Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems

Seyed Mohammadjavad Seyed Talebi and Hamid Tavakoli, *UC Irvine;*
Hang Zhang and Zheng Zhang, *UC Riverside;*
Ardalan Amiri Sani, *UC Irvine;* Zhiyun Qian, *UC Riverside*

https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

This paper is included in the Proceedings of the
27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems

Seyed Mohammadjavad Seyed Talebi*, Hamid Tavakoli*, Hang Zhang†, Zheng Zhang†,
Ardalan Amiri Sani*, Zhiyun Qian†
**UC Irvine,* †*UC Riverside*

Abstract

Mobile systems, such as smartphones and tablets, incorporate a diverse set of I/O devices, such as camera, audio devices, GPU, and sensors. This in turn results in a large number of diverse and customized device drivers running in the operating system kernel of mobile systems. These device drivers contain various bugs and vulnerabilities, making them a top target for kernel exploits [78]. Unfortunately, security analysts face important challenges in analyzing these device drivers in order to find, understand, and patch vulnerabilities. More specifically, using the state-of-the-art dynamic analysis techniques such as interactive debugging, fuzzing, and record-and-replay for analysis of these drivers is difficult, inefficient, or even completely inaccessible depending on the analysis.

In this paper, we present Charm[1], a system solution that facilitates dynamic analysis of device drivers of mobile systems. Charm's key technique is *remote device driver execution*, which enables the device driver to execute in a virtual machine on a workstation. Charm makes this possible by using the actual mobile system only for servicing the low-level and infrequent I/O operations through a low-latency and customized USB channel. Charm does not require any specialized hardware and is immediately available to analysts. We show that it is feasible to apply Charm to various device drivers, including camera, audio, GPU, and IMU sensor drivers, in different mobile systems, including LG Nexus 5X, Huawei Nexus 6P, and Samsung Galaxy S7. In an extensive evaluation, we show that Charm enhances the usability of fuzzing of device drivers, enables record-and-replay of driver's execution, and facilitates detailed vulnerability analysis. Altogether, these capabilities have enabled us to find 25 bugs in device drivers, analyze 3 existing ones, and even build an arbitrary-code-execution kernel exploit using one of them.

1 Introduction

Today, mobile systems, such as smartphones and tablets, incorporate a diverse set of I/O devices, e.g., camera, display, sensors, accelerators such as GPU, and various network devices. These I/O devices are the main driving force for product differentiation in a competitive market. It is reported that there are more than a thousand Android device manufacturers and more than 24,000 distinct Android devices seen just in 2015 [1]. Therefore, one smartphone vendor might use a powerful camera so that its smartphone would stand out in this market, while another might be the first to incorporate a fingerprint scanner.

Such diversity has an important implication for the operating system of mobile systems: *a large number of highly diverse and customized device drivers are required to power the corresponding set of distinct I/O devices.* Device drivers run in the kernel of the operating system and are known to be the source of many serious vulnerabilities such as root vulnerabilities [78]. Therefore, security analysts invest significant effort to find, analyze, and patch the vulnerabilities in them. Unfortunately, they face important deficiencies in doing so. More specifically, performing dynamic analysis on device drivers in mobile systems is difficult, inefficient, or even impossible depending on the analysis. For example, some dynamic analyses, including introspecting the driver and kernel state with a debugger (such as GDB) and record-and-replay, requires the driver to run within a controlled environment, e.g., a virtual machine. Unfortunately, doing so for device drivers running in the kernel of mobile systems is impossible. As another example, a kernel fuzzer, such as kAFL [65] or Google Syzkaller [7], can be used to find various types of bugs in the operating system kernel including device drivers. Unfortunately, fuzzing the device drivers in mobile systems encounters various disadvantages. First, using kAFL requires running the driver in an x86-based virtual machine, which is not possible for mobile drivers.

[1] Charm is open sourced: https://trusslab.github.io/charm/

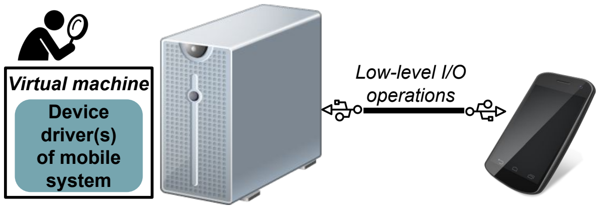


Figure 1: *Charm enables a security analyst to run a mobile I/O device driver in a virtual machine and inspect it using various dynamic analysis techniques.*

Second, using Syzkaller directly on mobile systems is challenging due to (*i*) lack of support for latest fuzzing features, such as new kernel sanitizers [9–12] and (*ii*) lack of access to the system's console without using a specialized adapter [8].

In this paper, we present Charm, a system designed to facilitate dynamic analysis of device drivers of mobile systems in order to find and investigate the vulnerabilities in them. Our key contribution in Charm that makes this possible is a system solution for the execution of mobile I/O device drivers within a virtual machine on a different physical machine, e.g., a workstation. Such a capability overcomes the aforementioned deficiencies. That is, since the device driver executes within a virtual machine, it enables the analyst to use various dynamic analyses including manual interactive debugging, record-and-replay, and enhanced fuzzing.

Executing a mobile system's device driver within a workstation virtual machine is normally impossible since the driver requires access to the exact hardware of the I/O device in the mobile system. We solve this problem using a technique called *remote device driver execution*. With this technique, the device driver's attempts to interact with its I/O device are intercepted in the virtual machine by the hypervisor and routed to the actual mobile system over a customized low-latency USB channel. In this technique, while the actual mobile system is needed for the execution of the infrequent low-level I/O operations, the device driver runs fully within a virtual machine and hence can be analyzed. Figure 1 shows the high-level idea behind Charm.

Remote device driver execution raises two important challenges, which we address in this paper. First, interactions of a device driver with its corresponding I/O device are time-sensitive. Hence the added latency of communications between the workstation and mobile system can easily result in various time-out problems in the I/O device or driver, as our own experience with our earlier Charm prototypes demonstrated. We address this challenge with a customized USB channel. Quite importantly, *our solution does not require any customized hardware* for the connection to the mobile system. It leverages the commonly available USB interface and hence makes our solution immediately available to security analysts.

Second, in addition to interacting with the I/O device's hardware, a device driver interacts with several other modules in the operating system kernel including a bus driver, the power management module, and the clock management module. These modules, which we refer to as "resident modules", cannot be moved to the virtual machine since they are needed in the mobile system for the usage of the USB channel. We address this challenge with a Remote Procedure Call (RPC) interface for the remote driver to interact with these modules in the mobile system. We build our RPC solution at the boundary of common Linux APIs. Therefore, different device drivers of different mobile systems can use the same RPC interface, reducing the engineering effort to apply Charm to new device drivers.

We implement Charm's prototype using an Intel Xeon-based workstation and three smartphones: LG Nexus 5X, Huawei Nexus 6P, and Samsung Galaxy S7. We implement remote device driver execution for two device drivers in Nexus 5X, namely the camera and audio drivers, for the GPU device driver in Nexus 6P, and for Inertial Measurement Unit (IMU) sensor driver in Samsung Galaxy S7. Altogether, these drivers encompass 129,000 LoC. We choose four distinct device driver from three vendors to demonstrate the ability of Charm to support a diverse set of device drivers in various mobile systems. We have released the source code of Charm as well as the kernel images configured for the supported drivers. The former enables security analysts to support new device drivers, while the latter enables them to immediately apply different dynamic analysis techniques to the set of device drivers that Charm already supports.

Our current prototype of Charm only supports open source device drivers. Fortunately, kernel source code (including drivers) is often available for Android devices. In practice, the kernel is often released by vendors soon after launch, e.g., in the case of Samsung Galaxy S9 and S9+ [19]. Moreover, kernels released by the vendors are integrated into custom Android projects (such as LineageOS, which supports 200 devices at the time of this writing [18]), providing bootable Android images. These projects also provide instructions to unlock the bootloader on supported devices in order to deploy these images. Therefore, we believe that Charm is useful for many (if not most) Android devices. However, there are still a large number of closed source device drivers, which Charm cannot currently support. Therefore, as part of our future work, we plan to support closed source drivers in Charm too (§8).

Using extensive evaluation, we demonstrate the following. First, we show that it is feasible to add support for new device drivers in Charm in a reasonable amount of time. Second, we show that despite the overhead of remote device driver execution, Charm's performance is on par with actual mobile systems. More specifically, we show that a fuzzer can execute about the same number of fuzzing programs in Charm and hence achieve similar code coverage in the driver. Third, we show that Charm enables us to find 25 bugs in drivers including 14 previously unknown bugs (several of which we have already reported) and two bugs detected by a kernel sanitizer not available on the corresponding mobile system's kernel. Fourth, we show that we can record and replay the execution of the device driver, which, among others, can help easily recreate a bug without needing the mobile system's hardware. Finally, we show that it is feasible to use a debugger, i.e., GDB, to analyze various vulnerabilities in these drivers. Using this ability, we have analyzed three publicly reported vulnerabilities and managed to build an arbitrary-code-execution kernel exploit using one of them.

2 Motivation

Our efforts to build Charm is motivated by our previous struggles to analyze the device drivers of mobile systems in order to find and understand vulnerabilities in them. In this section, we discuss three important dynamic analysis techniques: manual interactive debugging, record-and-replay, and fuzzing. We discuss the current challenges in applying them to device drivers of mobile systems and briefly mention how Charm overcomes these challenges.

2.1 Manual Interactive Debugging

Security analysts often use a debugger, such as the infamous GDB, to analyze a vulnerability or a reported exploit. A debugger enables the analyst to put breakpoints in the code, investigate the content of memory when and where needed, and put watchpoints on important data structures to detect attempts to modify them. Unfortunately, performing these debugging actions on device drivers is typically infeasible as they run in the kernel of the mobile system's operating system. Kernel debugger, KGDB, tries to address this challenge by providing support for interactive debugging for the operating system kernel. However, using KGDB for the kernel of mobile systems is either infeasible, is difficult to use, or requires a specialized adapter. More specifically, KGDB requires console access, which can be made available through the UART hardware. Unfortunately, some mobile systems do not have the UART hardware, and hence do not support KGDB. Moreover, some other systems, e.g., some Xperia smartphones, have the UART hardware, but accessing it requires opening up the system, finding the UART pins, and soldering connections [14], which is a difficult and error-prone task. Finally, some systems have the UART hardware and connect it to the audio jack for easy access, e.g., Nexus devices [20]. Console access in this case is relatively easier but still requires a specialized adapter cable [15].

Charm solves this problem. It enables the security analysts to analyze the device driver since the driver runs within a virtual machine. To demonstrate this point, we have used GDB to analyze 3 vulnerabilities in Nexus 5X camera driver (reported on Android Security Bulletins [2]). Moreover, we have also used GDB to help construct an exploit that can gain arbitrary code execution in the kernel using one of these vulnerabilities.

2.2 Record-and-Replay

Record-and-replay is an invaluable tool for analyzing the behavior of a program, including device drivers. It enables an analyst to record the execution of the device driver and replay it when needed. Imagine that a certain run of a device driver results in a crash (e.g., when being fuzzed). Recreating the crash might not be trivial since it might depend on a race condition that is triggered in a certain interleaving of driver execution and incoming interrupts from the I/O device. However, if the execution is recorded, it can be simply replayed and analyzed (e.g., with GDB). What is extremely useful about this technique is that *the replay of the driver does not even require having access to the actual mobile system*. Therefore, anyone with access to a virtual machine can replay the device driver execution and analyze it.

While any virtual machine record-and-replay can be used in Charm, we have implemented our own solution. It records all the interactions of the driver with the remote I/O device in the hypervisor and then replays them when needed.

2.3 Fuzzing

Fuzzing is a dynamic analysis technique that attempts to find bugs in a software module under test by providing various inputs to the module. In case of device drivers, the input to the driver is through system calls, such as `ioctl` and `read` system calls. While fuzzing is an effective technique to find bugs in software, it often suffers from low code coverage when inputs are randomly selected. Therefore, to increase coverage, feedback-guided fuzzing techniques collect execution information and use that to guide the input generation process. One such fuzzing tool is kAFL [65], which uses the hypervisor to collect execution information of the virtual machine by leveraging the Intel Processor Tracer (PT) hardware. Using kAFL to fuzz the device drivers of mobile systems is currently impossible because most of the commodity mobile devices use ARM processors, which do not have the Intel PT hardware. Moreover, hypervisor support is not enabled on these systems. However, by running the driver in a virtual machine in an x86 machine, Charm enables the use of kAFL.

Another such fuzzing tool, which is capable of fuzzing kernel-based device drivers, is Syzkaller [7], recently released by Google. Syzkaller uses a compiler-based coverage information collector, i.e., KCOV [4], and use that to guide its input generation. Since the coverage information collector is inserted into the kernel using the compiler, it is possible to use Syzkaller to directly fuzz the device driver running inside a mobile system. Yet, using Syzkaller with Charm provides three important advantages. First, Syzkaller can benefit from other dynamic analysis techniques only available for virtual machines. Specifically, record-and-replay can facilitate the analysis of the bugs triggered by Syzkaller, as discussed earlier.

Second, it is easier to leverage new kernel sanitizers of Syzkaller in a virtual machine compared to a mobile system. Kernel sanitizers instrument the kernel at compile time to allow Syzkaller to find non-crash bugs by monitoring the execution of the kernel. Examples are KASAN [9], which finds use-after-free and out-of-bounds memory bugs, KTSAN [11], which detects data races, KMSAN [10], which detects the use of uninitialized memory, and KUBSAN [12], which detects undefined behavior. Unfortunately, these sanitizers are not often supported in the kernel of mobile systems. To the best of our knowledge, only the Google Pixel smartphone's kernel supports KASAN [16]. In contrast, in Charm, one can simply choose a virtual machine kernel with support for these sanitizers. For example, we show that we can easily use KASAN in Charm by simply porting our drivers to a KASAN-enabled virtual machine kernel.

Finally, Syzkaller can more effectively capture and analyze crash bugs when fuzzing a virtual machine compared to a mobile system. Syzkaller reads the kernel logs of the operating system through its "console". It needs the kernel logs at the moment of the crash to capture the dump stack. The console of the virtual machine is reliably available by the hypervisor at the time of a crash. On the other hand, getting the console messages from a mobile system at the time of the crash is more challenging and requires a specialized adapter [8], which is not available to all analysts and is not easy to use. Indeed, kernel developers are familiar with the difficulty of having to use a serial cable on a desktop or laptop to get the last-second console messages from a crashing kernel in order to be able to debug the crash. Getting the console logs from a crashing mobile system is as challenging, if not more. When such debugging hardware is not available, one can try to read the kernel messages through the Android Debug Bridge (ADB) interface, the main interface used over USB for communication to Android mobile systems. Unfortunately, the interface cannot deliver the kernel crash logs since the ADB daemon on the phone crashes as well. One can attempt to read the crash logs after the mobile system reboots, but crash logs are not always available after reboot since a crash might corrupt the kernel, hindering its ability to flush the console to storage. These challenges are also confirmed by the Syzkaller's developers: "Android Serial Cable or Suzy-Q device to capture console output is preferable but optional. Syzkaller can work with normal USB cable as well, but that can be somewhat unreliable and turn lots of crashes into lost connection to test machine crashes with no additional info" [8]. Running the device driver in a virtual machine significantly alleviates this problem.

In our prototype, we use Syzkaller as one of the analysis tools used on top of Charm. We choose Syzkaller in order to be able to compare its performance with that of fuzzing directly on mobile systems. However, note that Charm can also support a fuzzer such as kAFL, which is impossible to use directly on a mobile system.

3 Overview

Our goal in this work is to facilitate the application of existing dynamic analysis techniques to mobile I/O device drivers.

3.1 Straw-man Approaches

Before describing our solution, we discuss two straw-man approaches that attempt to run a device driver inside a virtual machine. The first approach is to try to run the device driver in an existing virtual machine in a workstation (without the solutions presented by Charm). Unfortunately, this approach does not work out of the box since the driver requires access to the I/O device hardware in the mobile system. As a result, at boot time, the driver will not get initialized by the kernel since the kernel does not see the I/O device. If forced (e.g., by forcing the call to initialize the driver), the driver will immediately throw an error (since it will not be able to interact with the I/O device hardware), potentially resulting in a kernel panic in the virtual machine.

In this case, one might wonder whether we can emulate the I/O device hardware for the virtual machine in software. Unfotunately, doing so requires prohibitive engineering effort due to the diversity of I/O devices in mobile systems today.

The second approach is to run the device driver in a virtual machine in the mobile system and use the direct device assignment technique [21, 24, 43, 53, 54] (also known as device passthrough) to enable the virtual machine to access the underlying I/O device. This approach suffers from two important limitations. First, existing implementations of direct device assignment mainly support PCI devices common in x86 workstations, but not I/O devices of mobile systems. Second, running a hardware-based virtual machine within commodity mobile systems is impossible. While many mobile systems today incorporate ARM processor with hardware virtualization support, the hypervisor mode is disabled on these devices to prevent its use by rootkits. This leaves us with the option of software-based virtualization, which suffers from poor performance.

3.2 Charm's Approach

We present Charm, a system solution to facilitate the dynamic analysis of device drivers of mobile systems. Charm decouples the execution of the device driver from the mobile system hardware. That is, it enables the device driver to run in a virtual machine on a different physical machine, i.e., a workstation.

As mentioned earlier, a device driver needs access to its I/O device for correct execution. Our key idea to achieve this in Charm is to reuse the physical I/O devices through *remote device driver execution*. That is, we connect the physical mobile system directly to the workstation with a USB cable. The device driver executes *fully in the workstation* and only the *infrequent low-level I/O operations* are forwarded and executed on the physical mobile system.

In Charm, the latency of remoting the low-level I/O operations to the mobile system is of critical importance. High latency would result in various time-out problems in the device driver or I/O device. First, device drivers often wait for a bounded period of time for a response from the I/O device. In case the response comes later than expected, the device driver triggers a time-out error. Second, the I/O device might require timely reads and writes to registers. For example, after the device triggers an interrupt, it might require the driver to clear the interrupt (by writing to a register) in a short period of time. If not, the device might re-trigger the interrupt, potentially repeatedly.

