DIFUZE: Interface Aware Fuzzing for Kernel Drivers

Jake Corina
UC Santa Barbara
jcorina@cs.ucsb.edu

Aravind Machiry
UC Santa Barbara
machiry@cs.ucsb.edu

Christopher Salls
UC Santa Barbara
salls@cs.ucsb.edu

Yan Shoshitaishvili
Arizona State University
Yan.Shoshitaishvili@asu.edu

Shuang Hao
University of Texas at Dallas
shao@utdallas.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Device drivers are an essential part in modern Unix-like systems to handle operations on physical devices, from hard disks and printers to digital cameras and Bluetooth speakers. The surge of new hardware, particularly on mobile devices, introduces an explosive growth of device drivers in system kernels. Many such drivers are provided by third-party developers, which are susceptible to security vulnerabilities and lack proper vetting. Unfortunately, the complex input data structures for device drivers render traditional analysis tools, such as fuzz testing, less effective, and so far, research on kernel driver security is comparatively sparse.

In this paper, we present DIFUZE, an interface-aware fuzzing tool to automatically generate valid inputs and trigger the execution of the kernel drivers. We leverage static analysis to compose correctly-structured input in the userspace to explore kernel drivers. DIFUZE is fully automatic, ranging from identifying driver handlers, to mapping to device file names, to constructing complex argument instances. We evaluate our approach on seven modern Android smartphones. The results show that DIFUZE can effectively identify kernel driver bugs, and reports 32 previously unknown vulnerabilities, including flaws that lead to arbitrary code execution.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Vulnerability scanners*;

KEYWORDS

Fuzzing, Kernel drivers, Interface aware

1 INTRODUCTION

Smartphones and other mobile devices occupy a central part of our modern lives. They are the last thing many of us interact with at night and the first thing we reach for in the morning. We use them to carry out financial transactions and to communicate with family, friends, and coworkers, and allow them to record location, audio, and video. Increasingly, they are used not just for personal and commercial purposes, but also to facilitate government activity.

The importance of the security of these devices is obvious. If an adversary compromises the device that has become our gateway to the connected world, he gains an enormous amount of power. Therefore, much effort has gone into ensuring the security of smartphones. This security is achieved using sophisticated application sandboxing, by leveraging many attack mitigation techniques targeting userspace applications (such as Address Space Layout Randomization, Data Execution Protection, and SELinux), and by making security a first-tier development goal. However, there is a weakness in the security of mobile devices: their kernels.

Unlike userspace applications, for which several vulnerability mitigation techniques are available and used, the kernels of modern operating systems are relatively vulnerable to attack despite available protections [43]. As a result, as vulnerabilities in userspace applications become rarer, attackers turn their focus on the kernel. For example, over the last three years, the share of Android vulnerabilities that are in kernel code increased from 4% (in 2014) to 39% (in 2016) [62], highlighting the need for techniques to detect and eliminate kernel bugs.

The kernel can itself be split into two types of code: core kernel code and device drivers. The former is accessed through the system call (*syscall*) interface, allowing a user to open files (the `open()` system call), execute programs (the `execve()` system call), and so on. The latter, on POSIX-compliant systems (such as Linux/Android and FreeBSD/iOS which cover over 98% of the mobile phone market), are typically accessed via the `ioctl` interface. This interface, implemented as a specific system call, allows for the dispatch of input to be processed by a device driver. According to Google, 85% of the bugs reported against the Android kernel (which is a close fork of Linux) are in driver code written by third-party device vendors [62]. With the continually growing number of mobile devices in use, and with the criticality of their security, automated approaches to identify vulnerabilities in device drivers before they can be exploited by attackers are critical.

