Emulating Android Device Drivers via Replicated Execution Context

Alex Le Blanc

A thesis
in
The Department
of
Concordia Institute for Information Systems Engineering (CIISE)

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Applied Science (Information Systems Security)
Concordia University
Montréal, Québec, Canada

November 2023

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Alex Le Blanc**

Entitled: **Emulating Android Device Drivers via Replicated Execution Context**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair and Examiner
Dr. Mohammad Mannan

_____ Examiner
Dr. Amr Youssef

_____ Supervisor
Dr. Ivan Pustogarov

Approved by _____
Dr. Jun Yan, Graduate Program Director
Concordia Institute for Information Systems Engineering

_____ 2023 _____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Emulating Android Device Drivers via Replicated Execution Context

Alex Le Blanc

The Android operating system is characterized by the many variants of its kernel, each variant being specific to the manufacturer and the hardware it is running on. At scale, this makes emulating these kernels highly challenging, as existing emulators implement only a handful of hardware boards, and extending them is impractical due to the plethora of devices in existence today. Inability to emulate Android kernels means that dynamic analysis, which can be very effective in finding security vulnerabilities, is only possible on the device itself. This not only makes such analysis more expensive, as one needs to purchase physical copies of the device, but it also limits its usefulness, as some techniques require fine-grained monitoring of the internal state, which can only be achieved in an emulated environment.

In this thesis, we present LiLi, a framework that allows security analysts to emulate selected bug-prone parts of an Android kernel. This limits the number of hardware peripherals that we need to deal with, and also allows for a more targeted analysis. It takes advantage of the fact that the Android OS is based on Linux, whose default configuration is supported by existing emulators. LiLi executes a modified stock Linux kernel within an emulator, wraps and injects the Android kernel under test into the same memory space, connects the two kernels, and successfully redirects execution to any portion of the Android kernel, while providing it a valid execution context.

We evaluate our approach by further extending LiLi with coverage-based fuzzing and testing 57 Android device drivers from ten different Android kernels, from a total of four vendors. For 40% of the drivers, LiLi is able to successfully restore a valid execution context and enable correct emulation. Using LiLi, we were able to discover 4 zero-day vulnerabilities (2 of which are high-severity) which were confirmed by the Google security team and were awarded bounties totaling 6,000 USD.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Ivan Pustogarov, for his continued support and guidance throughout my master's degree. His expert insights have been invaluable not only for the realization of this project, but also for my development as a researcher. Additionally, I would like to thank the Concordia Institute for Information Systems Engineering, as well as Concordia University, for providing both an excellent learning environment and a variety of resources that have been key to this project's success. This includes the university's administrative personnel, who have been highly helpful in facilitating the completion of my degree. A special word of thanks as well to the examining committee for their astute and helpful suggestions. I would also like to recognize Google's security team for their responsiveness and friendliness throughout our communications.

I am deeply grateful to my parents, whose unwavering support and affection have been indispensable in this process. Their encouragement and faith in my abilities have been a constant source of motivation, both in my academic journey, and in my life in general. I would also like to extend a similar thanks to the rest of my family, with a particular emphasis on my grandparents, siblings, and step-family. Indeed, they all substantially contribute to my determination, happiness, and success. Lastly, I would like to acknowledge my close friends, particularly those who had an impact, in one way or another, on this project (David, Marc, Michael, and Steven). I am grateful to be able to relax and play games with them, and they have always been happy to hear about my work and to bounce off ideas.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Motivation . . . 1
1.2 Research objective . . . 2
1.3 Contributions . . . 3
1.4 Outline . . . 4

2 Background 6

2.1 Linux device drivers . . . 6
2.2 Dynamic analysis and fuzzing . . . 7
2.3 ELF symbols and relocations . . . 7
2.4 Loadable kernel modules . . . 9
2.5 The evasion kernel . . . 9
2.6 Kernel images: vmlinux and vmlinux.o . . . 10
2.7 Syzkaller . . . 10
2.7.1 Writing syscall descriptions . . . 11
2.7.2 Specifying syzkaller configuration options . . . 11

3 Literature Review 12

4 Overview of LiLi 16

4.1 Compiling kernels . . . 18
4.2 Standardizing kernel-driver API . . . 18
4.3 Using LiLi . . . 18

4.4 Satisfying dependencies in evasion kernel . . . 19
4.4.1 Software dependencies . . . 19
4.4.2 Hardware dependencies . . . 20
4.5 Loading module with QEMU . . . 20
4.6 Fuzzing module with syzkaller . . . 20

5 First Iteration of LiLi **22**
5.1 Implementation . . . 22
5.1.1 Preparing an empty module . . . 23
5.1.2 Identifying driver's symbols . . . 23
5.1.3 Recursively copying relevant relocations . . . 25
5.1.4 Patching certain ELF sections . . . 27
5.1.5 Setting module entry point . . . 28
5.2 Limitations . . . 30
5.2.1 Needing source files . . . 30
5.2.2 Many potential edge cases . . . 31
5.2.3 Limited availability of vmlinux.o . . . 31

6 Second Iteration of LiLi **32**
6.1 Implementation . . . 32
6.1.1 Preparing an empty module . . . 33
6.1.2 Identifying the driver's symbols . . . 33
6.1.3 Patching the driver's functions . . . 34
6.1.4 Recursively copying relevant relocations . . . 38
6.1.5 Setting the module entry point . . . 39
6.2 Limitations . . . 41
6.3 Potential directions for improvement . . . 41

7 Testing LiLi **43**
7.1 Introductory remarks . . . 43
7.2 Improving the evasion framework . . . 43
7.3 Finding fuzzable drivers . . . 44
7.3.1 Studied kernels . . . 44
7.3.2 Determining candidate drivers . . . 44
7.3.3 Fixing kernels . . . 46

7.3.4 Testing candidate drivers for fuzzability 47
7.4 Fuzzing experiments 47
7.4.1 The fuzzer 48
7.4.2 Computer specifications 50
7.4.3 Fuzzing results 50
7.4.4 New vulnerabilities 52

8 Conclusion and Future Work **55**

Appendices **62**

Appendix A Compiling the Studied Kernels **63**

Appendix B Driver Information **66**

Appendix C Driver Loading Instructions **69**

List of Figures

1 Overview of entire tool-chain 16
2 First iteration of LiLi 22
3 Second iteration of LiLi 32

List of Tables

1 Relocation types for different instructions 35
2 Example relocation entries 38
3 Studied Linux 4.9 kernels 45
4 Driver fuzzability test: does each driver have reachable IOCTLs? . . . 48
5 Fuzzing results 51
6 Newly discovered vulnerabilities 53
7 How to obtain and compile the kernels 63
8 Driver information 66
9 Special driver loading instructions 69

Chapter 1

Introduction

1.1 Motivation

Android is among the most popular mobile and embedded operating systems occupying about 70% of the market share and powering millions of devices today [17, 35]. The central part of any Android-based device is the Linux kernel that manufacturers modify and extend to add support for new hardware and new features. Usual modifications include drivers for new peripherals (e.g. cameras, touch screens, temperature sensors, interrupt controllers, etc), new subsystems, as well as changes to the existing kernel's core subsystems such as the memory allocator. The code added by the manufacturers is usually much less audited compared to the core kernel subsystems and historically has been a source of many vulnerabilities.

Dynamic analysis is one of the proven and effective methods in finding security bugs, and it achieves its full potential in an emulated environment. Indeed, 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 recompile the code and purchase physical devices. Unfortunately, emulating custom Android or Linux kernels is problematic. For correct emulation, every new physical peripheral requires the corresponding emulated version, and existing emulators do not provide them. In fact, one of the more popular emulators, QEMU [6], can only properly boot a handful of hardware boards. The naive way to solve this problem would be to develop an emulated version for each peripheral. In practice, there are too many of them to make it economically viable.

We note, however, that in most cases a security analyst, rather than analyzing the whole kernel, would be interested in only specific bug-prone parts of this kernel, e.g., network packet parsing code [33], or a system call handler [28, 38]. Because of this, it is natural to ask if it is possible to emulate only specific isolated kernel parts[1]. Unfortunately, executing and emulating code starting from an arbitrary location 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 manifested as a memory access violation error when it tries to interact with uninitialized kernel subsystems.

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. With such expected and valid execution context, 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's 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.

1.2 Research objective

The primary goal of this thesis is to develop a framework that allows for the execution of arbitrary parts of an Android kernel within an emulated alien environment.

[1] An alternative would be to reconfigure the kernel under test and to try to remove all the modifications added by the manufacturer except for a small number of them that we would be interested in analyzing. This way, we could potentially reduce the custom kernel to one that is close enough to the stock Linux kernel (e.g., in terms of dependencies and configuration options) to be emulated. In practice, however, this approach is not generic and often infeasible. First, it requires a good knowledge of the kernel under test. Second, the modifications are often quite complex and usually cause a chain of dependency issues. We elaborate further in Subsection 7.3.1.

Moreover, we aim to fuzz-test these kernel portions in order to discover real-world vulnerabilities. In other words, we set out to answer the following research questions:

1. **Question 1**: Is there a way to execute a portion of an Android kernel in binary format within an alien environment that approximates an intended execution context?

2. **Question 2**: Can we show that this approximation is precise enough to discover real-world vulnerabilities via fuzz-testing?

1.3 Contributions

In this thesis, we develop a framework for reconstructing valid execution contexts for isolated Android kernel parts, and we focus on device drivers. This, in turn, allows us to emulate them, and thus apply dynamic analysis techniques. In doing so, we were able to answer our two research questions, both positively.

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 the driver. And 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 start by running in an emulator the stock Linux kernel, and we 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 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, and in order to deal with missing peripherals, we adopt an approach similar to [28], where we allocate memory buffers at the ranges where the driver expects to find the peripheral control and data registers.

To evaluate our approach, we first explored ten different Android kernels, from

a total of four vendors (Lineage[2], Huawei, HTC, and Samsung), in order to build a corpus of 57 unique drivers. Of these drivers, 40% could be emulated by our framework without major interventions. With emulation possible, we fuzzed these drivers, and ultimately discovered four zero-day vulnerabilities in Google smart TV kernels. These 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 (i.e., ones depending on the availability of ION memory pools).

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

1. **LiLi (Linux in Linux).** We have developed a framework that allows the execution of portions of kernel code in an alien environment. We use this to enable the off-device dynamic analysis of arbitrary parts of Android kernel binaries. Off-device analysis is more efficient than on-device, and therefore can be an interesting approach for security analysts who are faced with time and cost constraints.

2. **Fuzzing emulated drivers.** We show how to emulate drivers using our approach, as well how to fuzz the IOCTLs of these emulated drivers.

3. **Discovery of zero-day vulnerabilities.** Following fuzzing experiments, we found four zero-day vulnerabilities, all of which were confirmed by Google's Android Security Team. Two of them were rated "High" severity, while another was rated "Moderate" severity. We also received bounties totaling 6,000 USD.

1.4 Outline

The rest of the thesis is structured as follows. In Chapter 2, we briefly explain some concepts and technologies that are fundamental to the understanding of our work. Then, in Chapter 3, we provide a review of academic literature that allows us to contextualize and further motivate this research. In Chapter 4, we provide an overview of LiLi, as well as the entire the tool-chain that it fits into. We further

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

elaborate on our framework in Chapters 5 and 6, where we present two different iterations of our approach (based on relocatable and statically linked kernel images, respectively). We then present fuzzing experiments conducted on drivers emulated using LiLi, and discuss vulnerabilities that we found along the way in Chapter 7. Finally, in Chapter 8, we provide some concluding thoughts, as well as a discussion of future research directions.

Chapter 2

Background

In this chapter, we discuss certain key concepts that this thesis relies on.

2.1 Linux device drivers

There are three main parts that go into the proper execution of an operating system: the user space (i.e., where user applications run), the kernel space, and the hardware. Whenever an application needs to interact with hardware, the kernel will always act as an intermediary, both for security and convenience reasons. To carry out this communication, hardware devices will have a variety of corresponding drivers in the kernel. Once loaded, these drivers create a device file in the user space's `/dev/` directory, to which applications can issue system calls. The device files relay these calls to the relevant kernel driver, which works together with the rest of the kernel to perform some operation on a device. Once done, the same path is taken in reverse to communicate the result of the operation to the user space application. For further reading on Linux device drivers, we refer to [12].