In Charm, we leverage an x86 virtual machine in the workstation to execute the device driver. Given that mobile systems use ARM processors, one might wonder why we do not use an ARM virtual machine. Indeed, in our first prototype of Charm, we used a QEMU ARM virtual machine with ARM-to-x86 instruction interpretation on our x86-based workstation and implemented Charm fully in QEMU. Unfortunately, the overhead of instruction interpretation slowed the execution down to a point that our device drivers triggered various time-out errors. This made us realize that native execution is needed to meet the device driver's latency requirements, and hence we used a hardware-virtualized x86 virtual machine and reimplemented Charm in KVM.

Note that it is possible to use an ARM workstation in order to have native ARM execution for the Charm's virtual machine. However, while x86 workstations are easily available, ARM workstations are not yet commonplace. Therefore, we did not adopt this approach since we want Charm to be available to security analysts immediately.

3.3 Potential Concerns

There are two potential concerns with Charm's design. Fortunately, as we will report in our evaluation, we have managed to show that Charm overcomes both concerns. The first concern is potentially poor performance. Remoting I/O operations can significantly slow down the execution of the device driver. This can result in incorrect behavior due to time-outs. Even if there are no time-outs, it can slow down the dynamic analysis' execution, e.g., fuzzing time. In this paper, we show that by leveraging native execution of an x86 processor and a customized low-latency USB channel, we can not only eliminate time-outs but also achieve performance on par with the execution of the analysis running directly on the mobile system.

The second concern is that the disparity between the ARM Instruction Set Architecture (ISA) used in mobile systems vs. the x86 ISA used in the virtual machine may result in incorrect device driver behavior, which can affect the analysis, e.g., false positives in bugs detected by a fuzzer. Fortunately, as we will show, that is not the case. For example, we have not yet encountered a confirmed false positive bug detected by Charm. Moreover, we have verified that several Proof-of-Concept codes (PoC's) publicly reported for a device driver are also effective in Charm. The reason behind this is that device drivers are written almost fully in C and they suffer from bugs in the source code, which are effective regardless of the ISA that they are compiled to. We do, however, note that "compiler bugs", e.g., undefined behavior bugs [70], can show different behavior in the mobile system vs. Charm. This is because a compiler bug present in a C x86 compiler might not be present in a C ARM compiler, and vice versa. Therefore, Charm might result in false compiler bug reports (although we have not yet come across one) . However, note that bugs due to undefined behavior are not necessarily false positives since they happen due to the driver code wrongly relying on an undefined behavior of the language. Finally, Charm might result in false negatives for ARM compiler bugs as well.

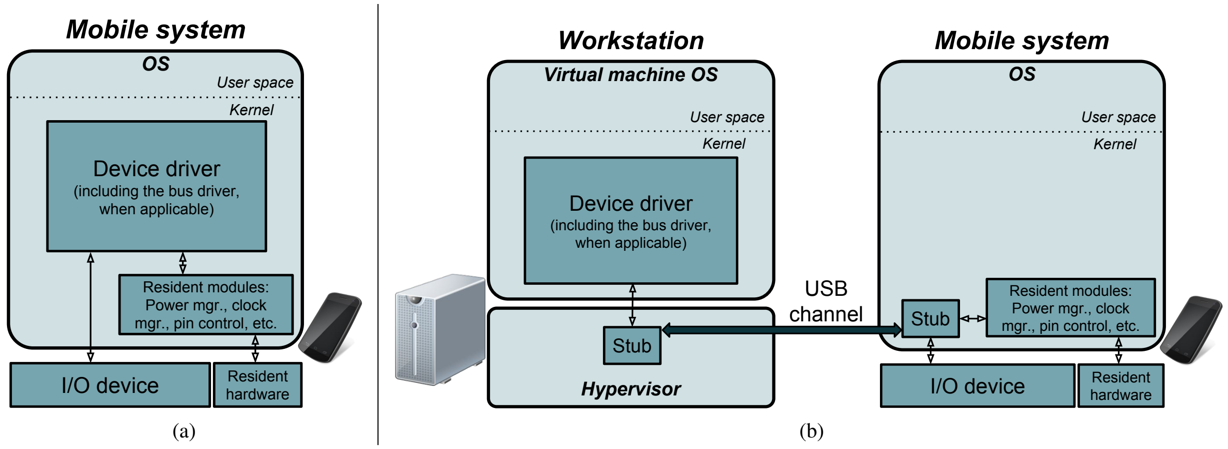


Figure 2: *(a) Device driver execution in a mobile system. (b) Remote device driver execution in Charm.*

4 Remote Device Driver Execution

The key enabling technique in Charm is the remote execution of mobile I/O device drivers. In this technique, we run the device driver in a virtual machine in the workstation. We then intercept the low-level interactions of the driver with the hardware interface of the I/O device and route them to the actual mobile system through a USB channel. Similarly, interrupts from the I/O device in the mobile system are routed to the device driver in the virtual machine. Figure 2 illustrates this technique. We will next elaborate on the solution's details.

4.1 Device and Device Driver Interactions

The remote device driver technique requires us to execute the device driver in a different physical machine from the one hosting the I/O device. At first glance, this sounds like an impossible task. The device driver interacts very closely with the underlying hardware in the mobile system. Therefore, this raises the question: *is remote execution of a device driver even possible?* We answer this question positively in this paper. To achieve this, a stub module in the workstation's hypervisor communicates with a stub module in the mobile system to support the device driver's interactions with its hardware. These interactions are three-fold: accesses to the registers of the I/O device, interrupts, and Direct Memory Access (DMA). Charm currently supports the first two. We will demonstrate that these two are enough to port and execute many device drivers remotely. In §8, we will discuss how we plan to support DMA in the future.

Register accesses. Using the hypervisor in the workstation, we intercept the accesses of the device driver to its registers. Upon a register write, we forward the value to be written to the stub in the mobile system. Upon a register read, we send a read request to the stub module, receive the response, and return it to the device driver in the virtual machine.

Interrupts. The stub module in the mobile system registers an interrupt handler on behalf of the remote driver. Whenever the corresponding I/O device in the mobile system triggers an interrupt, the mobile stub forwards the interrupt to the stub in the workstation, which then injects it into the virtual machine for the device driver to handle.

4.2 Device Driver Initialization

For the device driver to get initialized in the kernel of the virtual machine, the kernel must detect the corresponding I/O device in the system. Therefore, for a remote device driver to get initialized in the virtual machine, we must enable the kernel of the virtual machine to "detect" the corresponding I/O device as being connected to the virtual machine. ARM and x86 machines use different approach for I/O device detection. In an ARM machine, a *device tree* is used, which is a software manifest containing the list of hardware components in the system. In this machine, the kernel parses the device tree at boot time and initializes the corresponding device drivers. In an x86 machine, hardware detection is mainly used through the Advanced Configuration and Power Interface (ACPI). In an x86 virtual machine, the ACPI interface is emulated by the hypervisor.

The first solution that we considered was to add a remote I/O device to the hypervisor's ACPI emulation layer so that the virtual machine kernel can detect it. However, this solution would require significant engineering effort to translate the device tree entries into ACPI devices. Therefore, we take a different approach. We have the x86 kernel parse and use device trees as well. That is, we first allow the kernel to finish its ACPI-based device detection. After that, the kernel parses the device tree to detect the remote I/O devices. This significantly reduces the engineering effort. To support the initialization of a new device driver, we only need to copy the device tree entries corresponding to the I/O device of interest from the device tree of the mobile system to that of the virtual machine.

4.3 Low-Latency USB Channel

We use USB for connecting the mobile system to the workstation as USB is the most commonly used connection for mobile systems. USB provides adequate bandwidth for our use cases. For example, the USB 3.0 standard (used in modern mobile systems) can handle up to 5 Gbps.

In Charm, in addition to bandwidth, the latency of the channel between the workstation and the mobile system is of utmost importance. High latency can result in time-out problems in both the I/O device and the device driver. In our initial prototypes of Charm, we experienced various time-out problems in the device driver and I/O device due to high latency of our initial channel implementation. In this prototype, we used a TCP-based socket over the ADB interface. However, our measurements showed that this connection introduces a large delay (about one to two milliseconds for a round trip). This latency is due to several user space/kernel crossings both in the virtual machine and mobile system. To address this problem, we implement a low-level and customized USB channel for Charm. In this channel, we create a USB gadget interface [13] for Charm and attach five endpoints to this interface. Two endpoints are used for bidirectional communication for register accesses. Two endpoints are used for bidirectional communication for RPC calls (explained in §4.4). And the last endpoint is used for unidirectional communication for interrupts (from the mobile system to the workstation). Both in the mobile system and in the workstation, our stub modules read and write to these endpoints directly in the kernel (the host operating system kernel in the case of the workstation) hence avoiding costly user/kernel crossings. Therefore, this channel eliminates all user space/kernel crossings, significantly reducing the latency.