While automatic analysis of the system call interface has been thoroughly explored by related work [28, 34], `ioctls` have been neglected. This is because, while interaction with syscalls follows a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134069

unified, well-defined specification, interaction with `ioctl`s varies depending on the device driver in question. Specifically, the `ioctl` interface comprises structured arguments for each of a set of valid commands, with both the commands and the data structures being driver-dependent. While this has security implications (i.e., pointers, dynamically-sized fields, unions, and sub-structures in these structures increase the chance of a vulnerability resulting from the mis-parsing of the structure), it also makes these devices hard to analyze. Any automated analysis of such devices must be *interface-aware*, in the sense that, to be effective, it must interact with `ioctl`s using the command identifiers and data structures expected by them.

In this paper, we present DIFUZE, a novel combination of techniques to enable *interface-aware fuzzing*, and facilitate the dynamic exploration of the `ioctl` interface provided by device drivers. DIFUZE performs an automated static analysis of kernel driver code to recover their specific `ioctl` interface, including the valid commands and associated data structures. It uses this recovered interface to generate inputs to `ioctl` calls, which can be dispatched to the kernel from userspace programs. These inputs match the commands and structures used by the driver, enabling efficient and deeper exploration of the `ioctl`s. The recovered interface allows the fuzzer to make meaningful choices when mutating the data: i.e., typed fields like pointers, enums, and integers should not be handled as simply a sequence of bytes. DIFUZE stresses assumptions made by the drivers in question and exposes serious security vulnerabilities. In our experiments, we analyzed seven modern mobile devices and found 36 vulnerabilities, of which 32 were previously unknown (4 vulnerabilities found by DIFUZE were patched during the course of our experiments), ranging in severity from flaws that crash the device in question causing Denial of Service (DoS) to bugs that can give the attacker complete control over the phone.

In summary, our paper makes the following contributions:

Interface-aware fuzzing. We design a novel approach to facilitate the fuzzing of interface-sensitive targets, such as the `ioctl` kernel driver interface on POSIX systems.

Automated driver analysis. We developed a fuzzing framework, that can automatically analyze the kernel sources of a device. For every driver the tool identifies all the ioctl entry points, as well as the corresponding structures, and device file names. We apply our technique to analyze seven devices, identifying 36 vulnerabilities. These vulnerabilities, ranging from DoS to code execution flaws, demonstrate the efficacy and impact of our approach. We are in the process of responsibly disclosing these vulnerabilities to the respective driver vendors.

DIFUZE prototype. We are releasing DIFUZE as an open-source tool at www.github.com/ucsb-seclab/difuze in the hope that it will be useful for future security researchers.

2 BACKGROUND AND RELATED WORK

In this section, we will explain the unique challenges that we must overcome (and why these challenges make existing state-of-the-art systems inapplicable to `ioctl` fuzzing), introduce the platform (Android) in which our fuzzing tool operates, and compare previous work on finding program vulnerabilities.

2.1 POSIX Device Drivers

The POSIX standard specifies an interface for the interaction of userspace applications with device drivers. This interface supports interaction with the device through *device files*, which are special files that represent the userspace presence of the kernel-resident device drivers. After a userspace application obtains a handle to the device file with the `open()` system call, there are multiple ways in which the application can interact with these files.

Different devices require different system calls to fulfill their functionalities. For example, `read()`, `write()`, and `seek()` are presumably applicable for a hard drive device file (showing the contents of the hard drive as, essentially, a single file). For an audio device, `read()` might read raw audio data from the microphone, and `write()` might write raw audio data to the speakers, and `seek()` might be unused.

However, some functionality cannot be implemented through traditional system calls. For example, for the audio device, how would a userspace application configure the sampling rate at which to record or play audio? Such out-of-band actions are supported by the POSIX standard through the `ioctl()` interface[1]. This system call allows drivers to expose functionality that is hard to model as a traditional file.

To support generality, the `ioctl()` interface can receive arbitrary driver-specified structures as input. It's C prototype looks like `int ioctl(int file_descriptor, int request, ...)`, where the first argument is the open file descriptor, the second argument is an integer commonly known as the *command identifier*, and the type and quantity of the remaining arguments are dependent on the driver and the command identifier.

