SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers

Yu Hao*, Guoren Li*, Xiaochen Zou*, Weiteng Chen*, Shitong Zhu*, Zhiyun Qian* and Ardalan Amiri Sani†
*University of California, Riverside, †University of California, Irvine
*{yhao016, gli076, xzou017, wchen130, szhu014}@ucr.edu, zhiyunq@cs.ucr.edu, †ardalan@uci.edu

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179298

***Abstract*—Fuzz testing operating system kernels has been effective overall in recent years. For example, syzkaller manages to find thousands of bugs in the Linux kernel since 2017. One necessary component of syzkaller is a collection of syscall descriptions that are often provided by human experts. However, to our knowledge, current syscall descriptions are largely written manually, which is both time-consuming and error-prone. It is especially challenging considering that there are many kernel drivers (for new hardware devices and beyond) that are continuously being developed and evolving over time. In this paper, we present a principled solution for generating syscall descriptions for Linux kernel drivers. At its core, we summarize and model the key invariants or programming conventions, extracted from the "contract" between the core kernel and drivers. This allows us to understand programmatically how a kernel driver is initialized and how its associated interfaces are constructed. With this insight, we have developed a solution in a tool called SyzDescribe that has been tested for over hundreds of kernel drivers. We show that the syscall descriptions produced by SyzDescribe are competitive to manually-curated ones, and much better than prior work (*i.e.,* DIFUZE and KSG). Finally, we analyze the gap between our descriptions and the ground truth and point to future improvement opportunities.**

1. Introduction

Fuzzing has become one of the most popular methods for finding bugs and vulnerabilities due to its practicability and effectiveness. For example, in the context of large-scale and stateful programs such as operating system kernels, the state-of-the-art operating system kernel fuzzer, syzkaller [1], has found more than 4.6k bugs in the Linux kernel over the past few years, more than 3.6k of which have been fixed [2].

One necessary component of syzkaller is a collection of syscall descriptions that are often provided by human experts. As the primary interface between the user space and kernel space is syscall, the fuzzer needs to be aware of the syscalls that are available on an operating system kernel, values of interest for each syscall, and explicit dependencies between syscalls [3]. For syzkaller, such syscall descriptions can be provided using a declarative language called syzlang [4]. However, to our knowledge, current syscall descriptions are largely written manually [4], which is both time-consuming and error-prone. Indeed, in our study, we find that existing manually-curated syscall descriptions can miss dependencies and/or syscall interfaces, and even incorrect or out-of-date descriptions. It has also been shown that incomplete syscall descriptions is a major root cause limiting the code coverage achieved by syzkaller [3]. Moreover, syscall description generation is not a one-time effort. Descriptions need to be continuously updated as the kernel evolves (*e.g.,* there are on average 200 commits daily to Linux mainline [5]). Last but not least, not all kernel drivers have manually-curated syscall descriptions [6]. Compared to the core Linux kernel, kernel drivers account for about 71.9% SLoC of the kernel code and therefore a large fraction of the attack surface that needs to be tested as well [3].

There exist limited solutions to generate syscall descriptions. DIFUZE [6] attempts to generate syscall descriptions automatically through static analysis of the source code of kernel drivers. Unfortunately, without a principled modeling of the programming convention in kernel drivers, the generated descriptions are significantly inaccurate. KSG [7] achieves the same goal partially through a dynamic analysis to recover drivers and associated interfaces. However, it has limited coverage of drivers which have to be loaded already on a live system, and the requirement of setting up such a live system with a recompiled and instrumented kernel can be a high bar for adoption. Unfortunately, there exists no comprehensive evaluation of the quality of these automatically generated descriptions due to lack of comparison and ground truth. As such, we believe it remains an important and unsolved problem to accurately and automatically generate syscall descriptions.

In this paper, we present a principled solution to tackle this problem, focusing on Linux kernel drivers. At its core, we summarize and model the key invariants and programming conventions regarding kernel driver development. This allows us to understand programmatically how a kernel driver is initialized and how its associated interfaces are constructed. At a high level, we can then statically reconstruct the initialization of a kernel driver by faithfully following the summarized/modeled initialization process. As will be shown in our design, our modeling is chosen to balance accuracy and generality (*e.g.,* the convention should not be changing from version to version).

We have implemented our solution in a tool called SyzDescribe, which is able to statically, accurately and

DOI 10.1109/SP46215.2023.00133

automatically generate syscall descriptions for Linux kernel drivers. We evaluate SyzDescribe against static based tool DIFUZE, dynamic based tool KSG, manually-written syzkaller descriptions and even a ground truth dataset which we curate, demystifying the differences between these solutions. We find that SyzDescribe has a much better accuracy and coverage than DIFUZE and KSG, generating many more descriptions compared to manually-written syzkaller descriptions, and is closest to the ground truth. More specifically, we find that the syzkaller descriptions cover only less than half of the kernel drivers that SyzDescribe generates. Interestingly, even when a driver is covered by syzkaller already, we still find many "bugs" in them due to human mistakes or lack of ongoing maintenance (kernel code and descriptions become out-of-sync). Furthermore, we evaluate the effectiveness of SyzDescribe with a number of fuzzing experiments, and show that SyzDescribe also performs the best among all existing solutions. Moreover, by applying the solution to the kernel drivers of the Pixel 6 Android smartphone, where no existing syscall descriptions are available, we are able to find 18 unique crashes.

We summarize our main contributions as below:

- We present a principled solution that can automatically and statically generate syscall descriptions for Linux kernel drivers, based on the modeling of the well-established contract between the core kernel and drivers.
- We evaluate SyzDescribe comprehensively and show that SyzDescribe is capable of generating descriptions that are highly accurate, better than manually-curated syzkaller descriptions and closest to the ground truth.
- We investigate and summarize the root causes for the gap between the results of SyzDescribe and the ground truth. We then point out future directions for continuing to improve automated syscall description generation.
- We open source the implementation of SyzDescribe [8] and the generated syscall descriptions [9] to facilitate the reproduction of results and future research.

2. Background and Motivation

In this section, we first briefly introduce Linux kernel drivers, then explain how syzkaller currently describes them, and finally summarize existing state-of-the-art solutions for automatically constructing such descriptions.

2.1. Linux Kernel Drivers and Descriptions

Kernel Drivers. In Linux, drivers are in kernel modules that can be loaded during either boot time or later on demand [10]. Each kernel module has well-marked entry and exit points usually defined by macros such as `module_init(x)` and `module_exit(x)`. A single driver is usually defined within a single kernel module. However, it is also possible that a driver is defined across multiple kernel modules that have dependencies between each other, *e.g.,* two kernel modules defined by `subsys_initcall(alsa_sound_init)` and `module_init(alsa_seq_init)` collectively constitute a sound sequencer driver. Conversely, a single kernel module can also define multiple kernel drivers, *e.g.,* loop-control driver and loop driver are defined in `module_init(loop_init)`.

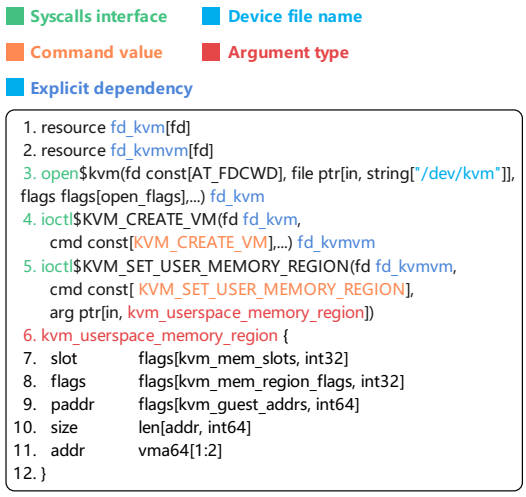

Figure 1: Example syscall descriptions of KVM in syzkaller

In summary, the initialization of a kernel driver takes place at one or more entry points, during which the interfaces exposed to the user space will be defined — first, one or more device file names will be defined and exposed under the `/dev` directory; second, a number of syscall handlers will be defined and registered, *e.g.,* `open()` and `ioctl()`. Together, this initialization allows for a user-space application to interact with the driver (we provide a concrete example in §3.1).

Conceptually, every kernel driver is associated with a certain type of device, as its goal is to "drive" the device. There are three first-level types of devices in Linux kernel: character device, block device and network interface [11]. In our work, we consider character devices and block devices because they can be accessed from user space by device files in the `/dev` directory. There are also a limited number of sub-types within each first-level device type, which we will describe later on.

A device is uniquely identified by the *device number*, which is the combination of a major number and a minor number. The device number not only uniquely identifies a device and its file name(s), but is also associated with a set of syscall handlers, *e.g.,* `ioctl()`. Even though these details are often hidden from a user-space application that is interested in interacting with a kernel driver, they are in fact critical for generating syscall descriptions to exercise the kernel driver. We will present an example kernel driver with all these low-level details in §3.1.

Syscall Descriptions. We now illustrate an example syscall description for the KVM kernel driver shown in Figure 1, which includes the following necessary components:

- **Syscall interface**, which allows user-space applications to interact with the driver. For example, we see `open()` and `ioctl()`, which are the most common. There can

also be `read()`, `write()`, and others.