To further minimize the latency of communication over this channel, we perform an optimization: *write batching*. That is, we batch consecutive register writes by simply sending the write request over the USB channel and receiving the acknowledgment asynchronously, hence removing the wait-for-ack latency between these consecutive writes.

4.4 Dependencies

A device driver does not merely interact with the I/O device hardware interface. It often interacts with other kernel modules in the mobile system. We use two solutions for resolving these dependencies. First, if a kernel module is not needed on the mobile system itself, we move that module to the workstation virtual machine as well. The more modules that are moved to the virtual machine, the better we can analyze the device driver behavior. Consider fuzzing as an example. Fuzzing the device driver in the virtual machine will manage to also find bugs in these other modules if they are moved to the virtual machine. An example of a dependent module that we move the virtual machine is the bus driver. Many I/O devices are connected to the main system bus in the System-on-a-Chip (SoC) via a peripheral bus. In this case, the device driver does not directly interact with its own I/O device. Instead, it uses the bus driver API.

Second, if a module is needed on the mobile system, we keep the module in the mobile system and implement a Remote Procedure Call (RPC) interface for the driver in the virtual machine to communicate with it. We have identified the minimal set of kernel modules that cannot be moved to the virtual machine. We refer to these modules as "resident modules". These modules (which include power and clock management system, pin controller hardware, and GPIO) are in charge of hardware components that are needed to boot the mobile system and configure the USB interface. We refer to these hardware components as "resident hardware". Figure 2b illustrates this design.

Note that we implement Charm's RPC interface at the boundary of generic kernel APIs. More specifically, we use the generic kernel power management, clock management, pin controller, and GPIO API for RPC. This allows for the portability of the RPC interface. That is, since the kernel of all Android-based mobile systems leverage mostly the same API (although different kernel versions might have slightly different API), Charm's RPC implementation can be simply ported, requiring minimal engineering effort.

4.5 Porting a Device Driver to Charm

Supporting a new driver in Charm requires *porting the driver to Charm*. At its core, this is similar to porting a driver from one Linux kernel to another, e.g., porting a driver to a different Linux kernel version or to the kernel used in a different platform. Device driver developers are familiar with this task. Therefore, we believe that porting a driver to Charm will be a routine task for driver developers. Moreover, we show, through our evaluation, that non-driver developers should also be able to perform the port as long as they have some knowledge about kernel programming, which we believe is a requirement for security analysts working on kernel vulnerabilities.

Porting a device driver to run in Charm requires the following steps. The first step is to add the device driver to the kernel of the virtual machine in Charm. This requires copying the device driver source files to the kernel source tree and compiling them. Moreover, if the device driver has movable dependencies, e.g., a bus driver, the dependent modules must be similarly moved to the virtual machine kernel. One might face two challenges here. The first challenge is that the virtual machine kernel might have different core Linux API compared to the kernel of the mobile system. To solve this challenge, it is best to use a virtual machine kernel as close in version to the kernel of the mobile system as possible. This might not fully solve the incompatibilities. Hence, for the leftover issues, small changes to the driver might be needed. We have faced very few such cases in practice. For example, when porting the Nexus 6P GPU driver, we noticed that the Linux memory shrinker API in the virtual machine kernel is slightly different than that of the smartphone. We addressed this by mainly modifying one function implementation. The second challenge is potential incompatibilities due to the virtual machine kernel being compiled for x86 rather than ARM. This is due to the potential use of architecture-specific constants and API in the driver. To solve these, it is best to support the ARM constants and API in the x86-specific part of the Linux kernel instead of modifying the driver. We have faced a couple of such cases. For example, Linux x86 support does not provide the `kmap_atomic_flush_unused()` API, which is supported in ARM and hence used in some drivers. Therefore, this function needs to be added and implemented in Charm.

The second step is to configure the driver to run in the virtual machine given that the actual I/O device hardware is not present. To do this, the device tree entries corresponding to the I/O device hardware must be moved from the mobile system's device tree to that of the virtual machine (as discussed in §4.2). In doing so, dependent device tree entries, such as the bus entry, must be moved too.

The third step is to configure Charm to remote the I/O operations of the driver to the corresponding mobile system. This includes determining the physical addresses of register pages of the corresponding I/O device (easily determined using the device tree of the mobile system) as well as setting up the required RPC interfaces for interactions with modules in the mobile system. The latter can be time-consuming. Fortunately, it is a one-time effort since the RPC interface is built on top of generic Linux API shared across all Linux-based mobile systems (as mentioned in §4.4). Hence, many of the RPC interfaces can simply be reused.

Mobile System	I/O Device	Device driver LoC
LG Nexus 5X	Camera	65,000
LG Nexus 5X	Audio	30,000
Huawei Nexus 6P	GPU	31,000
Samsung Galaxy S7	IMU Sensors (accelerometer, compass, gyroscope)	3,000

Table 1: *Device drivers currently supported in Charm.*

The last step is to configure the mobile system to handle the remoted operations. This needs to be done in two sub-steps. First, Charm's stub needs to be ported to the kernel of the mobile system. This step is trivial and requires adding a kernel module and configuring the USB interface to work with the module. Second, the device drivers that are ported to the virtual machine must be disabled in the mobile system (since we cannot have two device drivers managing the same I/O device). This is easily done by disabling the device driver in the kernel build process. Alternatively, one can remove the corresponding device tree entries of the I/O device from the mobile system's device tree.

5 Implementation & Prototype

We have ported 4 device drivers to Charm: the camera and audio device drivers of LG Nexus 5X, the GPU device driver of Huawei Nexus 6P, and the IMU sensor driver of Samsung Galaxy S7. Table 1 provides more details about these drivers. It shows that these drivers, altogether, constitute 129,000 LoC. We extract these drivers from LineageOS sources for each of the phones. The Linux kernel versions of the operating system for Nexus 5X, Nexus 6P, and Galaxy S7 are 3.10.73, 3.10.73, and 3.18.14. We port these drivers to a virtual machine running Android Goldfish operating system with Linux kernel version 3.18.94.

As mentioned in §4.1, we do not currently support DMA operations. DMA is often used for data movement between CPU and I/O devices. Therefore, the lack of DMA support does not mostly affect the behavior of the driver; it only affect the data of I/O device (e.g., a captured camera frame). However, this is not always the case, and DMA can be used for programming the I/O device as well. One device driver that does so is the GPU driver. It uses DMA to program the GPU's command streamer with commands to execute. We cannot currently support this part of the GPU driver, and we hence disable the programming of the command streamer in the driver. Regardless, we show in §6.2 and §6.4 that we can still effectively fuzz the device driver and even find bugs.

We use a workstation in our prototype consisting of two 18-core Xeon E5-2697 V4 processors (on a dual-socket SeaMicro MBD-X10DRG-Q-B motherboard) with 132 GB of memory and 4 TB of hard disk space. We install and use Ubuntu 16.04.3 in the workstation with Linux kernel version 4.10.0-28.32. To support the remoting of I/O operations, we have modified the QEMU/KVM hypervisor (QEMU in Android emulator 2.4, which we use in our prototype). Note that while we use a Xeon-based machine in our prototype, we believe that a desktop/laptop-grade processor can be used as well, although we have not yet tested such a setup. This is because, as we will show in §6.2, the virtual machine does not need a lot of resources to achieve good performance for the device driver. A virtual machine with 6 cores and 2 GB of memory is adequate.

We write device driver templates for Syzkaller. A template provides domain knowledge for the fuzzer about the structure of the system calls supported by the driver. Our experience with Syzkaller is that without the templates, the fuzzer is not able to reach deep code within the driver. We use these templates for all our experiments with Syzkaller in §6. Alternatively, one can use an automated tool for template generation, such as DIFUZE [36].

We faced a challenge in supporting interrupts. That is, the x86-based interrupt controllers supported in the virtual machine only supports up to 24 interrupt line numbers. The ARM interrupt controller, on the other hand, supports interrupt line numbers as large as 987. Hence, we extended the number of supported interrupt line numbers in our virtual machine to 128 and implemented an interrupt line number translation in the hypervisor.

6 Evaluation

We answer the following questions in this section: (*i*) Is it feasible to support various device drivers of different mobile systems in Charm? (*ii*) Does remote device driver execution affect the performance of the device driver? (*iii*) Is Charm's record-and-replay effective? (*iv*) Can Charm be effectively used for finding bugs in device drivers? Does using an x86 machine (vs. ARM) result in false positives? and (*v*) Can manual debugging of a device driver, enabled by Charm, enable the security analyst to understand a vulnerability and/or build an exploit?

6.1 Feasibility

It is important that Charm supports diverse device drivers in different mobile systems. We evaluate how long it takes one to port a new driver to Charm. To do this, we report the time it took one of the authors to port the GPU driver of Nexus 6P and the IMU sensor driver of Samsung Galaxy S7. This author ported these drivers to Charm after the implementation of Charm was almost complete, hence he could mainly focus on the port itself.

The port of these two drivers was mainly performed by a different author from the author who ported the first two drivers (i.e., camera and audio drivers of Nexus 5X). Therefore, this author had to learn about the port process in addition to performing the port. These two new drivers are each on a different smartphone compared to Nexus 5X used for camera and audio drivers. Therefore, the port of these drivers required adding Charm's component to these smartphones' kernels as well.

