Ex-vivo dynamic analysis framework for Android device drivers

Ivan Pustogarov
University of Toronto
i.pustogarov@utoronto.ca

Qian Wu
University of Toronto
qianch.wu@mail.utoronto.ca

David Lie
University of Toronto
lie@eecg.toronto.edu

***Abstract*—The ability to execute and analyze code makes many security tasks such as exploit development, reverse engineering, and vulnerability detection much easier. However, on embedded devices such as Android smartphones, executing code in-vivo, on the device, for analysis is limited by the need to acquire such devices, the speed of the device, and in some cases the need to flash custom code onto the devices. The other option is to execute the code ex-vivo, off the device, but this approach either requires porting or complex hardware emulation.**

In this paper, we take advantage of the observation that many execution paths in drivers are only superficially dependent on both the hardware and kernel on which the driver executes, to create an ex-vivo dynamic driver analysis framework for Android devices that requires neither porting nor emulation. We achieve this by developing a generic *evasion* framework that enables driver initialization by evading hardware and kernel dependencies instead of precisely emulating them, and then developing a novel *Ex-vivo AnalySIs framEwoRk (EASIER)* that enables off-device analysis with the initialized driver state. Compared to on-device analysis, our approach enables the use of userspace tools and scales with the number of available commodity CPU's, not the number of smartphones.

We demonstrate the usefulness of our framework by targeting privilege escalation vulnerabilities in system call handlers in platform device drivers. We find it can load 48/62 (77%) drivers from three different Android kernels: MSM, Xiaomi, and Huawei. We then confirm that it is able to reach and detect 21 known vulnerabilities. Finally, we have discovered 12 new bugs which we have reported and confirmed.

I. Introduction

The Android kernel is an attractive target for malicious actors: it powers millions of mobile devices and vulnerabilities in it are particularly dangerous as they can be exploited by malicious code to gain high privilege execution level. Historically, the first place to look for kernel vulnerabilities is driver code [4], [6], [18], [25], [26]. Since Android devices ship with different peripherals (such as cameras or accelerometers), manufacturers customize the stock Android kernel by adding corresponding drivers to support these peripherals. As a result, a significant part of these custom Android kernels consists of driver code that may not be as rigorously audited as main kernel components.

While vulnerabilities can be found using both static and dynamic analysis, the latter makes triaging, understanding and fixing (or exploiting) such security vulnerabilities much easier. For example, coverage-based fuzzing and symbolic execution have proven to be very efficient in finding vulnerabilities; setting breakpoints, pausing execution, and peeking into memory state is important when identifying the exact cause of a crash; taint tracking is very useful in reverse engineering; and memory integrity checkers, such as AddressSanitizer [22] are critical in detecting memory corruption vulnerabilities.

Dynamic analysis can be performed either on-device (in-vivo) or off-device (ex-vivo). Unfortunately, while in-vivo analysis might be ideal due to its accuracy, it is often impractical, especially at scale. This is because drivers reside in an operating system kernel and dynamic analysis of a kernel often requires either special privileges or special hardware, neither of which are commonly available on Android devices. For example, syzkaller [28] relies on running a custom kernel on the device, but many Android devices will only boot signed kernels. Similarly, kAFL [20] relies on the Intel PT hardware tracing, which is not available on the ARM processors that dominate smartphones. Finally, on-device analysis would require a smartphone of a particular model for each analysis instance, which does not scale considering the number of smartphone devices that exist.

The alternative to in-vivo analysis is off-device, ex-vivo analysis in a hardware emulator. Emulated hardware has many benefits, including efficiency, enabling monitoring that is unavailable on the real hardware and the ability to parallelize and scale on commodity CPU clusters. Unfortunately, it is challenging to employ emulation to dynamically analyze device drivers for two reasons. First, drivers have *hardware dependencies* on proprietary, device-specific hardware components that are not provided by the current emulators. Unless these dependencies are satisfied, the drivers will not execute properly in an emulated environment. Unfortunately, the proprietary nature of the devices means that their specifications are not available, and while the specifications can be reverse engineered, the effort required precludes scaling to the plethora of Android devices that exist. Second, drivers have *software dependencies* on the host kernels on which they were meant to run[1]. Unfortunately, hardware dependencies between the host kernels and the device hardware also prevent those kernels from running in an emulator. In fact, very few hardware-specific kernels can boot on commonly available emulators—for example, the Qemu emulator only supports a few board-specific kernels (such as `vexpress` or `versatile`). The inability to boot host kernels in an emulator has been a challenge for other work as well [3]. Running an Android driver inside an emulator would require porting the driver to a kernel that can be emulated, but this also requires a significant amount of effort.

[1] E.g. Qualcomm drivers depend on the MSM Android kernel subsystems that work with Qualcomm hardware.

Some approaches such as [29] and [32] avoid having to emulate hardware by splitting the execution between the emulated and real systems. For example, in [29], whenever the driver tries to access the peripheral the corresponding calls are redirected to the physical device. However, such approaches still have the two drawbacks we mentioned—they require porting of the driver to an off-device kernel, as well as manual splitting of the driver, and while they do not require emulation of the hardware device, they require a real hardware device for each dynamic analysis instance, limiting scalability.

Our approach. In this paper, we present *evasion*, a technique for detecting and analyzing vulnerabilities using dynamic ex-vivo device driver analysis for Android phones. Evasion enables the ex-vivo dynamic analysis of unmodified driver code without: (a) having to port it and its dependencies, and (b) the requirement to have the physical device. Without emulated peripherals and the physical device, this goal becomes a trade-off between emulation completeness and availability. Our key enabling insight, gained through extensive analysis of driver code, is that while there are many execution paths that do have hardware and software dependencies, these dependencies are superficial. For example, they may only depend on the ability to read a device register, but not on the actual value returned; or they may depend on a certain function returning a success code, but not the actual semantics of the function. Most importantly, we observe that there are many such *superficially dependent* paths and such paths contain vulnerabilities. Thus, instead of having to do the precise and rigorous work of emulating all dependencies, we instead propose the alternative approach of "evading" dependencies, and embody this idea in an evasion kernel, which superficially claims to satisfy the dependency, but does so in a generic way that does not have to be faithful to the true software or hardware component on which the driver depends.

We demonstrate the usefulness of evasion by developing the *Ex-vivo AnalySIs framEwoRk (EASIER)*, which we use to target privilege escalation vulnerabilities in driver IOCTL system call handlers. IOCTLs are historically the biggest source of such local privilege escalation vulnerabilities in drivers, and are a critical component in remote exploit chains which are considered a high-value targets in Android ecosystem [7]. One advantage of ex-vivo analysis is that we are able to perform this analysis with standard userspace vulnerability detection tools, such as the AFL fuzzer [33] and the Manticore [16] symbolic execution library. This ability is beneficial because the number and power of userspace tools greatly exceeds those of in-kernel tools, e.g. syzkaller [28] and S2E [5], are among the most well-known, and still, they are hard to setup and use.

EASIER works by initializing a driver with our evasion kernel and then taking a snapshot of the driver with its initialized state. *EASIER* then uses a CPU-only ARM emulator, which does not need to emulate any devices, to run the snapshot while injecting fuzzed inputs from AFL to search for vulnerabilities. We evaluate *EASIER* on a corpus of 72 platform device drivers from three different Android kernels and different kernel subsystems, and were able to successfully load and initialize 77% of them up to the point where analysis of IOCTL handlers was possible (i.e. we could execute driver code without crashes). When tested on 26 known vulnerabilities, *EASIER* was able to trigger 81% of them. We then used *EASIER* to fuzz the drivers and discovered a total of 29 vulnerabilities. Manual analysis of the discovered vulnerabilities showed that 12 are zero-day vulnerabilities in the Xioami Android kernel. We have reported and confirmed all of these and received bug bounties for 5 of them. The remaining 17 vulnerabilities are from the MSM kernel, but since they were found in an older version, we are currently in the process of verifying whether they are present in the latest kernel version before reporting them.

Our contributions. In summary we provide two separate contributions:

1) **The evasion kernel and framework.** We develop a set of techniques and a kernel that allow us to load and initialize an Android driver inside an alien environment and carry out dynamic analysis of its system call handlers. Such framework alone can be used to analyze,verify and prepare PoC for known bugs.
2) ***EASIER***. We introduce a way to transplant and analyze parts of the kernel in userspace and develop fuzzing and symbolic execution tools to analyze driver IOCTL system calls.

The rest of the paper is organized as follows. In Section II, we provide the necessary background on Linux kernel modules. We describe our evasion kernel in Section III and *EASIER* in Section IV. We then evaluate both evasion and *EASIER* in Section V. In Section VI, we discuss related work and Section VII concludes the paper.

II. Background

In this section, we give an overview of Linux kernel modules, device tree files, the platform bus, and the IOCTL system call.

A. Loadable kernel modules

In this paper, we target Android systems which are based on Linux. In Linux, most device drivers can be compiled as loadable kernel modules that can be dynamically linked into the kernel after the system has booted. Once a Linux module has been loaded it has the same privileges as any other kernel code and can compromise the kernel just like any other kernel code.

B. Platform bus and device tree files

Android smartphones today have most components/peripherals integrated into a single board. Such integrated peripherals can be a part of the SoC or use I2C or AMBA buses, none of which supports device discovery (as opposed to PCI or USB device buses). The Linux kernel uses a virtual "platform bus" for these integrated peripherals and system developers configure the kernel to manage the peripherals with a *device tree file*.

A device tree file describes hardware configuration for a specific board and provides a plain-text description for each

```
qcom,csid@fda00000 {
   compatible = "qcom,csid"; /* Device ID */
   cell-index = <0>;
   reg = <0xfda00000 0x100>;
   reg-names = "csid";
   interrupts = <0 50 0>;
   interrupt-names = "csid";
   qcom,csi-vdd-voltage = <1200000>;
   qcom,mipi-csi-vdd-supply = <&pm8110_l4>;
};
```

Listing 1: Device tree entry for `msm_ispf.ko`; the `compatible` property identifies the device.

