. Previous work explored multiple reads of user-space memory from OS kernels or from trusted execution environments [45], [59], [67], [76], [79], and multiple reads of memory shared between different hypervisor domains [78]. They either use static analysis (e.g., static code pattern matching [76] and symbolic execution [79]), or dynamic analysis (e.g., memory access tracing followed by pattern analysis [45], [59], [78] and cache behavior-guided fuzzing [67]). PERIFUZZ is also a dynamic approach, but targets a different attack surface: I/O memory mappings shared between peripheral devices and kernel drivers.

PERIFUZZ and DECAF are currently the only two tools that are sufficiently generic to support double-fetch fuzzing without instrumentation or manual analysis of the target code [67]. DECAF cannot fuzz double-fetches from MMIO and DMA coherent mappings, however, because these mappings are typically uncached, and DECAF relies on cache side channels to detect double-fetches.

IX. CONCLUSION

The interactions between peripherals and drivers can be complex, and hence writing correct device driver software is hard. Unfortunately, as has been recently demonstrated, vulnerabilities in wireless communication peripherals and corresponding drivers can be exploited to achieve remote kernel code execution without invoking a single system call. Nonetheless, no versatile framework has existed until now that analyzes the interactions between peripherals and drivers.

This paper presents PERISCOPE, a generic probing framework that addresses the specific analysis needs of the two peripheral interface mechanisms MMIO and DMA. Our fuzzing

component PERIFUZZ builds upon this framework and can help the end user find bugs in device drivers reachable from a compromised device; uniquely, PERIFUZZ can expose double-fetch bugs by fuzzing overlapping fetches, and by warning about overlapping fetches that occurred before a driver crash. Using these tools, we found 15 unique vulnerabilities in the Wi-Fi drivers of two flagship Android smartphones, including 9 previously unknown ones.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback. The authors also thank Paul Kirth and Joseph Nash for their help with proofreading this paper. This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research under contract N00014-15-1-2948, N00014-17-1-2011, and N00014-17-1-2782, and by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government. The authors also gratefully acknowledge a gift from Oracle Corporation, and from Google's Anti-Abuse group.

REFERENCES

[1] "Facedancer11." [Online]. Available: http://goodfet.sourceforge.net/hardware/facedancer11

[2] "ktap: A lightweight script-based dynamic tracing tool for Linux." [Online]. Available: https://github.com/ktap/ktap

[3] "LTTng." [Online]. Available: https://lttng.org

[4] "SystemTap." [Online]. Available: https://sourceware.org/systemtap

[5] "Using the Linux kernel Tracepoints." [Online]. Available: https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[6] "Kernel probes (Kprobes)," 2004. [Online]. Available: https://www.kernel.org/doc/Documentation/kprobes.txt

[7] "ftrace - function tracer," 2008. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[8] "Memory mapped I/O trace," 2014. [Online]. Available: https://nouveau.freedesktop.org/wiki/MmioTrace

[9] "Project Triforce: Run AFL on everything!" 2016. [Online]. Available: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything

[10] "ProtoFuzz: A protobuf fuzzer," 2016. [Online]. Available: https://blog.trailofbits.com/2016/05/18/protofuzz-a-protobuf-fuzzer

[11] "BlueBorne vulnerabilities," 2017. [Online]. Available: https://armis.com/blueborne

[12] "BlueZ: Official Linux Bluetooth protocol stack," 2018. [Online]. Available: http://www.bluez.org

[13] "Fluoride Bluetooth stack," 2018. [Online]. Available: https://android.googlesource.com/platform/system/bt

[14] "Trinity: Linux system call fuzzer," 2018. [Online]. Available: https://github.com/kernelslacker/trinity

[15] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, "Defending against malicious peripherals with Cinch," in *Proceedings of the USENIX Security Symposium*, 2016.

[16] N. Artenstein, "BroadPwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets," *Black Hat USA*, 2017.

[17] D. Aumaitre and C. Devine, "Subverting Windows 7 x64 kernel with DMA attacks," *HITBSecConf Amsterdam*, 2010.

[18] 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 ACM European Conference on Computer Systems (EuroSys)*, 2006.

[19] I. Beer, "pwn4fun spring 2014 - Safari - part II," 2014. [Online]. Available: https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html

[20] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[21] G. Beniamini, "Over the air - vol. 2, pt. 2: Exploiting the Wi-Fi stack on Apple devices," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-2-exploiting-wi-fi.html

[22] ——, "Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html

[23] ——, "Over the air: Exploiting Broadcom's Wi-Fi stack (part 1)," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[24] ——, "Over the air: Exploiting Broadcom's Wi-Fi stack (part 2)," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html

[25] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[26] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[27] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in Linux," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[28] A. Cama, "A walk with Shannon: A walkthrough of a pwn2own baseband exploit," *OPCDE Kenya*, 2018.