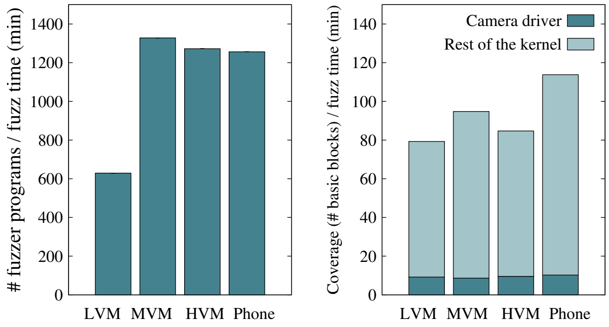
It took the author less than one week to port the GPU driver and, after that, about 2 days to port the sensor driver. This author is familiar with kernel programming and device drivers. We believe that this is the profile of a security analyst who works on device drivers.

6.2 Performance

Charm adds noticeable latency to every remoted operation (i.e., register accesses, interrupts, and interactions with the resident modules as discussed in §4.4). Therefore, one might wonder if Charm impacts the performance of the device driver significantly.

To evaluate the performance of the device driver, we perform two experiments. In the first experiment, we use the Syzkaller fuzzing framework. That is, we configure Syzkaller to fuzz the driver by issuing a large number of syscalls to the camera driver of Nexus 5X both directly in the mobile system and in Charm. Syzkaller operates by creating "programs", which are ensembles of a set of syscalls for the driver, and then executing these programs. We run Syzkaller for one hour in each experiment and measure the number of executed programs as well as the code coverage.

Figure 3a shows the results for the number of executed fuzzer programs per minute. We show the results for 4 setups: *LVM*, *MVM*, *HVM*, and *Phone*. The first three setups (standing for Light-weight VM, Medium-weigh VM, and Heavy-weight VM) represent fuzzing the device driver in Charm while the last one represents fuzzing the device driver directly on the Nexus 5X smartphone. LVM is a virtual machine with 1 core and 1 GB of memory. MVM is a virtual machine with 6 cores and about 2 GB of memory (similar to the specs of the Nexus 5X). HVM is a virtual machine with 16 cores and 16 GB of memory. Moreover, we configure Syzkaller to launch as many fuzzer processes (one of the configuration options of the framework that controls the degree of concurrency) as the number of cores. The results show that MVM achieves the best performance amongst the virtual machine setups. It outperforms the LVM due to availability of more resources needed for execution of fuzzing



(a)

(b)

Figure 3: *(a) Execution speed of the fuzzer. (b) Coverage of the fuzzer.*

programs. It also slightly outperforms the HVM. We believe that this is due to the high level of concurrency in the HVM experiment, which negatively impacts the performance. Finally, the results also show that MVM and HVM slightly outperform the phone's performance. This result is important: it shows that Charm's remote device driver execution does not negatively impact the performance of the driver and hence the driver can be used for various analysis purposes.

Figure 3b also shows the code coverage of the fuzzing experiments. It shows the coverage for the camera device driver and the rest of the kernel. The results show that Charm achieves similar code coverage in the driver compared to fuzzing directly on the smartphone. Note that the results show that the coverage in the rest of the kernel is different in Charm and in the smartphone. This is because the kernel in these two setups are different. While they are close in version, one is for x86 and one is for ARM and hence the coverage in the rest of the kernel cannot be directly compared in these setups.

In the second experiment, we choose a benchmark that significantly stresses Charm: the initialization of the camera driver in Nexus 5X. This initialization phase, among others, reads a large amount of data from an EEPROM chip used to store camera filters and causes many remote I/O operations (about 8800). We measure the driver's initialization time on the smartphone and in MVM to be 555 ms and 1760 ms, respectively. This shows that I/O-heavy benchmarks can slow down the performance of the driver in Charm. Yet, we do not anticipate this to be the case for many dynamic analysis tools that we target for Charm, including fuzzing (as seen previously).

6.3 Record-and-Replay

We demonstrate the feasibility of record-and-replay in Charm. As mentioned in §2.2, we implement a simple record-and-replay solution for Charm. It only records and replays the interactions of the device drivers and the I/O device (including register accesses and interrupts). Replaying register accesses is simple: a write access is simply ignored while a read access receives a value from the recorded log. Replaying interrupts is done by injecting the interrupt after observing all the preceding register accesses. Our simple record-and-replay implementation does not support concurrent execution of threads within the driver.

To demonstrate the effectiveness of Charm's record-and-replay, we record the execution of a PoC (related to bug #2 discussed in §6.4). We are then able to successfully replay the execution of the PoC and its interactions with the device driver without requiring a mobile system. Such a replay capability is significant help to understanding this bug.

We also evaluate the overhead of recording and the execution speed of the replay. For this purpose, we record the initialization phase of the camera device driver in Nexus 5X and successfully replay it without needing a Nexus 5X smartphone. We measure the recorded initialization and the replayed initialization to take 1843 ms and 344 ms, respectively. As mentioned in the previous section, the normal initialization of this driver in Charm takes 1760 ms. The results show that (*i*) recording does not add significant overhead to Charm's execution and (*ii*) the replay is much faster than the normal execution (indeed, the replay is even faster than the initialization time on the smartphone itself, which is 555 ms). The latter finding is important: replay accelerates the analysis, e.g., for that of a PoC.

6.4 Bug Finding

We investigate whether Charm can be used to effectively find bugs in device drivers. We use Syzkaller for this purpose and fuzz the drivers supported in Charm. One key question that we would like to answer is whether using an x86 virtual machine for a mobile I/O device driver would result in a large number of false positives, which can make the fuzzing more difficult for the analyst as s/he will have to filter out these false positives manually.

Table 2 shows the list of 25 bugs that we have found in the camera and GPU drivers (we did not find any bugs in the other drivers). The table also shows that we confirmed the correctness of these bugs through various methods (i.e., developing a PoC, checking against the latest driver commits, and manual inspection). We use PoC development and manual inspection to confirm the bugs that we detect in the latest version of the drivers (many of which we have reported). However, in addition to the latest version of the drivers, we also fuzz slightly older versions of them (i.e., not the latest publicly available commit of the driver). This allows us to check the bugs detected by Syzkaller against the latest patches and confirm their validity. We label the bugs confirmed using this method as LC in Table 2. More specifically, by looking at the latest version of the driver, we can find a patch for the bug, which confirms its validity. We find the correct patch using its commit message as well as the location in the code to which the patch is applied to.

We also port the camera driver to a KASAN-enabled virtual machine for fuzzing with this sanitizer. KASAN detected one out-of-bounds bug and one use-after-free bug in the camera driver (bug #1 and bug #13 in Table 2). This shows an advantage of Charm. Not only it facilitates fuzzing, it enables newer features of the fuzzer that is not currently supported in the kernel of the mobile system.

Our analysis showed that these bugs belong to 7 categories: one unaligned access to I/O device registers, 19 NULL pointer dereferences, one invalid pointer dereference, one use-after-free, one out-of-bounds access, one divide-by-zero, and one explicit `BUG()` statement in the driver.

Fuzzing with Charm uncovered 14 previously unknown bugs. We have managed to develop PoCs for many of these bugs and reported nine of them to kernel developers already. The developers have acknowledged our reports, assigned a P2-level severity [6] to them, and are analyzing several of them at the time of this writing. They have already closed our reports for two of the bugs for which we did not have a PoC (bugs #13 and #22) and for one that they believe is not a security bug (bug #2).

Note that 3 of our PoCs do not trigger the same bug in the mobile system itself. We investigated the reasons behind this. For bug #14, the PoC rely on some prior device driver's system calls not being issued. On the mobile system, the user space camera service issues these system calls at boot time hence preventing the bug to be triggered afterwards. In Charm, however, we do not execute the user space camera service, allowing us to find the bug. We leave this to the user of the system to decide whether s/he wants to initialize the user space camera service in Charm, in which case such bugs would not be triggered by the fuzzer. We also studied a similar issue for bugs #23 and #24, which are also triggered in Charm (but not in the mobile system) for a similar reason.

We believe that these results demonstrate that Charm can be used to effectively find correct bugs in device drivers through fuzzing. However, note that false positives are possible either as a result of x86 compiler bugs or an incomplete driver port. For example, as mentioned in §5, we have not supported the DMA functionalities of

	Device driver	**Bug type**	**Confirmed? (How?)**
1	Camera	Out-of-bounds memory access in msm_actuator_parse_i2c_params (**Detected by KASAN**)	Yes (LC)
2	Camera	Unaligned reg access in msm_isp_send_hw_cmd() (**Reported to kernel developers**)	Yes (PoC)
3	Camera	NULL ptr deref. in msm_actuator_subdev_ioctl()	Yes (PoC, LC)
4	Camera	NULL ptr deref. in msm_flash_init()	Yes (PoC, LC)
5	Camera	NULL ptr deref. in msm_actuator_parse_i2c_param()	Yes (LC)
6	Camera	NULL ptr deref. in msm_vfe44_get_irq_mask()	Yes (LC)
7	Camera	NULL ptr deref. in msm_csid_irq()	Yes (LC)
8	Camera	Invalid ptr deref. in cpp_close_node()	Yes (LC)
9	Camera	NULL ptr deref. in msm_ispif_io_dump_reg()	Yes (LC)
10	Camera	NULL ptr deref. in msm_vfe44_process_halt_irq()	Yes (LC)
11	Camera	NULL ptr deref. in msm_csiphy_irq()	Yes (LC)
12	Camera	NULL ptr deref. in msm_csid_probe()	Yes (LC)
13	Camera	Use-after-free in msm_isp_cfg_axi_stream (**Detected by KASAN**) (**Reported to kernel developers**)	Yes (MI)
14	Camera	NULL ptr deref. in msm_private_ioctl() (**Reported to kernel developers**)	Yes (PoC)
15	Camera	NULL ptr deref. in msm_ispif_io_dump_reg() (**Reported to kernel developers**)	Yes (PoC)
16	Camera	NULL ptr deref. in msm_vfe44_axi_reload_wm() (**Reported to kernel developers**)	Yes (PoC)
17	Camera	NULL ptr deref. in msm_vfe44_axi_ub() (**Reported to kernel developers**)	Yes (PoC)
18	Camera	NULL ptr deref. in msm_vfe44_stats_cfg_ub() (**Reported to kernel developers**)	Yes (PoC)
19	Camera	NULL ptr deref. in msm_vfe44_reset_hardware() (**Reported to kernel developers**)	Yes (PoC)
20	Camera	NULL ptr deref. in msm_vfe44_stats_clear_wm_irq_mask() (**Reported to kernel developers**)	Yes (PoC)
21	Camera	NULL ptr deref. in msm_vfe44_reg_update() (**Reported to kernel developers**)	Yes (PoC)
22	Camera	Divide-by-zero in msm_isp_calculate_bandwidth() (**Reported to kernel developers**)	Yes (MI)
23	GPU	NULL ptr deref. in _kgsl_cmdbatch_create()	Yes (PoC)
24	GPU	NULL ptr deref. in kgsl_cmdbatch_destroy()	Yes (PoC)
25	GPU	kernel BUG() triggered in adreno_drawctxt_detach()	Yes(MI)

Table 2: *Bugs we found in device drivers through fuzzing with Charm. MI and LC refer to confirming the bug by Manual Inspection and by checking the driver's Latest Commits, respectively.*

```
    /* in msm_csid_cmd(): */
 1  for (i = 0; i < csid_params.lut_params.num_cid; i++) {
      ...
 2    if (copy_from_user(vc_cfg, (void *)
       csid_params.lut_params.vc_cfg[i], sizeof(struct
       msm_camera_csid_vc_cfg))) {
        ...
 3      for (i--; i >= 0; i--)
 4        kfree(csid_params.lut_params.vc_cfg[i]);
 5      rc = -EFAULT;
 6      break;
 7    }
 8    csid_params.lut_params.vc_cfg[i] = vc_cfg;
 9  }
    ...
10  rc = msm_csid_config(csid_dev, &csid_params);


    /* in msm_csid_cid_lut(): */
    ...
11  if (csid_lut_params->vc_cfg[i]->cid >=
      csid_lut_params->num_cid ||
      csid_lut_params->vc_cfg[i]->cid < 0) {
      ...
12  }
```

(a) Vulnerable code snippet of CVE-2016-3903

```
1  int16_t step_index = 0;
2  uint16_t step_boundary = 0;
   ...
3  for (; step_index <= step_boundary; step_index++) {
     ...
4    if (cur_code < max_code_size)
5      a_ctrl->step_position_table[step_index] = cur_code;
     ...
6  }
```

(b) Vulnerable code snippet of CVE-2016-2501

```
1  int i = stream_cfg_cmd->stream_src;
2  if (i >= VFE_AXI_SRC_MAX) {
     ...
3    return -EINVAL;
4  }
   ...
5  memset(&axi_data->stream_info[i], 0, sizeof(struct
      msm_vfe_axi_stream));
   ...
6  axi_data->stream_info[i].session_id =
      stream_cfg_cmd->session_id;
7  axi_data->stream_info[i].stream_id =
      stream_cfg_cmd->stream_id;
```

(c) Vulnerable code snippet of CVE-2016-2061

Figure 4: *Vulnerable code snippets.*

the GPU driver. This can result in false positives. In addition, false negative bugs are possible either for ARM compiler bugs or due to execution in a virtual machine, which might affect some characteristics of driver execution, such as timing. As a result, there might be real bugs (e.g., timing sensitive bugs), which we did not find using Charm.

6.5 Analyzing Vulnerabilities with GDB

Charm enables us to use GDB to analyze vulnerabilities in device drivers. To demonstrate this, we have analyzed three publicly reported vulnerabilities in the Nexus 5X camera driver: CVE-2016-2501, CVE-2016-3903, and CVE-2016-2061. We leverage the available PoCs in our analysis. The PoCs crash the kernel using the reported vulnerability. We use the kernel crash dump to identify the crash site. We then insert a breakpoint before the crash site in a GDB session to investigate the root cause of the crash. Since we compile the driver and kernel with debugging information, GDB can also display source lines, making the debugging much easier.

CVE-2016-3903. The vulnerable code is shown in Figure 4a. The crash site is at line 11 (in function `msm_csid_cid_lut()`). At a first glance, this appears to be an out-of-bounds access bug, but our investigation (described next) showed that this is a use-after-free bug. We performed our investigation as follows. By using a watchpoint, we find that the index variable `i` at the crash site is always within a normal range (and not negative). We then try to inspect other pointer values at the crash site with GDB and finally identify that `vc_cfg[i]` holds an invalid address. To trace the origin of the array `vc_cfg`, we utilize watchpoints to trace its parent structure `csid_lut_params` and finally locate another function, `msm_csid_cmd`, which is responsible for initializing the structure. By single-stepping through the initialization code, we find that if an error occurs during the `vc_cfg` initialization at line 2, it will be freed at line 4 and then the initialization loop will terminate at line 6. However, the function call at line 10 will continue to use the `csid_params` structure regardless of its `vc_cfg` subfield having been freed, thus causing a use-after-free vulnerability.

CVE-2016-2501. The vulnerable code is shown in Figure 4b. The crash site is at line 5. When the breakpoint at the crash site is triggered, we can infer that it is likely an out-of-bounds array access. Next, we set a watchpoint for the index variable `step_index`, tracing its value change. Indeed, its value is negative when the crash occurs. Upon a closer look, as a loop index, it is compared against `step_boundary` at line 3, which is a 16-bit register holding the value of `0xffff`. However, `step_index` is a signed integer and can take negative values before it reaches `0xffff` to terminate the loop (note that the comparison is unsigned). Therefore, when it is used as array index at line 5, out-of-bounds access occurs. In the end, we also set a watchpoint for `step_boundary` and find that its value comes from a function argument passed from user space, which is untrusted.

CVE-2016-2061. The vulnerable code is shown in

Figure 4c. A first glance at the crash site suggests a possibility that `memset()` at line 5 zeroes an invalid memory region, which causes the kernel crash. Indeed, by inspecting the various variable values involved in the crash at the crash site, we find that `i` takes a negative value as an array index, leading to an out-of-bounds access. To fully understand why `i` can be negative, we trace it back with the help of watchpoints and find that the value of `i` comes from a user controlled parameter (line 1). Besides, the sanity check at line 2 cannot filter the negative `i`, unfortunately. We then find out that this is a critical vulnerability. This is because starting from line 6, the right side of the assignment statements is also controlled by a parameter `stream_cfg_cmd` originated in user space. Together with the user controlled index variable `i`, this vulnerability becomes an ideal target for privilege escalation, which we show we can achieve next.

6.6 Building a Driver Exploit using GDB

Our analysis in the previous subsection show that CVE-2016-2061 can be potentially used for a full compromise of the kernel given that it can perform write operations at unintended locations. To further demonstrate the capabilities of Charm, we use GDB on the driver code and attempt to develop an exploit against it.

The first step is to check if the "vulnerable object" (`struct vfe_device`, where the out-of-bounds write occurs) is a kernel heap or stack object. With GDB, we are able to confirm that it is allocated using `kzalloc()`, indicating that it is a heap object. To gain the ability of arbitrary code execution from heap-related vulnerabilities, we attempt heap feng shui [40, 55], which is a technique to arrange the heap layout in a deterministic fashion to facilitate the write operation. However, this vulnerability only allows a very limited form of write. First of all, it cannot write to absolute addresses (only relative addresses to the base of an object). Secondly, when it writes, 480 bytes are written continuously (most are 0s due to the `memset()` at line 5), with only a few fields controlled by the attacker. Such a large memory footprint can destroy the integrity of data stored nearby and cause a kernel crash.