Compared to other parts of the kernel, the device drivers are notoriously bug-prone [9, 11, 25, 34, 36]. One of the main reasons for this is the near-constant development of new peripherals by third-party manufacturers, for which drivers of varying code quality are written. This is especially problematic because drivers act as a bridge between the user space and the kernel space, meaning that exploits will often result in an escalation of privilege. For these reasons, we place a particular focus in this thesis on device drivers, as opposed to other parts of the kernel.

2.2 Dynamic analysis and fuzzing

In terms of automated vulnerability detection, there are two main classes of techniques available: static and dynamic analysis. In the former, source code is directly audited, and no execution of the code is required. On the contrary, dynamic analysis is characterized by its evaluation of a program's behaviour as it executes. For the research discussed in this thesis, we focus on the latter.

One of the most efficient, proven, and popular dynamic techniques is fuzzing. In this approach, randomized inputs to a program are automatically generated, and the resulting execution of the program is analyzed. The goal is to provide unexpected inputs that would trigger a crash, or some other unintended behaviour. Older kernel fuzzers, such as sysfuzz [39] and tsys [22], will generate these inputs entirely randomly. Others, such as iknowthis [2] and Trinity [4], will do so with knowledge of valid inputs (e.g., their expected type). These can serve as a starting point for the fuzzing, for instance by mutating them to obtain slightly invalid inputs, thereby reducing the chances of wasting time on wildly incorrect inputs that can be caught by checks at several levels. Finally, some of the more state-of-the-art fuzzers, such as syzkaller [15] and the many variants of AFL [1] (e.g., [31], [16], [3]), use heuristics to guide the fuzzer towards unexplored paths. Here, one common technique is to consider a set of various previous inputs, and to mutate whichever ones achieved the greatest code coverage. Due to these advantages, we use syzkaller in our fuzzing experiments.

2.3 ELF symbols and relocations

The executable and linkable format (ELF) is the standard format for any type of compiled file in Linux (e.g., executables, object files, etc.). While it sees much use in the user space, it also has applications in the kernel space, such as in kernel images (e.g., `vmlinux`) and in loadable kernel modules. In particular, there are two main types of metadata that it specifies that are especially of interest to us, namely symbols and relocations. Relocations allow for the dynamic patching of a binary based on the symbols they refer to. For instance, a program that calls the `printk` function could have a relocation that modifies the calling instruction to use the address given by the `printk` symbol. We also occasionally refer to a third type, sections, which are just contiguous parts of a binary that serve a 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. 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 in some rare cases, a checksum (in the case of CRC symbols, discussed later on)
3. Binding: the visibility of the symbol, typically 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 in the linking process.

ELF relocation entries, on the other hand, provide a way for executables to dynamically modify parts of the binary that might be difficult to pre-determine during the compilation process. For instance, if we write a module that needs to store the address of an object that we have defined, we will need the help of the kernel in which the module is loaded to know that address at run-time. For this, a relocation entry is created during the compilation process, and is usually stored in a `.rela` section that is linked with the section its entries will modify (e.g., `.rela.text`'s relocation entries modify instructions at various offsets in `.text`). These relocation entries will contain 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 the field of a structure, rather of the structure itself).

In this thesis, we take advantage of relocations to connect the injected and emulated kernels together. For instance, for a call in the Android binary to a standard kernel function (e.g., `printk`), we can add a relocation entry to a `printk` symbol that is undefined (i.e., does not point to an address). At insertion-time, the emulated kernel will patch the corresponding instruction to use its own `printk` function, rather

than the equivalent one in the injected binary (which will not have a proper execution context).

2.4 Loadable kernel modules

Many drivers are compiled within kernels, and are therefore automatically initialized at boot-time (these are known as built-in modules). The alternative is to insert a loadable kernel module after the booting process, which allows a separately compiled driver to be loaded at run-time. In this thesis, we make use of these types of modules to inject the Android kernel into another kernel. In order to specify the precise location in the injected kernel that we wish to redirect execution to, we take advantage of the fact that loadable kernel modules can designate an entry point via a pointer stored in the `.gnu.linkonce.this_module` section. While only one such pointer can be stored, it is still possible to specify multiple entry points in this way. We achieve this by injecting a custom function that calls a given list of initialization functions, and designating this custom function as the entry point.

2.5 The evasion kernel

The evasion kernel, which is a part of the EASIER framework [28], is a modified version of a stock Vanilla Linux kernel that allows one to insert loadable kernel modules that were compiled against Android host kernels. It accomplishes this in part by altering relocations that use missing symbols to instead use function stubs, thereby "evading" software dependencies. The EASIER framework also provides ways to resolve hardware dependencies, as well as to patch any data structure format differences between the evasion and Android host kernels. We opted to use evasion kernels for all our testing, because they allow us to perform dynamic analysis without the challenges of in-vivo analysis, while also being more user-friendly than other approaches that avoid hardware emulation.

2.6 Kernel images: vmlinux and vmlinux.o

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 some booting information to produce a bootable kernel image: zImage (or bzImage). Therefore, `vmlinux` is effectively an uncompressed version of the bootable kernel image (minus the booting metadata). The difference between `vmlinux` and `vmlinux.o` is that the latter will have relocations for almost any reference to a location outside of a given symbol, whereas the former, having been statically linked, will have these references baked into the binary itself. Both files should normally be available after the compilation process. If one only has access to the bootable image though, it is actually possible to recover `vmlinux`, but not `vmlinux.o`. Thus, it might happen that one has access to `vmlinux` and not `vmlinux.o`, whereas the opposite should not typically occur.

2.7 Syzkaller

The fuzzing software that we use in our experiments is syzkaller [15]. It is a coverage-guided fuzzer, in that it uses the number of basic blocks of code reached in the kernel as a heuristic to determine which execution paths are worth exploring further. It is used to fuzz kernels (primarily Linux), and the inputs it randomizes are system calls, often called `syscalls` (in the case of drivers, these will usually be `IOCTL`s). However, in order to properly target them, syzkaller requires some information about the drivers' `syscalls`, such as the types of variables that they input and output. These are referred to as syscall descriptions. With this in mind, there are two main preliminary steps to be completed prior to fuzzing.

1. Writing `syscall` descriptions
2. Specifying syzkaller configuration options

We elaborate on both in the following subsections.

2.7.1 Writing syscall descriptions

In order for syzkaller to fuzz intelligently and efficiently, it needs to know all kinds of information about our module, such as the command numbers of the IOCTLs we wish to fuzz, information about any structures used by the IOCTLs that would normally be provided from the user space, as well as any specifications about the fields of these structures (e.g., maybe one field is an integer, but is only ever expected to be a number from 0-9). All this information about the driver's interface would need to be provided to syzkaller via `syzlang`, a syscall description language. This may be done either entirely manually or with the assistance of certain interface recovery tools, such as DIFUZE [13].

2.7.2 Specifying syzkaller configuration options

Configuration options can be specified in `.cfg` files. Here, one can designate, for instance, the `syscalls` that they wish syzkaller to target. Another important parameter is `type`, which determines which fundamental "mode" syzkaller will operate in. Since we will be fuzzing kernels emulated in QEMU, there are two different modes that are of interest to us, namely isolated host mode and QEMU mode. The former, hence its name, will treat our QEMU instance as an isolated host, connecting to it and fuzzing directly in that instance (as if it were fuzzing on a phone). The latter will open its own QEMU instances and fuzz within them in parallel. Many configuration options will depend on this `type` parameter. For instance, for QEMU mode, one may also specify the number of parallel virtual machines to run, as well as the path to the filesystem image that syzkaller will be using when opening these QEMU instances.

Chapter 3

Literature Review

In this chapter, we discuss works related to the automated detection of vulnerabilities in Linux kernels. In doing so, we will attempt to identify trends that appear in the literature, with the goal of finding some gaps in research.

Given the variety of challenges surrounding dynamic techniques, in many cases, researchers will limit the scope of their work and focus on a single isolated part of the kernel. For example, many techniques will target USB drivers in particular. This is the case with vUSBf [32], 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 [26] 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 [27] 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 [19], with IEEE 802.11 devices), typically with the goal of finding vulnerabilities in the syscall interface. In contrast, PeriScope [33] 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 are diverse and perform well in their niche. However, they naturally lack the breadth that we aim for. Learning an entirely new framework to test a driver, and being faced with the

possibility that it may not work (e.g., hardware procurement or emulation difficulties), can be a non-negligible deterring factor.

As aforementioned, there are also more generalized tools that allow for the automated analysis of the Linux kernel and its drivers. A lot of the time, these techniques will try to find ways to execute the kernel in a virtual environment, which can help make the bug-finding process more accessible and scalable. For instance, Charm [38] sets out to do exactly this. In particular, it 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 [20] and AVATAR [40] 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. 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. This is yet another obstacle for security analysis in the Linux kernel.

One prevalent solution to the problem of missing peripherals is the use of symbolic execution. For instance, SymDrive [30] 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 [10] platform, which can be used to symbolically execute an entire operating system's stack. Another example is FIE [14], which can be used to detect vulnerabilities in MSP430 microcontroller firmware. It uses the KLEE [8] 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 opt for a static approach to vulnerability detection. For instance, Coccinelle [37] is a tool that uses pattern-matching to detect stack-based buffer overflows and use-after-free vulnerabilities in the Linux kernel, among other codebases. DR. CHECKER [23] is a utility that employs pointer and taint analysis to identify broader classes of bugs, but at the cost of some precision (i.e., it results in more false positives than other techniques). The authors mention that in order to maximize the efficacy of their tool, they target one of the most bug-dense parts of the Linux kernel, the device drivers. On the other hand, Coverity [7] is known for its low rate of false positives. Despite this, it still has some issues with overall actionability. For instance, Imtiaz et al. [18] found that roughly 60% of alerts made by Coverity in the Linux OS were unactionable. The advantage of static approaches is that they allow to entirely bypass the challenge of code execution. This can be especially complicated in the case of Linux kernel driver execution, which will usually require the presence of peripherals. However, as explained in all of these papers, 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.

In summary, we first note that there is a particular focus of Linux kernel security research on device drivers. Authors will usually explain that this is because many devices and their corresponding drivers are produced by third-party developers, leading to wildly varying code quality. Security analysis of drivers does come with its challenges though: static techniques usually have precision issues, whereas dynamic approaches will normally require physical hardware, or complicated emulation. The evasion framework mitigates some of these issues, for instance by providing an easy way to resolve hardware and software dependencies, or 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 standalone binary 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 loadable kernel modules 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.

Chapter 4

Overview of LiLi

In this chapter, we introduce our framework, which we call LiLi (short for Linux in Linux) to make it easier to reference it in the following text. The LiLi framework can be used to effectively inject an Android kernel into a stock Linux kernel running inside an emulator. Once inside, arbitrary parts of this injected kernel can be executed, and therefore dynamically analyzed. In Figure 1, we present a diagram that depicts, at a high-level, the entirety of the tool-chain that we propose alongside LiLi (note that we focus here on `vmlinux`, but the same holds for `vmlinux.o`).