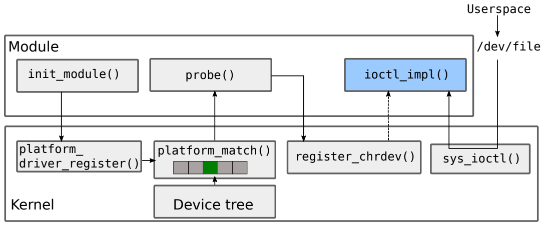

Fig. 1: Driver initialization flow

peripheral present. For example, Listing 1 shows one of the entries describing the Camera ISP (Image Signal Processor) for the Qualcomm Snapdragon 200 SoC. In particular, it includes properties that the kernel/driver will need during initialization such as MMIO memory ranges (field `reg`) and interrupt numbers (field `interrupts`). The compiled device tree file containing this entry is passed to the kernel at boot time (e.g. using `devicetree_image` option in the case of uBoot), which makes the kernel aware of what hardware devices are on the board. Each peripheral in the device tree file is identified by its `compatible` property.

Though device tree files are the predominant approach for conveying device information to the kernel, another, older option are *board files*, which describe devices in pure "C" and are compiled directly into the kernel. Board files are obsolete and used only by older kernels.

C. Module loading and initialization

When a module is loaded into the kernel, it first goes through a linking process that performs symbol relocation: modules may call functions from the host kernel at runtime, but the locations of these functions is not known during module compilation. Thus, after a module is loaded into the kernel, the kernel links these function calls to their implementations using module's relocation table. If the kernel is missing any of the functions required by the module, the module loading is aborted.

Once relocation is finished, the kernel calls several functions in the module to complete driver and peripheral initialization as shown in Figure 1. Each module defines an `init_module` function and optionally a `probe` function. The `init_module` function is executed by the kernel once relocation is finished. Usually, in the case of platform devices, this function registers the driver with one of the peripheral buses and provides a pointer to its `probe` function, for example by using the `platform_driver_register` API. The bus-related code then goes through the list of existing devices (pre-populated from a device tree file) and tries to find the matching device. The exact matching process is bus-specific; in the case of the platform bus, it matches the driver and the device based on: a) the `compatible` property in the device tree file; b) the device name; or c) driver ID's, whichever piece of information is present. If a matching device is found, the bus code calls the driver's `probe` function. The probe function usually creates a new file in `/dev/` folder and registers IOCTL/read/write system call handlers with `register_chrdev()`. System calls made on the device file are then handled by the registered handlers.

D. The IOCTL system call

In order to expose features of the driver and hardware devices to userspace programs, drivers register IOCTL system call handlers. The IOCTL system call provides more functionality than standard read/write system calls. It allows userspace to issue different commands to the driver, as well as transfer and receive arbitrary information in the form of C structures. Generally, IOCTL system calls contain a "command" field, which can be set to a number of values depending on the driver, where each value corresponds roughly to a feature of the peripheral, and an "argument" pointer, which can point to an arbitrary C structure whose format is defined by the driver and may depend on the command.

E. Kernel-userspace safe data copying

When a userspace program issues an IOCTL system call it provides an argument pointer. A driver can exchange data with userspace programs by copying data to and from the structure that the argument points to. When doing this, drivers use the kernel-provided `copy_from_user()`/`copy_to_user()` functions, which accept a pointer and the size of data to be copied to or from the kernel. These functions implement security checks to make sure that the pointer copied to or from falls within the address space of the process making the system call.

III. The Evasion Kernel

EASIER uses the *evasion kernel* to load and initialize "alien" drivers from foreign host kernels by "evading" missing software dependencies, missing hardware dependencies and API data structure incompatibilities. It is important that drivers are initialized before being passed to *EASIER* as drivers use initialization to setup their execution context, which includes important kernel structures, global variables the driver uses, kernel API structures used in system calls (for example the `file->private_data`), and resources such as MMIO memory ranges and interrupts. The driver's execution context is used extensively whenever the driver and the kernel try to speak through the kernel-driver API. If the context is not initialized and one still tries to execute some of the driver's

```
1  static long mdss_rotator_compat_ioctl(struct
        file *file, unsigned int cmd,
2  unsigned long arg)
3  {
4  ...
5  if (!rot_mgr)
6  return -ENODEV;
7  if (atomic_read(&rot_mgr->device_suspended))
8  return -EPERM;
9  if (!file->private_data)
10 return -EINVAL;
11 private = (struct mdss_rot_file_private *)file
        ->private_data;
12 if (!(mdss_rotator_file_priv_allowed(rot_mgr,
        private))) {
13 return -EINVAL;
14 }
15 ...
```

Listing 2: Code snippet of the `ioctl` handler from the `mdss_rotator.ko` driver.

functions, e.g. a system call, it will likely result in undefined behavior and false positive crashes that are not the result of true vulnerabilities.

To illustrate the problems that may arise if a driver is executed without a properly initialized context, consider the code snippet in Listing 2 that shows the beginning of the IOCTL handler for the `mdss_rotator.ko` driver (susceptible to CVE-2016-5344). At line 5, the handler checks if the global variable `rot_mgr` has been initialized. Later, at line 9, the `file->private_data` field maintained by the kernel is checked. If any of them is `NULL`, the handler returns immediately, and no analysis is possible. Note also that simply instrumenting the code and setting them to non-`NULL` values will not work: the variables are used later in the code (line 12), and thus should dereference to structures with properly set fields. Thus, our goal for the evasion kernel is to successfully initialize drivers.

Once a driver is loaded by the evasion kernel, *EASIER* can then extract the driver into userspace where fuzzing and symbolic execution can be performed. We describe the evasion kernel in this section and *EASIER* in the next section (Section IV).

A. Overview

The need for the evasion kernel comes from the fact that driver code is not self-contained code and in order to be able to run it needs to be loaded into a kernel; ideally, into the host kernel, i.e. the kernel against which the driver was compiled. However, as explained in Section I, the vast majority of Android host kernels cannot execute in an emulator. Instead, we modify an emulator-supported kernel so that it can load and initialize drivers using evasion, making it our evasion kernel. The evasion kernel used in our experiments is based on the stock Vanilla Linux kernel for the `vexpress` board for arm32 and `virt` board for arm64.

Before we discuss the challenges with loading a module into the evasion kernel, we first pose a simpler question: why not simply recompile drivers for the Qemu-supported kernel? Unfortunately, without porting, this can only be done in very simple cases since the driver relies on the host kernel's specific subsystems, header files, and configuration. While the effort needed to port the driver depends on the driver and the host kernel, in this paper we try to avoid any such porting that requires understanding of driver semantics all together. Instead of porting each driver, our goal is to develop a kernel that would handle missing driver dependencies in a generic way. We start by listing the evasion kernel requirements.

Evasion kernel requirements. Our ultimate goal is to be able to load alien drivers, execute their system calls without crashes, and reach real vulnerabilities. Since a driver lives in-between the kernel and the hardware, these are the only two components with which it can interact directly. Thus for the driver's proper operation we need to make sure that communications between the driver and the kernel and attempted communications from the driver to the hardware do not lead to a crash or failures and allow the driver to initialize correctly. Given that, our main goal translates into the following requirements. The evasion kernel should:

1) Satisfy driver's software dependencies,
2) Mask hardware dependencies, and
3) Ensure consistency in data structure formats passed between the kernel and the driver.

We note that the third dependency is also a software dependency, however its nature is quite different and solving it also requires a different approach. Because of this we put it into a separate category.

Each host kernel might have a different configuration and a unique set of dependencies. In order to account for that and achieve better precision in satisfying the dependencies, the evasion kernel is reconfigured for each host kernel (i.e. once for all drivers for this kernel) based on the host kernel's configuration. As a result, the evasion kernel contains components (i.e. scripts and logic) outside of the kernel itself, that are responsible for reconfiguring and in some cases, recompiling the evasion kernel with new configurations. However, for the sake of brevity, we will refer to the evasion kernel and such external tooling simply as the "evasion kernel". We now discuss each of the requirements and our design choices in detail.

B. Software dependencies

Drivers rely on the host kernel functionality and thus, when compiled as standalone modules, reference the host kernel's functions. At the same time, the evasion kernel should be generic enough to support loading and initializing different kinds of drivers from different host Android kernels. Consequently, as a trade-off, it will lack some of the functionality, and thus symbols, present in the host kernel and required by the driver. As a result, during the module's relocation phase, these symbols can't be resolved as they are simply not present. To solve this problem, the evasion kernel has a custom module loading subsystem that intercepts relocation requests for missing symbols and patches them with stub functions. It then continues on with the loading making the driver believe that the evasion kernel does implement the dependent function. Note that our main goal is not to completely replicate the behavior of the function in the original host kernel, but rather to prevent driver from crashing.

By default the evasion kernel substitutes missing functions with one of two stubs depending on the return type of the missing function. In the case of integer return type, the evasion kernel replaces the function with `stub0` that always returns zero. This is based on the common Linux kernel convention for functions to return zero on success, and a non-zero error code if an error happened. In case the function returns a pointer, the evasion kernel replaces it with `stubP` that allocates a memory region. This region in turn contains pointers to valid memory locations. This is to deal with cases when a function returns pointers to structures that in turn contain pointers. In order to infer missing function return types, the evasion kernel gets the list of the module's exported functions from the module's binary and checks their signatures in the host kernel source. It attaches this information as a part of the binary module itself as a new ELF section. The evasion kernel's custom module loading subsystem then uses this information to decide what function stub to choose.

Some functions may return non-zero code on success, and zero otherwise. The evasion kernel has a method to deal with this case. We define a third stub `stub1` which always returns 1 and patch relocation entries to use `stub1` instead of `stub0` if the module crashed or did not create a file in `/dev`. Only 2 out of the 72 drivers we analyzed requires `stub1` so we never automated this process. However, automating it would be straightforward: if a driver fails to load (i.e. a crash or no new file in `/dev`), the evasion kernel will try different combinations of `stub0` and `stub1` starting with the most recent relocation that was evaded. In order to give an upper bound estimate of the total number of combinations, the last column of Table V in the Appendix shows the total number of stub *uses* by drivers (i.e. includes repeated calls of the same function). In the 2 cases that needed `stub1`, changing the last evaded relocations enabled the driver to initialize.

C. Hardware dependencies

The main idea behind hardware evasion is the following: a) we make the driver believe that the peripheral exists so that the driver can finish initialization, and b) we then ignore each device-driver communication attempt. Though the core idea to evade the hardware dependency is conceptually similar for all drivers, the exact implementation would differ for different bus types. In this paper, we focus on the platform bus, which represents a large class of devices that do not support automatic discovery (e.g I2C or AMBA), and is found on the majority of Android and embedded devices today.