To address the first problem, we borrow the heap feng shui idea from the exploit of CVE-2017-7308 [5] to precisely co-locate the "vulnerable object" with one or more "target objects" (where one of their function pointer fields is the target for overwriting). To verify the feasibility of this approach, we use GDB to track the location of the vulnerable object. It turns out that the object is allocated in the beginning when the kernel boots, as part of the driver initialization procedure. In addition, its address changes from boot to boot, making it difficult to predict. When we attempt to allocate target objects (e.g., `struct sock`), their addresses shown by GDB are never close to the vulnerable object, due to the fact that they are allocated much later after the kernel boots completely. This means that the strategy of precisely co-locating the objects is not feasible. However, from GDB, we do notice that the address ranges of the vulnerable object and target objects more or less stay the same. This means that we can potentially spray a large number of target objects and try to arrange the target objects to be at a desired offset from the vulnerable object.

To address the second problem, where a 480-byte overwrite may crash the kernel unintentionally, it is necessary to know the size of the target object and how likely they will align with the vulnerable object. As it turns out, the vulnerable object is always at the start of a page. After exhausting the slab caches, we know that target objects (we use `struct inet_sock` which has a size of 896 bytes) are allocated in blocks whose addresses are aligned to be multiples of 4 pages. This allows us to calculate the desired offset at which the write should occur, where the `sk_destruct` function pointer can be overwritten. As a proof-of-concept, we use GDB to ensure that the target objects can indeed fall in the desired address range. By calling `close()` on the socket from user space, we can indeed cause the kernel to jump to any arbitrary location to execute code. Otherwise, we can simply spray enough objects and hope that the write will probabilistically succeed. Alternatively, we need a kernel arbitrary read vulnerability (similar to what Meltdown [52] provides) so that the attack can be deterministic.

Still, we need to make sure that the 480-byte overwrite does not crash the kernel. After all, the function pointer is towards the end of the `struct inet_sock` object, and the 480-byte overwrite will corrupt the next object adjacent to it. Fortunately, since we know `struct inet_sock` objects are allocated sequentially from low addresses to high addresses in a block, we can simply iterate the `close()` on each and every socket from user space and stop as soon as we notice a redirection of the control flow, ensuring that no one will touch the corrupted object.

7 Related Work

7.1 Remote I/O Access

The closest to our work are Avatar [77] and SURROGATES [50], solutions for dynamic analysis of binary firmware in embedded devices, such as a hard disk bootloader, a wireless sensor node, and a mobile phone baseband chip. Since performing analysis in embedded devices is difficult, they execute the firmware in an emulator and forward the low-level memory accesses (including I/O operation) to the embedded device. The remoting boundary in these solutions is similar to the boundary used in Charm. However, they focus on very different software and hardware. More specifically, they focus on binary firmware of embedded devices whereas Charm focuses on open source device drivers of mobile systems. Moreover, the connections to the embedded devices are low-bandwidth UART or JTAG interfaces in Avatar and a custom FPGA bridge in SURROGATES. In contrast, Charm uses a USB interface. This, in turn, results in different underlying techniques used in these systems. First, in its full separation mode, Avatar forwards all memory accesses to the embedded device, unlike Charm that ports the device driver fully to the virtual machine and only forwards I/O accesses. This results in poor performance in Avatar unlike Charm, which achieves performance on par with that of native mobile execution. To optimize, Avatar uses heuristics to perform some memory access locally. It also executes some or all of the firmware code directly on the embedded device. In contrast, Charm runs all the device driver code in the virtual machine. And for performance optimizations, it devises a custom low-latency USB channel and leverages the native execution speed of x86 processors. SURROGATES, on the other hand, tries to overcome the performance bottleneck in Avatar using a custom FPGA bridge that connects the host machine's PCI Express interface to the embedded device under test. In contrast, Charm does not require custom hardware. These technical differences also make these solutions useful for different analysis techniques. For example, Charm can fuzz the device driver fully in a virtual machine.

Other forms of remote I/O exists for mobile systems as well, such as Rio [22] and M+ [60]. The main difference between Charm and these systems is the boundary at which I/O operations are remoted. Rio uses the device file boundary and M+ uses the Android binder IPC boundary. In contrast, Charm uses the low-level software-hardware boundary. Therefore, Charm uniquely enables the remote execution of the device driver. In both Rio and M+, the device driver remains in the machine containing the I/O device.

Code offload has been an important topic in mobile computing research [35,38,44,45] in an effort to improve performance and reduce energy consumption. The idea is to offload heavy computation to a server to reduce the load on the mobile system itself. In Charm, in contrast, we "offload" the I/O operations from the workstation to real mobile systems.

7.2 Analysis of System Software

Over the years, many static and dynamic analysis solutions have been invented for a wide range of applications such as safety, reliability, and security. In recent years, popular analysis techniques include taint tracking [34, 41, 59, 76], symbolic and concolic execution [27, 28, 30, 31, 39, 73], unpacking and reverse engineering [47, 49, 74, 79], malware sandboxing [3, 25, 71], and fuzzing [29, 42, 69, 72].

Many of these analysis frameworks are built on top of the virtualization technology and can support full-system analysis, including the low-level code such as kernel and device drivers [33, 34, 59, 75, 76]. For instance, Panorama [76] and DroidScope [75] can analyze the entire Windows and Android operating systems, respectively. Aftersight [33] uses virtual machine replay to feed recorded logs from a production system to a testing system in real time where more expensive analysis is run. kAFL is a hardware-based feedback-driven kernel fuzzer [65]. It uses the Intel Processor Tracer (PT) to collect execution traces in the hypervisor and use that to guide the fuzzer. Digtool is a kernel vulnerability detection solution based on a customized hypervisor, which can monitor various events in the kernel such as memory allocation and thread scheduling. Keil et al. fuzz wireless device drivers in a QEMU virtual machine [48]. To enable the driver to run in a virtual machine, they emulate the wireless interface hardware in software. Dovgalyuk et al. perform reverse debugging of device drivers in a QEMU virtual machine. They use GDB as well as record-and-replay in their debugging. Unfortunately, none of these solutions can be applied to device drivers of mobile systems. They can only support system software running within a virtual machine, e.g., device drivers for emulated and virtualized I/O devices (including direct device assignment for PCI-based I/O devices). Charm addresses this problem and is complementary to all of these solutions. In other words, Charm enables all of these dynamic analysis solutions to be applied to device drivers of mobile systems as well.

Fuzzing is an effective dynamic analysis technique, which can be applied to the operating system kernel and device drivers as well. Peach Fuzzer fuzzes the device drivers by running a fuzzer in a separate physical machine than the one with the I/O device [17]. While superior to running the fuzzer and driver in the same machine, their approach suffers from similar challenges that Syzkaller suffers from when fuzzing a mobile system directly (§2.3). Charm solves these problems by making it possible to run the device driver in a virtual machine.

In [57], Mendonça et al. fuzz the Wi-Fi interface card driver. They perform the fuzzing directly on a Windows Mobile Phone. In contrast, Charm enables the fuzzing to be performed in a virtual machine in a workstation, providing significant usability benefits.

DIFUZE automatically generates templates for fuzzing the kernel device drivers directly on mobile systems [36]. IMF improves input generation by inferring a model for the system under test [46]. It learns the model by inspecting how application use the APIs of this system. Skyfire deploys data driven seed generation to enable fuzzing deep parts of the code [67]. Charm approach is orthogonal and it can benefit from DIFUZE, IMF and Skyfire for template generation.

VUzzer boosts the fuzzing effectiveness using static analysis [63]. It helps the fuzzer to spend most of its time reaching deeper parts of the code. Bohme et al. introduced a directed greybox fuzzing technique, which encourages the fuzzer to trigger specified part of the code [26]. VUzzer and directed greybox fuzzing can be used alongside Charm to improve the code coverage.

Slowfuzz enables finding non-crash bugs [62]. Charm can benefit from Slowfuzz since it generally broadens the scope of the fuzzers' use cases.

The diversity of device drivers and their direct interactions with physical I/O devices create challenges for dynamic analysis. Static analysis, therefore, has been extensively used on device drivers [23, 32, 61]. Examples are symbolic execution solutions such as in SymDrive [64], S2E [30, 31], and DDT [51] and taint and pointer analyses such as in DR. CHECKER [56]. Static analysis has the benefit of eliminating the need for the presence of actual devices. However, static analysis tools cannot uncover all the bugs and vulnerabilities in the drivers. They can only detect those which the analyzer explicitly checks for. Moreover, static analysis solutions often suffer from large false positive rates due to imprecision.

Analysis of firmware running inside embedded devices faces similar challenges stemming from diversity as analysis of device drivers. Both static analysis [37] and dynamic analysis [66, 77] solutions have been used for firmware analysis as well. In contrast to this line of work, Charm focuses on modern mobile systems.

7.3 Mobile Testing

Several mobile testing frameworks have recently emerged. BareDroid analyzes Android apps directly on mobile systems [58]. SPOKE analyzes the access control policies of Android by running test cases directly on mobile systems [68]. The main motivation behind this line of work is that the system software of mobile systems are unique and device-specific and hence these tests cannot be simply performed on virtual machines. Our motivation is in line with these systems. However, directly analyzing the device drivers in mobile systems is challenging, as we extensively discussed in the paper. Therefore, we enable these device driver to execute in a virtual machine for enhanced analysis.