- **Device file name**, which is used as an argument in syscall `open()`, *e.g.,* `"/dev/kvm"`.
- **Command value**, which decides sub-interface of `ioctl()` and denotes the value of the second argument of `ioctl()` — see the constant string that succeeds the `"$"`, *e.g.,* `KVM_SET_USER _MEMORY_REGION` in line 5. This explicit separation of sub-interfaces in `ioctl()` is due to the fact that each command value often leads to an independent set of functionalities. Treating them as separate syscall interfaces allows syzkaller to generate test cases more effectively.
- **Argument type**, which are other arguments to syscall interfaces, *e.g.,* the third argument of `ioctl()`, *e.g.,* `kvm_userspace _memory_region`) declared in line 5 and defined from line 6 to line 12.
- **Explicit dependency** [3] between those syscalls, mainly involving file descriptors being returned from one syscall interface (*e.g.,* return value of `open()`) and used in another (*e.g.,* first argument of `ioctl()`). Explicit dependency descriptions allow syzkaller to generate valid test cases that honor the dependencies (*i.e.,* allowing the subsequent syscalls to proceed without early termination). Note that file descriptors can also be returned by syscall interfaces other than `open()` as well for those complex driver drivers. For example, from `ioctl$KVM_CREATE_VM` at line 4, we can see that it returns `fd_kvmvm`, which will be used as the first argument of `ioctl$KVM_SET_USER_MEMORY_REGION` at line 5. We refer to such dependencies specifically as **non-open file descriptor dependency**. Note that it is almost impossible to fuzz the syscall interface `ioctl$KVM_SET_USER_MEMORY_REGION` correctly without knowing this **non-open file descriptor dependency**.

2.2. Current Attempts at Generating Syscall Descriptions

Currently, the project repository of syzkaller contains a number of syscall descriptions for various kernel drivers. As we confirmed with the maintainers of syzkaller, such descriptions are manually curated and face a number of challenges. First, these descriptions may be incomplete or even incorrect, given that the kernel drivers can be complex and their logic can evolve over time. Second, maintaining such descriptions can be costly as it needs to be continuously updated. Third, it is not scalable as there is a constant stream of new kernel drivers being developed, *e.g.,* the various Android kernel drivers to support OEM-specific device drivers [12], [6].

There do exist limited solutions proposed in recent years to generate syscall descriptions automatically. However, none of them is able to generate high quality syscall descriptions that are competitive with the existing manually-written ones. Below, we briefly describe these solutions categorized by dynamic and static solutions and discuss their pros and cons.

Dynamic Solutions. CoLaFUZE [13] and KSG [7] are recent solutions mainly using dynamic analysis to identify kernel drivers and recover their interfaces. First, they scan all the device files available under `/dev`, and retrieve file descriptors of those device files by syscall `open()`. For syscall handler discovery, they look for specific syscall handler structures during the execution of syscall `open()`, where the structure would be explicitly referenced. This way, they can easily pair the device file name and the corresponding syscall handler structure. As we will describe later in §3.1, by convention, Linux kernel syscall handler structures are defined as structures that contain a number of function pointers, *e.g.,* `open`, `ioctl`. After syscall handler recovery, they both apply symbolic execution to recover the command values and argument types.

Static Solutions. DIFUZE [6] is the only existing solution based on static analysis. To recover syscall handlers, it first attempts to identify syscall handler structures from a pre-defined list of struct types, *e.g.,* `struct cdrom_device_ops` (it is unclear how the list is generated). Then it attempts to identify the corresponding device file name (any constant string) used near the reference of the structure and pair them up. After syscall handler recovery, it performs a static inter-procedural analysis on the `ioctl()` handler to find command values through all the equality constraints (*i.e.,* `switch cases` and `if` conditions), as well as to find the argument type by inspecting `copy_from_user()`, a common kernel function used to copy data from the user space to the kernel space.

Static vs. Dynamic Solutions Static and dynamic solutions can both achieve the same goal of recovering syscall handlers, as attempted by existing solutions. At a high level, we believe static and dynamic solutions are complementary. Dynamic solutions can directly observe what occurs on a live system with certainty. For instance, it can directly observe the syscall handler structure as it is already set up after the driver is loaded. On the other hand, dynamic solutions have limited coverage as it requires the driver to be loaded and initialized properly before it can be analyzed. Static solutions can cast a wider net, identifying drivers that are not necessarily loaded, but they may not be very precise compared to dynamic solutions.

In this paper, we choose to tackle the problem statically. There are a few high-level reasons. (1) Dynamic solutions alone will not discover all fuzzable drivers/modules, as a module may be loadable depending on various factors, *e.g.,* depending on another module by specifying the flag of MODULE_SOFTDEP, or requiring certain hardware. We believe it is helpful to point out the existence of drivers, so human experts can intervene, *e.g.,* setting them up properly for fuzzing. (2) Human experts have the need to write syscall descriptions without any runtime testing environment. For example, Android vendors may want to start the description writing process before real devices are available for fuzzing. Only static solutions can help generate descriptions that can aid human experts in such cases. (3) Dynamic solutions need additional engineering work, *e.g.,* hardware-based tracing,

recompiling the kernel with instrumentation [7] or installing a kernel module [13], which can be challenging. For instance, Android devices may have locked bootloaders that prevent a new kernel image from being flashed. Furthermore, recompiling the kernel and enabling specific features (*e.g.,* eBPF and kprobe) can have compatibility issues based on our own experience.

To substantiate the first reason, we evaluated KSG and show in §5.2.2 that it indeed incurs significant false negatives, *i.e.,* missing drivers, device files and syscall interfaces. Interestingly, we realize another reason that contributed to false negatives: dynamic solutions still need to model certain programming conventions, the lack of which results in direct failure in retrieving the syscall handlers.

Unfortunately, developing a good static solution is non-trivial. When evaluating DIFUZE (details in §5.2.1), we find that it incurs both significant false positives and false negatives in discovering syscall handlers and device file names. Fundamentally, there is a lack of proper modeling of the programming convention in kernel drivers. For instance, it does not have a principled method of discovering all syscall handler structures. Instead, it relies on a pre-defined list of struct types which can be incomplete and out-of-date as kernel drivers evolve over time. Another example is that it does not model how the device file names and syscall handler structures are associated. Instead, it relies on a simple but unreliable heuristic that treats strings near syscall handler references as device file names.

3. SyzDescribe Design

SyzDescribe aims at fully automating the generation of accurate syscall descriptions for kernel drivers, descriptions that can be directly loaded and used by syzkaller. In this section, we describe the design choices of SyzDescribe, and the details of the key components. Starting with a motivating example, we first elaborate on the intuition behind SyzDescribe and why it can overcome challenges faced in prior solutions conceptually. We then move to its overall workflow, and eventually dive into the concrete components that SyzDescribe comprises.

3.1. A Motivating Example

Figure 2 shows an example kernel driver implementation (simplified from real drivers), which helps us to elaborate on the programming conventions and invariants. As mentioned earlier, since all drivers are defined in kernel modules, there must be a module init function declared by the macro `module_init` (line 16). Its definition can be found in line 5. Next, we can see that the driver's device file is initialized in the module init function (lines 6-11), which involves the creation of two objects. According to the Linux kernel driver development convention, one object corresponds to the "driver" (`struct cdev`) containing the description of a set of syscall handlers (see line 12) and the other corresponds to the "device" (`struct device`) containing the description of the device file name (see line 13). Note

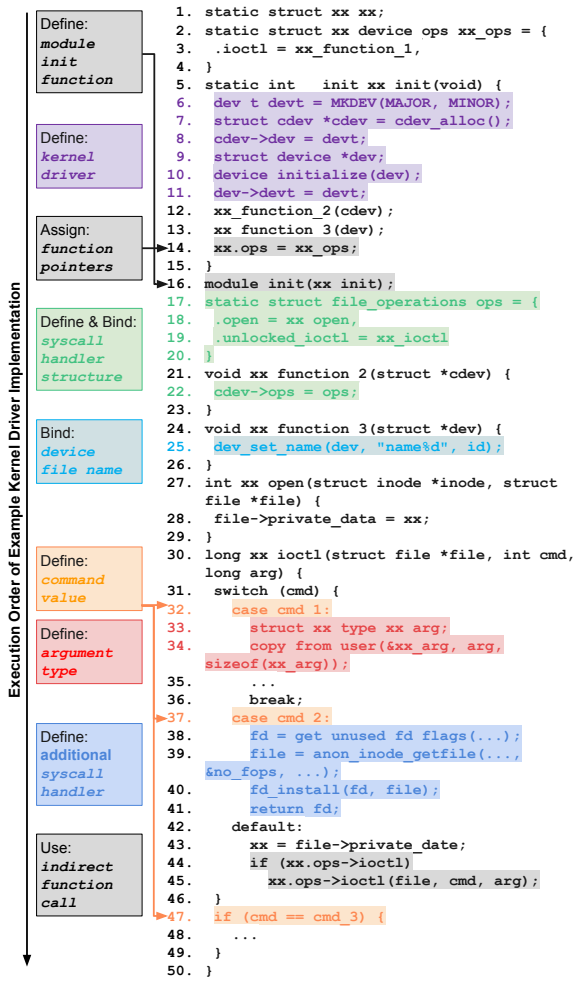

Figure 2: Simplified example kernel driver implementation

that a single set of syscall handlers can support multiple devices (*e.g.,* with different device file names), hence the two separate objects. Furthermore, each device must have a unique device number (line 6), which is associated with both the driver and device objects, allowing the two objects to be paired together to form a complete driver interface (*i.e.,* both syscall handlers and the device file name).

We can see the definition of the actual syscall handlers, *e.g.,* `xx_open()` (line 18) for handling the syscall `open()` of the device file and `xx_ioctl()` (line 19) for handling the syscall `ioctl()`. In the `ioctl()` handler, we can see the command value in `switch case` statements (from line 31 to line 46) or `if` conditions (line 47). Also, we can see the third parameter of the `ioctl()` handler (line 34) declared as a long type but treated as a pointer to a specific struct type, under a given command value (line 32). This is because a given command value may completely change the behavior the syscall `ioctl()` and hence require custom data structures passed in as the third argument. Besides, we can see that a non-open file descriptor being installed and returned from `ioctl()` (line 38-41), under a specific command value.