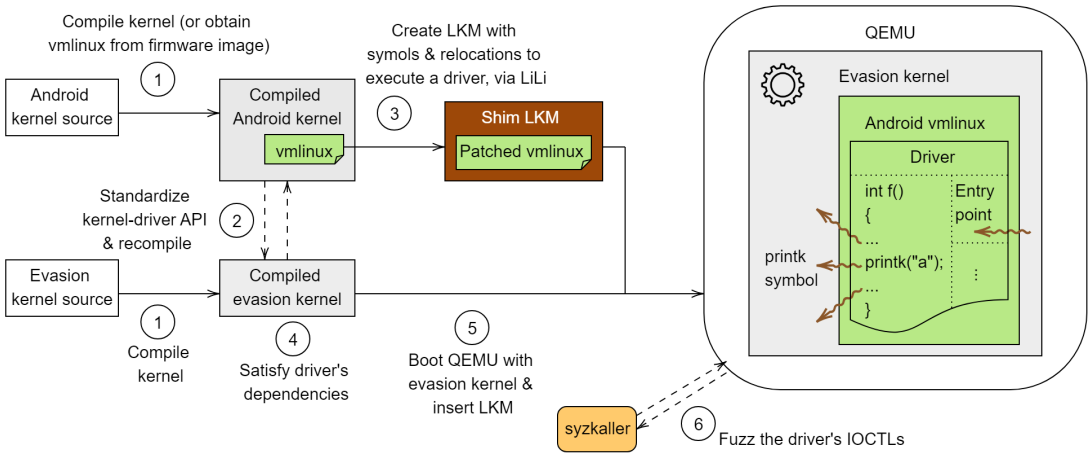

Figure 1: Overview of entire tool-chain

Suppose that we have some part of an Android kernel binary that we wish to fuzz. The first step is to obtain this binary, which can either be done by compiling source code, or by extracting it from firmware images (e.g., via [24]) distributed online

by vendors whenever firmware updates occur. Since this Android binary cannot be simply executed (as it lacks a valid execution context), we need to find some way to emulate it, which normally requires complex hardware emulation. Stock Linux kernels (modified here to be an evasion kernel) are easier to emulate, so we considered ways to load the Android image alongside an emulated stock Linux kernel. The most natural method that we discovered was to create and insert a shim loadable kernel module, which contains the entirety of the Android kernel image (here, "shim" refers to the fact that this LKM only exists as a convenient bridge between the two kernels).

The next problem is that we need to establish some way for the two kernels to interact with each other, in order to take advantage of the existing execution context. To accomplish this, we redirect standard kernel function calls (e.g., `printk`) and variable references from the Android kernel to the relevant symbols in the evasion kernel. We do this by adding relocation entries to the shim module that specify this. Finally, we must communicate the desired entry point of the driver that we wish to emulate, so that the evasion kernel knows where in the Android image to start executing. For this, we once more draw on features of loadable kernel modules. Indeed, they allow for the designation of an initialization function as their entry point, via a pointer stored in a particular ELF section. We use this to specify our own entry point, and we also put forth a novel technique to list several different entry points, should we wish to execute multiple drivers or programs.

Other steps from Figure 1 (e.g., 2 and 4) are only needed to increase the odds of our driver being compatible with the evasion kernel (e.g., by simulating hardware peripherals). Finally, once the driver is correctly executing within the evasion kernel, we may fuzz its various `syscalls`. In summary, we propose six main steps to emulate arbitrary kernel parts, which we list here and elaborate upon in the following sections:

1. Compile the kernels (Android & evasion)
2. Standardize the kernel-driver API between kernels
3. Use LiLi to prepare an injectable Android kernel image
4. Satisfy driver's hardware & software dependencies
5. Load the module in the evasion kernel via QEMU
6. Fuzz the module with syzkaller

4.1 Compiling kernels

There are two kernels that we must compile – the source Android kernel that contains drivers that we want to test, and the evasion kernel, where we will inject the Android kernel. The evasion kernel itself will just be a vanilla Linux kernel that has been patched with the evasion subsystem.

The reason we must compile the Android kernel is that we will need the kernel image (in this case `vmlinux` or `vmlinux.o`), which we can transform into an injectable kernel binary. Again, this could also be done in `vmlinux`'s case by extracting it from an Android firmware image, but because the option is available to us, we opt for compilation. In all likelihood, a cross-compiler will be needed for this.

4.2 Standardizing kernel-driver API

To interface with the kernel, drivers will rely on certain key structures, such as the `file` and `dev structs`. When loading a module, the kernel will look for fields in these structures based on the expected offsets of these fields in the structure. These offsets will vary from one kernel to another, depending, for instance, on the kernel's configuration options (which might enable or disable certain fields).

Because our drivers will have been compiled in an Android kernel, it is quite unlikely that the offsets of all of these fields will match between the two kernels. Therefore, we need to make sure that, at the very least, the `file` and `dev` structures have aligned offsets for their fields in both kernels. The evasion framework provides a tool to identify which fields have misaligned offsets, and which configuration options can be enabled or disabled to realign them. Naturally, once the alignment has been done, any kernel to which modifications have been made should be recompiled.

4.3 Using LiLi

We can now prepare a loadable kernel module that be used to inject this newly obtained Android kernel image, via LiLi. Our approach starts with the kernel image (e.g., `vmlinux`), a list of driver source files, and the evasion kernel's `System.map` file, and then outputs a Linux kernel module file (i.e., one with a .ko extension). The list of source files it takes are the files that would normally be linked together to form a

single driver. This will often end up being a single file, as many driver are entirely contained within just one file. However, there are also some that are written across several files, such as `ionvideo_drv` in the Lineage Amlogic kernel, which is a single module produced from the linking of both `ionvideo.o` and `ppmgr2.o`.

Starting with an almost empty `.ko` file, which we henceforth refer to as our blob, LiLi starts by copying key parts of the kernel image binary into the empty module, and modifying or adding to this imported binary. Such modifications include copying relocations relevant to our driver from the kernel image to our module, as well as adding relocations to undefined symbols for instructions in the driver's functions that refer to symbols defined outside of the driver. The latter is needed because, at load-time, the running kernel will need, as much as possible, to use its own functions and variables, rather than the corresponding copies in the extracted binary. Finally, a custom module entry point is set, such that all of the module's initialization functions are called in the correct order. If attempting to work with multiple distinct modules, one can also consider all of their source files to be the input for a single run of LiLi, and have the entry point once again call the initialization functions of these different modules (this time in a manually specified order). While these general strategies hold for both of LiLi's primary iterations, the mechanics will differ considerably. Because much of this requires the manipulation of ELF metadata, we rely heavily upon the ELFIO [21] library.

4.4 Satisfying dependencies in evasion kernel

Since the goal is to have our evasion kernel behave as similarly as possible to our Android kernel, we need to resolve certain software and hardware dependencies.

4.4.1 Software dependencies

Our driver will expect certain functions that were available in the source Android kernel to also be present in the evasion kernel. To evade these dependencies, the evasion kernel uses a custom subsystem that intercepts relocation requests and patches them with stub functions. In order to best approximate the expected execution, a different stub function would be used depending on what the absent function returns

(a pointer, or just an error code). To do this, it stores the return type of every non-local function called by the module in a separate ELF section, within the module binary. That way, at load-time and run-time, the evasion subsystem will know which stub function to use for each missing function's relocation request.

4.4.2 Hardware dependencies

Our driver will probably also expect certain peripherals to be present. In particular, in order for the module's probe function to be successfully called, the kernel's device tree needs to contain the nodes corresponding to the devices the module expects to communicate with. This will typically involve copying nodes from the Android kernel's device tree blob (`dtb` file) to a `dtb` file in the evasion kernel that will be used in the booting process. We use the same techniques as in the evasion framework to identify and copy these nodes.

4.5 Loading module with QEMU

To run the evasion kernel, we use QEMU. In QEMU's launch command, we will need to include some key details, such as the path to our `dtb` file, the path to the evasion kernel's image, and some networking information that we can use to communicate with the running kernel.

Once our module has been appropriately patched (as discussed in Subsection 4.4), it suffices to simply copy the module over to the evasion kernel (perhaps via `scp`) and to insert it regularly using `insmod`. At this point, the evasion subsystem should take care of resolving all the aforementioned dependencies, and the module should load correctly.

4.6 Fuzzing module with syzkaller

Now that our module has been loaded, we can proceed to fuzzing the contained driver's IOCTLs. First, we will need to communicate the driver's interface to syzkaller, so that we can fuzz in a targeted manner. For this, we write parameter and data structure information relevant to our IOCTLs in `syzlang`, which we opt to do manually.

With some small details being dealt with first, which we further explore later, our driver can finally be fuzzed.

Chapter 5

First Iteration of LiLi

5.1 Implementation

In this chapter, we discuss our strategy for the creation of a loadable kernel module that contains everything needed to execute a driver from within a relocatable kernel image (i.e., `vmlinux.o`). Given the fact that loadable modules and `vmlinux.o` are both ELF relocatable, this represents a natural first step for LiLi. We present a high-level diagram of this iteration of LiLi in Figure 2.

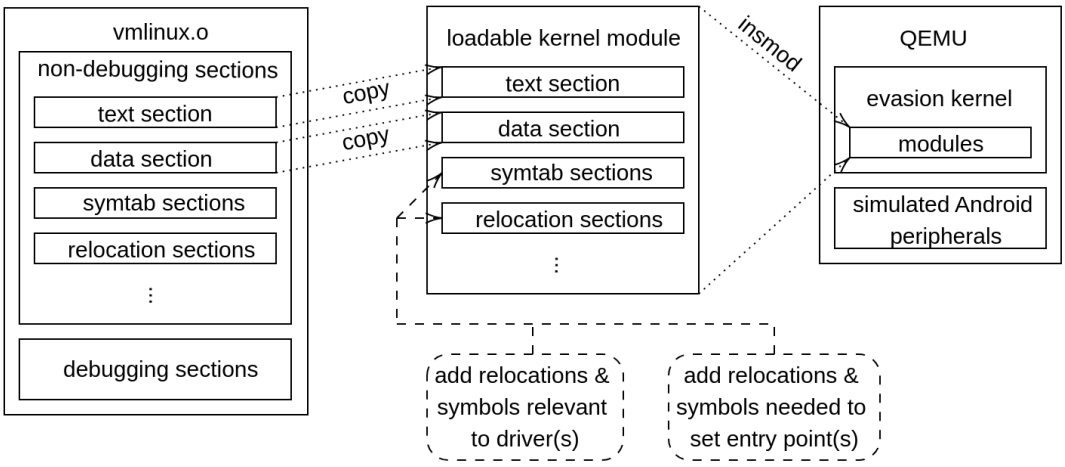

Figure 2: First iteration of LiLi

Put simply, starting with an empty `.ko` file that we create, we recursively look for relocations and symbol table entries relevant to our driver. The idea is to only copy

over ELF sections from `vmlinux.o` to our `.ko` file whenever their content is used in a symbol or relocation. We then add other necessary metadata, which we elaborate upon. This approach can be summarized in five main steps:

1. Preparing an empty module
2. Identifying the driver's symbols
3. Recursively copying relevant relocations
4. Patching certain ELF sections
5. Setting the module entry point

In order to better understand the inner workings of this approach, we will briefly explain each of the above parts in the following subsections.

5.1.1 Preparing an empty module

A C file that contains basic ELF sections that are essential in a loadable kernel module (i.e., `.gnu.linkonce.this_module` and `.modinfo`) is compiled into a `.o` file. This object file is then linked with `vmlinux.o`'s `.text` section to form a `.ko` file, which will serve as our blob, eventually becoming a proper loadable kernel module. Other basic sections are added to the blob, namely `.rela.text` and a `.bss` section of equal size to `vmlinux.o`'s `.bss`.

5.1.2 Identifying driver's symbols

In order to produce a relocatable file that behaves like a given driver, we have to start by figuring out which symbols in `vmlinux.o` correspond to any declarations made in the driver's source files. For this, we consider two main classes of symbols, namely local symbols and non-local symbols. Note that each class of symbol will require a different approach, which we elaborate upon here.

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). We first note that when compiling a C file, the GCC compiler will normally place all these local symbols that correspond to function and variable definitions together in the symbol table right after a single debugging symbol of type "FILE" whose name is the name of the compiled source file. When linking multiple `.o` files together, the GNU linker (LD) will place these groups of local symbols sequentially in the new symbol table, each group still preceded by the corresponding "FILE" symbol.

Now, we recall that `vmlinux.o` is effectively an amalgamation of many `.o` files that have been linked together, and among these linked files are the driver's compiled source files. Thus, LiLi can simply search the symbol table for everything between the FILE symbol with a source file's name and the next FILE symbol. More specifically, it will take all such symbol table entries for symbols that have non-zero size and add them to a list for later use. We discard empty local symbols as they are only relevant to debugging.

Non-local symbols

As for non-local symbols (i.e., symbols with "GLOBAL" and "WEAK" bindings), we cannot use the same approach, as 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). Instead, we need to directly scan the source file itself to recover the names of these global symbols. For this, we use the `ctags` utility, which can be used to output information (e.g., names and descriptors) about all the functions and variables defined in a source file. Setting the `file-scope` flag to `no`, we get specifically the non-local functions and variables. For each of these, we then scan the `vmlinux.o` symbol table for the GLOBAL symbol with the same name. The only exception is if a function has the "`__weak`" descriptor, in which case we scan the symbol table for the WEAK symbol with the same name. In either case, we then add this symbol table entry to our list.

Missed symbols

Because all local symbols defined by our driver will be grouped together, our technique will necessarily recover all such symbols. However, our technique for global symbol recovery, which depends on source file parsing, is not as sure-fire. Indeed, we

found that some variable declarations can be hidden behind the processing of macros, making them difficult to find with only access to source files. Rather than adding these to our list of symbols right away, we will instead include a test to determine whether or not symbols that we wish to add later on are supposed to be defined by the driver.

If we have a non-local object symbol that is not a part of the list of module symbols, we search the target kernel's (in this case evasion kernel) `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. Note that our assumption will not always hold true, but the case where it is false is not too problematic, because any relocations using these symbols will just resolve against the `vmlinux.o` version of that symbol, as opposed to resolving against a stub (or against nothing at all, if not using an evasion kernel).

5.1.3 Recursively copying relevant relocations

By this point, we have a list of the symbol table entries corresponding to functions and data declared by our driver's source files. We will use these `.symtab` entries as a starting point for the ELF section and metadata extraction process.

The first thing to do is to copy these symbols into our module's symbol table. Naturally, this alone will not be sufficient; it is possible that `vmlinux.o` will contain relocation entries that modify memory belonging to the range covered by one of our symbols. For instance, if our driver defines a function `my_func` that calls `printk`, then there will a relocation entry referring to `printk`, with an offset somewhere in `my_func`'s memory range. Therefore, we will also need to copy over any relocations used by the symbols we've added, as well as the symbols that those relocations refer to.

vmlinux.o relocations

Being a relocatable file, `vmlinux.o` is produced right before the linking stage during kernel compilation. This means that any reference in the data and code to some offset whose eventual exact relative position is yet unknown will need to handled with a relocation. At the very least, this includes any reference made within the range of a

symbol to somewhere outside of that range, because there is no guarantee that the distance between any two symbols will remain the same post-linking.

Our approach

In order to add all the relocations affecting any memory belonging to our symbols, we use the function `symbol_extract_v1`, presented in Algorithm 1.

Algorithm 1 symbol_extract_v1

```
1:  function SYMBOL_EXTRACT_V1(SYMBOL s)
2:      M = [s.offset, s.offset + s.size]
3:      get list (R) of relocations involved in M
4:      for r in R do
5:          let z be the symbol r refers to
6:          if z is not yet in our module then
7:              let a be true if z is not in System.map, and false otherwise
8:              let b be true if z has type OBJECT, and false otherwise
9:              if z is local OR (a AND b) then
10:                 copy z's section to our module, if absent
11:                 copy z to the symtab
12:                 symbol_extract_v1(z)
13:             else
14:                 add UND symbol for z to the module
15:         add r's .rela section, if absent
16:         add r to the appropriate .rela section
```

This function takes as input a `vmlinux.o` symbol table entry, as obtained from "`readelf -s`". Let us refer to the relevant symbol as s. It first builds a list (R) of all the relocation entries that modify anything at offsets in s's range. Then, for every such relocation entry, it adds the symbol that it refers to if needed, and then adds that relocation entry into our module. Determining if we need to add a symbol is straightforward: we simply check if that symbol is already present in our module. Figuring out the details of the symbol to be added, however, is somewhat more involved.

Adding a relocation's referenced symbol

When adding new symbols, we first have to determine whether or not we want to add an undefined ("UND") symbol. At module load-time, relocations refering to UND symbols will resolve against a global symbol with the same name in the running kernel (in this case, an evasion kernel). Because we are only considering symbols that have not been added to our module at this point, the global symbols that we look at will normally correspond to externally-defined variables and functions (e.g., `printk`). Thus, broadly speaking, we will want to add UND symbols for non-local symbols we come across, and regular symbols that represent some region of our module for the local symbols that we find.

The exception to this rule is for some non-local object symbols. Because they may have been missed by our driver symbol recovery algorithm (as discussed in Subsection 5.1.2), we will use the aforementioned test and check if a non-local object symbol is absent from the target kernel (via the `System.map` file). If it is, then we assume it is a driver-defined symbol (as opposed to being externally-defined), and therefore elect to add a regular symbol, rather than an UND symbol.

Note that when we choose to add a regular symbol, there will be a few more steps. Naturally, in order for the symbol to be meaningful, we will need to copy the ELF section that this symbol belongs to (via its section index) from `vmlinux.o` to our module, if that has not already been done. Finally, we will also need to copy over the relocations whose offsets are anywhere in the newly added symbol's region. For this, we can just call `symbol_extract_v1` again, this time on the new symbol.

5.1.4 Patching certain ELF sections

During the module insertion process, the module-loading subsystem will go section by section, either copying them in their entirety, or performing some special operations based on their content and metadata (or both). In the latter case, we may need to make some changes to our copied section in order for it to behave as intended.

Consider, for instance, the `__param` section, which contains information about variables whose values can be set via command line parameters. At load-time, the kernel will attempt to infer the number of parameters by dividing this section's size by the expected size of a single `kernel_param` structure. Thus, if we just copy the entire `__param` section from `vmlinux.o` to our module, then the unnecessarily massive size

of this section will cause the kernel to significantly overestimate the number of module parameters. This will cause issues later on in the module loading process, because while the kernel will be trying to retrieve the data from all of the `__param` entries, we will only have added relocations relevant to our module's `__param` entries. This means that all of the entries will have empty fields, leading to null pointer dereferences.

Our solution is to simply construct these problematic sections on our own. For instance, we can create a new `__param` section whose size is equal to the sum of the sizes of symbols that were previously in `__param` (there will only be symbols for our driver's parameters). Then, we can just update the offsets of `.rela__param`'s relocations to match the new positions of the `__param` entries that we moved. This way, the kernel will correctly infer the precise number of parameters that our module has. It should be noted that there are other ELF sections that the kernel treats similarly; these can all be found in `module.c`'s `find_module_sections` function. In our own testing, we only found it necessary to patch the `__param` section in this way, but it is possible that with further testing, it is found that some of the other sections also need to be dealt with. In such a case, the approach would no doubt be highly similar.

5.1.5 Setting module entry point

By this point, we have an almost complete loadable kernel module, but with one missing detail. When loading the module, the kernel needs a way to know what the module's initialization and exit entry points are. Since the scope of this thesis is limited to the fuzzing of IOCTLs, which does not involve module removal, we concern ourselves only with initialization.

Basics of entry points

To know the module's initialization entry point, the kernel will look at the module's `.gnu.linkonce.this_module` section at a particular offset, where it expects to find a pointer to the initialization function. When a source file is compiled into a loadable kernel module, the function that is designated in the `module_init` macro will become the module's initialization entry point. A relocation entry in the corresponding `.rela` section (i.e., `.rela.gnu.linkonce.this_module`) pointing to the right offset is then added, with a reference to the initialization function's symbol. Thus, to add such a

relocation to our blob and to replicate this process, we need two things: the correct offset for the initialization function in `.gnu.linkonce.this_module` and the name of this function. To determine the offset, we can simply compile a dummy module, and observe the offset of the initialization function's relocation entry in the aforementioned `.rela` section. The process of determining the initialization function, however, is a bit more involved, as initialization of built-in modules works quite differently.

There are two key differences between built-in and loadable kernel module initialization:

- A built-in module can have several initialization functions.
- A built-in module can use a variety of different macros to indicate its initialization functions (e.g., `early_initcall`, `device_initcall`, `late_initcall`). These macros specify the order in which initialization functions should be called at boot-time. For instance, a function marked by `early_initcall` will be called earlier than one marked by `late_initcall`.

Recovering init functions and their order

Regardless of the `initcall` macro used, a symbol with the initialization function's name prefixed by "`__initcall_`" will be added to `vmlinux.o`'s symbol table. Moreover, based on the `initcall` macro used, a different suffix will be added to said symbol's name. Thus, by looking for the "`__initcall_`" prefix among the module's LOCAL symbols, we can recover the names of the module'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.

Dealing with multiple entry points

There is just one more problem: recall that a loadable kernel module only has one initialization entry point, whereas we've recovered potentially several from our built-in module. The solution is to create a custom initialization function that calls each of the built-in module's initialization functions in the correct order.

To do this, our implementation compiles an automatically-generated C file with a function (which we will call `my_init_func`) that simply calls an empty dummy function as many times as there are initialization functions to add. The resulting object

file's binary is then copied to the beginning of our module's `.text` section. Because this `.text` section is the same as `vmlinux.o`'s text section, there is plenty of system code at the beginning that is irrelevant to our module, meaning we can overwrite anything there without issue. We then search for all the `bl` assembly statements in the newly copied `my_init_func`, and we add relocation entries for the initialization functions at the offsets of these `bl` statements, making sure to also preserve the order in which these functions should be called (recall that we previously determined this order). Now, it is as if we had a single initialization function, meaning that we can add a relocation entry in `.rela.gnu.linkonce.this_module` that refers to our custom function. At module insertion time, the kernel will call `my_init_func`, which will call all of our built-in module's initialization functions in the correct order. Naturally, this technique also allows us to store two completely distinct drivers (e.g., `msm.c` and `msm_flash.c`) into one file and to have their initialization functions called, though the ordering may have to be manually specified. In any case, we finally have a complete loadable kernel module.

5.2 Limitations

In this section, we discuss limitations that are inherent to the approach that we have proposed. In particular, we have identified three main challenges:

1. We require access to the module's source files
2. There are many potential edge cases to handle
3. Limited availability of `vmlinux.o`

We discuss these limitations in the following subsections, while also describing either solutions or mitigating factors, as relevant.

5.2.1 Needing source files

Firstly, in order to recover the driver's symbols, we currently need to scan the driver's source files (in particular, to obtain the global symbols). If there was a way to identify which global symbols are declared by any of our driver's source files by simply parsing the symbol table, we would no longer need these files at all, making the process even

more straightforward and flexible. That being said, it should be noted that one will usually have access to the source files of the drivers they wish to fuzz anyways. After all, they will often be needed to set up the fuzzer (as is the case with syzkaller), as well as to interpret the results of the fuzzing experiments. Indeed, simply knowing that a driver crashes somewhere, without even knowing if it is a false positive, is not typically meaningful. One will usually use this crash information as a starting point to then manually review the source code for a potential corresponding bug.

5.2.2 Many potential edge cases

We recall that this technique obtains the driver's binary by copying entire ELF sections from `vmlinux.o` to our module, whenever that section contains anything that our driver defines. However, whereas its statically linked counterpart will typically contain around 40 or 50 ELF sections, `vmlinux.o` will contain several tens of thousands of them. While they of course will not all be relevant to our driver, this still makes it difficult to prepare for any edge cases. For instance, we saw in Subsection 5.1.4 that the `__param` section needs to be treated differently than other sections. The most obvious solution to this problem is to find a way inject `vmlinux` instead, which is what we discuss in the following chapter.

5.2.3 Limited availability of vmlinux.o

As previously touched upon, the flexibility of this approach is somewhat dampened by the rather specific conditions under which one would have access to `vmlinux.o`. Indeed, the only typical scenario where one would have it is when they can compile a kernel, producing the corresponding `vmlinux.o` and `vmlinux` files. There is not currently a known way to reverse-engineer Android firmware images, for instance, in order to obtain `vmlinux.o`, which would remove the need to compile the whole kernel ourselves. This is possible with `vmlinux`, however. Therefore, by injecting a statically linked kernel image, we can make our approach more diverse in its applications. As aforementioned, we explore this alternate type of extraction in the next chapter.

Chapter 6

Second Iteration of LiLi

6.1 Implementation

In the previous chapter, we showed how to make an injectable relocatable kernel image that allows the execution of arbitrary drivers, and we discussed some limitations inherent to this technique. In this chapter, we explore a similar strategy, but this time with statically linked kernel images (i.e., `vmlinux`). Transforming statically linked code into relocatable code (which is the case for loadable modules) certainly is less straightforward, but it does allow us to address some of the aforementioned limitations. This therefore represents the second iteration of LiLi. Figure 3 shows the functioning of this new iteration, at a high-level.

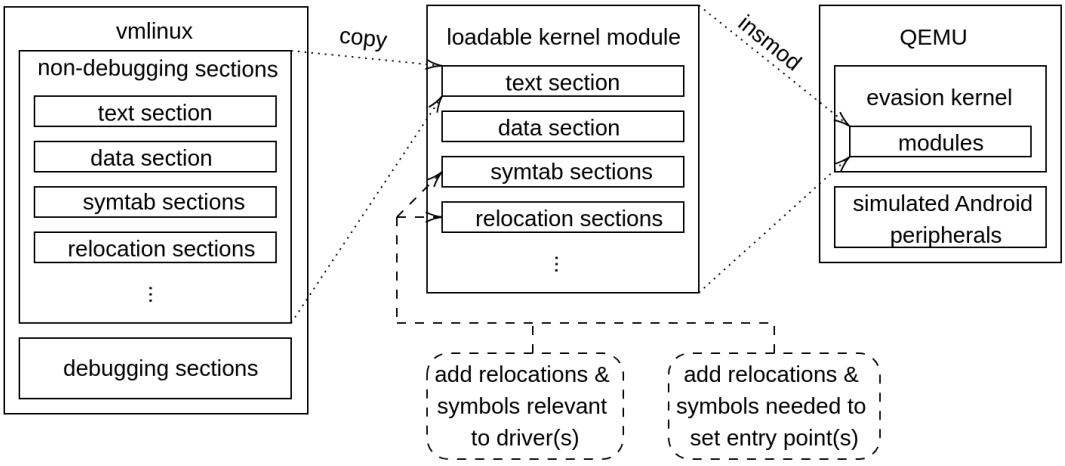

Figure 3: Second iteration of LiLi

The general idea behind this new approach to this is to take the entirety of the `vmlinux` binary, and to store it into a single ELF section of an empty `.ko` that we create. We then add some necessary relocations, as well as metadata essential to the correct operation of our module. This approach can be summarized in five main steps:

1. Preparing an empty module
2. Identifying the driver's symbols
3. Patching the driver's functions
4. Recursively copying relevant relocations
5. Setting the module entry point

In order to better understand the inner workings of this approach, we will briefly explain each of the above parts in the following subsections.

6.1.1 Preparing an empty module

We start by extracting the entirety of `vmlinux`, debugging sections aside, into a single binary file. The reason we even copy parts that may not be necessary for our particular module is that all the code in `vmlinux` is statically linked and PC-relative, meaning that we will need to maintain the same offsets between instructions and their destinations.

A C file that contains basic ELF sections needed in a loadable kernel module (i.e., `.gnu.linkonce.this_module` and `.modinfo`) is compiled into a `.ko` file. We then insert the previously extracted binary into this file's `.text` section. It is important that the whole binary go 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.

6.1.2 Identifying the driver's symbols

In order to determine which parts of the `vmlinux` binary correspond to our driver's code and data, we will need to find the symbol table entries corresponding to that

driver's function and object declarations. To accomplish this, we can use exactly the same strategy that we were using with `vmlinux.o`. We refer the reader to Subsection 5.1.2 for a detailed description, but we also summarize the key points here.

There are two main sorts of symbols to deal with: local symbols (visible only within their source file) and non-local symbols (visible outside). Local symbols for a single source file are all grouped together, along with a special symbol whose name is the relevant source file name. Thus, retrieving the symbol table entries for a source file's local symbols is straightforward. Non-local symbols, however, are dispersed throughout the symbol table, without a clear way to connect them to a particular file. For these, we need to parse the source code to get the names of global declarations. Because non-local symbols need to have unique names, we then just search the symbol table for symbols whose names correspond to the global declarations that we found in the source code.

6.1.3 Patching the driver's functions

As it turns out, we cannot just leave all the instructions to use their original destinations. For instance, many kernel functions, such as `mutex_lock`, will only work properly when using the running kernel's version, rather than the copy of that function in the previously extracted binary. The same goes for global variables declared by the kernel; it is possible that these variables need to interact with various kernel subsystems, meaning that the copy of this variable in our module might end up out of sync with the rest of the kernel. Therefore, we need to scan the instructions of all of the driver's functions, and find all the instructions whose destination is a global symbol not defined by the driver. Once this is done, we then add relocations that refer to an undefined symbol with the same name, allowing these instructions, at load-time, to refer to the running kernel's symbols, rather than the equivalent copy in the extracted binary.

For the sake of simplicity, we separate this process into two main steps.

- Add relocations in `vmlinux` for any instruction whose destination matches the value of a symbol.
- Copy the previously added relocations from `vmlinux` to our module if they refer to a GLOBAL symbol not defined by our driver.

In our own implementation, we use the Capstone engine [29] to disassemble functions and to recover all of the instructions' parameters.

Adding relocations to vmlinux for function instructions

There are four main instructions to watch out for: `bl`, `adrp`, `add`, and `ldr`. A `bl` instruction is an entirely self-contained instruction that branches execution to some relative offset, making it rather straightforward to determine its destination and to add a relocation as needed. The `adrp` instruction, on the other hand, is always paired with either `add` or `ldr`, to respectively store the address or contents of an object in a register. Since these pairs are non-atomic (in fact, different pairs can even overlap), a bit more sophistication is required in order to recover the destination. One could use, for instance, a list to keep track of unpaired `adrp` instructions, to then compare the registers used by subsequent `add` and `ldr` instructions with those of the `adrps` 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 present this idea in Algorithm 2.

Note that each instruction will require a different relocation type. These relocation types are listed in Table 1.

Table 1: Relocation types for different instructions

Instruction	Relocation Type
`bl`	`R_AARCH64_CALL26`
`adrp`	`R_AARCH64_ADR_PREL_PG_HI21`
`add`	`R_AARCH64_ADD_ABS_LO12_NC`
`ldr`	`R_AARCH64_LDST64_ABS_LO12_NC`

Edge cases

Since our instruction-parsing algorithm goes in order from the first to the last instruction (by address), some issues could theoretically arise from `adrp` pairs that are separated by some type of branching instruction. This could, for example, cause our

Algorithm 2 patch_funcs

```
1: function PATCH_FUNCS("FUNC symbol" s)
2:     Let f be the code designated by symbol s
3:     List instr = list of instructions in f
4:     List unmatched_adrps = []
5:     for i in size(func_instrs) do
6:         if instr[i].type is BL then
7:             Dest = instr[i].dest
8:             if vmlinux has a symbol z with value Dest then
9:                 Add relocation at instr[i].address referring to z
10:        else if instr[i].type is ADRP then
11:            unmatched_adrps.add(instr[i])
12:        else if instr[i].type is ADD or LDR then
13:            Let reg1 be the register instr[i] reads from
14:            for j in size(unmatched_adrps) do
15:                Let reg2 be the register unmatched_adrps[j] writes to
16:                if reg1 == reg2 then
17:                    Dest = unmatched_adrps[j].dest + instr[i].dest
18:                    if vmlinux has a symbol z with value Dest then
19:                        Add relocation at instr[i].address referring to z
20:                        Add relocation at unmatched_adrps[j].address referring to z
21:                    unmatched_adrps.remove(index j)
```

static computation of that pair's destination to be incorrect, assuming the common register they use is modified in that branch.

However, during testing, the only instances of this that we observed were `adrp` pairs that were separated by `bl` statements, where the register being written to by the `adrp` and read by the `add/ldr` is a callee saved register. In `arm64`, these registers (i.e., `X19-X28`) are saved and restored by any subroutines that use them. Consider also that `bl` instructions, contrary to `b`, will always return execution to the next instruction after the branch once the subroutine completes. This means that so long as an `adrp` pair uses callee saved registers, we need not worry about their two parts being separated by a `bl` instruction. Since this is the only edge case we observed, we can be fairly confident about the correctness of this algorithm.

Copying relocations using global symbols not defined by our driver

Now, we just need to copy over some of the relocations we added. Determing whether or not a symbol is global is trivial, as that information is provided in the symbol table. Figuring out if a symbol is defined by our driver, on the other hand, can be a bit more involved. To do this, we first consult the list of symbols that we collected, as discussed in Subsection 6.1.2. This list was built using a combination of the `ctags` utility and parsing of the symbol table. However, there might be some driver symbols missing from this list, particularly objects. Indeed, we found that some variable declarations can be hidden behind the processing of macros, making them difficult to find with only access to source files. This is mainly an issue for GLOBAL symbols, because they can only be identified by parsing the source files. 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 target kernel's (in this case evasion kernel) `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. In this case, this means that we will not add a relocation that will refer to the target kernel's version of the symbol. Note that our assumption will not always be true, but the case where it is false is not too problematic, because the alternative would be to resolve that symbol against a stub anyways.

6.1.4 Recursively copying relevant relocations

By this point, we have created some new relocations in our module's functions to ensure that it uses the running kernel's symbols where necessary. However, there are also other relocations that still need to be added, namely those that were already in `vmlinux` at offsets relevant to our driver.

vmlinux relocations

Because `vmlinux` is statically linked, almost any reference from one part of the binary to another will be embedded directly into the binary, as opposed to relying on relocations. The one exception is when an absolute address needs to be stored somewhere, as this cannot be predetermined. For this, `vmlinux` uses `R_AARCH64_RELATIVE` relocations, which store the absolute address corresponding to an offset (specified by the addend) at a given destination. Naturally, the primary use case of such a relocation is for the storage of pointers.

Our approach

We start by considering two different examples of relocation entries obtained from one of the `vmlinux` files we were working with, which we present in Table 2.

Table 2: Example relocation entries

Offset	Info	Addend
ffffff8009934b18	0000000000000403	-7ff66a297c
ffffff8009948358	0000000000000403	a16971b

In the former example, we have a typical case where the addend is pointing to some address in the binary (consider the addend's complement). For most of these cases, we can just copy the relocation entry, adjusting both the offset and addend to work in our module. The exception to this is if the addend points to a symbol that is not defined by the driver: in this case, we should have the relocation refer to an undefined symbol, so that it can use the running kernel's version of that symbol.

In the latter example, we have an addend that is clearly not pointing to a proper address (for instance, this particular image starts at `ffffff8008080000`). This is a

special case, where the relocation implicitly refers to a CRC symbol. These symbols are created when the `EXPORT_SYMBOL` macro is used, and their value parameter is a checksum of the exported symbol (whereas the value of a symbol is usually its position in a binary). In these cases, we will create a CRC symbol and add relocations that refer to these CRC symbols.

We combine all of these considerations into function `symbol_extract`, presented in Algorithm 3. It takes as input a `vmlinux` symbol table entry, and searches for all relocations within that symbol's range. For each such relocation, we then check if it implicitly refers to a CRC symbol or a symbol declared outside of the driver, and add a corresponding relocation accordingly. Note that, like in the previous subsection, we consider any GLOBAL object symbol not present in the target kernel's `System.map` to be declared by the driver. If such a symbol was not a part of our initial list of driver symbols, we call `symbol_extract` on it as well, as it might contain relocations that we will need to copy. If the relocation does not refer to a symbol, or if it is not covered by any of the cases mentioned above (e.g., LOCAL symbols declared by the driver), we simply add a relocation that refers to the corresponding address in our module, rather than to a symbol.

We can then just call `symbol_extract` on the list of driver symbols that we previously created. Once `symbol_extract` completes for every symbol from our initial list, we will have a `.ko` file with almost everything it needs to run properly, save for just one detail.

6.1.5 Setting the module entry point

By now, our loadable module has almost all the code, data, and metadata that it needs to correctly operate. The only thing that remains to be done is to set the module's entry point, so that the kernel knows what code to execute upon module insertion. For this, we can use the same technique as for `vmlinux.o`. This is thoroughly discussed in Subsection 5.1.5, but we also provide a summary of the key points here.

The module's entry point is set in the `.gnu.linkonce.this_section` ELF section, where a pointer to the initialization function is stored. Because some built-in modules have several initialization functions, while only one entry point can be set, we instead set a custom function as the entry point. This custom function calls all the initialization functions that we want to be called at insertion time. Moreover, we

Algorithm 3 symbol_extract

```
1: procedure SYMBOL_EXTRACT(SYMBOL s)
2:     M = [s.offset, s.offset + s.size[
3:     get list (R) of relocations involved in M
4:     for r in R do
5:         s = start address of vmlinux
6:         o = r's offset - s
7:         if ∃ sym z s.t. z's value = r's addend then
8:             if z is a CRC sym then
9:                 Add relevant CRC sym if not present
10:                Add relocation at o referring to CRC sym
11:                Continue to next iteration
12:            else if z is not defined by the driver then
13:                Add relevant UND sym if not present
14:                Add relocation at o referring to UND sym
15:                Continue to next iteration
16:            else if z has not been added to the module then
17:                v = z's value - s
18:                Create copy z' of symbol z at v
19:                symbol_extract(z)
20:        a = r's addend - s
21:        Add relocation at y referring to .text with addend a
```

parse some metadata in the symbol table to recover the intended order that these functions should be called in, which our custom function will reflect.

Note that in this particular case, because the entirety of the `vmlinux` binary is already in the loadable module, we also use this same technique to combine two entirely distinct drivers that need to be loaded together (e.g., `msm.c` and `msm_flash.c`) into one loadable kernel module. This helps reduce the total size of modules in the running kernel, while also avoiding the complications of inter-module references (e.g., managing exported functions).

6.2 Limitations

In Section 5.2, we discussed the three main limitations of extraction from relocatable kernel images, namely:

1. We require access to the driver's source files.
2. The fact that `vmlinux.o` has so many different ELF sections means there are many potential edge cases to handle.
3. Limited availability of `vmlinux.o`.

In the case of this second iteration of LiLi that extracts from statically linked kernel images, only the first issue is still present. We after all do not currently know of an alternative to the parsing of the driver's source files in order to recover the global symbol names. As we discussed in that section, though, the latter two issues are no longer as relevant with this change towards the use of statically linked kernel images.

It should also be noted that this approach does not present any obvious new weaknesses either, when compared to our previous one.

6.3 Potential directions for improvement

In this section, we consider two different directions that could possibly lead to further generalization of this approach, broadening its scope of applications.

Firstly, we note that while this technique was developed for and tested on Android kernel drivers, there does not seem to be very much that is obviously Android-specific.

Thus, it may be straightforward to generalize this approach to work with device drivers in any Unix-like system.

Another interesting avenue to explore would be to see what advantages there might be to just keeping references to functions that are defined outside of the driver, but that are no t present in the evasion kernel. Currently, if our module refers to a function that is defined outside of the module, we create an undefined symbol and refer to that symbol (this is the typical behaviour for regularly compiled loadable kernel modules). Therefore, at load-time, relocations using that symbol will resolve against the evasion kernel's version of that symbol. Since we use the evasion subsystem to load modules, any time these undefined symbols cannot resolve against anything in the evasion kernel, they instead just resolve against a function stub. This can be problematic if we reach an execution path that is not only superficially dependent on that function. To mitigate this, we could check if a function being referred to is not in the evasion kernel (e.g., via `System.map`), and if so, we could keep these references within the extracted binary, rather than referring to the evasion kernel's symbol. The main case where this would not work is when the function depends on the initialization of some subsystem that only exists in the extracted binary, meaning it would not have the right execution context. One solution to this issue would be to observe if a crash happens following a function call, and to fall back to a function stub if indeed a crash occurred.

Chapter 7

Testing LiLi

7.1 Introductory remarks

In this thesis, we also decided to fuzz a variety of kernel drivers, with a special focus on their IOCTL system calls. There are two main motivations for experiments in this context.

1. We can test the validity of LiLi.
2. If we discover new vulnerabilities, these can be reported, which is a direct improvement to the security of kernel drivers.

As for the methodology of these experiments, it can be broken down into two main parts: finding fuzzable drivers, and fuzzing them. We elaborate on these two parts along with relevant results in this section. We also start by briefly describing some improvements to the evasion framework that we needed to make.

7.2 Improving the evasion framework

Since we chose to work with the evasion framework, we needed to make some improvements to mitigate the fact that it sometimes has some trouble correctly replicating the intended behaviour of modules, whether it be at insertion-time, or during the execution of its system calls.

One primary source of problems comes from its replacing of references to functions absent from the running kernel with references to function stubs. If an explored

execution path is not just superficially dependent on a function, then replacing it with a generic stub may produce unintended behaviour. To solve this, we would sometimes copy over source code from the Android kernels to the evasion kernels, which in some cases involved porting entire subsystems from one to the other.

Another issue was that some the Android code would sometimes call functions that were present in the evasion kernel, but were not working properly for various reasons. This was the case, for instance, with the `regmap_read` function. In these situations, we needed to create custom function stubs, so that relocations that refer to these problematic symbols would instead be redirected to our custom stub.

7.3 Finding fuzzable drivers

7.3.1 Studied kernels

In order to identify which drivers can be fuzzed, we first need to select which kernels we will be considering. Because the evasion framework currently only works on specific versions of Linux, we choose kernels using the most recent version of Linux supported by the evasion framework, namely version 4.9. We list the studied kernels in Table 3. Furthermore, we provide instructions on how to obtain and compile all of these kernels in Appendix A.

One alternative to using the evasion framework would be to reconfigure the Android kernels with the stock Linux kernel's default configurations, and to only enable the drivers that we wish to test. In this way, the Android kernels would potentially be similar enough to the stock Linux kernel to be emulated by QEMU, circumventing the need for both the evasion and LiLi frameworks. However, in practice, it is unclear if this is feasible. For instance, in our own testing, none of these kernels could successfully be compiled against the default configuration (`defconfig`). Even if they could, there is no guarantee that we will be able to then enable the drivers that we want, and that this kernel can be loaded by QEMU. This clearly motivates the use of our framework on these kernels.

7.3.2 Determining candidate drivers

As for the types of drivers that we looked for, we identified several main requirements.

Table 3: Studied Linux 4.9 kernels

Vendor	**Kernel Name**
Samsung	Galaxy S9
	Galaxy Note 9
Huawei	P20 Pro
	Mate 10 Pro
HTC	Exodus
	U12+
Lineage	Fairphone_sdm632
	Xiaomi_msm8937
	Bq_msm8953
	Amlogic

1. It must be either an i2c or platform driver. For this, we check if the driver calls `i2c_add_driver` or `platform_driver_register`. This is because they are the only types of drivers supported by the evasion framework.

2. It must have an IOCTL handler, because we wish to fuzz the IOCTL syscalls in particular.

3. It must not be present in the vanilla Linux kernel. This is because, when making their own kernel, many Android developers wil keep unneeded drivers from the Linux kernel. By eliminating these drivers, we can better focus on drivers that were intended to be used in that kernel.

4. All of its compatible properties must correspond to device tree nodes present in its source kernel's `dtb` files. A compatible property is specified in the source file with the "`compatible =` " expression, and names some peripheral used by the driver. With this check, we can be even more certain that the driver was developed for the specific kernel being reviewed.

5. It must have at least one compatible property. This is to ensure that the drivers that we work with interact with some type of peripheral, which helps remove trivial drivers from consideration.

Using a custom script, we scanned our Android kernels for C files with all the above properties. Following this, we manually inspected the source files to confirm that they did follow the above properties. In some cases, they violated requirement 4 (e.g., because the source file needed to be linked with another file that referred to a device absent from any `dtb` files), but we tested them anyways, as the evasion framework does offer a method for `dtb` entry simulation (though it does not work in every case). In Appendix B, we provide a description of all of the drivers that we selected using these criteria. This description includes the kernel they were found in, as well as where they can be found in that kernel. We further describe special loading instructions for these drivers in Appendix C.

7.3.3 Fixing kernels

Before we can try inserting our candidate drivers into the evasion kernel, we need to do a bit of work on our kernels. Recall that in order to function correctly, we need to align certain key `struct` offsets between the source Android kernels and the evasion kernels. This will entail adding or removing configuration options, as well as modifying header files, which the evasion framework provides clear instructions for. Those aside, there are also two universal changes to evasion kernels that are required due to some details specific to LiLi, which we discuss here. Since we are working with Linux 4.9 kernels, the following might differ slightly for older and newer kernels.

Making the text section writable

The module loading process is, for the most part, handled by the `load_module` function in `kernel/module.c`. Among other things, this takes the module's sections and puts them into one of four regions in the kernel: text, read-only data, read-only after initialization, and writable data. These regions of memory are each marked by different flags, most importantly to determine if they are executable, writable, or neither. In order to set these flags, `module.c` defines functions such as `module_enable_ro` and `module_disable_ro`, which either makes the regions that should be read-only (e.g., text) non-writable, or makes them writable, respectively. Since we have put all of `vmlinux`'s sections into the module's `.text` section, we will need the kernel's text region to be both executable (for the code) and writable (for the data). The text will be executable by default, but in order to also ensure that it is writable, we must

change calls to `module_enable_ro` into `module_disable_ro` instead. In the case of Linux 4.9, there are two such calls, both in `module.c`.

Enabling adrp instructions

Most Linux kernels come with `CONFIG_ARM64_ERRATUM_843419` enabled. Among other things, this disables the use of `adrp` instructions, because they can lead to faulty memory accesses when used with Cortex-A53 central processing units (CPUs). However, both `vmlinux` and `vmlinux.o` make constant use of `adrp` instructions. Moreover, because we do not work with Cortex-A53 CPUs (the processor can be specified by QEMU), we can safely disable this configuration option, so that we can keep the compiled code as it is.

7.3.4 Testing candidate drivers for fuzzability

We then tested most of these drivers to see if they could be inserted into an evasion kernel, and if the `/dev` file it creates could be opened and closed with any issues. Furthermore, we tested to see if there is a reachable IOCTL handler from any device nodes created by the module. In Table 4, we report these observations, where the "Status" of the driver is set to "✓" if the module produces a device file with a reachable IOCTL handler. Several modules that can be found in multiple kernels appear in this table; we list a module as having reachable IOCTLs if there is at least one of these equivalent modules that meets our criterion. As we see, out of 57 considered modules, 23 have reachable IOCTL syscalls. With respect to the drivers that did not initialize as expected, the main issues were typically caused by the redirection of important functions to function stubs, as well as errors in copied device tree entries (e.g., `phandles` did not work well, which is expected given that they work like pointers).

7.4 Fuzzing experiments

Now that we have a list of drivers to fuzz, we can go ahead and fuzz them.

Table 4: Driver fuzzability test: does each driver have reachable IOCTLs?

Driver	Purpose	Status	Driver	Purpose	Status
maxim	Audio	✓	tfa98xx	Audio	X
anc_hs	Audio	✓	anc_hs_default	Audio	✓
hicam_buf	Camera	✓	hwcam_cfgdev	Camera	X
laser_module	Camera	X	hismart_ar	Contexthub	X
msm	Camera	X	msm_rng	Randomization	X
sde_rotator_dev	Rotator	X	msm_csiphy	Camera	✓
msm_vidc_4l2	Video	X	msm_glink_pkt	Packet Manager	X
qseecom	Security	X	qcedev	Cryptography	X
sensors_ssc	Sensor	✓	msm_actuator	Camera	X
radio-iris	Radio	X	msm_ispif	Camera	✓
msm_ispif_32	Camera	✓	msm_isp	Camera	X
msm_sensor_driver	Camera	X	msm_flash	Camera	✓
msm_csid	Camera	✓	msm_eeprom	Camera	X
msm_cpp	Camera	X	mdss_rotator	Rotator	X
cam_cci_dev	Camera	X	hbtp_input	Input	✓
msm_ir_led	Camera	✓	msm_ir_cut	Camera	X
msm_qca402	Wi-Fi	X	cam_eeprom_dev	Camera	X
cam_flash_dev	Camera	X	cam_actuator_dev	Camera	X
nq-nci	NFC	X	pn547	NFC	X
dolby_fw	Audio	✓	amaudio2	Audio	✓
smartcard	Smart card	X	audio_info	Audio	✓
aml_aucpu	CPU	X	efuse64	Audio	✓
vm	Camera	X	amlvideo2	Video	X
picdec	Video	X	ionvideo	Video	✓
video_composer	Video	✓	videotunnel	Video	X
meson_ion_delay_alloc	Memory	✓	meson_uvm_allocator	Memory	✓
vout2_serve	Video	✓	vout_serve	Video	✓
cvbs_out	Video	✓	wifi_dt	Video	X
msm_smd_pkt	Packet Manager	X			

7.4.1 The fuzzer

General notes

The fuzzing software that we opted to use is syzkaller, primarily for both its efficiency and user-friendliness. As for the "mode" used, recall that because we are fuzzing in a kernel emulated by QEMU, we can choose between either isolated host mode and QEMU mode.

While the natural option was to gravitate towards the latter, we encountered some difficulties applying this mode to our particular use case, which we discuss later. We therefore started by trying isolated host mode. While we were able to get

it working, we did find its slowness to be too significant of a drawback. We therefore tried looking once more into QEMU mode, which we ultimately were able to use with some modifications to the source code. Thanks primarily to the parallelization of instances that QEMU mode allows, we were able to achieve a fuzzing speed orders of magnitude greater than we were previously getting.

Finally, we recall that in syzkaller, information about the driver's interface (e.g., what sorts of `arg` values are expected by the driver for an IOCTL) is provided by the user. This information can be partially or entirely recovered automatically with tools like DIFUZE [13] or syzkaller's `headerparser` tool. However, there does not currently exist a tool that will always fully recover all information about the interface. For instance, DIFUZE will not necessarily infer that an `int32` structure field might only ever be expected to be one of a few values in the code. As a result, it might take syzkaller a lot more time to find execution paths hidden behind checks for expected inputs. Thus, once more in the spirit of optimizing fuzzing efficiency, we opted to write these specifications manually, by simple code inspection.

Adapting syzkaller to our use case

Because we will be fuzzing in an evasion kernel, which attempts to satisfy certain hardware dependencies needed by drivers compiled against Android kernels, we need to load QEMU with an external device tree (i.e., in a `dtb` file) that satisfies them. Otherwise, the evasion kernel would just use its own innate default device tree, which will most likely not contain the nodes that these Android drivers would expect to have. However, while syzkaller does provide a way to specify some QEMU booting parameters via its configuration files, many such parameters are not covered by this interface (including this `dtb` file).

The main file that manages the launching of QEMU instances in syzkaller is `qemu.go`. Here, in the `boot` function, we can hardcode whatever QEMU booting parameters we wish. In our case, we needed to specify the path to the external device tree, as well as the type of the emulated machine (via the "`-M`" flag).

Finally, in order to report the cause of a crash, syzkaller uses the `dmesg` utility with the "`-w`" flag. This command will read the kernel logs, and due to the flag, will wait for any new messages. However, in some kernels (including the older vanilla kernels we work with), this flag is not available. Thus, in `console.go`, we need to

change this command to something equivalent to "`while true;do dmesg -c;sleep 1;done`", which achieves the same.

7.4.2 Computer specifications

All our tests were performed in AWS EC2 [5] instances running Ubuntu 20.04 with the arm64 architecture. We specifically chose the `c7g.metal` instance type, which has 64 threads and 128 GiB of RAM. The main reason we decided to use AWS EC2 instances is for the possibility to take advantage of virtualization. When emulating a kernel that uses a different architecture than the one supported by the host CPU, there needs to be a translation that occurs during communication between the host CPU and the emulated guest CPU. The alternative is virtualization, which is only possible when the host and guest use the same architecture. Here, CPU instructions don't need to be translated, which speeds up execution significantly. Because the kernels we selected use arm64 and we did not have access to arm64 machines, we opted to use AWS EC2 instances. Note that we specifically use a bare-metal instance, because we would otherwise be operating in a virtual machine, and AWS EC2 does not support nested virtualization.

7.4.3 Fuzzing results

In Table 5, we present the results of these fuzzing experiments. In particular, we discuss the number of randomized programs executed by syzkaller per second (execs/s), the number of these programs that produce a crash per minute (crashes/m), as well as the coverage (total number of basic blocks of kernel code reached throughout all executions).

There are three main characteristics that we can observe that support the validity of LiLi.

- Crashes: Firstly, the number of crashes for every program matches our expectations. In particular, the number of crashes is generally quite low, except for a few cases where there is some significant error encountered on most execution paths that can be identified in the code. This suggests that our extraction and injection process, generally speaking, likely did not create issues not already present in the code.

Table 5: Fuzzing results

Driver	Kernel	LOC	Fuzzing Statistics			
			Uptime	Execs/s	Crashes/m	Coverage
maxim	Huawei P20 Pro	879	3h6m	198	0.15	1182
anc_hs	Huawei P20 Pro	1080	4h35m	8149	0	802
anc_hs_default	Huawei P20 Pro	156	4h8m	164	65	1426
hicam_buf	Huawei Mate 10 Pro	630	3h2m	421	0	1158
dolby_fw	Lineage Amlogic	540	3h58m	10	70	1124
audio_data	Lineage Amlogic	224	3h11m	991	6.8	814
efuse64	Lineage Amlogic	879	3h4m	556	0	1205
amaudio2	Lineage Amlogic	1465	3h5m	0.42	20	1261
ionvideo	Lineage Amlogic	1566	3h2m	228	4.3	2453
video_composer	Lineage Amlogic	2544	3h7m	4845	1.1	1222
meson_ion_delay_alloc	Lineage Amlogic	500	3h26m	2145	13	1425
meson_uvm_allocator	Lineage Amlogic	395	3h4m	6793	0	1127
vout2_mod	Lineage Amlogic	1721	3h4m	6075	0	1148
vout_mod	Lineage Amlogic	1744	3h1m	6234	0	1144
cvbs_out	Lineage Amlogic	1881	3h6m	325	0	1272
sensors_ssc	Lineage Xiaomi	356	3h13m	191	65	1417
msm_ispif	Lineage Xiaomi	1832	3h2m	588	21	593
msm_ispif_32	Lineage Xiaomi	1326	3h1m	588	20	859
msm_csiphy	Lineage Xiaomi	2355	4h34m	514	20	598
msm_csid	Lineage Xiaomi	1153	3h3m	549	19	594
msm_flash	Lineage Xiaomi	1207	4h11m	609	22	593
hbtp_input	Lineage BQ	1387	3h20m	3415	0	1809
msm_ir_led	Lineage BQ	360	3h20m	645	37	861

- Coverage: Showing that the drivers don't crash excessively does not alone imply validity, however, as there could theoretically be an issue where an error in the extraction process simply prevents entire parts of the program from executing, thereby rendering it trivial. For this, we have our second metric: coverage. We see that all the tested programs achieved good coverage (they at a minimum reached several hundreds of code blocks in the kernel). This means that, given random inputs, the driver was reaching a variety of different blocks of code in the kernel, suggesting that the extraction process was producing modules that performed all sorts of tasks (i.e., not trivial).

- Vulnerabilities: We also manually reviewed the crashes to find any true positives. In particular, we looked at crashes for which syzkaller provided detailed information, such as the type of crash, the function where it happens, and an example program that triggers the crash. Some were ultimately false positives, but we did also find four vulnerabilities, all of which are previously unreported[1]. This suggests not only that modules emulated via LiLi can be used effectively in fuzzing, but that their correctness is robust enough to allow fuzzers to find bugs that have never been reported.

In summary, the first two observations together suggest that the emulated modules do things, but not things that lead to crashes that are not supposed to happen. This could mean either that they are functioning as intended, or that they are doing something else entirely, but doing it without crashing (which is theoretically possible, but terribly unlikely for obvious reasons). The former possibility is then further reinforced by our third observation, which itself suggests correctness. Putting everything together, we can be confident that the module, having been transformed from being built-in to loadable, is in fact operating as intended.

7.4.4 New vulnerabilities

We ultimately found four previously undiscovered bugs, all of which are located in the Lineage Amlogic kernel. Three of these were found directly through the fuzzing

[1]Indeed, the only vulnerabilities found were zero-day (none were rediscovered). Every other crash was either not investigated (i.e., due to imprecise information from syzkaller) or corresponded to a false positive. The former was the case for the majority of crashes. As for the false positives, they were mostly due to the redirection of some non-trivial functions to stubs.

experiments, and one was found upon manual code inspection of a driver that contained one of the other three bugs (i.e., it did not correspond to a bug found by syzkaller). Indeed, knowing that programs that contain one vulnerability may well contain others, we performed a manual review of the three vulnerable drivers, and in at least one case, this suspicion was confirmed.

Given that the Lineage Amlogic kernel is based on Android TV kernels, we have reported these bugs to Google's Android Security Team, who has confirmed all of them. In Table 6, we provide further details about these vulnerabilities, including their type and Google's severity assignment.

Table 6: 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

* Discovered via additional code review

meson_ion_delay_alloc double free

The `meson_ion_delay_alloc.ko` module defines the `UVM_IOC_ALLOC` IOCTL. 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 inside 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 the `UVM_IOC_ALLOC` IOCTL. This will register a `dma_buf` over and over, until the maximum number of given `fd`'s has been reached.

meson_ion_delay_alloc memory leak

Recall that the `meson_ion_delay_alloc` driver's `UVM_IOC_ALLOC` IOCTL allocates a `uvm_buffer` based on data passed through the IOCTL's `arg` parameter. If another driver knows this buffer's file descriptor (e.g., a user-space program is interacting with both drivers), then it can call `dma_buf_map_attachment` on it, which will eventually call `meson_uvm_map_dma_buf`, in turn calling `meson_uvm_alloc_buffer`. This last function uses `ion_alloc`, which allocates a buffer inside memory pools belonging to the ION subsystem, based on data from the aforementioned `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. Among other things, there needs to be a custom driver on top of the `meson` driver, as well as a user-space program that interacts with these two drivers in a very specific way.

ionvideo arbitrary write

The `ionvideo.ko` module defines the `vidioc_qbuf` IOCTL, which takes as user input a pointer `p` to a `v4l2_buffer`. Then, the `vidioc_qbuf` function writes to an array at index `p->index`, but without first having validated that `p->index` is a reasonable value. 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.

amaudio2 null ptr deref

The `amaudio2.ko` module 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.

Chapter 8

Conclusion and Future Work

In this thesis, we proposed a set of techniques which, put together, provides a way to execute a portion of an Android kernel in binary format within an alien environment. In particular, we focused on the execution of Android device drivers. To accomplish this, we started by gathering a list of candidate drivers to work with, and then ran experiments to verify that these injected drivers can be loaded and fuzzed. Following an analysis of the results of these experiments, it is clear that our approach operates correctly, and that it can be useful for fuzzing. In our case, we discovered 4 previously unknown vulnerabilities.

This approach can be used by developers and security researchers to easily and efficiently fuzz parts of kernels that would otherwise be difficult to dynamically analyze. We believe that using LiLi, which can be entirely automated into a single tool, can therefore be effective in reducing costs, as well as making the dynamic analysis of the Android kernel more accessible.

In terms of future research directions, perhaps the most interesting would be to determine which functions exist only in the Android kernel of interest and not in the corresponding evasion kernel, to then add relocations that refer internally to these functions. Modifying the implementation in order to do this is somewhat trivial, but an analysis of the consequences of this change could be highly valuable. If this change works well, it could help reduce the amount of times our modules call functions that do not actually exist in the evasion kernel, thereby potentially making these modules behave in way that is truer to their intended functioning.

Of course, anything that moves towards the generalization of this approach makes

for a highly practical avenue to explore as well. This could include, for instance, looking into whether or not LiLi could work with more recent versions of Android, or if it could work with non-Android systems (or perhaps even non-UNIX ones). Another interesting direction could be to see if this technique could be adapted to produce modules that can be inserted into something other than evasion kernels. Finally, while we only use this framework to emulate drivers in this thesis, there is nothing in principle that prevents its use on other parts of the kernel (e.g., subsystems). The question is just whether or not the software and hardware dependencies can be satisfied. In either case, approaches similar to what the evasion kernel proposes could potentially work (e.g., redirecting missing function and variable references to stubs, and copying entries to the device tree in the case of missing hardware). We believe that these could all be highly impactful ways to broaden the scope of the applications of this new framework.