1) Reusing device tree entries: As detailed in Section II, an Android kernel reads the list of peripherals present on the board from a device tree file. This suggests that in order to make the driver believe that the missing hardware is present, we need to add an entry with the corresponding `compatible` property to the evasion kernel's device tree. During initialization, drivers will often query the host kernel to retrieve yet other properties for the devices they are interacting with. Like software dependencies, some of these properties can be generically evaded without knowledge of the device tree entries themselves. However, there are cases where driver code relies on specific assumptions about the device properties. For example, one value may index into another property, a value may need to be within a specific range (e.g. gpio type that can only take values of 0 and 1), a property may need to match a string exactly.

Our evasion kernel thus implements two ways for satisfying device property dependencies. First, if a device tree for the host kernel is available, the evasion kernel uses it since it is more reliable and still does not require any porting or a hardware device. However, blindly copying the device tree entry from the host kernel to the evasion kernel does not always work. The original device tree entry might contain cross-references to other hardware that is absent in the evasion kernel, for example different interrupt controllers or clocks. We thus design the evasion kernel to replace references to interrupt controllers and clocks with analogs that are present in the evasion kernel (i.e. in `vexpress`, the base kernel on which we base our evasion kernel). In order to identify the `compatible` property expected by the driver, which must match the returned device property (see Section II for an explanation of this property) the evasion kernel first loads the driver with hardware evasion disabled. The evasion kernel can then observe the `compatible` property provided by the driver, search the host kernel's device tree for the matching entry, and load it into its current device tree.

Second, if the host kernel's device tree file is not available, or the required device tree entry is not present, then the evasion kernel falls back on a generic device tree entry that returns reasonable values for the properties that are most commonly requested by drivers, `reg` and `interrupts` (describe MMIO ranges and IRQ numbers). If the driver requests another (unknown in advance) property it is generated dynamically with value of 1 for integer type and random string for strings.

2) Board files: Some older drivers expect to work with board files instead of device tree files. Board files have been deprecated for quite some time now and are only used in much older drivers. We still added support for board files to our evasion kernel but in its current implementation we need to extract the corresponding board file entries manually, after which we load them dynamically into the evasion kernel. This copying is mechanical and does not require understanding of the driver code. The only reason we did not implement the functionality to automate the copying was the small number of instances (only seven cases in our experimental dataset) where it was necessary.

3) Ignoring driver-device communications: During initialization, most of the drivers will use values from device tree entries to register MMIO ranges with `ioremap`/`ioremap_wc` or `of_iomap` which map the corresponding pages. The evasion kernel intercepts the above functions and redirects them

```
struct device {
...
#ifdef CONFIG_PINCTRL
  struct dev_pin_info    *pins;
#endif
  struct device_node     *of_node;
...
}
```

Listing 3: Definition of struct device. The offset of field of_node depends on kernel configuration

to kzalloc. This results in that read and write operations are ignored. In our case of missing peripherals, reading from this memory will return arrays of zeroes, and writes will pass through.

Finally, the evasion kernel intercepts and replaces with custom implementations 15 existing kernel functions whose behavior depends on the presence of peripheral devices. We list these differently from the replaced software dependency functions as these are functions that have existing implementations in the vanilla Linux kernel, whereas software dependencies replace functions that only exist in the host kernel. The list of intercepted functions can be found in Appendix A.

D. Kernel-driver API structures layout

In order to successfully load a driver into the evasion kernel we must ensure that both the evasion kernel and the driver have the same memory layouts for structures that are a part of kernel-driver API. Since drivers only use a subset of the kernel APIs we only need to extract layouts of structures that are actually used by the driver to speak to the kernel. We found that for many drivers that we analyzed, only two structures needed to be aligned between the module and the kernel, namely struct device and struct file. The former is used to pass information from the driver to the kernel during the driver initialization/probing, and the latter is used when the userspace opens a /dev file. Depending on the host kernel and its configuration, these structures will either include or lack some fields, and if the driver and the evasion kernel layouts do not match, the kernel and the driver will read/write information at different offsets (which will lead to a crash most of the time).

As an example, consider the definition of struct device in Listing 3. Depending on the CONFIG_PINCTRL configuration option, this structure contains an additional field which shifts the offset of member of_node by 4 (on a 32 bit machine). If the module was compiled with CONFIG_PINCTRL set and the kernel has CONFIG_PINCTRL unset, then the driver will assume the offset of of_node is 4 bytes larger than it should be and access the wrong memory location. This will lead to memory corruption and prevent the driver from initializing correctly.

In order to avoid this problem, structure layouts in the the evasion kernel must be compatible with those in the driver. While aligning these two structures layouts can be done manually, our evasion kernel uses a technique that enables it to do this alignment automatically by extracting the appropriate layout and recompiling the evasion kernel to use a layout that matches the driver. It learns the structure layouts used by the driver and then suggests configuration options and additional fields for the evasion kernel. With this information an external script can then reconfigure and recompile the evasion kernel with the new data structure layout. This is done by exploiting the fact that both the host kernel and the evasion kernel are Linux kernels, and thus will share a great deal of code that accesses the data structure whose layout needs to be extracted. In addition, we also have the source code for the evasion kernel. We describe our approach using only struct_device for the sake of brevity; recovering layouts of other structures is similar. First we identify a small set of Linux kernel functions, that a) accept struct_device as an argument, and b) use as many of its fields as possible. Currently, we use functions i2c_new_device, device_resume, and device_initialize for struct device layout recovery and __dentry_open for struct file recovery. If the source code for the host kernel is available we can skip this and simply make our own kernel module that lists all the fields of struct_device. The objective in either cases is to have binary code compiled from identical source code that accesses the same structure under both the evasion and host kernel configurations.

By comparing the resulting binaries, the evasion kernel can then recover corresponding field offsets for the structures in both itself and the host kernel. Then using the evasion kernel source code, the evasion kernel's external tooling identifies the necessary configuration settings to make the offsets identical. It then sets the build configuration and then recompiles the evasion kernel. In some cases, it is possible for the host kernel to contain a field option that is not present in the evasion kernel—simply having the host kernels .config file is not sufficient because of this. In addition, imprecision may result in case we don't use our own module but use existing functions that use only a subset of fields. In the first case, where a corresponding configuration option doesn't exist, the tooling will insert padding to cause the appropriate fields to align. In the latter case, the tooling generates several possible configurations. Currently, this requires manual intervention to compile and check which option works, but in our experiments, there was never more than one configuration, so manual intervention was not necessary. Moreover, we believe this trial and error process can be easily automated if necessary.

With our own module that lists all the fields, the evasion kernel was able to recover all kernel data structure layouts in our evaluation. We further tested the technique that uses existing kernel functions (i2c_new_device and others) on 8 different MSM kernel configurations that produce 5 different structure layouts and added 3 more manually selected layouts (see Appendix F). In all 8 cases, the correct configuration options and correct additional fields (i.e. where to add them and of what size) were produced.

E. Surrogate modules

In most cases, if the kernel initializes correctly, a device file in the kernel's `/dev` directory is created. This device file is needed later to send IOCTLs to the driver. However, an intermediate case arises when due to some missing dependencies the context is initialized only partially but sufficiently enough for the driver to work, but the device never creates the device file. This may happen for example if at the end of initialization the driver requires the peripheral to return a specific value, which fails since evasion is imprecise and cannot account for all possible values. Without the device file, a user process has no way of invoking the driver's system call handlers, which prevents them from being analyzed. In case no device file is created, the evasion kernel provides functionality to create a "surrogate" device file and attach the driver's system call handlers to the device (by looking up the handlers in the memory with `kallsyms_lookup_name()`). This allows *EASIER* to invoke the driver's system call handlers from the userspace and analyze them.

IV. Fuzzing and symbolic execution with *EASIER*

Once the driver is loaded and initialized with missing dependencies taken care of, we can manually analyze the driver, verify vulnerabilities and prepare proof-of-concept exploits for known bugs or bugs found via static analysis. However, our real objective is fuzzing and symbolic execution of drivers for automatic bug discovery. *EASIER* enables this by "running" drivers as userspace programs.

EASIER first extracts the initialized state of the driver and kernel into a memory snapshot. Then inputs from an input file are read and injected into the memory snapshot and the image is loaded and executed in a custom CPU-only emulator called dUnicorn which is based on the Unicorn library [30]. To execute the extracted memory snapshot as a userspace program, certain kernel functions, which cannot execute without hardware are replaced by our custom equivalents. Additionally, because each driver has its own IOCTL input format, dUnicorn dynamically infers the format of the IOCTL inputs so that our analysis can produce semantically valid inputs. Using the image and dUnicorn, *EASIER* can proceed to perform fuzzing and symbolic execution of IOCTL system calls, identify vulnerabilities, and automatically generate bug-triggering programs. We describe each of the components in more detail in this section.

A. Memory snapshot

EASIER takes a snapshot of the entire evasion kernel image by first issuing a system call (using a userspace program running on the evasion kernel). Once execution enters the evasion kernel *EASIER* pauses the emulator and dumps all memory pages and register values (see Appendix B-A for implementation details). Our *EASIER* prototype currently uses Qemu as the emulator to run the evasion kernel and produce the snapshot.

B. dUnicorn

To run the snapshot, dUnicorn loads it into emulated memory and restores the saved register values. It then uses an input file to write specified values into memory and registers. For example, when executing a system call handler in a driver (e.g. IOCTL, read, write), dUnicorn reads the input file and uses its content as the argument to the system call. This is done by copying the file content into unused space in the emulated memory, and setting the corresponding register to point to that memory. This allows one to test the system call against different potentially malicious inputs. Control is then transferred to the system call entry point where the snapshot was taken. dUnicorn then emulates CPU instructions up to the point where it is about to leave the system call and return back to userspace (specifically, once the execution reaches `ret_fast_syscall`). If some instruction tries to access emulated memory that was not mapped, dUnicorn raises a SEGFAULT event. In other words, dUnicorn runs as a userspace program that crashes on some input only if the driver inside the kernel would crash on the same input, and finishes successfully otherwise.

The biggest advantage over static analysis is that in our approach the execution context and kernel structures are defined and initialized. Thus, the analysis becomes much more precise. For example, alternatives like under-constrained symbolic execution [10] (UC) treat all uninitialized state as symbolic, and then later tries to find both an input and a state that can trigger a bug. However, for complex programs like an OS kernel, the symbolic state can become large and lead to path explosion. Moreover, the large state can also become beyond what can be solved by a constraint solver. *EASIER* elides these problems by producing an initialized state that is precise enough for dynamic analysis.

C. Replacing kernel functions

As a CPU-only emulator, dUnicorn only emulates ARM instructions and no other hardware. Thus, dUnicorn needs to intercept and redirect accesses to hardware that dUnicorn doesn't emulate. The intercepted functions fall into three main categories. First, since there is no MMU in CPU-only emulation, the kernel code is not able to map new physical pages. In most cases, this does not create any difficulties as all the memory that the driver uses or is going to request is likely to be already mapped by the slub allocator. Nonetheless if a driver does allocate memory after initialization, dUnicorn intercepts and replaces memory allocation routines such as `kzalloc`, `krealloc`, and `kfree` with calls to a simple custom memory allocator that allocates chunks from unused emulated memory. Second, dUnicorn intercepts calls that switch context such as `_cond_resched`. Context switches are currently treated the same way as a successful return to userspace. Finally we can have dUnicorn optionally disable logging and output functions such as `printk`. Such logging functions can usually be safely skipped as they perform no useful function for the analysis and take long time to execute. In fact, we find that removing print functions can speed

```
struct msm_vfe_cfg_cmd2 {
  uint16_t num_cfg;
  uint16_t cmd_len;
  void __user *cfg_data;
  void __user *cfg_cmd;
};
```

Listing 4: IOCTL structure for `msm_isp` driver.

out the dynamic analysis of some drivers by as much as 3 times. However, this optimization has a trade-off of missing some read-past-the-buffer bugs when a char array was not properly null-terminated. dUnicorn is general enough to be able to intercept and redirect execution of any kernel function to a custom implementation, and we hypothesize that this functionality could be used in the future for other types of dynamic analyses.

D. IOCTL input format recovery

While the approach implemented in *EASIER* is generic and can be applied to testing any kernel component, we specifically target IOCTLs in this work, which requires additional information for automatic fuzzing or execution. System calls such as write and read only require providing a contiguous array and size as input, and differ from IOCTLs, which have more requirements on their inputs. Recall that an IOCTL system call has the following format:

`ioctl(int fd, long cmd, void *arg),`

The first argument is a file descriptor tied to the driver's device file in `/dev` and the second argument is an IOCTL command number. Finally, the last argument points to an arbitrary structure in memory whose format is again defined by the driver. The final argument is particularly complex as there is no way to tell beforehand what size the buffer that `*arg` points to should be. If it is too short, the driver will not get enough data and will most probably go to error handling code. If it is too long, a fuzzer will waste time mutating data that is never used by the driver. Moreover, the structure in `*arg` can embed pointers to other structures and arrays. Nested arrays can have dynamic sizes, and the fields responsible for their sizes are specific to the IOCTL command and driver.

To illustrate, consider Listing 4, which is a typical example of the format of an `*arg` input for a driver. The last two fields are pointers and should point to valid memory locations. If they are set to random values (e.g. by a fuzzer), the driver will fail while trying to access/copy this memory and is likely to immediately go to error handling code. In addition, the size of the data they point to is dynamic and depends on other two fields, `num_cfg` and `cmd_len`. Nested structures can in turn contain pointers to other structures and so on. As a result, it is important to extract which parts of the input to an IOCTL call can be set to arbitrary values and fuzzed, and which parts must have semantically correct inputs in order to not trigger spurious errors and enable exploration of driver code.

While other work has proposed extracting the layout of these structures statically, such an approach is necessarily incomplete as it cannot extract the size of dynamically allocated objects and arrays in the structure [8]. Instead, *EASIER* recovers IOCTL structure layouts dynamically. As a result, during fuzzing, one can issue an IOCTL system call with `arg` set to arbitrary values and dUnicorn will dynamically allocate the right sizes and insert pointers at correct places. The key observation behind our approach is that in order to copy data between userspace and kernel space drivers must use `copy_from_user()` kernel API call (not using it is a bug by itself that dUnicorn can catch). Functions from this family take as arguments, an address in userspace that points to data to be copied, and an integer giving the size of the data to be copied. dUnicorn uses this information to recover the format of the data that the driver expects at runtime as it makes these `copy_from_user()` calls. dUnicorn intercepts calls to `copy_from_user()` (see implementation details in Appendix B-B) and redirects them to its own implementation which dynamically, and on the fly, allocates the desired amount of memory, fills it with random data and returns it to the driver. This exploits the fact that if a pointer appears in `copy_from_user()` function, then the driver expects userspace data to be present at this location.

Consider an example in Figure 2. Assume the userspace calls `ioctl` libc function and sets `arg` to `0x10000000`[2] as shown in Figure 2a). When the driver calls `copy_from_user()` for the first time it will use this pointer as the first argument and `len1` as the size of the structure it expects (or array of structures). dUnicorn intercepts this call, pauses the emulation, and allocates `len1` bytes of emulated memory and fills it with random data as shown in Figure 2b). dUnicorn then continues execution.

Assume now that the driver expects a pointer at some (unknown to us) offset d inside the data it just copied (Figure 2c). It will try to use the (random) value from this offset, also stored in `src2`, in another call to `copy_from_user()`. dUnicorn intercepts this call and searches for `src2` inside the copied data; once found it gives us the offset d. dUnicorn then allocates new space of size `len2` and updates the values of both `src2` and the memory slot at offset d correspondingly (see Figure 2d). Note that our approach is insensitive to how `len2` was computed or whether it depends on other user input. Such dependencies can pose a problem to static analysis, but not for our dynamic recovery. dUnicorn continues this algorithm recursively which allows it to allocate the right amount of memory and deal with nested pointers.

E. Fuzzing

Now that *EASIER* can use dUnicorn to run the memory snapshot as a userspace program, instead of returning random data it can return fuzzed values from the fuzzer. Moreover, *EASIER* can use any userspace fuzzer to produce those values. In our implementation, we support coverage-based fuzzing using AFL, more specifically AFL in unicorn mode [31] (a.k.a.

[2] Note that since we are working on a memory snapshot we can treat userspace and kernel space memory separation to our liking.

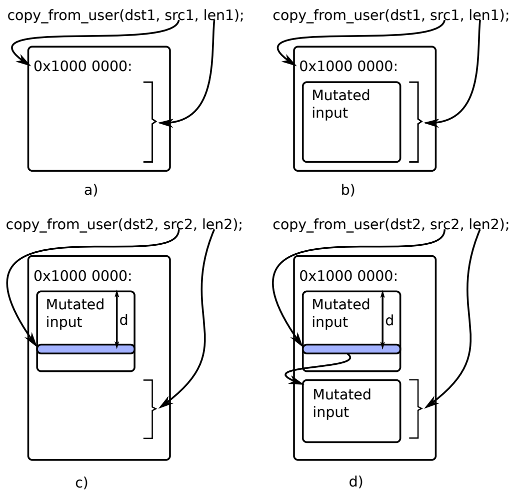
copy_from_user(dst1, src1, len1);
0x1000 0000:
a)
copy_from_user(dst1, src1, len1);
0x1000 0000:
Mutated input
b)
copy_from_user(dst2, src2, len2);
0x1000 0000:
Mutated input
d
c)
copy_from_user(dst2, src2, len2);
0x1000 0000:
Mutated input
d
Mutated input
d)

Fig. 2: IOCTL structure recover details

AFL-Unicorn). This mode provides integration with Unicorn library similar to how AFL-qemu mode integrates with Qemu and captures emulated basic blocks/edges.

AFL executes dUnicorn as any other userspace program. dUnicorn then does all the job of copying mutated input to proper memory locations, catching unmapped memory errors and raising SEGFAULT signals, dynamically intercepting and rewriting function calls, and dynamically recovering IOCTL structures.

F. Symbolic execution

With symbolic execution our goal is similar: being able to symbolically run a kernel memory snapshot. Our implementation is based on the Manticore [16] framework. Manticore is tailored to symbolically execute userspace ELF binaries and cannot run kernel code. We extend it and add a new mode that allows for restoring the execution state from the memory snapshot. Similar to dUnicorn, our symbolic execution tool allows for dynamic binary rewriting and replacing kernel functions with custom implementations, specifically we replace memory allocation functions with our custom memory allocator.

The symbolic execution component is generic and can potentially be used for different tasks (for example discovering new paths when the fuzzer gets stuck). In our implementation, we used it to develop a pass that recovers IOCTL command numbers to be used in the `cmd` argument. Recovering IOCTL commands was necessary since they are usually hard to guess by the fuzzer. Our approach to recovering IOCTL command numbers is based on the common convention of having a large `switch` statement inside IOCTL handlers. Each switch case is usually compiled into a conditional branch instruction. We assign register `r1` (containing `cmd` argument) a symbolic value and every time the execution state forks inside the IOCTL handler (but not inside its callees), we call the solver to produce a counterexample.

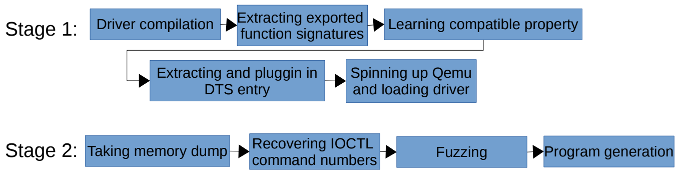
Stage 1:
Driver compilation
Extracting exported function signatures
Learning compatible property
Extracting and pluggin in DTS entry
Spinning up Qemu and loading driver
Stage 2:
Taking memory dump
Recovering IOCTL command numbers
Fuzzing
Program generation

Fig. 3: Experimental workflow

G. Program generation

For each of the IOCTL commands we launch an AFL/dUnicorn instance. If crash is found, our system uses the input that triggered the crash and the recovered IOCTL structure layout to automatically generate a C program that will trigger the crash when run on the real kernel. We have found this very useful when preparing a proof-of-concept code for a vulnerability report or filtering out false positive crashes. In fact, we were able to submit and get accepted, working proof-of-concept code for vulnerabilities in our reports without ever having the device on which the proof-of-concept is supposed to run.

V. Evaluation

In this section, we pose three questions: 1) whether our hypothesis about superficially dependent paths is correct and we can reach real vulnerabilities; 2) whether the evasion techniques are sufficient to load and initialize drivers; 3) whether we can find new bugs and how many false positive crashes we get during fuzzing. We start by describing experimental workflow, describe our experimental results and discuss the limitations of evasion.

A. Workflow

Figure 3 shows the sequence of steps that we followed during our experiments for each driver. Each individual step was automated except for driver compilation: our framework was designed to work directly on binary kernel modules (i.e. .ko files) but most Android drivers are currently compiled directly into the kernel rather than as modules. Compiling drivers as modules did not require any changes in the driver source. We had to either modify the corresponding Makefiles (by changing `obj-y` to `obj-m` directives) or navigate through the menuconfig. When modifying Makefiles, sometimes it might not be immediately clear what source files constitute a driver, this however can be solved through trial and error.

Going from one step to another requires providing the output of one step as input to the next step manually. Gluing all the steps together is mechanical (i.e. identical for all drivers and does not require any additional analysis by the analyst) and can be done with moderate engineering effort.

B. Reaching existing vulnerabilities

We start by answering: given that we can load some drivers, do superficially dependent paths contain real vulnerabilities and can we reach them? To answer this question, we looked at existing vulnerabilities in IOCTL system calls. We wanted to verify that we can reach vulnerabilities specifically in Android drivers (and not only in Vanilla Linux kernel drivers on which the evasion kernel is based on). Thus, we create a corpus of known vulnerabilities in Android device drivers that are not present in the vanilla Linux kernel and cannot be trivially compiled against it.

In order to find CVE's we used `cvedetail.com` database with "android" keyword and chose the first 21 CVE's that satisfied the above criteria. These CVE's belonged to 10 different drivers from Android for MSM kernel[3] that could be loaded and initialized by the evasion framework. The vulnerable versions of the drivers were for MSM kernel 3.4, 3.10, and 3.18, we thus prepared evasion kernels based on the same versions of the Vanilla kernel.

We also include 5 additional bugs (lines 22-26) that were known to the developers and patched in the latest version of the MSM kernel but not publicly reported as CVE's. We describe each of these 5 bugs in more detail in Appendix E. For each bug in our corpus, we created a test program that can inject an input that should trigger each of the bugs and confirm whether the bug is triggered or not.

Overall we were able to reach existing bugs in 21 (80%) of the 26 cases. This indicates that vulnerabilities often pertain to superficially dependent paths and confirms that *EASIER* is able to trigger them without having to precisely emulate either the host kernel or the hardware device. We describe some of the interesting CVE's in Appendix D.

C. Loading drivers in the evasion framework

We now evaluate the ability of the evasion framework to successfully load and initialize Android platform device drivers without peripherals and without porting drivers. To evaluate this, we use 62 drivers from three different kernels: the MSM kernel, the Xiaomi Redmi 6 kernel, and the Huawei P20 Pro kernel. Specifically, we searched for platform drivers that appear in the MSM kernel and are not present in the vanilla Linux kernel and then selecting the first 20 that contained IOCTL system call handlers (note that these drivers are different from those in the previous experiment). We were able to compile all of these as modules. We chose Xiaomi drivers by selecting drivers that contain an IOCTL system call handlers and are not present in the Vanilla kernel, resulting in another 50 drivers. Among those, we were able to compile 32 as modules. The most common reason for not being able to compile a driver was that the kernel source tree was missing header files included by the driver. We believe these drivers were not actually a part of the Xiaomi kernel but were left there from previous kernels and never meant to be compiled. We used a similar procedure for the Huawei kernel and added another 10 drivers to bring the total to 62 drivers. The drivers came from a variety of kernel subsystems, such as camera, network, radio, USB, video subsystem, and others. Driver loading experiments were conducted on machines running Ubuntu 16.04/18.04 (with a 4-core 2.70GHz CPU and 8 GB of RAM), Qemu version 3.10, and evasion kernels based on the Vanilla kernels 3.4 and 4.9. We now evaluate the success rate of driver initialization by the evasion kernel, analyze reasons for failures to initialize and outline how often the `stub1` evasion function was needed instead of `stub0`.

	CVE #	Module	Reach
1	CVE-2014-9785	qseecom.ko	✓
2	CVE-2014-9891	qseecom.ko	
3	CVE-2014-4322	qseecom.ko	
4	CVE-2014-9894	qseecom.ko	
5	CVE-2012-4220	diagchar.ko	✓
6	CVE-2015-9863	diagchar.ko	✓
7	CVE-2015-9863-1	diagchar.ko	✓
8	CVE-2014-9875	diagchar.ko	✓
9	CVE-2014-9782	msm_actuator.ko	✓
10	CVE-2014-9786	msm_actuator.ko	✓
11	CVE-2014-9777	vidc_vdec.ko	✓
12	CVE-2014-9880	vidc_vdec.ko	
13	CVE-2016-5344	mdss_rotator.ko	✓
14	CVE-2014-9866	msm_csid.ko	✓
15	CVE-2016-3903	msm_csid.ko	✓
16	CVE-2014-9867	msm_isp.ko	✓
17	CVE-2015-8941	msm_isp.ko	✓
18	CVE-2014-9871	msm_isp.ko	✓
19	CVE-2014-9868	msm_csiphy.ko	
20	CVE-2014-9882	iris-radio.ko	✓
21	CVE-2014-9881	iris-radio.ko	✓
22	diag-crash-1	diagchar.ko	✓
23	diag-crash-2	diagchar.ko	✓
24	actuator-crash-1	msm_actuator.ko	✓
25	actuator-crash-2	msm_actuator.ko	✓
26	isp-crash-1	msm_isp.ko	✓
		Total	21/26 (80%)

TABLE I: *EASIER*'s abaility to reach known vulnerabilities. The bug marked as `CVE-2015-9863-1` was unreported but similar in nature to `CVE-2015-9863`.

1) Success rate: The summary of the results is shown in Table II. For each driver we indicate whether we were able to load it, as well as the size of the driver. Out of 62 drivers, the evasion kernel was able to successfully load and initialize 48 drivers (77%). Among those, for 2 drivers (drivers #4 and #17) the last, `open`, stage failed with a crash, in which case the evasion kernel completed loading by using the surrogate module to which the IOCTL handlers of the driver were attached.

2) Reasons for initialization failure: We analyzed the reasons for initialization failure and observed that failures often happened either in `init` or `probe` functions due to evaded functions that had required functionality, such as initializing fields of a struct, that our stubs did not perform. This caused the driver to crash, skip the creation of the `/dev` file, or to go to error handling code that deleted the previously created `/dev` file. For example, in the case of the `dwc3-msm.ko` USB controller driver, the `probe` function finished successfully, but the execution did not take the path that creates the

[3] A fork of Android Open Source Project containing additional enhancements for Qualcomm chipsets.

#	Module	LOC	Init	#	Module	LOC	Init	#	Module	LOC	Init
MSM kernel v3.4											
1	avtimer.ko	369	✓	8	msm_rotator.ko	1,850	✓	15	adsprpc.ko	1,287	✓
2	msm_adc.ko	1,533	✓	9	msm_serial_hs.ko	3,447		16	vidc_venc.ko	4,149	✓
3	msm_led_flash.ko	273	✓	10	msm.ko	1,056	✓	17	msm_gemini.ko	2,399	∓
4	msm_jpeg.ko	2,303	∓	11	msm_ispif.ko	1,085	✓	18	msm_cpp.ko	1,951	✓
5	msm_vpe.ko	1,645		12	msm_rmnet.ko	841	✓	19	msm_rmnet_bam.ko	1,013	✓
6	msm_rmnet_sdio.ko	713	✓	13	msm_rmnet_smux.ko	934	✓	20	msm_rmnet_wwan.ko	750	
7	qfec.ko	3,056		14	dwc3-msm.ko	3,217					
Xiaomi Redmi6 kernel											
21	mtk_disp_mgr.ko	1,958	✓	32	btif.ko	7,554	✓	43	flashlight.ko	1,902	✓
22	flashlights-dummy.ko	689	✓	33	flahslights-leds191.ko	544		44	flashlights-lm3642.ko	728	✓
23	vcodec_kernel_driver.ko	4,996	✓	34	teei.ko	7,838		45	ccu_drv.ko	1,468	✓
24	dfrc.ko	1,920	✓	35	mtk_extd_mgr.ko	5,474	✓	46	mtk_auxadc.ko	2,000	✓
25	sub_lens.ko	2,883	✓	36	main2_lens.ko	4,081	✓	47	sub2_lens.ko	670	✓
26	main_lens.ko	15,624	✓	37	camera_isp.ko	15,330	✓	48	camera_dpe.ko	5,053	✓
27	camera_rsc.ko	3,660		38	3.5/camera_fdvt.ko	1,389	✓	49	5.0/camera_fdvt.ko	4,569	✓
28	4.0/camera_fdvt.ko	1,396	✓	39	camera_dip.ko	5,028	✓	50	sec.ko	399	✓
29	mtk_irtx_pwm.ko	412		40	eeprom_driver.ko	1,406	✓	51	jpeg_drv.ko	3,176	
30	imgsensor.ko	2,784		41	cmdq.ko	17,959		52	mmprofile.ko	2,429	✓
31	mtkbattery.ko	11,491	✓	42	mtkcharger.ko	5,079	✓				
Huawei P20 Pro kernel											
53	fingerprint.ko	2,284		57	maxim.ko	618	✓	60	tfa98xx.ko	497	✓
54	usb_audio_power.ko	593		58	usb_audio_power_v600	192	✓	61	usb_audio_common.ko	205	✓
55	anc_hs_default.ko	236	✓	59	anc_hs.ko	1,517	✓	62	ext_modem_power.ko	1,244	✓
56	hwcam_module.ko	5,471	✓								

Total successful: 48/62 (77%)

TABLE II: Evaluation of driver initialization. ∓ – surrogate module was used.

/dev file. From our analysis, it appeared that some of the missing functionality could be extracted from the host kernel and added to the stub functions in the evasion kernel, but we leave the exploration of this functionality for future work.

3) Using `stub1`*:* In the vast majority of cases, the evasion kernel's stub function returns 0 and this is the value expected by the driver. However, two drivers, `vidc_vdec.ko` and `mdss_rotator.ko` from Table I requires `stub1`. In both cases, the most recent evaded relocations needed to be changed, thus requiring only one attempt for the first driver, and two attempts for second driver to correct the issue.

D. Fuzzing results

Finally, we evaluate the ability of our framework to discover new bugs, the false positive rate due to evasion and the speed at which *EASIER* supports fuzzing.

1) Evaluation set: We fuzz a total of 32 drivers: all of the 24 Xiaomi drivers that loaded successfully and 8 MSM kernel drivers from section V-B. Since our current implementation of dUnicorn is for 32bit ARM binaries, and Huawei drivers could only be compiled for ARM64 we currently cannot fuzz them. We plan to add support for fuzzing arm64 binaries in future versions. We fuzzed each driver between 12 hours and 2 days on an 8-core machine with 8GB of memory running Ubuntu 18.04. In total we fuzzed the drivers for 715 hours.

2) Bugs discovered: We discovered bugs in the drivers of both the Xiaomi and MSM kernels. The number of bugs and their types are shown in Table IV. We found a total of 12 new bugs in the Xiaomi drivers, all of which were confirmed by the Xiaomi security. We also received bounties for 5 of them (submitted as 4 reports, one report combining two bugs into a single read-write primitive).

We also found a total of 17 bugs in the MSM kernel. These bugs were not known to us at the time of the experiments. However because an older version of the MSM kernel was used in our experiments (we originally used the MSM kernel to test known vulnerabilities), we cannot be sure they are zero-days. We are in the process of checking whether the bugs are previously known, silently fixed or still present in the most recent MSM kernel version.

Finally, our fuzzing experiments also used a checker that detects cases where a userspace application can cause a driver to attempt to allocate arbitrarily large memory buffers with `kmalloc`. This checker found 13 unbound `kmalloc` uses for Xiaomi and 1 unbound `kmalloc` for MSM kernel.

During fuzzing we were able to recover all IOCTL structures and catch all field interdependencies. For each of the bugs we automatically generated a C program that we used to triage bugs and prepare vulnerability reports

3) False positives: We observed 1 false positive in the MSM kernel and 4 false positives for the Xiaomi kernel (see

Table IV). Three of the Xiaomi false positives are due to a mismatch in the `struct pm_qos_request` definition between the Xiaomi kernel and the evasion kernel. Upon manual analysis we found that the Xiaomi's version included an additional field, resulting in an incompatibility between the driver and the evasion kernel that resulted in a false crash. Our current evasion kernel only guarantees the same layouts of common structs like `struct device` and `struct file` and did not handle this structure, though it could be extended to do so.

The fourth Xiaomi false positive was due to a loop where the driver reads a value from the device until it gets a non-zero value. Since *EASIER* always returns a zero whenever the driver tries to read from the non-existent peripheral the loop never terminates. Note that this issue can also be classified as a bug: ideally a kernel should not hang simply because of a malfunctioning peripheral device.

The one false positive for MSM kernel was a driver that failed to initialize completely, but still generated a device file so it appeared to have been properly initialized. In reality, a variable had not been properly initialized and then was used by the driver's IOCTL handler.

4) Fuzzing rate and execution paths: We tabluate the time spent fuzzing, number of code paths discovered and rate of fuzzing for each driver in Table III. On the average, *EASIER* fuzzed MSM kernel drivers at 1,167 executions per second and Xiaomi drivers at 525 executions per second (on an 8-core machine). The difference between the two kernels is due to the difference in snapshot sizes: 37Mb and 205Mb correspondingly. This fuzzing speed is an improvement of 1-2 orders of magnitude over previous hybrid systems such as Charm [29], which achieves roughly 20 executions per second by fuzzing drivers on a 16-core machine, but still forwarding low-level operations to be run in-vivo on a real device. Another advantage on our system was that the lack of a physical device removed the need to re-initialize or restart the device when needed (after a crash for example [8]).

On some drivers, fuzzing produced a low number of discovered paths. We found that it often was due to magic numbers that the fuzzer was not able to find rather than due to *EASIER* or the evasion kernel. For example we often observed that after IOCTL command switch statement drivers use yet another switch statement on user input subfields. We plan to use our symbolic execution tool together with the fuzzer to deal with this cases in future work.

5) Halts during context switch: As mentioned in Section IV-C, a driver may invoke a context switch if it needs to wait for a value from the peripheral. Since dUnicorn cannot simulate a context switch, it halts if this happens. Such context switches might prevent exploring new paths in the driver. During fuzzing, such halts happened only in 13/267 IOCTL commands for Xiaomi drivers and 3/137 IOCTL commands for MSM drivers.

Module	**Time, hrs**	**Paths**	**Speed, exec/s**
MSM kernel			
diagchar.ko	48	37	1357
qseecom.ko	49	42	1126
msm_isp.ko	50	251	1250
msm_csiphy.ko	17	14	1130
radio-iris.ko	16	172	1217
msm_actuator.ko	51	140	1188
vidc_vdec.ko	51	52	952
msm_csid.ko	20	32	1117
Xiaomi kernel			
mtk_disp_mgr.ko	19	58	485
mtk_btif.ko	22	11	671
mtk_flashlight.ko	22	36	932
ccu_drv.ko	20	90	537
camera_isp.ko	20	422	578
flashlights-dummy.ko	15	63	621
flashlights-lm3642.ko	15	63	711
vcodec.ko	14	40	617
dfrc.ko	16	54	458
mtk_extd_mgr.ko	17	21	501
mtk_auxadc.ko	17	5	548
subaf.ko	17	9	455
main2af.ko	17	9	466
sub2af.ko	17	7	466
mainaf.ko	17	8	393
camera_fdvt_v3.5.ko	12	118	441
camera_fdvt_v4.0.ko	23	124	520
camera_fdvt_v5.0.ko	12	145	527
camera_dip.ko	12	208	527
sec.ko	12	4	431
eeprom.ko	19	86	275
mmprofile.ko	19	16	638
mtkbattery.ko	19	19	691
mtkcharger.ko	20	1	121

TABLE III: Fuzzing statistics

Type of bug	**# MSM kernel**	**# Xiaomi kernel**
Memory read	0	3
Memory write	1	2
Buffer overflow	1	0
Out-of-bound index	6	4
Unchecked user pointer	6	0
`NULL` dereference	1	1
`ZERO_SIZE_PTR` deref.	1	1
Buffer overread	1	1
False positives	1	4
Total (excluding FP)	**17**	**12**

TABLE IV: Types of bugs found. Memory read/write bugs include bugs that allow for either arbitrary read/write or memory read/write below a certain address.

E. Limitations

Our framework has three main limitations. First, because of the lack of actual hardware it is currently most suitable for system call analysis, i.e. when the malicious input comes from userspace applications. It cannot reliably be used to find vulnerabilities that are exploitable by a malicious/compromised peripheral or an attacker who sends malicious input to the device. Second, our current implementation does not support interrupts and only supports platform and I2C buses in the case of ARM32 and platform bus in the case of ARM64. Finally, evasion can produce false positives, which is usually not the case for fuzzing.

VI. Related work

Charm [29] deals with the problem of missing peripherals by redirecting the corresponding I/O calls to the actual physical device through USB. Such an approach, while more precise in terms of emulation, requires porting the driver to a specific version of the kernel and also requires the physical device to be present. As reported by its authors the time required to port a driver for an experienced kernel developer varies between two days and two weeks. Our approach does not require driver porting and having access to a physical device. Our approach can also work for drivers that come as binary only. Avatar [32] is similar in spirit to Charm but works for low-level embedded firmware such as hard drive firmware.

FIE [9] analyzes self-contained barebone firmware for MSP430 microcontrollers by extending KLEE [2], a fully symbolic environment. In order to deal with missing peripherals, every time the firmware tries to access a hardware through memory-mapped registers, the framework returns either a new symbolic value or a constant. Such an approach is well suited for small barebone programs that access hardware using memory-mapped registers. In contrast, Linux kernel drivers are not self-contained code and use a diverse set of kernel API to access peripherals. Additionally, FIE is tailored for a limited set of MSP430 microcontrollers and peripherals. In the case of Linux drivers there are many more peripherals, and for many of them specification is not available.

Keil and Kolbitsch [12] focus on testing WiFi drivers in Qemu by emulating an IEEE 802.11 device. Periscope [24] instruments DMA buffers inside kernel to inject fuzzed data into WiFi drivers. Ma [14] develops emulated versions of USB devices and fuzz drivers against these emulated devices. Mueller [17] uses a Qemu provided virtual USB device. Schumilo et al [21] use USB redirection protocol to provide access to remote USB devices. Patrick-Evans et al [19] develop an emulation of a generic USB device. In the current work we target a wider range of device drivers and do not require emulated or physical peripherals.

Triforce [11] is a modified version of AFL that supports fuzzing using QEMU's full system emulation for x64 architecture. S2E [5] allows for symbolic execution of the full kernel stack and is tightly coupled with the KLEE symbolic executor. In this work, we support ARM, we deal with missing hardware, automatically recover `ioctl` system call structures, and allow any userspace fuzzer or binary symbolic execution tool to be used.

DR. CHECKER [15], Coccinelle [27] and Coverity [1] use static analysis to find bugs in Linux kernel drivers. The upside of static analysis is that it alleviates the need to have peripherals. The downside is that it is also imprecise and is known to produce a large amount of false positives that require further manual analysis. Static analysis also limits the analysis to bug finding only. In contrast, our goal is to enable dynamic analysis. Our approach provides a different set of capabilities such as coverage-guided fuzzing, symbolic execution, and interactive debugging. Since dynamic analysis produces a concrete input for a potential vulnerability, it can be mechanically tested on the real device for false positives.

Firmalice [23] statically analyzes embedded binary firmware images and identifies authentication bypass vulnerabilities. Firmadyne [3] emulates user space applications extracted from embedded devices. In this work, we target dynamic analysis instead, and we focus on kernel-level vulnerabilities.

VII. Conclusion

The main challenge to ex-vivo dynamic analysis of device drivers is the software dependencies they have on their host kernel, and the hardware dependencies they have on the hardware device they are supposed to manage. In this paper, we make the observation that for many execution paths in drivers these dependencies are in fact superficial. For example, such paths may only depend on the ability to read a memory-mapped device register, but not on the actual value returned. We hypothesize that a possible solution to ex-vivo dynamic analysis for such paths is *evasion*, where a specially constructed evasion kernel satisfies those dependencies by evading them.