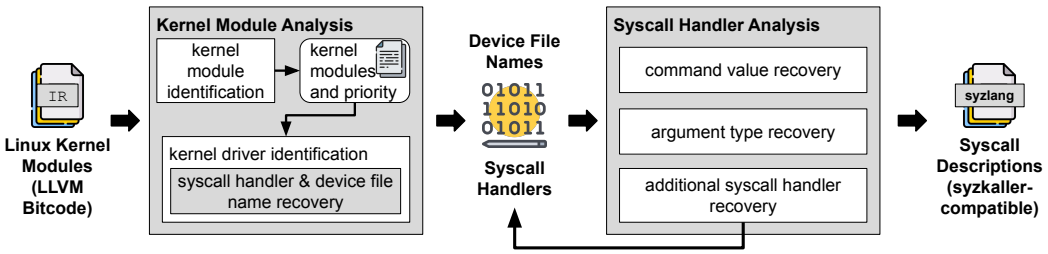

Figure 3: System Overview

Finally, there are two other structs defined in the beginning (line 1 to 4). Interestingly, `struct xx_device_ops` (which is reachable by the other struct) looks like a syscall handler struct that stores function pointers pointing to actual handler functions. However, as we can see, the struct is only really used inside a syscall handler (see line 43 to 45) and is not used to handle syscalls. This shows that a simple heuristic to search for structs that look like a syscall handler will not work.

3.2. Overview

Now we present the high-level workflow of SyzDescribe (as depicted in Figure 3). SyzDescribe requires LLVM bitcode [14] of Linux kernel modules as input and produces syscall descriptions in syzkaller-compatible format as output. As shown in Figure 3, there are two main stages in SyzDescribe:

- **Kernel Module Analysis**, SyzDescribe detects kernel modules by their initialization functions and associates modules with priorities that determine the order of execution during kernel boot time. SyzDescribe then recognizes the presence of any kernel driver that may span across more than one kernel module, and recovers the basic interfaces created and exposed to the user space, *i.e.,* supported syscalls (and the corresponding handlers) and device file name.
- **Syscall Handler Analysis**, For each discovered syscall handler, SyzDescribe attempts to recover additional details about these interfaces. This includes the command values and argument types supported by the `ioctl()` handler. In addition, SyzDescribe can recover additional syscall handlers which can lead to recovery of **non-open file descriptor dependencies**. Finally, SyzDescribe can translate the knowledge into syscall descriptions directly usable by syzkaller.

The premise of our solution is that kernel driver development follows certain *contracts* or interfaces between the core kernel and drivers. SyzDescribe depends on a minimal set of such contracts (most of which exist for over a decade), including (1) how a driver / module is initialized, *i.e.,* where the initialization functions are defined and their order of invocation; (2) key types of driver and device objects (*e.g.,* `struct cdev` and `struct device`) and how they are initialized/registered; (3) file-related objects and how they are associated with syscall handlers. We will provide more details on exactly what SyzDescribe assumes in §3.3 and §3.4, and how we model the initialization and file-related operations based on them. The faithful modeling of the critical kernel driver operations is the key feature that differentiates our solution from the prior work.

3.3. Kernel Module Analysis

Here, we focus on modeling the initialization process of kernel modules (and hence drivers). This component is a key contribution of SyzDescribe.

3.3.1. Kernel Module Identification. First of all, we need to detect all module init functions (defined in 2.1) to identify kernel modules (where kernel drivers are housed) in the bitcode. In the common case, this is straightforward as most kernel modules use the easily recognizable macro `module_init()` to declare module init functions (as seen in Figure 2). However, this is not the only macro that does this. For example, `subsys_initcall()` is used in the sound driver for declaring the init function as well. Regardless of how many there are, these macros will eventually use the lowest level macro named `__define_initcall`. It is important to note that `__define_initcall` takes two arguments — the first specifying the function to be declared as a module init function, and the second representing the priority of the corresponding module init function, determining the global order of execution by the kernel. Various higher-level macros that eventually rely on `__define_initcall` are defined (see Figure 7 in Appendix). Note the second argument being different — 0 being the highest priority, 1 being the next, and 1s being the one after, and so on. In Linux, all loadable kernel modules are declared by `module_init`, which means they will have a priority of 6. Based on this, our analysis can extract each declared module init function and its associated priority.

The order of execution of different kernel module init functions matters when a driver is defined across different kernel modules. This is because a driver can perform some partial initialization in one kernel module, *e.g.,* setting up certain indirect calls (discussed more in §4.2), which would influence the later kernel module init functions.

We rely on the above "contract", which has existed in the kernel since v2.6.19 [15].

3.3.2. Kernel Driver Identification. Now that we can identify all kernel modules globally and decide the order of their

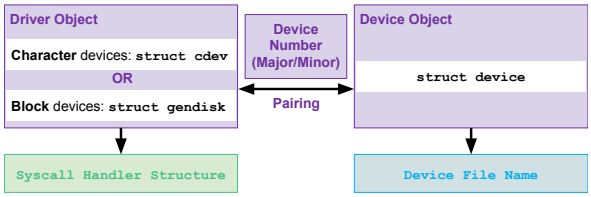

Figure 4: Relations among driver/device-related data structures in kernel (colors correspond to different parts in Figure 2)

initialization, our next task is to identify kernel drivers. As mentioned earlier, some drivers are defined across multiple kernel modules and it is not immediately clear which kernel modules collectively form a single kernel driver. Our solution is to iterate through all kernel modules and group the ones that share data structures or unique device numbers.

Driver and device object identification and pairing. From Figure 2, we know there are two key types of objects that are defined during the initialization of a kernel driver, and they collectively define the basic driver interface. Our goal is to identify the two types of objects and associate them (regardless of whether they are defined in the same kernel module or different ones). As shown in Figure 4, one type of objects is what we call *driver* objects (*e.g.,* `struct cdev`) that contain descriptions about syscall handlers (often defined as file operation structures). The other type is what we call *device* objects (*e.g.,* `struct device`) that contain descriptions about device file names.

Since we consider only character and block devices, there are only `struct cdev` and `struct gendisk` types, which are the corresponding *driver* objects. There is only `struct device` type that corresponds to the basic *device* objects.

To pair the two types of objects, we rely on the device numbers assigned to each type of object (*e.g.,* line 6, 8 and 11 in Figure 2). Different kernel drivers will have their unique device numbers. If the major number (*e.g.,* `MAJOR`) is already unique, then the minor number (*e.g.,* `MINOR`)) can be optional for the *driver* object. Otherwise, the two numbers combined need to be unique globally. Note that *device* objects must have minor numbers assigned. When multiple *device* objects (with different device file names) are supposed to be paired with a single *driver* object, they will all share the same major number, but the minor number of the driver object will not be set. This allows the pairing to rely on only the major number [11].

To summarize, as long as we can recover the major number and minor number assigned to each *driver* and *device* object, regardless of which kernel modules they are located in, we can pair them.

We rely on the above "contract" which has existed in the kernel since v2.6.12. We will explain more details on what additional types *device* objects there may be and how we handle them in §4.3. For now, we will assume that there are a few pre-defined ones, which means it is straightforward to identify the creation and initialization of both *driver* and *device* objects by type.

Syscall handler and device file name recovery. Now that we can track both the *driver* and *device* objects by type as mentioned above, we can simply inspect their related critical fields, at the time when the objects are registered through well-marked kernel functions (*e.g.,* `cdev_add()`) to recover what we need. As we show in Figure 4, syscall handlers are stored in file operation structures which are assigned to *driver* objects, whereas device file names are stored in *device* objects.

It is important to note that the various syscall handlers stored in the same operation structure naturally lead to **open file descriptor dependencies** being recovered. Specifically, by convention, an `open()` handler is assumed to return a file descriptor which is then fed into the co-located syscall handlers such as `ioctl()` and `read()` (in the same operation structure).

For example, we can identify store instructions that write into `cdev->ops`, the source operand of which will correspond to the syscall handler structure. Similarly, we can identify functions such as `dev_set_name(dev, "name%", id);` that ultimately sets the field of `dev->kobject.name`. For ease of implementation, we model a common set of APIs that perform the initialization (the complete list is in Figure 8 in appendix), including the support of common format string specifiers such as `%s` and `%d`. In general, these can be extended with minor engineering efforts. As shown in our evaluation, we are able to successfully recover 72% of all device names in our dataset (see §5.2).

3.4. Syscall Handler Analysis

As mentioned, once we recover the syscall handlers, we will need to analyze these handlers in more detail to recover even more specifics about the interfaces exposed to the user space. This primarily includes the arguments of syscalls and non-open file descriptor dependencies. Note that the prior work supports only the former, which can leave important code uncovered due to the lack of knowledge of dependencies.

3.4.1. Command Value Recovery. For most kernel drivers, the main driver logic is encoded in the `ioctl()` syscall handler. First, we identify the command values by checking their uses in `switch case` statements and `if` conditions (only considering equality comparisons). Then we extract the basic blocks behind a specific command value through a reachability analysis, *i.e.,* those that belong to the true branch of an `if condition` or a particular `switch case`. For example, in Figure 2, the reachable basic blocks of command value `cmd_1` are line 33 - 35, and the reachable basic blocks of command value `cmd_2` are line 38 - 41. Then we can analyze these basic blocks separately to recover more fine-grained syscall descriptions as will be described next. Besides, we have a tailored solution to resolve indirect calls (detailed in §4.2).

3.4.2. Argument Type Recovery. After the reachability analysis for each command value, we aim to recover the

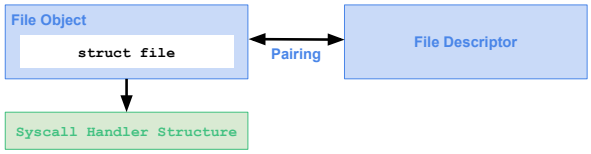

Figure 5: Relations of data structures around non-open file descriptors (colors correspond to different parts in Figure 2)

corresponding type of argument (the one right after the command argument). Similar to prior work, we model common kernel functions such as `copy_from_user()` in order to achieve this [11], [6], [7]. As we illustrated in Figure 2, in line 34, we can see the destination argument type of `copy_from_user()` being `struct xx_arg` (under command value `cmd_1`). This allows us to infer that the argument type is a pointer type pointing to a `struct xx_arg` object. In addition to `copy_from_user()`, we also model function `memdup_user()` that was missed in prior work. Note that our current solution does not support nested argument types [16], which we leave as future work.

3.4.3. Additional Syscall Handler Recovery. In addition to module init functions, syscall handlers themselves can also create and register additional syscall handlers. This is the most common in `ioctl()` handlers where additional `struct file` objects are created and registered. Inside a `struct file` object, there is a pointer to the `struct file_operations` object, representing the corresponding set of syscall handlers associated with the file object. In addition, as shown in Figure 5, by convention, a `struct file` object is paired with a file descriptor that will be returned to the user space. For example, in Figure 2, from line 37 to 41, we can see the `ioctl()` handler with command value `cmd_2` has created a file descriptor and a `struct file` object, and paired them through a specific function called `fd_install()`.

If we can discover the creation and initialization of a `struct file` object, as well as its association with a file descriptor, we will be able to infer two things: (1) the corresponding syscall handlers will be exposed to the user space, which we should continue to analyze recursively for command value, argument type, and additional syscall handler recovery; (2) a **non-open file descriptor dependency**, which is effectively formed because we know that the syscall handlers will operate exclusively on the paired file descriptor that is returned to the user space. In other words, following the example in Figure 2, we know the return value of the `xx_ioctl(fd, cmd_2, arg)` should be used as the first argument of the syscall handlers defined in `no_fops`.

Our solution models the `struct file` object and file descriptor based on the set of related kernel functions. Interestingly, we find that none of the prior work have modeled these behaviors and therefore will miss the additional interfaces and **non-open file descriptor dependencies** in syscall descriptions. The "contract" (including both the type of objects and the functions) we rely on has existed in the kernel since v2.6.12.

4. Implementation

SyzDescribe is implemented as a static analysis tool using the LLVM toolchain, based on LLVM 14. We separately handled the kernel module analysis and syscall handler analysis. For the former, SyzDescribe performs a top-down inter-procedural, context-sensitive, and field-sensitive analysis. As an optimization, we prune the functions that do not lead to driver or device related operations. For the latter, SyzDescribe performs a top-down inter-procedural, context-sensitive, and flow-sensitive analysis. Flow-sensitivity is necessary to differentiate the branches executed under different command values where corresponding argument types are extracted. In total, there are 8.2k (C++) lines of code for the whole system (including generation of syzlang format syscall descriptions) and 0.3k (Golang) lines of code for building and linking LLVM bitcode of the Linux kernel. In this section, we will describe in more detail several implementation aspects of our tool.

4.1. Kernel Module Identification

As mentioned, we rely on the macro `__define_initcall()` to identify the declaration of kernel modules. However, in practice, these macros are expanded during the pre-processing of the compiler. Since our analysis is on the LLVM intermediate representation (where macros are already expanded), we can no longer observe the macros. In fact, even at the source level, `__define_initcall()` is implemented through the inlined assembly. Such assembly code will carry over to the LLVM bitcode as well. Our current solution is to recognize the pattern of such assembly directly, which works for kernel since v4.19. For loadable modules (as opposed to built-in modules), there is a different expansion of the macro even though they are still declared using macro `module_init` [17]. The form is to set a global function pointer with the name `init_module`, pointing to the entry points of the kernel module. Our current solution is to search the global function pointer in LLVM bitcode by name, which works for a kernel since v2.6.12.

4.2. Indirect Call Resolution

The indirect call is a well-known challenge in static analysis of kernel code. No perfect solution exists given the kernel's complex multi-entry and stateful nature [18]. Nevertheless, there are several recent advances [19], [20] that rely on heuristics based on type information (*e.g.,* parent structure of a function pointer) to match potential indirect call targets. The downside of such solutions is that they still over-approximate and produce many false indirect call targets, which substantially lengthens the overall analysis time in some of our experiments.

In our solution, we apply a simple and effective filter on top of [19] (which has an open source version) to reduce the set of indirect call targets. Our observation is that the scope of our analysis is much more focused on module init

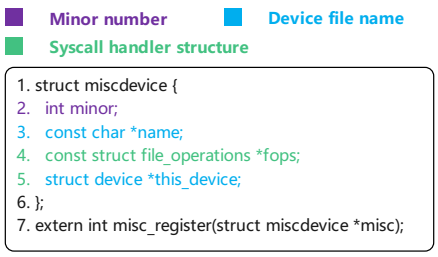

Figure 6: Miscellaneous structure

functions rather than syscalls. Both can set up indirect call targets and perform indirect calls. But there is an inherent ordering of these functions, which we can leverage to prune indirect call targets that are impossible. Namely, an indirect call target is feasible at an indirect call site only if the target has been set up in prior functions. In other words, given the initial targets of an indirect call site using the solution from [19], we further constrain the targets to those that have been seen (and used in some way) in functions that *can* be invoked earlier.

We know that different modules are initialized with different priority/ordering in §3.3.1. Loadable kernel modules are an exception because they can be loaded on demand at any point in time and therefore cannot be assigned a specific ordering. The ordering also applies beyond module init functions. By definition, all syscall handlers for a driver can only be invoked after the corresponding module init functions. Furthermore, the `open()` handler, if present, is always invoked earlier than other syscall handlers. For instance, let us say the targets of an indirect call site within a module init function are said to contain two targets (*e.g.,* `funcA` and `funcB`) according to [19]. Suppose `funcA` is used in another module init function with a higher priority, and `funcB` is used in a syscall handler for an unrelated module, we will retain `funcA` and prune `funcB`. This is because only `funcA` can possibly be set up before the indirect call site. This pruning is effective because oftentimes the type-based methods will match many false indirect call targets just because their types match, regardless of where the targets are (within the same modules or outside). To give some concrete evidence, we find that SyzDescribe successfully reduced the average number of indirect call targets per call site from 33.2 to 5.8 for the `allyesconfig`.

4.3. Additional Device Object Modeling

As mentioned in §3.3.2, a key step in identifying drivers and their interfaces is through recognizing and pairing *driver* and *device* objects. We mention that we only recognize `struct cdev` and `struct gendisk`.

For *device* objects, there is only one primitive `struct device` type, which sometimes can be encapsulated by other types. For example, `struct miscdevice` is such a type that encapsulates `struct device` — it contains a pointer to a `struct device` object, as shown in Figure 6. Such objects can be created and manipulated directly through another layer of abstraction (*e.g.,* using separate APIs). For example, kernel driver developers can choose to assign the device file name to the field of `struct miscdevice`. But then when the object is registered through `misc_register()`, the `name` will be copied into the field of the encapsulated `struct device` object ultimately. Similarly, the `minor` number will be propagated to the `struct device` object.

Note that syscall handlers are supposed to be defined by *driver* objects according to the contract we described in §3.3.2, and yet `struct miscdevice` contains a pointer named `fops` pointing to a set of syscall handlers as well. This may seem contradictory initially. However, in reality, for any driver that uses `struct miscdevice`, there will indeed be a driver object and its syscall handlers registered separately with a matching device number. What happens is that the new syscall handlers defined in `struct miscdevice` will be used to replace the original ones after it is registered through `misc_register()`.

In principle, we can automatically recognize such extended *device* objects which encapsulate `struct device` and analyze them accordingly. However, for ease of implementation, we recognize these objects by modeling the related APIs. By searching through the latest Linux kernel, we find only four such types, namely `struct miscdevice`, `struct usb_class_driver`, `struct drm_driver`, and `struct snd_minor`. The first three were defined since Linux v2.6.12 and the last one was defined in Linux v2.6.16.

5. Evaluation

In this section, we present our evaluation results. Specifically, we aim to answer the following questions:

- ***RQ1.*** What's the number of the syscall descriptions generated by SyzDescribe compared with other solutions? (§5.1)
- ***RQ2.*** What's the quality of the syscall descriptions generated by SyzDescribe? (§5.2)
- ***RQ3.*** How effective are syscall descriptions generated by SyzDescribe in fuzzing? (§5.3)

All experiments except Android fuzzing ones are conducted on a machine with an Intel(R) Xeon(R) Gold 6248 CPU and 512GB of RAM, running Ubuntu 18.04 LTS. The target Linux kernel version is v5.12, which is released on 04/25/2021. The version of syzkaller and its syscall descriptions are both based on the commit of 36c8823 (made on a closest date) from the git repository [1]. We use the `allyesconfig` for the Linux kernel to generate descriptions, and the configuration from syzbot [21] for fuzzing experiments. This is because the allyesconfig cannot be used to compile a bootable kernel, *e.g.,* only a subset of drivers from the same category should be enabled in a specific operating system. The syzbot config represents the decision made by Google when fuzzing the Linux kernel through QEMU.

Table 1: Syscall Descriptions Comparison of SyzDescribe

Name	Config	#HANDLER	#NAME	#CMD &TYPE	#N-OPEN	Time (hours)
SyzDescribe	*allyes*	415	653	3,379	58	3.76
DIFUZE	*allyes*	4,334	48	560	0	0.42
DIFUZE_F	*allyes*	4,403	60	761	0	0.16
SyzDescribe	*syzbot*	185	229	2,119	33	0.47
KSG	*syzbot*	88	1,098[1]	714[2]	0	2.38
syzkaller	36c8823	153	232	1,737	27	n/a

[1] KSG generates many similar device file names (*i.e.,* "usbmon0", "usbmon1", ..., "usbmon40"), which is "usbmon#" in SyzDescribe and syzkaller descriptions. If we group them, the number is 142.
[2] KSG generates some totally repeated CMD&TYPE. If we remove those repeated ones, the number is 485.

5.1. Overall Results

In this section, we present overall results of the syscall descriptions generated by SyzDescribe, and compare them against those from DIFUZE, and KSG, as well as the syzkaller descriptions (*i.e.,* manually-curated syscall descriptions from the official syzkaller repository).

Metrics. Our comparison focuses on the following metrics. #HANDLER is the number of syscall handler structures (from drivers and their non-open file descriptor dependencies). #NAME is the number of device file names. #CMD is the number of command values. #TYPE is the number of argument types. #CMD&TYPE is the number of valid combinations of them. #N-OPEN is the number of non-open file descriptor dependencies, which result directly from the additional handlers recovered as described in §3.4.3. We highlight the non-open dependencies because no existing solutions attempt to recover them. Moreover, such dependencies are always from "complex" device drivers and recognizing them do lead to significant new code coverage that would otherwise be missed completely (as will be shown in the kvm example later).

Analysis time. As we can see in Table 1, SyzDescribe spends 3.73 hours to finish the analysis of 7,554 kernel modules under *allyes*, averaging 1.66 seconds per module. The maximum time SyzDescribe needs to spend on one module is 86 seconds. DIFUZE runs significantly faster as it does not attempt to model the module initialization process. Finally, KSG runs slower than SyzDescribe in the Linux kernel with *syzbot* config.

5.1.1. SyzDescribe vs. DIFUZE and DIFUZE_F. Since our evaluation is based on a new version of Linux kernel (v5.12), we had to port DIFUZE to a newer version of LLVM/Clang, which is required by Linux v5.12. The results of this ported version are labeled as DIFUZE in the Table 2. However, due to the updates of LLVM/Clang and kernel, there are some compatibility issues that impacted DIFUZE. To be fair, we fixed these issues that fall under two categories: (1) correcting hard-coded offsets for certain types that have changed in kernel v5.12; (2) fixing failed tracking of `copy_from_user()` destination object due to changes in compiled LLVM bitcode (otherwise no argument type can be recovered). The fixed version is labeled as DIFUZE_F.

As we can see in Table 1, SyzDescribe recovers far less syscall handler structures than DIFUZE or DIFUZE_F (415 vs. 4,334 or 4,403). However, as will be shown in §5.2.1 and §5.3.1, most syscall handler structures found by DIFUZE or DIFUZE_F are actually false positive, given that they recovered only 48 or 60 device file names. For reference, in manually-curated syzkaller descriptions, there are only 153 handler structures from 232 device file names. In comparison, the 415 handler structures recovered by SyzDescribe correspond to 653 device file names. Regarding cmd&type combinations and non-open explicit dependencies, SyzDescribe recovered many more descriptions compared to DIFUZE and even the manually-curated ones.

5.1.2. SyzDescribe vs. KSG. To be fair, we run SyzDescribe using the same *syzbot* config which compiles a subset of drivers (compared to *allyesconfig* into the kernel, therefore lowering the number of descriptions. Even then, SyzDescribe still outperformed KSG significantly, recovering more syscall handler structures (185 vs. 88). Upon a closer look, the discrepancy was due to: 1) 64 drivers which are compiled into the kernel but need to satisfy some other dependencies (*e.g.,* plugin related hardware or execute other syscalls) to make the related device file exposed under the `/dev` directory. 2) 33 non-open file descriptor dependencies. 3) 49 drivers which need additional modeling. The first reason is clearly a limitation of a dynamic solution. The second and third reasons are more related to the modeling necessary to capture the syscall handlers and their associated interfaces. The third reason is especially interesting because we did not anticipate that even a dynamic solution still needs a similar modeling of the programming conventions. In this case, KSG did not model how to look for the syscall handlers of block devices, *i.e.,* they are not located through the struct file object's pointer field `f_ops` (which is what KSG assumes to be), thus missing all the handlers of block devices.

In terms of file names, SyzDescribe recovers fewer device file names than KSG (229 vs. 1,098). It would appear that SyzDescribe does a much worse job. However, the discrepancy is in fact much smaller than the data suggest. Specifically, the 1,090 file names are bloated because KSG counts variations of the same device file name, *e.g.,* "usbmon0", "usbmon", ..., "usbmon4" as separate ones whereas SyzDescribe (as well as syzkaller syscall descriptions) counts them as a single one based on the format string: "usbmon%" that appeared in the module initialization logic. Indeed, all of these file names point to the same set of syscall handlers. If we group such files for KSG's result, the number of file names goes down to 142.

Finally, SyzDescribe recovers many more cmd&type combinations than KSG (2,119 vs. 714) because (1) SyzDescribe finds more syscall handlers, and (2) SyzDescribe performs an inter-procedural static analysis with a more accurate indirect call resolution while KSG does an intra-procedural symbolic execution, which misses many command values and argument types that appear in the callees of the `ioctl()` handler. In fact, we find that

Table 2: Accuracy comparison of SyzDescribe vs. DIFUZE vs. Ground truth

Name	#HANDLER			#NAME			#CMD			#TYPE			#N-OPEN		
	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1
SyzDescribe	96	0	0.95	74	31	0.71	1,039	48	0.84	521	2	0.74	6	0	1.00
DIFUZE	49	30	0.53	16	4	0.26	269	24	0.32	0	0	0.00	0	0	0.00
DIFUZE_F	52	25	0.57	16	4	0.26	269	26	0.32	78	4	0.16	0	0	0.00
Ground truth	106	-	-	103	-	-	1,400	-	-	894	-	-	6	-	-

Table 3: Accuracy comparison of SyzDescribe vs. KSG vs. Ground truth

Name	#HANDLER			#NAME			#CMD			#TYPE			#N-OPEN		
	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1
SyzDescribe	74	0	0.99	58	24	0.75	837	42	0.81	399	2	0.70	6	0	1.00
KSG	43	2	0.71	45	0	0.77	223	303	0.26	64	15	0.16	0	0	0.00
Ground truth	76	-	-	72	-	-	1,192	-	-	732	-	-	6	-	-

1,270 of 2,119 combinations recovered by SyzDescribe require looking into the callees.

5.2. Accuracy of SyzDescribe's Descriptions

In this section, we try to understand the correctness of the generated syscall descriptions of various solutions against a set of ground truth descriptions which we manually construct.

Dataset. We randomly pick 100 kernel drivers and manually analyze them with best efforts which form the ground truth dataset. To mitigate potential errors we make during the process, we always cross-validate with the syscall descriptions generated with SyzDescribe and from syzkaller (the manually-curated ones) to see whether we missed anything. Overall, it takes more than one person month to collect the ground truth for the 100 drivers. To our knowledge, no prior work has built a ground truth dataset of this scale for evaluation.

We use the full 100-driver dataset for comparison against DIFUZE as shown in Table 2. Again, to be fair against KSG, we choose to include a subset of the 100 drivers that are compiled with the *syzbot* config, *i.e.,*, 70 drivers; the other 30 are compiled only under the *allyes* config. Similarly, because 57 drivers are missed completely in the syzkaller descriptions, to be fair against them, we choose to include a subset of drivers that are covered by them, *i.e.,*, 43 drivers.

5.2.1. SyzDescribe vs. DIFUZE and DIFUZE_F. The results are shown in Table 2. Compared with DIFUZE and DIFUZE_F, SyzDescribe has a significant advantage in all aspects. First, SyzDescribe performs better on both #HANDLER (0.95 vs. 0.57 of F_1) and #NAME (0.71 vs. 0.26 of F_1). For #NAME, although SyzDescribe has more false positives for some complex drivers, SyzDescribe significantly boosts the true positives and is much closer to the ground truth. Because SyzDescribe is more accurate than DIFUZE in both recovering syscall handlers and resolving indirect calls, it also finds more command values (0.84 vs. 0.32 of F_1) and argument types (0.74 vs. 0.16 of F_1).

Table 4: Accuracy comparison of SyzDescribe vs. syzkaller descriptions vs. Ground truth

Name	#HANDLER			#NAME			#CMD			#TYPE			#N-OPEN		
	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1	TP	FP	F_1
SyzDescribe	47	0	0.99	42	7	0.87	807	34	0.81	393	2	0.68	5	0	1.00
syzkaller	45	0	0.97	46	0	0.98	922	0	0.89	506	3	0.80	3	0	0.75
Ground truth	48	-	-	48	-	-	1,141	-	-	755	-	-	5	-	-

Table 5: Improvement of SyzDescribe vs. syzkaller descriptions

Category	Syscall handler structure	#	Commit time of related code in Linux kernel	Update time of latest syzlang (before 04/2021)
kernel drivers with CMD FN	lo_fops	1	05/2020	12/2019
	sg_fops	7	10/2014	01/2019
	usbdev_file_operations	11	08/2019	01/2020
	rfkill_fops	1	06/2009	03/2019
	snd_timer_f_ops	6	04/2018	03/2020
	snd_ctl_f_ops	1	05/2005	01/2020
	nbd_fops	1	04/2005	02/2021
	raw_fops	19	01/2020	06/2020
	ashmem_fops	2	12/2011	01/2018
	ppp_device_fops	4	12/2020	01/2019
	tun_fops	1	02/2018	03/2020
kernel drivers with TYPE FN	lo_fops	1	05/2020	12/2019
	usbdev_file_operations	5	01/2015	01/2020
	raw_fops	1	10/2015	06/2020
	sr_bdops	9	04/2005	08/2020
	hiddev_fops	5	03/2008	04/2020
	evdev_fops	3	08/2010	03/2020
kernel drivers with TYPE FP	snd_timer_f_ops	2	04/2018	03/2020
	snd_ctl_f_ops	1	12/2019	01/2020
kernel drivers with N-OPEN FN	udmabuf_fops	1	09/2018	02/2019 (fixed in 01/2022)
	lo_fops	1	05/2007	12/2019

Besides, SyzDescribe can recover 6 non-open file descriptor dependencies, which are completely missed by DIFUZE.

5.2.2. SyzDescribe vs. KSG. The results are shown in Table 3. Compared with KSG, SyzDescribe has advantages in all aspects except the device file names. For #HANDLER, SyzDescribe performs better (0.99 vs. 0.71 of F_1). For #NAME, we note that, for KSG, we group variations of the same device name such as "usbmon0" and "usbmon1" into a single device name to be consistent with how we count them in SyzDescribe. SyzDescribe has more true positives but also has more false positives, which makes its F_1 score slightly worse than KSG (0.75 vs. 0.77). It is worth noting that, as we will discuss later in §5.3.3, false positives generally do not have a big influence on fuzzing. SyzDescribe has an overwhelming advantage in #CMD (0.81 vs. 0.26 of F_1) and #TYPE (0.70 vs. 0.16 of F_1). Interestingly, we find that KSG generates many unspecified command values for `ioctl()` interfaces, because it is unable to pinpoint the specific values through its intra-procedural symbolic execution. We consider such CMDs as FPs, since KSG does generate an entry for them (sometimes multiple repeated ones) in the description file. Finally, SyzDescribe again recovers 6 non-open file descriptor dependencies, which are completely missed by KSG.

5.2.3. SyzDescribe vs. syzkaller Descriptions. Table 4 shows the results are competitive with the manually-curated syzkaller descriptions. Interestingly, SyzDescribe even recovered more syscall handlers (0.99 vs. 0.97 of F_1), which is because of two additional non-open file descriptor dependencies. Meanwhile, SyzDescribe generated 42 device file names (0.87 vs. 0.98 of F_1), 807 CMDs (0.81 vs. 0.89 of F_1) and 393 TYPEs (0.68 vs. 0.80 of F_1), which are marginally worse than the existing descriptions.

Note that the results vary from driver to driver. Regarding #CMDs, SyzDescribe generated more CMDs for 11 drivers, the same number for 17 drivers, and less for 13 drivers. Regarding #TYPEs, SyzDescribe generated more TYPEs for 6 drivers, the same number for 24 drivers, and less for 11 drivers. Combined together, SyzDescribe generated either more CMDs or more TYPEs for 13 drivers. We then analyze these 13 drivers in more depth to understand these "bugs" in human-generated descriptions. We summarize the results by types of "bugs" for these drivers (identified by their corresponding `ioctl()` handlers) in Table 5. We can see that SyzDescribe recovers 78 missed command values or argument types across a total of 13 kernel drivers[1]. In addition, we see manual syscall descriptions incur three false positives with regard to the argument types across two drivers, identified by `snd_timer_f_ops` and `snd_ctl_f_ops`. This drives us to investigate the reason of these bugs.

Interestingly, for the two drivers (*i.e.,* identified by `lo_fops` and `ppp_device_fops`), they have been changed with new command values and argument types after the last update of the syscall description, indicating that ongoing human maintenance is needed. In both cases, the last updates to the descriptions were in 2019 and clearly no updates were made up until 04/25/2021 (which is the version we analyzed). For the three false positives of syzkaller descriptions, we find that they are due to the evolution of the kernel code. In other words, the syscall descriptions were accurate in describing some argument types, but the definitions of those types have changed after some Linux kernel versions (04/2018 and 12/2019 respectively), which causes the syscall descriptions to become out-of-sync. More interestingly, even though there are other updates to the syscall descriptions (in 2020), the human experts have missed such problems. For other cases, the relevant changes in the kernel have been made much earlier, which technically are visible to human experts. Unfortunately, they are still missed likely because of the labor-intensive and error-prone nature of reading and understanding kernel code. In addition, we also inspected the two false negatives of the non-open file descriptor dependencies from Table 4.

We find that only one (*i.e.,* `udmabuf_fops`) of these "bugs" is eventually fixed on January 2022 before we started to report them. This analysis shows that these syscall description "bugs" can persist over an extended period of time. So far, we have reported all the bugs to syzkaller (all of which are fixed) and shared the syscall descriptions generated by SyzDescribe.

One last thing worth mentioning is that sometimes seemingly small improvements can be significant. One example is that SyzDescribe recovers 7 missing command values and 9 missing argument types for the driver identified by `sg_fops`, which eventually lead to 4 new crashes in our fuzzing experiments in §5.3.

5.2.4. SyzDescribe vs. Ground Truth. Table 2 illustrates how well SyzDescribe performs with respect to ground truth. In this section, we inspect the gap and summarize the reasons which will be helpful to facilitate further improvements of SyzDescribe.

- **Syscall handler.** SyzDescribe does not have any false positives of #HANDLER. The major reason for the false negatives (9 out of 10) is that syscall handler structures take dynamically constructed variables (as opposed to target functions directly), which we do not handle in our current implementation. The remaining one case is a special case as it delays the creation of the device object to be performed inside the `ioctl()` handler and not in any module init functions. However, currently SyzDescribe only tries to identify device file in module init functions.
- **Device file name.** The main reason for the false negatives of #NAME (28 out of 29 false negatives) is similar. That is, the device file names are constructed dynamically (not following the standard functions with format strings that we model). As for the false positives of #NAME, the reason is that their major number/device number are generated dynamically, so we can not match the driver object and device object accurately. In this situation, our current policy is to match all possible unmatched device file names within the same module init functions, leading to false positives.
- **Command value.** There are two reasons for the false negative of #CMD. The first reason is the false negatives of #HANDLER, which means we can not find the command value if we can not find the related `ioctl()` handler. There are 13 cases of this reason. The second reason contributes to the remaining 360 cases and is related to the implementation of `ioctl()` handlers. We find that some drivers employ non-trivial (custom) uses of the command values to determine which branches to enter. For example, the check `if(_IOC_NR(cmd) == _IOC_NR(HIDIOCSFEATURE(0)))` makes it difficult to decide what exact value we should supply. And the false positive of #CMD is also from this reason. Symbolic execution is a natural solution to overcome this, which we leave as future work. Another example is that the command values can be used in a function array to decide where to branch (*e.g.,* see Figure 9 in appendix). This can be improved with better modeling of such programming patterns.
- **Argument type.** There are three reasons for the false negatives of #TYPE. The first reason is again the false

1. It is possible that SyzDescribe still finds something missed in human-generated descriptions when SyzDescribe recovers the same number or even fewer number of command value or argument type (as our results may not overlap completely), and thus the 13 drivers represent a lower bound

negatives of #HANDLER or #CMD, which contributed to 216 cases. The second reason is the incomplete data flow tracking of `copy_from_user()` to determine the type of the argument. The third reason is the failure to model additional functions (defined in inlined assembly) that copy data from user space, *e.g.,* `get_user()`, which we do not currently handle.

- **Non-open file descriptor dependency.** SyzDescribe recovers all of them and there is no false positive or false negative.

5.3. Effectiveness of SyzDescribe in Fuzzing

In this section, we evaluate the effectiveness of SyzDescribe by running fuzzing on the generated syscall descriptions against DIFUZE, KSG, manually-curated syzkaller descriptions, as well as the ground truth descriptions we created. Overall, it is evident that the descriptions generated by SyzDescribe is much better than other automated solutions, competitive with the manually-curated ones (in many cases complementary), and not too far off from the ground truth.

Dataset and Setup. We run three fuzzing experiments. The first two are both against Linux kernels running in QEMU under the *syzbot* config. The last one is against an Android kernel running on a Pixel 6 device compiled through the official HWAddressSanitizer config [22].

For the first experiment, we pick 30 out of 100 kernel drivers in Table 2 based on whether they are compiled and available in QEMU. We fuzz each kernel driver individually for 24 hours with three runs. For "kvm" specifically, we run 120 hour fuzzing sessions because it is a much more complex kernel driver (with 3 non-open file descriptor dependencies). Our fuzzing session for a driver consists of 8 CPU cores (4 QEMU instances with 2 CPU cores each). The coverage and the number of crashes (unique) for each driver are averaged over three runs. The descriptions we use in the experiment include SyzDescribe, DIFUZE_F, syzkaller ones, and the ground truth. The ground truth descriptions are curated by fixing FNs and FPs in the syzkaller descriptions, where they not only command values and type arguments but also argument value ranges [4], making them even more powerful.

For the second experiment, we compare specifically against KSG by fuzzing the whole kernel using all available descriptions for 72 hours and repeat for three runs (which is how KSG itself is evaluated [7]). This is because the syscall descriptions generated by KSG in many cases incorrectly associate distinct drivers with the same set of syscall handlers. For example, `/dev/disks` and `/dev/dri` are two different drivers and should have their own `ioctl()` handlers and descriptions. However, KSG associate them with the same descriptions (*e.g.,* handlers and command values). Such mix-ups make it hard to tease out the descriptions that belong to specific drivers.

For the last experiment, we do not have any comparison and instead use all the descriptions generated by SyzDescribe for fuzzing. The fuzzing campaign lasted for one week.

5.3.1. SyzDescribe vs. DIFUZE_F. Compared with DIFUZE_F, SyzDescribe has a significant advantage in both coverage and number of crashes as shown in Table 6, which is expected given the accuracy results. Specifically, DIFUZE_F failed to generate any syscall descriptions for 15 drivers, whose coverage is 0 in Table 6. For other drivers, SyzDescribe generally produced more or similar coverage. It is worth noting that SyzDescribe achieved much more coverage and crashes for "kvm" than DIFUZE_F because SyzDescribe recovered 3 non-open file descriptor dependencies.

5.3.2. SyzDescribe vs. syzkaller Descriptions. As we can see, SyzDescribe still makes significant improvements in overall coverage as shown in Table 6. First of all, there are 10 drivers without any coverage because of the lack of syzkaller descriptions. For the other drivers, the coverage of the syscall descriptions generated by SyzDescribe are competitive (mostly comparable). Note that the coverage may not overlap completely even though their numbers look similar. This is because their corresponding descriptions can encode complementary information. For example, for these 20 drivers, we find that SyzDescribe produced more CMDs in 9 drivers, fewer CMDs in 3 drivers, and the same number of command values for the remaining ones. However, more code coverage makes sense but maximizing code coverage does not directly mean finding most crashes[23]. SyzDescribe achieves fewer crashes then syzkaller descriptions because manually-curated syzkaller descriptions can include things that are out-of-scope for SyzDescribe. For example, in addition to argument types, syzkaller descriptions would contain valid ranges of values of arguments (which SyzDescribe currently does not support). In §6, we list the additional features in descriptions that SyzDescribe can support in the future.