Bibliography

[1] afl. URL: `https://github.com/mirrorer/afl`.

[2] iknowthis. URL: `https://github.com/rgbkrk/iknowthis`.

[3] Triforcelinuxsyscallfuzzer. URL: `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer`.

[4] Trinity. URL: `https://github.com/kernelslacker/trinity`.

[5] Amazon. Amazon EC2. Accessed: 2023-11-09. URL: `https://aws.amazon.com/ec2/`.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 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.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, 2001.

[12] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[14] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[15] Google. Syzkaller, 2019. URL: `https://github.com/google/syzkaller`.

[16] Jesse Hertz and Tim Newsham. Triforceafl. URL: `https://research.nccgroup.com/wp-content/uploads/2022/09/NCC-Group-whitepaper-TriforceAFL-2017.pdf`.

[17] Josh Howarth. Iphone vs android user stats (2023 data), Oct 2023. visited on 2023-11-22. URL: `https://explodingtopics.com/blog/iphone-android-users`.

[18] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.

[19] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[20] Karl Koscher, Tadayoshi Kohno, and David Molnar. {SURROGATES}: Enabling {Near-Real-Time} dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[21] Serge Lamikhov-Center. ELFIO. URL: `https://github.com/serge1/ELFIO`.

[22] Tin Le. tsys, 1991. URL: `https://groups.google.com/g/alt.sources/c/V_B37EtnWKQ/m/NztsljVYV84J?hl=en`.

[23] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.

[24] Marin. vmlinux-to-elf. URL: `https://github.com/marin-m/vmlinux-to-elf`.

[25] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 305–318, 2011.

[26] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. POTUS: Probing Off-The-Shelf USB drivers with symbolic fault injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association. URL: `https://www.usenix.org/conference/woot17/workshop-program/presentation/patrick-evans`.

[27] Hui Peng and Mathias Payer. {USBFuzz}: A framework for fuzzing {USB} drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575, 2020.

[28] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105. IEEE, 2020.

[29] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 5(2):3–8, 2014.

[30] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. {SymDrive}: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, 2012.

[31] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*, pages 167–182, 2017.

[32] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat Europe*, 2014.

[33] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[34] Chad Spensky, Jeffrey Stewart, Arkady Yerukhimovich, Richard Shay, Ari Trachtenberg, Rick Housley, and Robert K Cunningham. Sok: Privacy on mobile devices-it's complicated. *Proc. Priv. Enhancing Technol.*, 2016(3):96–116, 2016.

[35] Statcounter Global Stats. visited on 2023-11-02. URL: `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

[36] Jeff Vander Stoep. Android: protecting the kernel, 2016. visited on 2023-11-17. URL: `https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf`.

[37] Henrik Stuart. Hunting bugs with coccinelle. *Master's Thesis*, 2008.

[38] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 291–307, 2018.

[39] Ilja van Sprundel. sysfuzz. URL: `https://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf`.

[40] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 14, pages 1–16, 2014.

Appendices

Appendix A

Compiling the Studied Kernels

In Table 7, we detail information needed to download the source code of every kernel we studied, as well as some basic compilation instructions (e.g., the `defconfig` that we used).

Table 7: How to obtain and compile the kernels

Kernel	**Notes**
Samsung Galaxy [S9 / Note 9]	1. Obtained from `github.com/djb77/samsung-kernel` (at branch [`G96XX-TW9` / `N96XX-TW9`]) 2. Defconfig: [`exynos9810-starlte_defconfig` / `exynos9810-crownlte_defconfig`]
Huawei [P20 Pro / Mate 10 Pro]	1. Obtained from `consumer.huawei.com/en/opensource/` (model [`CLT_PIE_EMUI9.0` / `ALP_PIE_EMUI9.0`]) 2. Defconfig: `merge_kirin970_defconfig`

HTC [Exodus / U12+]	1. Obtained from `htcdev.com/devcenter/downloads` (description [`2.45.2401.3` / `1.68.401.6`]) 2. Defconfig: `sdm845_defconfig`
Lineage Fairphone	1. Obtained from `github.com/LineageOS/android_kernel_fairphone_sdm632` (at branch `lineage-19.1`) 2. Defconfig: `sdm845_defconfig` 3. Note: May need to edit `gcc-compiler.h` so its check for `GCC_VERSION` is relaxed (e.g., change "`GCC_VERSION < 50100`" to `GCC_VERSION < 40600`)
Lineage Xiaomi	1. Obtained from `github.com/LineageOS/android_kernel_xiaomi_msm8937` 2. Defconfig: `lineageos_mi8937_defconfig` 3. Note: May need to edit `gcc-compiler.h` so its check for `GCC_VERSION` is relaxed (e.g., change "`GCC_VERSION < 50100`" to `GCC_VERSION < 40600`)

Lineage BQ	1. Obtained from `github.com/LineageOS/android_kernel_bq_msm8953` 2. Defconfig: `msm8953_defconfig` 3. Note: May need to edit `gcc-compiler.h` so its check for `GCC_VERSION` is relaxed (e.g., change "`GCC_VERSION < 50100`" to `GCC_VERSION < 40600`) 4. Note2: Add "-w" to `ARCH_CPPFLAGS` in the root Makefile. This disables warnings that can cause that copmilation to stop.
Lineage Amlogic	1. Obtained from `github.com/LineageOS/android_kernel_amlogic_linux-4.9` (at branch `lineage-19.1`) 2. Defconfig: `g12a_defconfig`

Appendix B

Driver Information

In Table 8, we list all the drivers that we chose to study, along with their location in their original kernels, with respect to the root folder. We note that while some drivers appear in several kernels (e.g., `msm.c`), we typically only tested drivers in one kernel, so as to avoid redundancy. Hence, every driver in this table is listed as appearing in exactly one kernel.

Table 8: Driver information

Driver	**Directory**
Huawei P20 Pro	
maxim	huawei_platform/audio/maxim
tfa98xx	huawei_platform/audio/tfa98xx
anc_hs	huawei_platform/audio/anc_hs_module/anc_hs
anc_hs_default	huawei_platform/audio/anc_hs_module/anc_hs_default
Huawei Mate 10 Pro	
hicam_buf	media/huawei/camera/hicam_buf
hwcam_cfgdev	media/huawei/camera
laser_module	media/huawei/camera/laser/vl53Lx
hismart_ar	contexthub/flp
HTC Exodus	
sde_rotator_dev	media/platform/msm/sde/rotator
msm_vidc_v4l2	media/platform/msm/vidc_3x

msm_glink_pkt	soc/qcom
qcedev	crypto/msm
Lineage Xiaomi	
sensors_ssc	sensors
msm	media/platform/msm/camera_v2
radio-iris	media/radio
msm_ispif	media/platform/msm/camera_v2/ispif
msm_ispif_32	media/platform/msm/camera_v2/ispif
msm_isp	media/platform/msm/camera_v2/isp
msm_sensor_driver	media/platform/msm/camera_v2/sensor
msm_flash	media/platform/msm/camera_v2/sensor/flash
msm_csiphy	media/platform/msm/camera_v2/sensor/csiphy
msm_csid	media/platform/msm/camera_v2/sensor/csid
msm_actuator	media/platform/msm/camera_v2/sensor/actuat or
msm_eeprom	media/platform/msm/camera_v2/sensor/eeprom
msm_cpp	media/platform/msm/camera_v2/pproc/cpp
qseecom	misc
msm_smd_pkt	char
msm_rng	char/hw_random
mdss_rotator	video/fbdev/msm
cam_cci_dev	media/platform/msm/camera/cam_sensor_modul e/cam_cci
Lineage BQ	
hbtp_input	input/misc
msm_ir_led	media/platform/msm/camera_v2/sensor/ir_led
msm_ir_cut	media/platform/msm/camera_v2/sensor/ir_cut
msm_qca402	media/platform/msm/qca402
cam_eeprom_dev	media/platform/msm/camera/cam_sensor_modul e/cam_eeprom
cam_flash_dev	media/platform/msm/camera/cam_sensor_modul e/cam_flash

cam_actuator_dev	media/platform/msm/camera/cam_sensor_module/cam_actuator
nq-nci	nfc
pn547	nfc
Lineage Amlogic	
dolby_fw	amlogic/dolby_fw
amaudio2	amlogic/amaudio2
smartcard	amlogic/smartcard_sc2
audio_data	amlogic/audioinfo
efuse64	amlogic/efuse
vm	amlogic/media/camera/common
amlvideo2	amlogic/media/video_processor/video_dev
picdec	amlogic/media/video_processor/pic_dev
ionvideo	amlogic/media/video_processor/ionvideo
video_composer	amlogic/media/video_processor/video_composer
videotunnel	amlogic/media/video_processor/videotunnel
meson_ion_delay_alloc	amlogic/media/common/uvm
meson_uvm_allocator	amlogic/media/common/uvm
vout2_serve	amlogic/media/vout/vout_serve
vout_serve	amlogic/media/vout/vout_serve
cvbs_out	amlogic/media/vout/cvbs
wifi_dt	amlogic/wifi
aml_aucpu	amlogic/dvb/aucpu

Appendix C

Driver Loading Instructions

In Table 9, we detail information needed to correctly load the drivers that we fuzzed, beyond the typical steps that are universal to all drivers.

Table 9: Special driver loading instructions

Driver	**Notes**
maxim	• Source file(s): `maxim.c` • Load the `max98925` driver first. • Make sure the emulated kernel has an i2c bus.
anc_hs	• Source file(s): `anc_hs.c` • Define `CONFIG_GPIO_HI6402` at the beginning of the probe function. • Load the `anc_hs_interface` driver first.
anc_hs_default	• Source file(s): `anc_hs_default.c` • Load the `anc_hs_interface` driver first.

hicam_buf	• Source file(s): `hicam_buf.c` • Load the `anc_hs_interface` driver first.
dolby_fw	• Source file(s): `dolby_fw.c`
audio_data	• Source file(s): `audio_data.c`
efuse64	• Source file(s): `efuse64.c`, `efuse_hw64.c`
amaudio2	• Source file(s): `amaudio2.c`
ionvideo	• Source file(s): `ionvideo.c`, `ppmgr2.c`
video_composer	• Source file(s): `video_composer.c`

meson_ion_delay_alloc	• Source file(s): `meson_ion_delay_alloc.c` • The `__init` function only calls `platform_driver_register` if `use_uvm` is false. Since this resolves to a stub, we can just remove this if statement to ensure that `platform_driver_register` is called. • Not necessary but recommended: porting `ion` subsystem from the Lineage Amlogic kernel.
meson_uvm_allocator	• Source file(s): `meson_uvm_allocator.c` • Not necessary but recommended: porting `ion` subsystem from the Lineage Amlogic kernel.
vout2_mod	• Source file(s): `vout2_serve.c`, `vout2_notify.c`, `vout_func.c`
vout_mod	• Source file(s): `vout_serve.c`, `vout_notify.c`, `vout_func.c`, `vout_reg.c`
cvbs_out	• Source file(s): `cvbs_out.c`, `cvbs_out_reg.c`, `enc_clk_config.c`
sensors_ssc	• Source file(s): `sensors_ssc.c`

msm_ispif	• Source file(s): `msm_ispif.c` • Load the `msm` driver first • Manually edit the probe function so that `msm_ispif_get_clk_info` returns 0 (or else it will not complete).
msm_csiphy	• Source file(s): `msm_csiphy.c` • Load the `msm` driver first
msm_csid	• Source file(s): `msm_csid.c` • Load the `msm` driver first
msm_flash	• Source file(s): `msm_flash.c` • Load the `msm` driver first
hbtp_input	• Source file(s): `hbtp_input.c` • Load the `msm` driver first
msm_ir_led	• Source file(s): `msm_ir_led.c` • Load the `msm` driver first