8 Limitations and Future Work

DMA. As mentioned in §4.1, Charm does not currently support DMA. We plan to support DMA by integrating a Distributed Shared Memory (DSM) implementation into our prototype. The memory pages accessed through DMA will be kept coherent by the DSM system. However, we might need to insert explicit update operations in the driver for performance optimization and in the mobile system's kernel stub to notify the DSM system of the completion of DMA.

Closed source (binary) drivers. Charm does not currently support closed source (binary) device drivers. We plan to support these device drivers in the future. To do this, we plan to use ARM virtual machines (instead of x86 virtual machines used in this paper). We will either run this virtual machine in an ARM workstation or in an x86 server with a ARM-to-x86 interpreter (note that we will need to improve the performance of this interpreter to overcome the limitations mentioned in §3.2).

Automatic device driver porting. As we showed in our evaluations in §6.1, it takes time and engineering effort to port a new driver to Charm. We plan to build a framework for automatic porting of device drivers to Charm. In this framework, the security analyst will only need to provide the driver's source code and the list of resident modules. The framework will implement all required RPCs and port the driver to Charm automatically.

9 Conclusions

We presented Charm, a system solution for running device drivers of mobile systems in a virtual machine running in a workstation. Charm enables application of various existing dynamic analysis solutions, e.g., interactive debugging, record-and-replay, and enhanced fuzzing to these device drivers. Our extensive evaluation showed that Charm can support various device drivers and mobile systems (e.g., 4 drivers of 3 different smartphones in our prototype), achieves decent performance, and is effective in enabling a security analyst to find, study, and analyze driver vulnerabilities and even build exploits.

Acknowledgments

The work was supported by NSF Awards #1617481 and #1617573. We thank the paper shepherd, Adwait Nadkarni, and the reviewers for their insightful comments.

References

[1] ANDROID FRAGMENTATION VISUALIZED (AUGUST 2015). https://opensignal.com/legacy-assets/pdf/reports/2015_08_fragmentation_report.pdf.

[2] Android Security Bulletins. https://source.android.com/security/bulletin/.

[3] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[4] Code coverage tool for compiled programs (KCOV). https://github.com/SimonKagstrom/kcov.

[5] Exploiting the Linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[6] Google Issue Tracker: Issues. https://developers.google.com/issue-tracker/concepts/issues.

[7] Google Syzkaller: an unsupervised, coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller.

[8] Instruction for using the Syzkaller to fuzz an Android device. https://github.com/google/syzkaller/blob/master/docs/linux/setup_linux-host_android-device_arm64-kernel.md.

[9] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki.

[10] The Kernel Memory Sanitizer (KMSAN). https://github.com/google/kmsan/blob/master/README.md.

[11] The Kernel Thread Sanitizer (KTSAN). https://github.com/google/ktsan/wiki.

[12] The Kernel Undefined Behavior Sanitizer (KUBSAN). https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html.

[13] USB Gadget API for Linux. https://www.kernel.org/doc/html/v4.13/driver-api/usb/gadget.html, 2004.

[14] Access UART ports of Xperia devices. https://developer.sony.com/develop/open-devices/guides/access-uart-ports, 2013.

[15] Building a Nexus 4 UART Debug Cable. https://www.optiv.com/blog/building-a-nexus-4-uart-debug-cable, 2013.

[16] Building a Pixel kernel with KASAN+KCOV. https://source.android.com/devices/tech/debug/kasan-kcov, 2017.

[17] Peach Fuzzer for Driver Fuzzing Whitepaper. https://www.peach.tech/datasheets/driver-fuzzing/peach-fuzzer-driver-fuzzing-whitepaper/, 2017.

[18] Devices supported by LineageOS. https://wiki.lineageos.org/devices/, 2018.

[19] Samsung publishes kernel source code for Galaxy S9/S9+ Snapdragon and Exynos models. https://www.androidpolice.com/2018/03/14/samsung-publishes-kernel-source-code-galaxy-s9-s9-snapdragon-exynos-models/, 2018.

[20] Serial debugging. https://wiki.postmarketos.org/wiki/Serial_debugging, 2018.

[21] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).

[22] AMIRI SANI, A., BOOS, K., YUN, M., AND ZHONG, L. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proc. ACM MobiSys* (2014).

[23] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys* (2006).

[24] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B. A. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI* (2010).

[25] BLASING, T., BATYUK, L., SCHMIDT, A.-D., CAMTEPE, S., AND ALBAYRAK, S. An Android Application Sandbox System for Suspicious Software Detection. In *Proc. IEEE International Conference on Malicious and Unwanted Software (Malware)* (2010).

[26] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed Greybox Fuzzing. In *Proc. ACM CCS* (2017).

[27] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. USENIX OSDI* (2008).

[28] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on Binary Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2012).

[29] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-Adaptive Mutational Fuzzing. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2015).

[30] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: a Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proc. ACM ASPLOS* (2011).

[31] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems (TOCS)* (2012).

[32] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP* (2001).

[33] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX Annual Technical Conference* (2008).

[34] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security* (2004).

[35] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proc. ACM EuroSys* (2011).

[36] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proc. ACM CCS* (2017).

[37] COSTIN, A., ZADDACH, J., FRANCILLON, A., BALZAROTTI, D., AND ANTIPOLIS, S. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proc. USENIX Security Symposium* (2014).

[38] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys* (2010).

[39] DAVIDSON, D., MOENCH, B., JHA, S., AND RISTENPART, T. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proc. USENIX Security* (2013).

[40] DRAKE, J. J. Stagefright: An android exploitation case study. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[41] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. USENIX OSDI* (2010).

[42] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proc. Internet Society NDSS* (2008).

[43] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., TSAFRIR, D., AND SCHUSTER, A. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS* (2012).

[44] GORDON, M. S., HONG, D. K., CHEN, P. M., FLINN, J., MAHLKE, S., AND MAO, Z. M. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proc. ACM MobiSys* (2015).

[45] GORDON, M. S., JAMSHIDI, D. A., MAHLKE, S., MAO, Z. M., AND CHEN, X. COMET: Code Offload by Migrating Execution Transparently. In *Proc. USENIX OSDI* (2012).

[46] HAN, H., AND CHA, S. K. IMF: Inferred Model-based Fuzzer. In *Proc. ACM CCS* (2017).

[47] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A Hidden Code Extractor for Packed Executables. In *Proc. ACM Workshop on Recurring Malcode (WORM)* (2007).

[48] KEIL, S., AND KOLBITSCH, C. Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment. *Black Hat Japan* (2007).

[49] KIRAT, D., AND VIGNA, G. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proc. ACM CCS* (2015).

[50] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)* (2015).

[51] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. USENIX ATC* (2010).

[52] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (Jan. 2018).

[53] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC* (2006).

[54] LIU, M., LI, T., JIA, N., CURRID, A., AND TROY, V. Understanding the Virtualization "Tax" of Scale-out Pass-Through GPUs in GaaS Clouds: An Empirical Study. In *Proc. IEEE High Performance Computer Architecture (HPCA)* (2015).

[55] LIU, Z. E. Advanced Heap Manipulation in Windows 8. In *Black Hat Europe* (2013).

[56] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proc. USENIX Security Symposium* (2017).

[57] MENDONÇA, M., AND NEVES, N. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *Proc. IEEE European Dependable Computing Conference (EDCC)* (2008).

[58] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proc. Annual Computer Security Applications Conference (ACSAC)* (2015).

[59] NEWSOME, J. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. Internet Society NDSS* (2005).

[60] OH, S., YOO, H., JEONG, D. R., BUI, D. H., AND SHIN, I. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. In *Proc. ACM MobiSys* (2017).

[61] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS* (2011).

[62] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proc. ACM CCS* (2017).

[63] RAWAT, S., JAIN, V., KUMAR, A., AN CRISTIANO GIUFFRIDA, L. C., AND BOS, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Proc. Internet Society NDSS* (2017).

[64] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing Drivers without Devices. In *Proc. USENIX OSDI* (2012).

[65] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium* (2017).

[66] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proc. Internet Society NDSS* (2015).

[67] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proc. IEEE Security and Privacy (S&P)* (2017).

[68] WANG, R., AZAB, A. M., ENCK, W., LI, N., NING, P., CHEN, X., SHEN, W., AND CHENG, Y. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proc. ACM ASIA CCS* (2017).

[69] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2010).

[70] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proc. ACM SOSP* (2013).

[71] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security Privacy* (2007).

[72] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling Black-box Mutational Fuzzing. In *Proc. ACM CCS* (2013).

[73] YADEGARI, B., AND DEBRAY, S. Symbolic Execution of Obfuscated Code. In *Proc. ACM CCS* (2015).

[74] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2015).

[75] YAN, L. K., AND YIN, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. USENIX Security* (2012).

[76] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. ACM CCS* (2007).

[77] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. Avatar: A framework to Support Dynamic Security Analysis of Embedded Systems Firmwares. In *Proc. Internet Society NDSS* (2014).

[78] ZHANG, H., SHE, D., AND QIAN, Z. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS* (2015).

[79] ZHANG, Y., LUO, X., AND YIN, H. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (2015).