[29] Q. Casasnovas, "[patch] kcov: add AFL-style tracing," 2016. [Online]. Available: https://lkml.org/lkml/2016/5/21/58

[30] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[31] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[32] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with RevNIC," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2010.

[33] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[34] P. Chubb, "Linux kernel infrastructure for user-level device drivers," in *Linux Conference*, 2004.

[35] 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 ACM Conference on Computer and Communications Security (CCS)*, 2017.

[36] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain, "Can you still trust your network card?" *CanSecWest*, 2010.

[37] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[38] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of microdrivers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[39] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-Miner: Uncovering memory corruption in Linux," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[40] B. Gregg, "Linux extended BPF (eBPF) tracing tools," 2018. [Online]. Available: http://www.brendangregg.com/ebpf.html

[41] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[42] J. Hertz and T. Newsham, "A Linux system call fuzzer using TriforceAFL," 2016. [Online]. Available: https://github.com/nccgroup/TriforceLinuxSyscallFuzzer

[43] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019, To appear.

[44] M. Jodeit and M. Johns, "USB device drivers: A stepping stone into your kernel," in *Proceedings of the European Conference on Computer Network Defense (EC2ND)*, 2010.

[45] M. Jurczyk and G. Coldwind, "Identifying and exploiting Windows kernel race conditions via memory access patterns," 2013.

[46] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

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

[48] A. Konovalov and D. Vyukov, "KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel," *LinuxCon North America*, 2015.

[49] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[50] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser, "User-level device drivers: Achieved performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654–664, 2005.

[51] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Checker: A soundy analysis for Linux kernel drivers," in *Proceedings of the USENIX Security Symposium*, 2017.

[52] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[53] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir, "DAMN: Overhead-free IOMMU protection for networking," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[54] M. Mendonça and N. Neves, "Fuzzing Wi-Fi drivers to locate security vulnerabilities," in *Proceedings of the European Dependable Computing Conference (EDCC)*, 2008.

[55] R. J. Moore, "A universal dynamic trace for Linux and other operating systems," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.

[56] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[57] V. Nossum and Q. Casasnovas, "Filesystem fuzzing with american fuzzy lop," *Vault*, 2016.

[58] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the USENIX Security Symposium*, 2018.

[59] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2017.

[60] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "POTUS: probing off-the-shelf USB drivers with symbolic fault injection," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[61] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "BootStomp: On the security of bootloaders in mobile devices," in *Proceedings of the USENIX Security Symposium*, 2017.

[62] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[63] A. Ryabinin, "UBSan: run-time undefined behavior sanity checker," 2014. [Online]. Available: https://lwn.net/Articles/617364

[64] M. Schulz, D. Wegemer, and M. Hollick, "The Nexmon firmware analysis and modification framework: Empowering researchers to enhance Wi-Fi devices," *Computer Communications*, 2018.

[65] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the USENIX Security Symposium*, 2017.

[66] S. Schumilo, R. Spenneberg, and H. Schwartke, "Don't trust your USB! how to find bugs in USB device drivers," *Black Hat Europe*, 2014.

[67] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[68] F. J. Serna, "MS08-061 : The case of the kernel mode double-fetch," 2008. [Online]. Available: https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch

[69] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[70] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[71] 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 *Proceedings of the USENIX Security Symposium*, 2018.

[72] D. J. Tian, A. Bates, and K. Butler, "Defending against malicious USB firmware with GoodUSB," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[73] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor, "Making USB great again with USBFILTER," in *Proceedings of the USENIX Security Symposium*, 2016.

[74] D. Vyukov, "kernel: add kcov code coverage," 2016. [Online]. Available: https://lwn.net/Articles/671640

[75] ——, "syzkaller - kernel fuzzer," 2018. [Online]. Available: https://github.com/google/syzkaller

[76] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel," in *Proceedings of the USENIX Security Symposium*, 2017.

[77] R.-P. Weinmann, "All your baseband are belong to us," *DeepSec*, 2010.

[78] F. Wilhelm, "Xenpwn: Breaking paravirtualized devices," *Black Hat USA*, 2016.

[79] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[80] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[81] M. Zalewski, "American fuzzy lop," 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl