Emulating Android Device Drivers via Borrowed Execution Context

Alex Le Blanc[1] and Ivan Pustogarov[2]

[1] University of Waterloo, Waterloo, Ontario, Canada
a6leblan@uwaterloo.ca
[2] Concordia University, Montreal, Quebec, Canada
ivan.pustogarov@concordia.ca

Abstract. The ability to emulate isolated code parts in binary low-level system code such as an operating system's kernel is often both necessary and beneficial from the security analysis point of view, as it significantly reduces the search space to more interesting parts, and also because complete kernel images are very difficult or often impossible to emulate as a whole using existing emulators. In this paper, we consider this problem of emulating isolated code compartments for Android binary kernel images.
To this end, we present a framework that allows emulation of device drivers directly from binary Android kernel images by "borrowing" emulated execution context from a running stock Linux kernel. It works by injecting the Android kernel under test into the same memory space as the stock Linux kernel, logically unlinking specific isolated parts from its original kernel, and reattaching them to the stock Linux kernel.
We evaluate the correctness of our approach on a set of 56 drivers from 10 different kernels, for which it was successful in borrowing the execution context in all cases. By further extending it with coverage-based fuzzing, we fuzzed a set of 23 IOCTL drivers and discovered 4 zero-day vulnerabilities (some high-severity) which were confirmed by Google's security team.

Keywords: Emulation · Android kernel · fuzzing · device drivers.

1 Introduction

Dynamic analysis is one of the proven and effective methods for finding security bugs. Moreover, it achieves its full potential in an emulated environment: emulation allows for a fine-grained control of the execution through internal state introspection, which, in turn, enables a number of useful types of instrumentation without the need to purchase physical copies of the device.

When it comes to dynamic analysis of binary low-level system code, such as an operating system's kernel (or firmware), one of the common tasks is to emulate and analyze only specific isolated parts rather the whole system. There are two reasons for this. First, it is more effective from a security analysis point of view to focus on specific bug-prone parts, e.g., network packet parsing code in a network driver, or a system call handler, as it significantly reduces the search space in otherwise large system. Second, emulating a complete operating system kernel can be problematic or impossible due to the absence of a large number of emulated versions of the hardware that the low-level system code would require to boot and operate. In fact, one of the more popular emulators, Qemu [1], can properly boot kernels compiled for only a handful of hardware boards[1]. This makes, in many cases, focusing on isolated code compartments the only possible solution in terms of the amount of the hardware whose emulated versions need to be added.

Unfortunately, isolating and emulating code starting from an arbitrary location without booting the kernel first is far from obvious. This is usually hindered by missing execution context, i.e., various kernel structures and subsystems that would otherwise be initialized during the normal boot process. Without a valid execution context, the portion of the code that we try to emulate will most likely result in undefined behavior. This usually manifests as memory access violation errors when it tries to interact with uninitialized kernel memory. In this paper, we approach this problem of emulating isolated code parts for binary Android kernels images and we focus on device drivers added by the manufacturers, which usually are much less audited compared to the core kernel subsystems and historically have been a source of many vulnerabilities.

More specifically, when the execution moves to a logically separate code compartment (i.e., a driver), which is a part of a larger system (i.e., the kernel), it is expected that registers and memory at specific address ranges are initialized (i.e., are set to particular values). It is also expected that specific kernel functions, which are not part of the driver itself, are present at specific memory addresses. If such expected/valid execution context is present, the emulation will correspond to an execution run on a real device, and it becomes possible to reason about the device's security properties. If, on the other hand, the execution context is set arbitrarily, then with overwhelming probability, the emulation will not correspond to an actual device behavior. Without being able to emulate the original Android kernel, getting a precise and valid execution context is non-trivial, and the space of all possible values is too large for exhaustive search.

Our key observation is that the execution context of the stock Linux kernel configuration (that can be emulated) might be close enough to the execution context expected by individual parts of Android kernels (e.g., device drivers). Our main idea is to load the Android kernel alongside the already booted (in an emulator) stock Linux kernel, and then redirect the execution to the driver for the analysis. More precisely, we run the stock Linux kernel and then inject the Android kernel's binary into the same memory space. In this way, the two kernels "live" alongside each other, but only the stock kernel is in charge of running the system, i.e., interacting with emulated peripherals and maintaining all the kernel structures required for proper operation. We connect the injected Android kernel to the running stock kernel by redirecting calls to standard kernel

[1] While some of the peripherals, such as camera, or IR sensor, might indeed not be needed, in general, there is always a number of hardware components that are critical and are required during the boot process.

functions (such as `printk`) and control data structures to the corresponding versions in the stock kernel. Once the kernels are connected, we finally identify the initialization functions of the driver and move the execution there.

A different approach for a related problem was proposed in [11] in which the authors suggested a way to insert drivers originally developed for custom Android kernels into the stock Linux kernel, thus enabling emulation of these drivers. While similar in spirit, that approach has two fundamental limitations which we also try to solve in the current work. First, the approach in [11] is limited to device drivers only, while we consider an approach that, in principle, can be used to direct execution to any part/subsystem of an Android kernel. Second, and most importantly, the system presented in [11] requires drivers to be recompiled from the source code. At the same time, kernel images are shipped to end users in binary form (including initial release and system updates). Because of this, there is no guarantee that, in general, the OEM's published source code[2] exactly matches the binary running on the devices; moreover, some bugfixes can be included only into the distributed kernel binaries. In addition, the source code for some drivers might simply not be available. Working on binary kernel images (e.g., extracted from system updates) does not have the aforementioned limitations.

We implemented our approach as the LiLi framework (as in Linux in Linux). To evaluate LiLi, we used drivers from ten different Android kernels, from four vendors (Lineage[3], Huawei, HTC, and Samsung). We used 56 unique drivers to test LiLi's correctness in transferring Linux execution context. LiLi was successful in all cases. We then further extended our framework with coverage-based fuzzing and tested it on a set of 23 drivers. We ultimately discovered four zero-day vulnerabilities in Google smart TV kernels while obtaining only a handful of false positives. All discovered vulnerabilities were confirmed by Google's Android Security Team, and two of them were assigned a severity rating of "High", while another received a "Moderate" rating. Moreover, one of the high-severity vulnerabilities was discovered in one of the core Android subsystems, ION, which potentially affects a much broader set of Android devices. We received a total of 6,000 USD as bounties for reporting the vulnerabilities.

Our contributions. In summary, we make three main contributions:

1. **LiLi (Linux in Linux).** We present LiLi, a framework that enables emulation and fuzzing of selected parts of custom Android kernels such as device drivers.
2. **Fuzzing emulated drivers.** We emulate and fuzz drivers from 10 different custom Android kernels.
3. **Discovery of zero-day vulnerabilities.** We discover 4 zero-day vulnerabilities, all of which were confirmed by Google's Android security team.

[2] Linux kernel is developed under GPL which in theory requires OEMs to publish their source code. In practice it remains a grey area especially when it comes to kernel modules and device drivers, many of which, in fact, come in binary form only.

[3] Lineage operating systems are custom modifications of existing kernels from other vendors. In this project, we look at four Lineage kernels based on Google, Huawei, Fairphone, and BQ kernels.

2 Background

Executable and linkable format (ELF). The executable and linkable format (ELF) is the standard file format for executable and relocatable object files for a number of Unix-based systems including Linux and Android. Linux kernel binaries themselves (e.g., `vmlinux`), as well as any drivers inserted at run-time, i.e., loadable kernel modules, follow this format. There are two main types of metadata that are of particular interest to us, namely symbols and relocations. We also refer to a third type, sections, which are contiguous parts of the binary that serve some common purpose (e.g., there is a "`.text`" section for code, a "`.rodata`" section for read-only data, etc.).

An ELF symbol provides a reference to some part of a binary. It can be thought of as a structure that holds information about a function or data object (e.g., a variable). These symbols are stored in the `.symtab` section. Examples of the information stored in these structures are: (1) *Section index*: the ELF section that this symbol belongs to; (2) *Value*: typically the offset of the symbol relative to its section, but can also be a CRC checksum; (3) *Binding*: the visibility of the symbol, typically denoted as either "local" if the scope of the symbol is limited to a single file, or "global" if it can be made available to other files during linking; (4) *Symbol name:* points to the symbol name in the string table section. A symbol may belong to undefined category (in which case the section index and offset are undefined). These are symbols that have been referenced in a compilation unit, but which have not been defined by this compilation unit. They are resolved once the value of a symbol is known (e.g., during linking process).

ELF relocation entries are added to a compilation unit by the compiler to keep track of instructions that reference symbols whose location is either yet unknown (i.e., undefined symbols) or might change. For instance, a loadable kernel module might have a branch instruction (`call` or `bl`) to kernel-defined `printk`. But during compilation, the actual address of `printk` is not yet known to the module (and might even be different for different target kernels). For this, ELF binaries include a *relocation entry* in a dedicated ELF section that signals to the kernel module loading subsystem that this instruction's target address should be patched/replaced by the actual address. Each relocation entry contains information such as: (1) *Offset*: the location where a relocation needs to be performed; (2) *Info*: contains both the index of the symbol referred to (such as for our object in the example above), and the type of relocation; (3) *Addend*: the offset from the relevant symbol that we are interested in (e.g., we might need the address of a field of a structure, rather than of the structure itself). The kernel (or program loader) then uses relocation and symbol entries to dynamically patch each instruction to point to the correct destination.

We use relocations to connect the injected and the emulated kernels together by adding new relocation entries.

Loadable kernel modules (LKMs). A device driver can either be compiled as a constituent part of the kernel and is therefore automatically initialized at boot-time; or be compiled as an LKM, i.e., a separate ELF binary (usually having

`.ko` filename extension) that can be loaded and removed from the kernel at run-time. LKMs designate their initialization routine via a pointer stored in the `.gnu.linkonce.this_module` section. LiLi makes use of LKM kernel subsystem to inject one Android kernel into another kernel. To specify the precise location in the injected kernel that we wish to redirect execution to, we take advantage of the initialization pointer.

The evasion kernel. The evasion kernel is a part of the EASIER framework [11], an ex-vivo dynamic analysis framework for Android device drivers. It is a modified version of a stock Vanilla Linux kernel that allows one to insert LKMs that were compiled for arbitrary Android host kernels. The EASIER framework provides a way to generate simplistic models for missing peripherals from the code itself with sufficiently good success rate. In this work, we use EASIER to generate hardware models for a small number of missing peripherals that are expected by the code compartments that we test. We note however that the Evasion kernel can be replaced by developing simplified hardware models of the required peripherals either manually or using another similar tool.

vmlinux. When compiling a kernel, the compiler starts by individually compiling the various source files it contains into separate object files. These object files are then linked together into a single object file, namely `vmlinux.o`. Then, `vmlinux.o` is statically linked to produce `vmlinux`, which is then compressed and combined with decompressing code to produce a bootable kernel image: zImage (or bzImage). Therefore, `vmlinux` is effectively an uncompressed version of the the bootable kernel image (minus the booting metadata).

3 Overview

If it was possible to emulate Android kernels as a whole with off-the-self emulators, running driver code would not be an issue either. Unfortunately, the vast majority of Android kernels will not boot in an emulator. This motivates the high-level goal of LiLi: if we cannot boot a complete kernel in an emulator, is it still possible to emulate its isolated parts, e.g., built-in device drivers? One of the first (and naïve) alternative approaches would be to simply copy the Android kernel binary under test into an emulated memory space, set the program counter to the first instruction of the function that we want to emulate (e.g., driver's entry function), and start to execute instruction by instruction, until we reach the end of the function. Such an approach might be valid, but only for very simple and, most importantly, self-contained code, which is not the case for the majority of drivers.

To better understand the problem, consider the code snippet in Listing 1.1. This code illustrates four different categories of problems with the naïve approach above. The code consists of two functions: a) `module_init()`, which is supposed to be called by the kernel when the driver is loaded, and b) `driver_probe()`, which is supposed to be called by the kernel once the driver is registered and the device is detected. Assume we set the instruction pointer at the first instruction of `module_init()` and start emulated execution.

Uninitialized pointers. When the execution reaches line 7, it tries to access global pointer `current`[4]. This pointer is defined/initialized outside of the `module_init()` function (and outside of the driver code), and, thus, will remain uninitialized for our naïve emulation. As a result, when `module_init()` tries to dereference it, a segmentation fault will occur. Moreover, the structure pointed by `current` contains a number of other pointers. With a large number of kernel-defined pointers, identifying all of them and manually setting them to correct values is not feasible.

Uninitialized global variables in standard kernel API. A related problem happens when the driver code tries to use standard kernel API functions in line 9, for example the `vmalloc()` memory allocation routine. Internally, this routine uses the `totalram_pages`[5] global variable to find where to allocate new memory. This global variable is supposed to be initialized during kernel boot; otherwise the behavior of `vmalloc()` would be undefined.

Unmapped memory. Another problem arises when the code tries to map physical pages (line 13). Usually the kernel maps all physical memory at a specific (virtual memory) offset (conventionally called *linear mapping*) early in the boot process. On many architectures, a call to `kmap()` will return the corresponding address from this mapping. If physical memory memory was not mapped (which is the case for the code snippet in Listing 1.1), the pointer dereference in line 14 will result in an unmapped memory exception.

```
 1 static struct platform_driver ex_plaform_driver = {
 2     .probe      = driver_probe,
 3 ...
 4 }
 5 int module_init() {
 6   ...
 7   struct files_struct *files = current->files; /* problem 1 */
 8   ...
 9   data = vmalloc(PAGE_SIZE);  /* problem 2 */
10   ...
11   struct page *p;
12   ...
13   v = kmap(p); /* problem 3 */
14   *(int *)v = 42;
15   ...
16   platform_driver_register(&ex_plaform_driver); /* problem 4 */
17   ...
18 }
19
20 int driver_probe() { ... }
```

Listing 1.1. Difficulties with naïve approach to emulation.

Asynchronous function calls. Properly initializing all required memory might be difficult even within the scope of one driver. This is because some functions are intended to be called by the kernel asynchronously. In line 16, the code calls `platform_driver_register`, which registers callback function `driver_probe` with the kernel. It is then up to the kernel to call this callback at an appropriate time. The driver itself never calls this function. This usually will leave a subset

[4] The `current` pointer refers to the user process currently executing. During the execution, for example (but not limited to) of a system call, the current process is the one that invoked the call. Kernel code can get process-specific information by using it [13]. On arm64 architecture, it is stored in the `sp_el0` machine-specific register.

[5] Defined in `mm/page_alloc.c`.

of driver-defined variables uninitialized, causing problems similar to those in the previous cases.

In this paper, we use the term *execution context* to denote all variables/structures that the driver (or isolated code portion) under test might need/use during the execution. The execution context must be initialized to proper values so that the emulated execution is identical to the execution on the physical device. Only in this case can we reason about code's security proprieties. We also note that these types of problems are not specific to kernel emulation, but applicable, to varying extent, to other types of code too (e.g., userspace programs).

The goal of LiLi is, thus, to reconstruct and provide a valid execution context for an isolated part of Android kernel code (in this paper, to a built-in Android device driver). Our key observation is that custom Android kernels and the stock Linux kernel share most of the core subsystems, and, thus, the execution context maintained by the stock Linux kernel (which can be booted and emulated) might be close enough to the execution context expected by Android kernel drivers. We refer to the stock Linux kernel that will provide the execution context as the *donor kernel*, the code/driver that we would like to test and that does not initially have an execution context as *orphan code*, and the kernel to which the *orphan code* originally belongs as the *original kernel*.

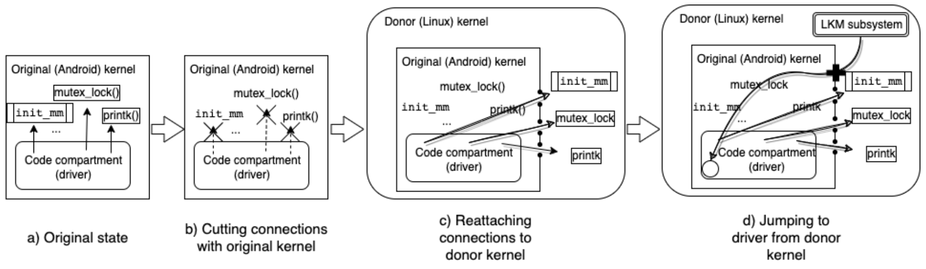
Original (Android) kernel
mutex_lock()
init_mm
printk()
...
Code compartment (driver)
a) Original state
Original (Android) kernel
mutex_lock()
init_mm
printk()
...
Code compartment (driver)
b) Cutting connections with original kernel
Donor (Linux) kernel
Original (Android) kernel
mutex_lock()
init_mm
printk()
...
init_mm
mutex_lock
printk
Code compartment (driver)
c) Reattaching connections to donor kernel
Donor (Linux) kernel
LKM subsystem
Original (Android) kernel
mutex_lock
init_mm
printk
...
init_mm
mutex_lock
printk
Code compartment (driver)
d) Jumping to driver from donor kernel

Fig. 1. Overview of LiLi: attaching orphan code to donor's execution context

LiLi's task is, through a series of transformations, to make the orphan code assimilate the execution context provided by the donor kernel. The overview of this process is shown in Figure 1. At the beginning (Step a), the orphan code is attached to the original kernel through a series of links; they connect it to the original kernel's (uninitialized) data structures and functions that operate on these structures. Thus, at Step b, LiLi cuts these original connections by replacing all references to these structures and functions with undefined symbols.

LiLi then reattaches (re-links) the orphan code to the donor Linux kernel at Step c. This is done in two sub-stages: first by adding new relocation entries that are specifically crafted to point to the donor's kernel code, and, during the second sub-stage, transforming the original kernel binary (that includes the orphan code) to a loadable kernel module. When this *surrogate module* is loaded into the donor kernel during the next steps, the donor kernel's module loading subsystem is forced, by the new relocations, to reattach the orphan code to itself. During this step, it might happen that the orphan code depends on a small number of functions/subsystems for which there are no alternatives in the donor kernel. In this case, we can redirect such calls to stub functions that return immediately (effectively skipping such calls). For references to data objects that have no alternatives in the donor kernel, LiLi refers to the objects from the original kernel, because either the object does not need to be initialized and everything works as intended, or it does need to be initialized, in which case the issues that arise are no different than if we had stubbed the object. Here, these objects are also recursively detached from the original kernel and reattached to the donor kernel, similar to the orphan code (e.g., if this objects stores a pointer to `printk`).

During Step c, we also need to make the donor kernel compatible with the orphan code as much as possible. For this, we align the donor's kernel-driver API to match the original kernel through its configuration. Once this is done, as the last step (Step d), LiLi instructs the donor kernel to pivot the execution to the code under test by adding a function pointer to a specific section in the surrogate module.

We finally proceed by booting the Linux donor kernel in Qemu, which creates the proper execution context. We then insert the original kernel transformed into a surrogate module (with includes the orphan code) into the donor kernel. We can then proceed to fuzzing the driver. More specifically, we focus on IOCTL system calls. For this step we chose to use syzkaller, but any other kernel fuzzer would work too.

4 Implementation

LiLi's goal is to make the orphan (driver) code run within the donor kernel's execution context. For this it first needs to be disconnected from the original kernel. The orphan code is functionally connected to its original kernel through the use of API calls[6] (e.g., `printk` or `vmalloc`) and global data structures (e.g., `init_mm`). To cut these connections we need to: (a) precisely *locate the code and data* that belongs to the driver (which will effectively define the border between the driver and the original kernel); (b) *find all instructions and data* within the driver code that reference functions/data outside of the driver code. In practical terms, cutting connections and reattaching them to the donor kernel means patching these instructions/data (in our specific implementation, the patching happens dynamically, i.e., when the donor kernel runs in the emulator).

4.1 Locating driver's code and data in the original kernel binary

The original kernel's `vmlinux` contains a symbol table[7], and some of the symbol entries will naturally belong to the driver code. These entries, in turn, will contain

[6] More precisely, ones exported by the original kernel functions and variables.

[7] For stripped kernel binaries which don't have the corresponding ELF section, we can still extract the symbol table: the kernel stores it internally as it needs it at run-time.

the offsets and sizes of `vmlinux` parts that store the driver's code and data. Our goal for this subsection is, thus, to identify these symbols.

In order to find the driver's symbols, we consider two main types: local and non-local. Each type of symbol will require a different approach.

Local symbols. For local symbols, we refer to symbols that have a binding of "LOCAL". These symbols represent functions and variables that are limited only to the scope of the file that defines them (e.g., static functions). When compiling a C file, the GCC compiler will normally place all these local symbols together in the symbol table right after a debugging symbol of type "FILE" whose name is the name of the compiled source file. When linking multiple `.o` files together, the GNU linker will place these groups of local symbols sequentially in the new symbol table, each group still preceded by the corresponding "FILE" symbol. Thus, we can search the symbol table for everything between the FILE symbol with a source file's name and the next FILE symbol. More specifically, we take all such symbol table entries for symbols that have non-zero size (to remove debugging symbols).

Non-local symbols. As for non-local symbols (i.e., symbols with "GLOBAL" and "WEAK" bindings), the challenge is that they are necessarily listed after all the LOCAL symbols in the symbol table, without a nearby symbol that identifies the source file that defines them (like the aforementioned FILE symbols). We must instead directly scan the source file itself to recover the names of these global symbols (via `ctags`). For each of these symbols, we then scan the `vmlinux` symbol table for the GLOBAL symbol with the same name (or WEAK symbol if the function has the "`__weak`" descriptor). We use the driver source code for the sole purpose of identifying function names and names of `.c` files. We never need to compile the driver, nor analyze its functions' bodies, and the code that we test during fuzzing comes directly from the kernel binary image. We discuss an alternative approach in Section 6.

Variable declarations behind preprocessing macros. Some variable declarations can be hidden behind macros which makes it more difficult to identify them without using a C preprocessor. To get around this, if we have a GLOBAL object symbol that is not a part of the list of driver symbols, we search the donor kernel's `System.map` file to find if that symbol exists there, and if it does not, we assume that this symbol should have been declared by the driver.

4.2 Instructions of interest

At this point, we have categorized all the symbols in the original kernel's `vmlinux` into those that belong to the driver, and those that do not. Moreover, from these symbols we also have located `vmlinux` segments[8] that contain the driver's code Our next task is to scan these segments for instructions that reference code and data outside of the driver. These will be the very connections that we cut later.

There are two types of instructions which are of interest to us: (a) branch instructions that call a function outside of the driver; (b) sequences of instructions that access data outside of the driver. Recovering the destination of a branch

[8] We mean contiguous parts of the binary here, not ELF program segments.

(e.g., `bl`) is relatively straightforward by looking at its argument[9]. We check if this destination address points to outside of the driver code, and if this is the case, we save the location of this instruction for future use. We also resolve its destination address to a symbol (e.g., `printk`) and save it too (saving the symbol name is important as it will be used to reattach the driver to the donor kernel).

Finding places where the driver accesses data objects external to the driver is more complicated. Unlike function calls where a single `bl` instruction immediately gives us the destination, access to data objects involves more than one instruction. By using the reference manual for ARMv8 we narrowed down two pairs of instructions that can cross page boundaries[10] for data access: `(adrp; add)`, and `(adrp; ldr)` (note that these instructions are always paired). The argument to `adrp` instruction gives us the page for the object, and the argument to either `add` or `ldr` gives us the offset of the object in this page. We, thus, find the relative (to PC) destination address of data object as

`(adrp_dest << 12 + add_dest)` or

`(adrp_dest << 12 + ldr_offset)`

depending on which pair of instructions was used[11]. For example, the code might contain the following instructions: (`adrp x0, n_pages; add x0, offset`) or, alternatively, (`adrp x0, n_pages; ldr x1, [x0, offset]`); the object address will be computed as `(pc + n_pages << 12 + offset)`

One of the difficulties that required a bit more sophisticated analysis is that these pairs of instructions are non-atomic, i.e., the two instructions in a pair can be interspersed by another unrelated instruction. Moreover, different pairs can even overlap. To tackle this problem, we use a list to keep track of unpaired `adrp` instructions, to then compare the registers used by subsequent `add` and `ldr` instruction with those of the `adrp`'s in the list. If the registers match, then the two destinations are added to obtain the precise destination of that pair (and the `adrp` can be removed from the list). We then resolve these destinations to specific symbols using the symbol table.

Finally, similarly to branch instructions, we save for future use the locations of `(adrp;add)`/`(adrp;ldr)` pairs together with the symbol names they reference.

4.3 Data of interest

While the original kernel's `vmlinux` is statically linked, it still contains relocations that patch data objects if the `CONFIG_RELOCATABLE` configuration option is enabled (e.g., for KASLR to shift objects in memory), which was always the case for all the kernels under test.

[9] There is the case of indirect calls, such as `bl x0`, but these are usually used to reference the code in the driver itself. This was also the case in all our experiments.

[10] Accesses within page boundaries will point back to the driver. No re-linking is required in this case.

[11] The destination address computed in this way may fall in the middle of a data object. While it was never the case in our experiments, one can use symbol start address and symbol size to find if an address falls within the boundaries of a data object.

LiLi therefore recreates these relocations in the surrogate module. Each such original relocation in `vmlinux` references an absolute address. We thus first search for the symbol corresponding to this address in `vmlinux`'s symbol table. LiLi then creates a relocation with the object's address and the found symbol. If `CONFIG_RELOCATABLE` is disabled (which should not be the case for production kernels), we can scan the driver's `data` section and create new relocations for each address that points to a symbol.

4.4 Re-linking

At this point, we should have a table that contains information about all connections (links) from the driver to the original kernel: for both functions and data objects. This table has the following format (*link_type, [from]:instr_address, [to]:destination_symbol*), where *link type* is either `func` or `data_object`, *instr_address* is the address of an instruction in the driver that references the original kernel, and *destination_symbol* is the name of the specific symbol that this instruction references. This table, in fact, can be seen as a *restored* simplistic and abbreviated version of the relocation table that was used (and discarded) when `vmlinux` itself was compiled and linked.

Now, equipped with this recovered relocation table, we can correctly disconnect the driver from the original kernel and reattach (re-link) it to the donor kernel. There are two ways to achieve this: we can either re-link the driver statically by rewriting the donor kernel binary, or we can do it dynamically (i.e., during donor kernel's runtime). In our implementation, we use a dynamic approach: ideally we would want to avoid any modifications to the donor kernel so that our approach is as generic as possible regarding kernel versions. But more importantly, the Linux kernel, at its very core, already has a way to accommodate additional code dynamically via the loadable kernel module subsystem; we are going to hijack this functionality.

From a high level, we transform the original kernel (together with the driver) into a loadable kernel module compatible with the donor kernel. We call the resulting module *surrogate module*. During this transformation, we incorporate the recovered relocation table into this new module. We then let the donor kernel do the rest: when we load the surrogate module, the donor kernel is forced by the added relocation entries to patch all necessary instructions, effectively reattaching the driver to itself. In the rest of this section, we will provide more technical details regarding our implementation.

Initializing surrogate module. We start by generating a simple C file that contains basic, minimally required ELF sections needed in a kernel module (e.g., `.modinfo`) and compiling it into a `.ko` file. LiLi then extracts the entirety of the original kernel's `vmlinux` image, debugging sections aside, and copies it into this file's `.text` section.

The reason we copy parts that may not be necessary for our particular driver is that all the code in `vmlinux` is statically linked and is position-independent (i.e., PC-relative), meaning that we will need to maintain the same offsets between instructions and their destinations. Moreover, it is important that the whole binary goes into one section, because otherwise upon module insertion, separate ELF sections may be loaded into different memory regions in the kernel, potentially causing the relative offsets used by the instructions to point to unintended destinations. Note that copying the entire `vmlinux` image like this, while formally preserving existing symbol and relocation tables in the output file, effectively disables them for kernel loading subsystems (as they are now residing in one single section together with everything else).

Adding symbols and relocations. Once we have the skeleton ELF file for the surrogate module, we can start adding symbols and relocations from our recovered simplistic relocation table. This procedure consists of two sub-steps. First, LiLi adds a new empty symbol section, and for each entry in the table, creates a new symbol entry with the same name but `UNDEF` as the value in case this name is present in the donor kernel. This will instruct the donor kernel that these symbols should be resolved, and it will resolve them to the donor kernel's version of these symbols. During the second sub-step, LiLi recreates the actual relocation table from our simplistic table, using the same values for the instruction address and symbol name. Once added, this should instruct the donor kernel to patch the corresponding instructions and point them to its own symbols.

This procedure with adding symbols and relocations effectively prepares the driver to be unlinked from the original kernel and linked to the donor kernel. In our experiments, this address resolution technique always worked correctly.

Driver entry points. As the last step, we need to: (a) identify the driver's entry/initialization functions, and (b) make the donor kernel jump to these functions. We first note that all initialization functions are prefixed by "`__initcall_`" in `vmlinux`'s symbol table at compile time. Moreover, depending on the intended call order, a different suffix is appended to the function's name by the compiler (suffix of 1 means it's called before ones with 1s, which are before 2, which are before 2s, etc). Thus, by looking for the `__initcall_` prefix among the driver's LOCAL symbols, we can recover the names of the driver's initialization functions . Then, by looking at the suffixes of these symbol names, we can recover the order in which these functions should be called.

In order to solve (b), we note that when loading a module, the kernel LKM subsystem looks at the module's `.gnu.linkonce.this_module` section at a particular offset, where it expects to find a pointer to the initialization function; we use this feature to finally redirect execution to the driver. In order to accommodate multiple initialization functions using a single available slot in `.gnu.linkonce.this_module`, we create a trampoline code that calls each of the built-in driver's initialization functions in the correct order. We inject this trampoline at the beginning of the surrogate module's `.text` section (since there is only irrelevant system code there anyways). We then add a relocation entry in `.rela.gnu.linkonce.this_module` that refers to the trampoline. This ensures that all the initialization functions are called at load-time.

5 Evaluation

In this section, we evaluate two aspects of LiLi. First, we check if LiLi can correctly unlink a built-in driver from its original kernel and then re-link it to the donor kernel. Second, we test if the re-linked drivers can: (a) run in the context of the donor kernel, and (b) be fuzzed. More importantly, we test if LiLi can be used to find new bugs, with our primary focus on IOCTL handlers.

For emulation and fuzzing, we use three components external to LiLi: Qemu as the emulator; the evasion kernel from the EASIER framework as the donor kernel; and syzkaller as the fuzzer. The reason for choosing the evasion kernel as the donor kernel is that Android drivers often require a specific peripheral. Without an emulated version of these peripherals (usually not implemented by Qemu), achieving acceptable code coverage is difficult. The EASIER framework can be used to substitute the missing peripherals with simplistic models derived from the driver code itself. This allows us to test LiLi without implementing emulated versions of the hardware. We also extended the evasion framework to accommodate more types of drivers. More specifically, we added support for I2C drivers in addition to platform drivers provided by EASIER. Finally, we extended EASIER's evasion kernel by porting several Android subsystems, thus, extending the set of drivers that EASIER can handle even further.

5.1 Experimental dataset

In order to test LiLi's re-linking capabilities, we use 56 different IOCTL drivers from 10 different Android kernels: Samsung (Galaxy S9, Galaxy Note 9), Huawei (P20 Pro, Mate 10 Pro), HTC (Exodus, U12+) and Lineage (Fairphone_sdm632, Xiaomi_msm8937, Bq_msm8953, Amlogic). As in this paper we focus on Android, we also verified that these drivers were not simultaneously a part of the vanilla Linux kernel.

For emulation and fuzzing experiments, we use 23 drivers from the same kernels. This is due to limitations of the EASIER framework to create emulated versions of the peripherals in all cases (indeed, this not a limitation of LiLi itself). See Appendix C for the list of drivers, and whether we were able to obtain a device models for each of them.

5.2 LiLi's correctness

In order to test LiLi's correctness in unlinking and re-linking built-in device drivers, we need to first verify that it fully recovers and restores all the relocations, and second, that it can be loaded to the donor kernel. To do that, we first notice that during normal driver compilation, the kernel build system, before creating the final `vmlinux` file, compiles each driver/subsystem separately, producing a `built-in.o` file. This intermediate file contains all the relocations that would be further used (and discarded) by the building system to link it to `vmlinux`. As we had access to the source code of the original kernels, we could use `builtin.o` files for each of the drivers to get their relocation and symbol tables[12]. We then check if they match the relocations and symbols reconstructed and extracted from `vmlinux` by LiLi.

Following this methodology, we verified that the relocation and symbol tables were correctly recovered by LiLi and were similar between the surrogate module and the corresponding built-in.o for all 56 drivers. In all these cases, the generated surrogate modules were accepted by the donor (evasion) kernel.

5.3 Fuzzing results and analysis

All our fuzzing tests were performed on arm64 `c7g.metal` AWS EC2 instances running Ubuntu 20.04 with 128 GiB of RAM. For instrumentation and address sanitizing, we use `KCOV` and `KASAN`, respectively. Table 1 summarizes these experiments. For each driver, the table shows its size, the amount of time it was fuzzed, code coverage, and the number of unique crashes. Each driver was fuzzed for the duration of 3 to 4.5 hours with the average of 1,134 covered blocks per driver (as reported by syzkaller). Based on the code coverage, and the fact that these drivers extensively use the kernel API, we conclude that all the drivers under test were able to successfully use the donor kernel's execution context. We further categorize the drivers into three groups based on type and number of crashes.

Group 1. This group includes drivers 1 through 11 in Table 1 (i.e., 48% of drivers). Fuzzing them resulted in discovering 4 zero day vulnerabilities. All these drivers produced sufficient code coverage without any false positives, i.e., they could be analyzed precisely. This also indicates that LiLi can be used to find new bugs.

Group 2. Drivers from the second group (12 to 17) resulted in a small number of unique crashes. However, we were not able to identify the exact cause of these crashes through additional manual analysis, and we thus classified them as false positives[13].

Group 3. The third group includes drivers from 18 to 23 which produced several false positive crashes. Drivers from this group internally use the `msm` Android subsystem which LiLi also re-linked to the evasion kernel. Upon manual analysis, we found that `msm` subsystem recursively depended on yet another subsystem not present in the donor kernel. In this case, our prototype of LiLi redirected all calls to that subsystem with function stubs. This means that the drivers from group 3 were operating under only partially recovered execution context. We note however that even in this case, we had only a handful of false positives, and moreover, the drivers from this group achieved reasonable code coverage. One solution to this problem would be to recursively re-link all Android kernel subsystems used by the driver instead of using a fixed depth level (2 in our case).

[12] We note that these object files were used only to obtain the ground truth about correct relocations; LiLi does not require these object files.

[13] We believe however that some of these crashes were caused by actual vulnerabilities.

Table 1. Fuzzing Results.

#	Driver	Kernel	LOC	Fuzzing Statistics		
				CPU-hours	Cov	Crashes
1	meson_ion_delay_alloc	Lineage Amlogic	500	219	1425	**1**
2	ionvideo	Lineage Amlogic	1566	194	2453	**2**
3	amaudio2	Lineage Amlogic	1465	197	1261	**1**
4	anc_hs	Huawei P20 Pro	1080	293	802	0
5	hicam_buf	Huawei Mate 10	630	194	1158	0
6	efuse64	Lineage Amlogic	879	196	1205	0
7	meson_uvm_allocator	Lineage Amlogic	395	196	1127	0
8	vout2_mod	Lineage Amlogic	1721	196	1148	0
9	vout_mod	Lineage Amlogic	1744	193	1144	0
10	cvbs_out	Lineage Amlogic	1881	198	1272	0
11	hbtp_input	Lineage BQ	1387	213	1809	0
12	dolby_fw	Lineage Amlogic	540	253	1124	3
13	audio_data	Lineage Amlogic	224	203	814	2
14	video_composer	Lineage Amlogic	2544	199	1222	1
15	sensors_ssc	Lineage Xiaomi	356	205	1417	1
16	anc_hs_default	Huawei P20 Pro	156	264	1426	1
17	maxim	Huawei P20 Pro	879	198	1182	2
18	msm_ispif	Lineage Xiaomi	1832	194	593	4
19	msm_ispif_32	Lineage Xiaomi	1326	193	859	3
20	msm_csiphy	Lineage Xiaomi	2355	292	598	3
21	msm_csid	Lineage Xiaomi	1153	195	594	3
22	msm_flash	Lineage Xiaomi	1207	267	593	3
23	msm_ir_led	Lineage BQ	360	213	861	4

CPU-hours = fuzzing time * 64 (the number of cores used)
Cov = total number of basic blocks of kernel code reached

5.4 Discovered vulnerabilities

The four previously unreported bugs were found in the Lineage Amlogic kernel[14]. Given that the Lineage Amlogic kernel is based on Android TV kernels, we have reported these bugs to Google's Android Security Team, and they were all confirmed. In Table 2, we provide the type of vulnerabilities and Google's severity assignment. We provide more technical details about each of the vulnerabilities in Appendix A.

6 Limitations

In our implementation, in order for LiLi to identify the driver's symbols, it needs to find the names of its functions and data objects. Currently, this is done by

[14] Three of these were found directly through the fuzzing experiments, and one was found upon manual code inspection of a driver that contained one of the other three bugs.

Table 2. Newly Discovered Vulnerabilities

Driver	Vulnerability Type	Severity
`meson_ion_delay_alloc`	Double free	High
`meson_ion_delay_alloc`	Memory leak	High
`ionvideo`	Arbitrary write	Moderate
`amaudio2`	Null ptr dereference	N/A

scanning the orphan code's source files, but we only need function declarations, and there is no requirement to parse function bodies, nor compile the files. An alternative approach that we can use is to recover function names directly from the `vmlinux` binary. To do this, we start with a single function name, e.g., the `ioctl` handler that we want to analyze. As a local function symbol, it should fall between two FILE symbols in the symbol table. Taking all entries between these two FILE symbols allows us to obtain the remaining local symbols that belong to the driver. In order to find all of the driver's global symbols, we analyze all call sites in `vmlinux` that either belong to this initial set or jump to this set. We then update our initial list of symbols (and filter out standard kernel API) by recursively repeating the same procedure on this updated set.

7 Conclusion

In this paper, we approached the problem of emulating arbitrary Android kernel images, the majority of which are not supported by existing emulators. We proposed LiLi, a tool that can be used to disconnect isolated parts of an Android kernel image, such as built-in drivers, and to re-link them to a version of the Linux kernel that can be emulated. We applied LiLi to a collection of Android kernel drivers from various vendors, which allowed us to fuzz test them. Following an analysis of the results of these experiments, we discovered 4 zero-day vulnerabilities, all of which were confirmed by the manufacturer (Google).

We believe the approach implemented by LiLi can be used to simplify dynamic analysis of various parts of custom Android kernels which can otherwise be difficult to test, making the dynamic analysis of the Android kernel more accessible. To this end, we make LiLi available as open-source on Github[15], where we also provide extended experimental results.

References

1. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track. vol. 41, p. 46. California, USA (2005)
2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM **53**(2), 66–75 (2010)

[15] https://github.com/AlexLB99/LiLi

3. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
4. Chipounov, V., Kuznetsov, V., Candea, G.: S2e: A platform for in-vivo multi-path analysis of software systems. Acm Sigplan Notices **46**(3), 265–278 (2011)
5. Davidson, D., Moench, B., Ristenpart, T., Jha, S.: FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: 22nd USENIX Security Symposium (USENIX Security 13). pp. 463–478 (2013)
6. Keil, S., Kolbitsch, C.: Stateful fuzzing of wireless device drivers in an emulated environment. Black Hat Japan (2007)
7. Koscher, K., Kohno, T., Molnar, D.: SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15) (2015)
8. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR.CHECKER: A soundy analysis for linux kernel drivers. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1007–1024 (2017)
9. Patrick-Evans, J., Cavallaro, L., Kinder, J.: POTUS: Probing Off-The-Shelf USB drivers with symbolic fault injection. In: 11th USENIX Workshop on Offensive Technologies (WOOT 17). USENIX Association, Vancouver, BC (Aug 2017), https://www.usenix.org/conference/woot17/workshop-program/presentation/patrick-evans
10. Peng, H., Payer, M.: USBFuzz: A framework for fuzzing USB drivers by device emulation. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2559–2575 (2020)
11. Pustogarov, I., Wu, Q., Lie, D.: Ex-vivo dynamic analysis framework for android device drivers. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1088–1105. IEEE (2020)
12. Renzelmann, M.J., Kadav, A., Swift, M.M.: SymDrive: Testing drivers without devices. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 279–292 (2012)
13. Rubini, A., Corbet, J.: Linux Device Drivers. Nutshell handbooks, O'Reilly & Associates (2001), https://books.google.ca/books?id=97eyKSX0oCYC
14. Schumilo, S., Spenneberg, R., Schwartke, H.: Don't trust your usb! how to find bugs in usb device drivers. Blackhat Europe (2014)
15. Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.P., Franz, M.: Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In: 2019 Network and Distributed Systems Security Symposium (NDSS). pp. 1–15. Internet Society (2019)
16. Stuart, H.: Hunting bugs with coccinelle. Master's Thesis (2008)
17. Talebi, S.M.S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A.A., Qian, Z.: Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 291–307 (2018)
18. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS. vol. 14, pp. 1–16 (2014)

A Discovered vulnerabilities: technical details

In this appendix, we elaborate on each discovered vulnerability from Section 5.

Double free. Driver `meson_ion_delay_alloc` defines IOCTL `UVM_IOC_ALLOC`. Here, it calls `uvm_alloc_buffer`, which allocates a `uvm_buffer` based on data passed through the IOCTL's `arg` parameter, and stores a pointer to the buffer in a `dma_buf`. This same function then tries to give the `dma_buf` a file descriptor `fd`, and if this fails, it then `kfree`'s the `uvm_buffer`. However, the `dma_buf`'s reference count will also be dropped, causing the release function `meson_uvm_release` to be called, where the `dma_buf`'s `uvm_buffer` is once again `kfree`'d. Hence, we have a pointer that is double freed, which produces undefined behaviour. One way to reliably cause the `fd` registration to fail is to repeatedly call `UVM_IOC_ALLOC`. This will register `dma_buf`'s over and over, until the maximum number of given `fd`'s has been reached.
Memory leak. Command `UVM_IOC_ALLOC` from driver `meson_ion_delay_alloc` allocates a `uvm_buffer` based on data passed through the IOCTL's `arg` parameter. If another driver has the file descriptor for this buffer (e.g., if a user-space program is interacting with both drivers), then it can call `dma_buf_map_attachment` on this buffer, which will eventually call `meson_uvm_alloc_buffer`. This uses `ion_alloc`, which allocates a buffer inside memory pools belonging to the `ion` subsystem, based on data in the `uvm_buffer`. This includes the buffer's size, which it takes from a field in the `uvm_buffer` allocated as a result of the `UVM_IOC_ALLOC` IOCTL. Since the `meson` driver does not contain any checks for this buffer size provided by the user-space, this means that an application interacting with these two drivers could create buffers of arbitrary size in the `ion` memory pools, potentially draining them and denying other processes from using these pools. Note that this would not normally be found by fuzzing, as its exploitation requires a fair amount of set-up (e.g., there needs to be a custom driver on top of the `meson` driver).
Arbitrary write. The `ionvideo` driver defines IOCTL `vidioc_qbuf`, which takes as input a pointer `p` to a `v4l2_buffer`. Then, the `vidioc_qbuf` function writes to an array at index `p->index`, but without first validating the value of `p->index`. Since the contents of the `v4l2_buffer` that `p` points to are user-provided, a user could control the write address of this particular operation.
Null ptr deref. The `amaudio` driver creates multiple `dev` files at insertion time (`amaudio2_out`, `amaudio2_in`, etc). Several of these files use the same IOCTL handler, wherein some commands call `mutex_lock` on `amaudio->sw.lock` and `amaudio->hw.lock`. However, these locks are only initialized via `mutex_init` in the `open` function if the file being opened is `amaudio2_out`. Thus, invoking IOCTLs that lock these locks with other `dev` files that use the same IOCTL handler (e.g., `amaudio2_in`) will result in an attempt to lock an uninitialized lock, ultimately causing a null pointer dereference.

B Related Works

Given the variety of challenges surrounding dynamic analysis techniques, in many cases, researchers limit the scope of their work to a single type of driver. vUSBf [14], which provides a framework that increases the performance of USB device driver fuzzing, by using the USB redirection protocol to communicate with these devices in virtual environments (with virtualization enabled). Similarly, POTUS [9] also enables the fuzzing of USB drivers in virtual machines, but allows for emulation of arbitrary USB devices as well, improving ease of use. Peng and Payer [10] would later propose a similar tool, but this time with a lesser reliance on symbolic execution, mitigating the associated overhead and scalability issues. Another popular area of focus is the kernel's WiFi drivers and devices. Some approaches will emulate certain WiFi devices in order to fuzz drivers that use these devices (e.g., Keil and Kolbitsch [6], with IEEE 802.11 devices), typically with the goal of finding vulnerabilities in the syscall interface. In contrast, PeriScope [15] explores the hardware-OS boundary by monitoring the two primary types of read accesses (MMIO and DMA) issued by drivers to their devices, and injecting fuzzed values whenever such a read is encountered. Overall, these techniques perform well in their niche, however, they naturally lack the breadth that we aim for.

There are also more generalized tools that allow for the automated analysis of the Linux kernel and its drivers. For instance, Charm [17] runs the device driver in a virtual machine, and provides a way for that driver to communicate with physical devices. It achieves this by redirecting I/O calls issued by the driver through a customized USB channel. SURROGATES [7] and AVATAR [18] offer similar functionality, but for embedded systems. These techniques opt to redirect I/O accesses to physical devices using FPGA bridges and the JTAG debugger backend, respectively. All of these approaches take an additional step toward complete driver emulation, but they still rely on the presence of physical hardware, which can be expensive and difficult to acquire. Moreover, even with the problem of hardware being resolved, there are still some challenges on the software side. This is especially evident in the case of Charm, which deals with kernel drivers. In order to resolve all the associated software dependencies, the authors mention that an experienced security analyst would normally take several days to port a driver to a custom kernel that can be emulated.

One prevalent solution to the problem of missing peripherals is the use of symbolic execution. For instance, SymDrive [12] allows for the creation of symbolic devices that specialized instrumented x86 Linux kernel drivers can interact with. This is done with the help of the S2E [4] platform, which can be used to symbolically execute an entire operating system's stack. Another example is FIE [5], which can be used to detect vulnerabilities in MSP430 microcontroller firmware. It uses the KLEE [3] symbolic execution engine, and intercepts a driver's accesses to memory-mapped registers (used for hardware interaction), returning custom symbolic values provided by FIE. In a similar way to PeriScope (sans the symbolic execution), FIE is therefore able to execute a driver without the presence of its corresponding devices. This technique only targets simple firmware programs relevant to MSP430 microcontrollers, meaning there is no guarantee that a similar technique could be used for the analysis of more complex drivers. More generally, symbolic execution techniques tend to suffer from slowness caused by the constraint solving problem, as well as path explosion issues. Moreover, in order to find bugs, custom checkers need to be used to solve the constraints produced as output of symbolic execution. Different kinds of bugs will require different checkers, and writing these takes time and experience.

Other techniques (e.g., [16, 8, 2]) opt for a static approach to vulnerability detection. The advantage of static approaches is that they allow to entirely bypass the challenge of code execution. However, getting false positives is a flaw that is universal across static analysis techniques. The more false positives there are, the more time and expertise is needed to identify real bugs.

Finally, the EASIER framework [11] mitigates some of the challenges of dynamic analysis (e.g., needing hardware or complex emulation) by resolving certain hardware and software dependencies, and by taking advantage of its dynamicity to allow for easy verification of false positives. However, it, like similar dynamic techniques, still depends on the insertion of a separately compiled stand-alone module into an emulator, rather than the insertion of a portion of a kernel binary. In other words, using it requires access to the source code of the entire kernel, as well as the ability to compile LKMs against that kernel. Not only is this not always possible, but it also limits itself to only the emulation of drivers, as opposed to any other selected part of the kernel.

C Experimental dataset

In Table 3, we include the 56 drivers from our initial testing set. We also indicate in the DM (Device Model) column those for which the Evasion framework was able to reconstruct a correct device model. For all these drivers, LiLi was able to reconstruct the correct relocation table.

Table 3. Experimental dataset

Driver	**DM**	**Driver**	**DM**	**Driver**	**DM**	**Driver**	**DM**
audio							
maxim	✓	tfa98xx	X	anc_hs	✓	anc_hs_default	✓
dolby_fw	✓	amaudio2	✓	audio_info	✓	efuse64	✓
camera							
hicam_buf	✓	hwcam_cfgdev	X	laser_module	X	msm	X
msm_csiphy	✓	msm_actuator	X	msm_ispif	✓	msm_ispif_32	✓
msm_isp	X	msm_sensor_driver	X	msm_flash	✓	msm_csid	✓
msm_eeprom	X	msm_cpp	X	cam_cci_dev	X	msm_ir_led	✓
msm_ir_cut	X	cam_eeprom_dev	X	cam_flash_dev	X	cam_actuator_dev	X
vm	X						
video							
msm_vidc_4l2	X	amlvideo2	X	picdec	X	ionvideo	✓
video_composer	✓	videotunnel	X	vout_serve	✓	vout2_serve	✓
vbs_out	✓	wifi_dt	X				
other							
hismart_ar	X	msm_rng	X	sde_rotator_dev	X	msm_vidc_4l2	X
msm_glink_pkt	X	qseecom	X	qcedev	X	sensors_ssc	✓
radio-iris	X	mdss_rotator	X	hbtp_input	✓	nq-nci	X
pn547	X	smartcard	X	aml_aucpu	X	meson_ion_delay_alloc	✓
meson_uvm_allocator	✓	msm_smd_pkt	X				