5.3.3. SyzDescribe vs. Ground Truth. The results are shown in Table 6. We can see that overall the ground truth results are better with respect to both coverage and number of crashes (which is also the case for most individual drivers). This is expected as the ground truth descriptions are the most complete (without FNs or minimal FNs). Interestingly, for some drivers, the coverage by SyzDescribe is marginally better than the ground truth whereas the number of crashes by SyzDescribe is much smaller. We find that this is because, for such drivers, SyzDescribe descriptions and ground truth descriptions share similar true positives, while the FPs in SyzDescribe actually helping to uncover some error handling logic that would otherwise not be covered (and there is also the randomness of fuzzing). Nevertheless, such error handling logic is typically shallow, and in most cases, the ground truth descriptions still find more crashes even if its coverage is slightly smaller.

To summarize, FPs do not appear to affect coverage significantly, especially given the long fuzzing sessions.

Table 6: Effectiveness comparison of SyzDescribe

Device Name	SyzDescribe		DIFUZE_F		syzkaller		Ground truth	
	#Cov	crash	#Cov	crash	#Cov	crash	#Cov	crash
“loop%d”	18,644	5.3	0	0.0	15,016	5.0	18,438	6.3
“loop-control”	7,799	1.0	7,789	1.0	6,422	0.7	7,800	1.0
“rtc%d”	14,513	4.0	0	0.0	13,061	4.3	14,153	3.0
“sg%d”	17,017	5.3	0	0.0	17,136	6.0	17,307	5.7
“sr%d”	15,554	2.0	0	0.0	15,264	2.0	15,400	2.3
“ptmx”...	15,195	4.0	0	0.0	15,239	5.7	15,833	6.0
“usbmon%d”	13,898	3.7	0	0.0	13,619	1.7	13,717	3.0
“snapshot”	4,099	0.3	4,070	0.0	3,422	0.0	3,968	0.0
“rfkill”	3,427	0.0	3,595	0.0	2,276	0.0	3,141	0.3
“controlC%d”	14,429	3.3	0	0.0	13,888	3.3	14,610	3.7
“timer”	4,364	0.0	0	0.0	2,977	0.7	4,334	0.5
“nbd%d”	15,606	3.7	0	0.0	15,423	5.3	15,234	2.3
“qat_adf_ctl”	3,779	0.3	0	0.0	2,545	0.0	4,056	1.0
“udmabuf”	2,505	1.0	2,285	0.0	1,391	0.0	2,520	1.0
“i2c-%d”	7,347	1.0	0	0.0	12,576	3.7	12,576*	3.7*
“uinput”	6,070	0.0	6,136	0.0	6,318	1.0	6,003	1.3
“ppp”	7,557	0.3	0	0.0	6,350	0.0	7,605	0.3
“ashmem”	3,799	0.0	0	0.0	3,300	0.0	3,684	0.7
“fuse”	3,423	0.0	3,603	0.0	1,737	0.0	3,409	0.0
“kvm”	16,932	4.0	6,093	1.7	21,593	9.7	24,289	7.0
“btrfs-control”	4,053	0.0	3,684	0.0	0	0.0	4,053*	0.0*
“capi20”	3,756	0.0	0	0.0	0	0.0	3,756*	0.0*
“fd%d”	13,872	3.3	0	0.0	0	0.0	14,127	6.7
“mISDNtimer”	3,546	0.0	3,662	0.0	0	0.0	3,708	0.0
“vhost-net”	4,469	0.0	4,367	0.0	0	0.0	4,469*	0.0*
“vhost-vsock”	4,398	0.7	4,524	0.0	0	0.0	4,398*	0.7*
“vmci”	6,860	2.0	5,320	1.3	0	0.0	6,154	2.0
“vsock”	3,620	0.0	3,359	0.0	0	0.0	3,620*	0.0*
“nvram”	3,732	1.0	3,701	0.0	0	0.0	3,732*	1.0*
“hpet”	3,254	0.3	3,587	0.0	0	0.0	3,254*	0.3*
Sum	247,516	46.7	65,777	4.0	189,553	49.0	259,334	59.8

* The fuzzing results are directly from SyzDescribe or syzkaller because the syscall descriptions are the same.

Table 7: Effectiveness of SyzDescribe vs. KSG

	#Cov	crash
KSG	30,345	7.7
SyzDescribe	34,097	9.0
KSG+syzkaller	50,049	11.3
SyzDescribe +syzkaller	58,201	15.3

This is evident when comparing SyzDescribe descriptions and ground truth descriptions where SyzDescribe produced more FPs but achieves similar coverage in many cases. On the other hand, FNs would significantly reduce both coverage and number of crashes, *e.g.,* ground truth descriptions achieved more coverage and crashes than syzkaller descriptions because of less FN.

5.3.4. SyzDescribe vs. KSG. As shown in Table 7, SyzDescribe had better results in both coverage and number of crashes. Although there are some device file names missed by SyzDescribe that are discovered by KSG, SyzDescribe has a significant advantage in CMDs, TYPEs and non-open file descriptor dependency (as seen in Table 3). In other words, SyzDescribe can go deeper than KSG and get more coverage and crashes. Interestingly, when we combine the syscall descriptions of SyzDescribe or KSG with syzkaller ones, SyzDescribe + syzkaller gains even more improvement in both coverage and number of crashes. The reason is that the main advantage of KSG is that it can open more device files, and also call `read()`, `write()`, *etc.*, which SyzDescribe currently does not incorporate into the descriptions. But this is something covered quite well by syzkaller descriptions. In contrast, SyzDescribe is more complementary to syzkaller descriptions.

5.3.5. Fuzzing Android Kernel Drivers of Pixel 6. Overall, SyzDescribe recovers 154 syscall handlers corresponding to 139 kernel drivers. Because some crashes are captured by RAMDUMP MODE in Pixel kernel (no public documentation about it), which can not be captured by syzkaller automatically, we have to manually record those crashes. In the end, although we bricked several Pixel 6, we still manage to find 18 crashes in Pixel 6 as shown in Table 8, demonstrating the effectiveness of the syscall descriptions. Unfortunately, due to the lack of detailed crash reports and documentation of the RAMDUMP MODE, it is difficult to understand the root causes of the bugs.

6. Limitations and Future Work

There are a few limitations of SyzDescribe, which can be areas for future work.

Specific values or value ranges. Currently we recover only the type of argument and do not support specific values or ranges of values that the last argument of `ioctl()` should take. One recent work [24] has explored this through dynamic analysis.

Other syscalls. SyzDescribe already identifies the syscall handlers and can in principle generate any syscall interfaces other than `open()` and `ioctl()`, *e.g.,* `read()`, `write()`, `mmap()`. The challenge though is to infer the appropriate argument types and values, which we leave as future work.

Other explicit dependencies. SyzDescribe supports only file-descriptor-related explicit dependencies but not other dependencies [3]. It will require a separate analysis such as the one proposed in HFL [16], which we consider being complementary.

Merging syscall descriptions. As we saw in §5.2.3 and §5.3, descriptions produced by SyzDescribe are complementary in various aspects, *e.g.,* non-overlapping CMDs and TYPES. This means that it is beneficial to merge the two syscall descriptions into a more complete description. We have not implemented the solution but envision that it is a promising direction to continuously maintain a single copy of description by both human and automated solutions.

7. Related Work

Linux Kernel Fuzzing. There are a number of recent studies improving various aspects of kernel fuzzing. Most notably, they are based on the state-of-the-art kernel fuzzer syzkaller. For example, MoonShine [25] generates seeds for syzkaller from traces of existing test cases, which improves

Table 8: Crashes found in Pixel 6

Kernel PANIC: KP: Asynchronous SError Interrupt	WARNING in lwis_ioctl_handler
Kernel PANIC: KP: Oops: Fatal exception: __skb_ext_put	WARNING in gvotable_cast_vote
Kernel PANIC: KP: Oops: Fatal exception: dit_enqueue_reg_value_with_ext_lock	WARNING in irq_set_irq_wake
Kernel PANIC: KP: BRK handler: Fatal exception: dit_hal_ioctl	WARNING in kbase_mem_pool_grow
Kernel PANIC: KP: BRK handler: Fatal exception: dit_hal_get_netdev	WARNING in drm_mode_object_add
Kernel PANIC: KP: BRK handler: Fatal exception in interrupt: comm:init, swapper/3-7	WARNING in gpio_to_desc
APC Watchdog: itom triggering err_fatal from HSIO USB31DRD_LINK to Refe	WARNING in corrupted
PMUCAL Watchdog: pmucal_local_disable: error on handling disable sequence. (pd: blkpwr_bo)	Emergency Restart
WARNING in drm_atomic_helper_commit_modeset_disables	INFO: corrupted

the bootstrapping stage of fuzzing. SyzVegas [26] improves mutation algorithm in syzkaller with reinforcement learning. HEALER [27] improves the quality of generated test cases and maximizes the coverage by learning relations between syscalls, which relies on the existing syscall descriptions in syzkaller. Besides, some tools are built for fuzzing specific Linux kernel subsystems, *e.g.,* file systems [28] and device drivers [29], [30]. PrIntFuzz[30] first enables more device drivers in QEMU via automated virtual device simulation, then generates the syscall descriptions for those drivers using DIFUZE[6] and fuzzes them from multiply interfaces. A recent study [3] measured the uncovered code after extensive fuzzing sessions and highlighted insufficiencies in syscall descriptions. Kernel fuzzing has also been used for purposes such as evaluating the impact and exploitability of bugs [31], [32], [33].

HFL [16] combines syzkaller with symbolic execution to enhance various aspects of kernel fuzzing, *e.g.,* resolving nested argument types, and inferring non-open file descriptor dependencies. Even though HFL does not aim to directly generate syscall descriptions, the inferences made during the process can be in principle ported to generate complementary aspects of syscall descriptions. Another very related work is SyzGen [24] that aims to generate syscall descriptions of closed-source macOS drivers from existing traces through binary analysis. The fundamental difference from SyzDescribe is that traces SyzGen relies on already included the basic device file names and syscall interfaces, which is the main result of SyzDescribe. Given the syscall interfaces, SyzGen also tries to infer the valid ranges of values for syscall arguments, which is complementary to SyzDescribe.

Programming Convention Modeling in Linux kernel. The contract between the core kernel and drivers is not the only programming convention that can be modeled. For example, Linux kernel maintains a list of error codes for different purposes, which have been modeled in prior to discover security check failures [34], *e.g.,* `-EACCESS` represents permission denied. Common APIs and data structures are also common targets for modeling, *e.g.,* locks and synchronization APIs have been modeled in Linux kernel for concurrency bug discovery [35]. Hecaton [36] relies on the kernel programming convention, where error handling (cleanup) code can be associated with specific state-changing statements, to undo the effects of syscalls. LinKRID [37] models the usage of internal reference counters of the Linux kernel to correctly perform reference counting. SADA leverages the Linux kernel convention for programming Direct Memory Access (DMA) in drivers to find unsafe DMA accesses [38]. DIFUZE [6] models how syscall handlers are often represented by operation structures that contain a number of function pointers, in an attempt to pair them with a device file name. Unfortunately, the modeling does not include the *driver* and *device* objects which lead to incorrect results in many cases, as we extensively explained in prior sections.

Linux Kernel Static Analysis. In terms of the scope of the Linux kernel, there are generally two styles of static analysis tools. The first is a focused analysis of specific parts of the kernel (*e.g.,* certain drivers). This includes DR.CHECKER [39], K-Miner [40], Juxta [41] and more recently SUTURE [18], which make it possible to conduct a precise analysis, *e.g.,* with inter-procedural flow-, context-, field-, index-, or opportunistically path-sensitive. Another style is to analyze all functions in the Linux kernel, *e.g.,* the study of bugs in Linux [42], [43], μchex [44], Graspan [45], INCRELUX [46], UBITect [5], which have better scalability but often need to trade off other aspects (such as precision and the kinds of analysis). Interestingly, none of these mainstream static analysis tools focused on the module init functions, which are completely separate from syscalls. For built-in modules, their module init functions are executed only once at the time of kernel boot. As we show, analyzing them in the correct order recognizes important data structures and context that can benefit the subsequent syscall analysis. We believe this is a missed opportunity that should be investigated further for other applications.

8. Conclusion

In conclusion, we present a principled solution to automatically generate syscall descriptions for Linux kernel driver with static analysis. The solution hinges on understanding the process of module initialization and the contract between the core Linux kernel and drivers. We evaluate SyzDescribe comprehensively and summarize the root causes for the gap between the results of SyzDescribe and ground truth, which is necessary for the future directions to improve automated syscall description generation.

Acknowledgment

The authors would like to thank the reviewers for their valuable feedback during the revision process. This work was supported in part by the NSF grants CNS-1953933, CNS-1652954 and CNS-1953932.

References

[1] Google. (2022) syzkaller. https://github.com/google/syzkaller. [Online]. Available: https://github.com/google/syzkaller

[2] ——. (2022) Syzbot. https://syzkaller.appspot.com/upstream. [Online]. Available: https://syzkaller.appspot.com/upstream

[3] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. Sani, "Demystifying the dependency challenge in kernel fuzzing," in *44rd IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, 22-27 May 2022*. IEEE, 2022, pp. 1–11.

[4] Google. (2022) Syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md. [Online]. Available: https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md

[5] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 221–232. [Online]. Available: https://doi.org/10.1145/3368089.3409686

[6] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[7] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "{KSG}: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[8] (2023) Syzdescribe. https://github.com/seclab-ucr/SyzDescribe. [Online]. Available: https://github.com/seclab-ucr/SyzDescribe

[9] (2023) Syzdescribe_syscall_description. https://github.com/seclab-ucr/SyzDescribe_Syscall_Description. [Online]. Available: https://github.com/seclab-ucr/SyzDescribe_Syscall_Description

[10] T. kernel development community. (2021) Driver basics. https://www.kernel.org/doc/html/v5.12/driver-api/basics.html. [Online]. Available: https://www.kernel.org/doc/html/v5.12/driver-api/basics.html

[11] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[12] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 291–307. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[13] T. Mu, H. Zhang, J. Wang, and H. Li, "Colafuze: Coverage-guided and layout-aware fuzzing for android drivers," *IEICE TRANSACTIONS on Information and Systems*, vol. 104, no. 11, pp. 1902–1912, 2021.

[14] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[15] T. kernel development community. (2021) https://github.com/torvalds/linux/blob/master/include/linux/init.h. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/init.h

[16] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: hybrid fuzzing on the linux kernel," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[17] T. kernel development community. (2021) https://github.com/torvalds/linux/blob/master/include/linux/module.h. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/module.h

[18] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in OS kernels," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 811–824. [Online]. Available: https://doi.org/10.1145/3460120.3484798

[19] K. Lu and H. Hu, "Where does it go?: Refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1867–1881. [Online]. Available: https://doi.org/10.1145/3319535.3354244

[20] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1205–1220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zhang-tong

[21] Google. (2022) stable-5.4-kasan.config. https://github.com/google/syzkaller/blob/master/dashboard/config/linux/stable-5.4-kasan.config. [Online]. Available: https://github.com/google/syzkaller/blob/master/dashboard/config/linux/stable-5.4-kasan.config

[22] ——. (2022) Hwaddresssanitizer. https://source.android.com/devices/tech/debug/hwasan. [Online]. Available: https://source.android.com/devices/tech/debug/hwasan

[23] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[24] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 749–763. [Online]. Available: https://doi.org/10.1145/3460120.3484564

[25] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 729–743. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[26] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2741–2758. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[27] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "HEALER: relation learning guided kernel fuzzing," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[28] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 147–161. [Online]. Available: https://doi.org/10.1145/3341301.3359662

[29] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[30] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: fuzzing linux drivers via automated virtual device simulation," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 404–416. [Online]. Available: https://doi.org/10.1145/3533767.3534226

[31] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 781–797. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[32] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng

[33] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "Syzscope: Revealing high-risk security impacts of fuzzer-exposed bugs in linux kernel," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3201–3217. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zou

[34] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Annual Network and Distributed System Security Symposium, (NDSS)*, 2017.

[35] P. Deligiannis, A. F. Donaldson, and Z. Rakamarić, "Fast and precise symbolic analysis of concurrency bugs in device drivers," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015.

[36] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2381–2398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/talebi

[37] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "Linkrid: Vetting imbalance reference counting in linux kernel with symbolic execution," in *31th USENIX Security Symposium, USENIX Security 2022*. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/liu-jian

[38] J. Bai, T. Li, K. Lu, and S. Hu, "Static detection of unsafe DMA accesses in device drivers," in *Proc. USENIX Security Symposium*, 2021.

[39] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1007–1024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[40] D. Gens, S. Schmitt, L. Davi, and A. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_05A-1_Gens_paper.pdf

[41] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015.

[42] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proc. ACM SOSP*, 2001.

[43] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten Years Later," in *Proc. ACM ASPLOS*, 2011.

[44] F. Brown, A. Nötzli, and D. Engler, "How to Build Static Checking Systems Using Orders of Magnitude Less Code," in *Proc. ACM ASPLOS*, 2016.

[45] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-scale Systems Code," in *Proc. ACM ASPLOS*, 2017.

[46] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger *et al.*, "Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel," in *29th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2022.

Appendix

```
#define pure_initcall(fn)           __define_initcall(fn, 0)
#define core_initcall(fn)           __define_initcall(fn, 1)
#define core_initcall_sync(fn)      __define_initcall(fn, 1s)
#define postcore_initcall(fn)       __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn)           __define_initcall(fn, 3)
#define arch_initcall_sync(fn)      __define_initcall(fn, 3s)
#define subsys_initcall(fn)         __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)    __define_initcall(fn, 4s)
#define fs_initcall(fn)             __define_initcall(fn, 5)
#define fs_initcall_sync(fn)        __define_initcall(fn, 5s)
#define rootfs_initcall(fn)     __define_initcall(fn, rootfs)
#define device_initcall(fn)         __define_initcall(fn, 6)
#define device_initcall_sync(fn)    __define_initcall(fn, 6s)
#define late_initcall(fn)           __define_initcall(fn, 7)
#define late_initcall_sync(fn)      __define_initcall(fn, 7s)
#define __initcall(fn) device_initcall(fn)
#define module_init(x) __initcall(x);
```

Figure 7: List of initcalls

```
int kobject_set_name_vargs(struct kobject *kobj,
     const char *fmt, va_list vargs);
int kobject_set_name(struct kobject *kobj, const
     char *fmt, ...);
int kobject_add_varg(struct kobject *kobj, struct
     kobject *parent, const char *fmt, va_list vargs
     );
int kobject_add(struct kobject *kobj, struct kobject
      *parent, const char *fmt, ...);
int kobject_init_and_add(struct kobject *kobj,
     struct kobj_type *ktype, struct kobject *parent
     , const char *fmt, ...);
int dev_set_name(struct device *dev, const char *fmt
     , ...);
struct device *device_create(struct class *class,
     struct device *parent, dev_t devt, void *
     drvdata, const char *fmt, ...);
struct device *device_create_with_groups(struct
     class *class, struct device *parent, dev_t devt
     , void *drvdata, const struct attribute_group
     **groups, const char *fmt, ...);
struct device * device_create_groups_vargs(struct
     class *class, struct device *parent, dev_t devt
     , void *drvdata, const struct attribute_group
     **groups, const char *fmt, va_list args);
int sprintf(char * buf, const char *fmt, ...);
```

Figure 8: Functions manipulating device file names

```
static const struct ioctl_handler {
  unsigned int cmd;
  int (*func)(struct snd_seq_client *client, void *
      arg);
} ioctl_handlers[] = {
{SNDRV_SEQ_IOCTL_PVERSION, snd_seq_ioctl_pversion},
...};
```

Figure 9: Example case of function array