To test this hypothesis, we developed an evasion kernel that can load and initialize platform device drivers and an *EASIER* tool that can then extract and run those drivers for analysis as userspace processes. We find that using evasion, our kernel is able to successfully initialize 48/62 (77%) of foreign platform drivers. Moreover, *EASIER* can trigger 21/26 (80%) vulnerabilities, showing that one does not need detailed porting or emulation to find bugs. Finally, to fully test the hypothesis, we fuzzed 32 drivers for a total 715 hours and found a total of 29 bugs, 12 of which have been confirmed to be new bugs. From this, we conclude that evasion and *EASIER* make Android ex-vivo driver analysis possible without porting or hardware (either real or emulated). Moreover, we conclude that these techniques are effective for discovering and analyzing vulnerabilities.

Our approach allows for dynamic analysis of Android drivers with sufficiently high precision and without requiring physical nor emulated devices. We believe that the ability to work without the need to use complex debugging interfaces or to reflash the device and the ability to analyze the driver in userspace will help to lower the bar for Android kernel driver analysis.

VIII. Acknowledgements

The research in this paper was made possible with generous support from Telus Corporation and an NSERC CRD Grant 535902-18.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[2] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[3] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for Linux-based embedded firmware,” in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf

[4] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. Frans Kaashoek, “Linux kernel vulnerabilities: State-of-the-art defenses and open problems,” *Proceedings of the 2nd Asia-Pacific Workshop on Systems, APSys’11*, 07 2011.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950396

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’01. New York, NY, USA: ACM, 2001, pp. 73–88. [Online]. Available: http://doi.acm.org/10.1145/502034.502042

[7] C. Cimpanu. (2019) Android exploits are now worth more than iOS exploits for the first time. [Online]. Available: https://www.zdnet.com/article/android-exploits-are-now-worth-more-than-ios-exploits-for-the-first-time

[8] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, “DIFUZE: Interface aware fuzzing for kernel drivers,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[9] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, “FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[10] D. Engler and D. Dunbar, “Under-constrained execution: Making automatic code destruction easy and scalable,” 01 2007, pp. 1–4.

[11] J. Hertz and T. Newsham. (2019) TriforceAFL. AFL Qemu fuzzing with full-system emulation. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[12] S. Keil and C. Kolbitsch, “Stateful fuzzing of wireless device drivers in an emulated environment,” 05 2019.

[13] K. Lu, M.-T. Walter, D. Pfaff, S. Nuernberger, W. Lee, and M. Backes, “Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying,” 01 2017.

[14] Z. Ma. (2019) Massive scale USB device driver fuzz without device, BlueHat v18. [Online]. Available: https://www.slideshare.net/MSbluehat/bluehat-v17-massive-scale-usb-device-driver-fuzz-without-device

[15] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, “DR. CHECKER: A soundy analysis for Linux kernel drivers,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1007–1024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[16] Manticore. (2019) Symbolic execution tool. [Online]. Available: https://github.com/trailofbits/manticore

[17] T. Mueller. (2019) Virtualised usb fuzzing using qemu and scapy. breaking usb for fun and profit. [Online]. Available: https://www.ekoparty.org/archivo/2011/ekoparty2011_Muller_usb_fuzzing.pdf

[18] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, “Faults in Linux: Ten years later,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 305–318. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950401

[19] J. Patrick-Evans, L. Cavallaro, and J. Kinder, “POTUS: Probing off-the-shelf USB drivers with symbolic fault injection,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/patrick-evans

[20] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kAFL: Hardware-assisted feedback fuzzing for OS kernels,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[21] S. Schumilo and R. Spenneberg. (2019) Don’t trust your USB! how to find bugs in USB device drivers. [Online]. Available: https://www.blackhat.com/docs/eu-14/materials/eu-14-Schumilo-Dont-Trust-Your-USB-How-To-Find-Bugs-In-USB-Device-Drivers-wp.pdf

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[23] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware.” in *NDSS*. The Internet Society, 2015. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2015.html#Shoshitaishvili15

[24] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, “Periscope: An effective probing and fuzzing framework for the hardware-OS boundary,” in *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2019, pp. 1–15.

[25] C. Spensky, J. Stewart, A. Yerukhimovich, R. Shay, A. Trachtenberg, R. Housley, and R. K. Cunningham, “Sok: Privacy on mobile devices–its complicated,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 3, pp. 96–116, 2016.

[26] J. V. Stoep. (2016) Android: protecting the kernel. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf

[27] H. Stuart, “Hunting bugs with Coccinelle,” Master’s Thesis, May 2008.

[28] Syzkaller. (2019) Kernel fuzzer. [Online]. Available: https://github.com/google/syzkaller

[29] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, “Charm: Facilitating dynamic analysis of device drivers of mobile systems,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 291–307. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[30] Unicorn engine. (2019) The ultimate CPU emulator. [Online]. Available: https://www.unicorn-engine.org

[31] N. Voss. (2019) AFL-Unicorn: Fuzzing arbitrary binary code. [Online]. Available: https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf

[32] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares,” in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[33] M. Zalewski. (2019) American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl

Appendix A
List of functions intercepted by the evasion kernel

In this appendix, we list functions that are intercepted by the evasion kernel in order to evade hardware dependencies.

`request_firmware`	Replaced with a stub that allocates and returns 1024 bytes of zeroed memory.

`ioremap`
`ioremap_wc`
`of_iomap`
	Replaced with kzalloc

`clk_get`
`devm_clk_get`
`of_clk_get_by_name`
	Replaced with a stub that returns clock "oscclk0" present at `virt` board or "apb_pclk" present at `vexepress` board

`clk_set_rate`	Replaced with `stub0`
`i2c_smbus_write_byte_data` `i2c_smbus_read_byte_data`	Replaced with `stub0` to avoid read/write to missing periphreal

`wait_for_completion`
`request_threaded_irq`
`_raw_spin_lock`
`_raw_spin_unlock`
	Replaced with `stub0` to avoid waiting for interrupts from missing peripheral

`wait_for_completion_timeout`	Replaced with `stub1` to avoid waiting for interrupts from missing peripheral

APPENDIX B
IMPLEMENTATION DETAILS

A. Extracting kernel state snapshot

Once a driver is loaded and initialized and the corresponding `/dev/` filesystem entry is created in Qemu, *EASIER* executes a user space program that opens that file and issues an IOCTL system call. *EASIER* uses Qemu's gdb interface to set a breakpoint at the beginning of the corresponding system call entry in the kernel: this address can be found in `/proc/kallsyms`. Note that in order to distinguish the system call that reaches the handler under test from system calls issued by other programs (e.g. by the parent terminal) *EASIER* puts a unique value to the list of arguments and uses a conditional breakpoint that checks for this value (in our implementation we conveniently reused the IOCTL cmd numbers which are usually sufficiently unique).

Once the execution reaches the kernel, *EASIER* dumps the guest memory and CPU registers values. Unfortunately, Qemu's control port command for dumping ARM guest memory, `dump-guest-memory`, only works if the host also runs on the ARM architecture[4]. Due to this, we use the gdb interface instead, which allows one to dump arbitrary memory regions. In order to use this approach, *EASIER* needs to know memory regions to dump in advance. While getting the userspace memory mapping is trivial through the `/proc` filesystem, getting the kernel space memory layout is a bit more complicated for ARM32. On Intel and aarch64 architectures the kernel mapping can be obtained through `debugfs` if the kernel was compiled with the `CONFIG_PTDUMP` option. Unfortunately, there is no such option for ARM32. Fortunately, a corresponding patch was published at `lwn.net` that enables similar functionality[5]. With a few tweaks we were able to make it work for our case. For the guest CPU registers we use Qemu's QMP interface.

B. IOCTL recovery additional details

When recovering IOCTL structures, we use `copy_from_user()` and `copy_to_user()` functions to learn the pointer value and the size of the corresponding memory region. To do this, we intercept and replace these functions in dUnicorn. One of the difficulties is that these functions are inlined which means that we cannot simply put a code hook at one specific address (as with non-inlined functions). Instead we need to find all places inside the driver that include the corresponding code. Fortunately, in Linux kernel both functions are defined using inline assembler which means that they are always compiled into the same type of instructions and with the same order. We use this fact to statically search for the corresponding sequence of instructions and redirect execution at those points.

C. Restoring coprocessor registers

When restoring the kernel state inside dUnicorn, we also need to restore coprocessor registers. Unfortunately the Unicorn library does not support setting such registers directly. In order to set them our framework automatically generates assembly code (which uses `mcr` instructions to set the registers) and executes it before starting IOCTL handler emulation.

APPENDIX C
DRIVER INITIALIZATION ADDITIONAL DETAILS

In this Appendix, we look further at how often stub functions were used to satisfy software dependencies at various stages of the driver initialization process. Table V shows the number of stub invocations for 25 drivers from the MSM kernel for which we were able to reach IOCTLs. To give a rough measure of driver complexity, column 2 shows the number of lines of code contained in the driver and column 3 specifies how many undefined functions the driver had (i.e. those present in the Android kernel but absent in our evasion kernel). Columns 4-7 specify how many times the evasion kernel used stub functions to satisfy a dependency in each step of the initialization process (including repeated calls to a missing function). We can see that the total number of invocations for most of the drivers is moderate and is often less than 10, but could be as high as 50 in the case of `msm_cpp.ko`.

In Table VI, we provide a breakdown of different loading phases for MSM drivers. For each driver, the 3rd and 4th columns specify if the driver's `init` and `probe` functions succeeded; `n/a` in the `probe` column means that the driver does not have the corresponding function and all the initialization happens inside `init`. Column "`dev`" `file` specifies whether during the initialization, the driver succeeded in creating the corresponding file in `/dev`; the `open` column specifies whether this file could be successfully opened.

APPENDIX D
KNOWN CVE'S: CASE STUDIES

In this appendix, we describe three existing vulnerabilities that were reachable in the evasion kernel, CVE-2014-9786, CVE-2014-9785 and CVE-2014-9783, in more detail. The former demonstrates that our system is capable of reaching deep bugs. The second, is an example of a bug that can be reached even in case the driver is supposed to actively

[4] https://lists.gnu.org/archive/html/qemu-devel/2015-11/msg04481.html

[5] https://lwn.net/Articles/572320/

Module	LOC	Missing Symbols	Stubs in Init	Stubs in Probe	Stubs in open	Stubs in close	Stubs total
avtimer.ko	369	5	0	0	2	4	**6**
msm_rotator.ko	1,850	16	0	12	0	0	**12**
adsprpc.ko	1,287	19	3	-	5	5	**13**
msm_adc.ko	1,533	8	0	1	0	0	**1**
vidc_venc.ko	4,149	38	2	-	5	1	**8**
msm_led_flash.ko	273	2	0	2	0	0	**2**
msm.ko	1,056	2	0	0	0	0	**0**
msm_gemini.ko	2,399	11	0	0	0	-	**0**
msm_jpeg.ko	2,303	20	0	3	0	-	**3**
msm_ispif.ko	1,085	5	0	0	0	0	**0**
msm_cpp.ko	1,951	23	0	19	10	21	**50**
msm_rmnet.ko	841	10	0	-	-	-	**-**
msm_rmnet_bam.ko	1,013	8	0	-	-	-	**-**
msm_rmnet_sdio.ko	713	0	0	-	-	-	**-**
msm_rmnet_smux.ko	934	0	0	0	-	-	**-**
qseecom.ko	3,391	18	0	10	0	0	**10**
diagchar.ko	8,781	17	1	-	0	0	**1**
msm_actuator.ko	907	15	0	0	0	0	**0**
msm_cci.ko	1,149	6	0	1	0	0	**1**
vidc_vdec.ko	2,691	49	1	-	5	3	**9**
mdss_rotator.ko	3,085	49	0	6	0	0	**6**
msm_csid.ko	669	7	0	0	0	0	**0**
msm_isp.ko	3,747	22	0	6	1	0	**7**
msm_csiphy.ko	774	4	0	0	0	0	**0**
iris-radio.ko	4,561	0	0	0	0	0	**0**

TABLE V: Details of driver initialization process

communicate with the hardware. The third, is an example of how we used our framework to show that CVE-2014-9783 is, in fact, a false positive and cannot really be triggered, even on a device.

CVE-2014-9786. According to the CVE, this vulnerability affects the camera actuator sensor driver in Google Nexus 5 and 7 devices. The vulnerability happens in an IOCTL used to configure the sensor; triggering it requires issuing two IOCTLs in a sequence. The first IOCTL is used to allocate an array of a user-defined size. Inside the second IOCTL this array is parsed inside a loop whose upper bound can be controlled by the attacker. This is a challenging stress test for our system since reaching vulnerable code requires the driver to stay in a consistent state between the two IOCTLs.

CVE-2014-9785. This vulnerability happens in `qseecom.ko` driver which enables communication between the Linux kernel and QSEE, Qualcomm's TrustZone implementation, which is present on a wide variety of Android devices. The driver works by issuing a Secure Monitor Call to request services inside QSEE, and thus its functionality depends on TrustZone returning specific values. Dependencies on specific functionality that cannot be evaded without detailed information about the hardware is what prevents three other `qseecom.ko` CVEs from being triggered: in order for the driver to reach vulnerable code QSEE needs to return a correct version number. However, even though execution in this driver depends on the hardware most of the time, this particular vulnerability could be executed and reached in our evasion kernel. The vulnerability is due to the use of `__copy_from_user()` function that does not verify the pointer provided by the userspace.

CVE-2014-9783. Finally we consider a CVE describing a bug in a camera control interface driver, `msm_cci.ko`. We could not cause a crash in our evasion kernel and at first decided that it was due to our framework. However after additional manual analysis, we found that the bug can not be in fact be triggered on a real device either. According to the CVE, the bug is due to unchecked "`size`" field which in theory should cause an out of bound memory access. However `msm_cci.ko` gets the user-provided data from the v4l2 subsystem which passes only the number of bytes as encoded by the first, `cmd`, argument. We found that there was another bug in the definition of the `cmd` argument. As a result instead of copying the whole second argument, only 4 bytes (a 32-bit pointer) were copied from the userspace to the kernel space by the v4l2 subsystem. This prevents the malicious payload ("`size`" field) from ever reaching the IOCTL code.

Appendix E
Five known but unreported bugs in the MSM kernel

In our experiments, we used 5 bugs that were fixed in the MSM kernel but not reported as CVE's. We describe these bugs in more detail in this section.

actuator-crash-1. This bug affects the camera actuator sensor driver, `msm_acutuator.ko`, and is due to an attacker-controlled upper bound of a loop. The vulnerable code we analyzed pertains to the MSM kernel commit `212da48` and `ioctl` cmd `VIDIOC_MSM_ACTUATOR_CFG`. In more

#	**Module**	**Init**	**Probe**	**“dev” file**	**Open**
1	avtimer.ko	✓	✓	✓	✓
2	msm_rotator.ko	✓	✓	✓	✓
3	adsprpc.ko	✓	n/a	✓	∓
4	msm_adc.ko	✓	✓	✓	✓
5	msm_serial_hs.ko	✓	×	–	–
6	vidc_venc.ko	✓	n/a	✓	✓
7	msm_led_flash.ko	✓	✓	✓	✓
8	msm.ko	✓	✓	✓	✓
9	msm_gemini.ko	✓	✓	✓	**S**
10	msm_jpeg.ko	✓	✓	✓	**S**
11	msm_ispif.ko	✓	✓	✓	✓
12	msm_cpp.ko	✓	✓	✓	∓
13	msm_vpe.ko	✓	∓	–	–
14	msm_rmnet.ko	✓	n/a	✓	✓
15	msm_rmnet_bam.ko	✓	n/a	✓	✓
16	msm_rmnet_sdio.ko	✓	n/a	✓	∓
17	msm_rmnet_smux.ko	✓	✓	✓	∓
18	msm_rmnet_wwan.ko	∓	n/a	–	–
19	qfec.ko	∓	–	–	–
20	dwc3-msm.ko	✓	✓	–	–
21	qseecom.ko	✓	✓	✓	✓
22	diagchar.ko	✓	n/a	✓	✓
23	msm_actuator.ko	✓	✓	✓	✓
24	msm_cci.ko	✓	✓	✓	✓
25	vidc_vdec.ko	✓	✓	✓	✓
26	mdss_rotator.ko	✓	✓	✓	✓
27	msm_csid.ko	✓	✓	✓	✓
28	msm_isp.ko	✓	✓	✓	✓
29	msm_csiphy.ko	✓	✓	✓	✓
30	iris-radio.ko	✓	✓	✓	✓

TABLE VI: Loading phase evaluation. ✓– step succeeded; ∓ – function returned a negative error code; × – function crashed; ’–’ – step failed due to errors at previous steps; **n/a** – functionality not present; ✗– ioctl analysis failed; S – surrogate module was used.

```
1   static int32_t msm_actuator_init(...) {
2     ...
3     if (copy_from_user(&a_ctrl->region_params, (
          void *)set_info->af_tuning_params.
          region_params, a_ctrl->region_size *
          sizeof(struct region_params_t)))
4       return -EFAULT;
5    ...
6   }
7   ...
8   static int32_t msm_actuator_init_step_table(
        struct msm_actuator_ctrl_t *a_ctrl,...)
9   {
10    ...
11    step_boundary = a_ctrl->region_params[
          region_index].step_bound[MOVE_NEAR];
12    for (; step_index <= step_boundary;
          step_index++) {
13      ...
14      a_ctrl->step_position_table[step_index] =
            cur_code;{
15    ...
16  }
```

Listing 5: Bug 1 in msm_actuator.ko, commit 212da48

```
1   static int32_t msm_actuator_config(...)
2   {
3   ...
4   switch (cdata.cfgtype) {
5    case CFG_SET_ACTUATOR_INFO:
6      rc = msm_actuator_init(a_ctrl, &cdata.cfg.
           set_info);
7      ...
8
9    case CFG_SET_DEFAULT_FOCUS:
10     rc = a_ctrl->func_tbl->
           actuator_set_default_focus(a_ctrl, &
           cdata.cfg.move);
11     ...
```

Listing 6: Bug 2 in msm_actuator.ko, commit e6edf78

detail, structure a_ctrl->region_params is copied from userspace in Listing 5, line 3. Later, step_bound field of this structure is used as a loop upper bound (lines 11 and 12) with a crash at line 14.

actuator-crash-2. This bug also affects the camera actuator sensor driver, msm_acutuator.ko, and is caused by an uninitialized function pointer. Note that such bugs can be sometimes exploited to get arbitrary code execution [13]. The vulnerable code we analyzed pertains to MSM kernel commit e6edf78 and ioctl cmd VIDIOC_MSM_ACTUATOR_CFG. By fuzzing this driver we immediately found that issuing the CFG_SET_DEFAULT_FOCUS subcommand in Listing 6 causes the kernel to crash. This is due to a_ctrl->func_tbl in line 10 not being initialized without first issuing the CFG_SET_ACTUATOR_INFO subcommand in line 6.

diag-crash-1. This bug pertains to diagchar.ko, commit 1414d4a. This bug requires first to issue two ioctl system calls in sequence: DIAG_IOCTL_SWITCH_LOGGING and DIAG_IOCTL_COMMAND_REG to switch the driver state; and then a write system call to trigger a crash. The reason for the crash is a missing check for the lenght field (see lines 4 and 6 in Listing 7).

diag-crash-2. This bug pertains to diagchar.ko, commit 1414d4a. The bug is due to copying a user pointer directly instead of using copy_from_user function in DIAG_IOCTL_COMMAND_REG ioctl.

isp-crash-1. This bug happens in msm_isp.ko driver, commit 83789a7935f9. The bug is in VIDIOC_MSM_VFE_REG_CFG ioctl which is used to write data to the peripheral using writeb_relaxed in which the attacker controls both the data written and the offset from the start of the control register. The driver does not check this offset which leads to arbitrary memory write primitive.

```
int diag_send_dci_pkt(struct diag_master_table entry, unsigned char *buf, int len, int index)
{
  ...
  len = len - 4;
  ...
  for (i = 0; i < len; i++)
        driver->apps_dci_buf[i+9] = *(buf+i);
  ...
}
```

Listing 7: Bug 1 in diagchar.ko, commit 1414d4a

APPENDIX F
LIST OF DEFCONFIGS FOR AUTOMATIC RECOVERY

In this appendix, we list the eight layouts with default configurations that were used to test our structure layout recovery component.

msm7627a_defconfig
8226_defconfig, 8610_defconfig, 8960_defconfig
8660_defconfig
8974_defconfig
9615_defconfig, 9625_defconfig
All struct device options enabled
All struct device options disabled
PM_RUNTIME disabled but DEBUG_SPINLOCK enabled