Challenges. The aforementioned property makes `ioctl` system calls especially susceptible to vulnerabilities: First, unlike with `read()` and `write()`, the data provided to an `ioctl()` call are often instances of extremely complex, non standard, data structures. Parsing of such structures is not trivial, and any mistake could introduce critical vulnerabilities directly into the kernel context. Second, the generality of the data structure also makes the analysis of `ioctl()` interfaces difficult, as an analyst must have knowledge of how the driver in question processes different command identifiers, and what type of data it expects for the optional arguments.

These are the core problems that we aim to solve. We designed DIFUZE to automatically recover command identifiers and structure information, build the required complex data structures, and fuzz devices with `ioctl()` interfaces to find security vulnerabilities, with minimal human intervention.

2.2 Android Operating System

Android is designed as an operating system for smartphones. A recent report shows that Android has dominated the smartphone OS market, with an 86.8% share in 2016 Q3 [15]. Although Android designers take cautious steps to safeguard the devices, there are several vulnerabilities in smartphone systems [20]. Given the popularity and increasing security problems of Android, we choose

[1] In the original standard, this interface was only designed for certain types of devices, but this has changed in modern implementations.

Android systems as our main target platform to evaluate our analysis approach. Note that DIFUZE also works on other Unix-like systems.

Android is based on the Linux kernel, which has a monolithic architecture. Although kernel modules (such as device drivers) provide a certain level of modularity, the design principle is still monolithic, in the sense that the entire kernel runs in a single memory space, with all its parts being equally privileged [65]. Therefore, any vulnerability in a device driver could compromise the entire kernel. Indeed, in 2016 more than 80% of the bugs reported in the Android kernel were from driver code written by vendors [62]. The Android Open Source Project allows vendors (e.g., Sony, HTC) to customize Android kernel drivers to support new hardware, such as digital cameras, accelerometers, or GPS devices. Because security often takes a back seat to time-to-market for such companies, their development process is susceptible to the introduction of security vulnerabilities. Thankfully, the openness of the Android system makes the source code publicly available under the GNU General Public License [22]. This facilitates our approach, as it provides access to a high-level, semantically rich information about a driver.

2.3 Fuzz Testing

Fuzzing is a well-known technique for program testing by generating random data as input to the programs [45]. It has drawn much research attention, such as SPIKE [3], Valgrind [47], and PROTOS [55].

Fuzzing. The key prospect of fuzzing is to generate "mostly-valid" inputs to execute a target program, exercise a wide range of functionality, and trigger some corner case leading to a vulnerability. Dynamic taint tracking is a widely-used strategy to generate potential inputs. Dowser [30] and BuzzFuzz [21] use taint tracking to generate inputs that are more likely to trigger certain classes of vulnerabilities. However, for `ioctl` functions, which require highly constrained inputs, these techniques are less effective. Approaches based on taint analysis exist to recover the input format used by the underlying program [13, 40], but they cannot recover the cross-dependency between values, e.g., given a particular *command identifier* an `ioctl` handler will expect a further argument of a particular type.

Evolutionary techniques represent another common input generation strategy in fuzzing systems [19, 41, 69]. VUzzer [53], and SymFuzz [12] combine static analysis with mutation-based evolutionary techniques to efficiently generate inputs. However, these techniques are ineffective in generating highly constrained input. DIFUZE solves this problem by first collecting possible `ioctl` command values and then fuzzing only the unconstrained values with the expected input format.

If the input format of a program is known, fuzzing can be enhanced with a specification of the valid inputs. Peach [49] is one of the industry standard tools. However, it cannot generate *live data* (i.e., data containing active pointers to other data), and, as we show in Section 8, many device drivers require input structures that contain pointers. Grammar-based techniques have been used to fuzz file formats [29], interpreters [23, 31], and compilers [18, 37], but these techniques require inputs to have a fixed format.

Kernel and driver fuzzing. Fuzzing operating system interfaces or system calls is a practical approach to testing the operating system kernel [28, 34]. Most drivers use `ioctl` functions, a POSIX standard, to interact with userspace. As discussed in Section 3.1, `ioctls` are complex, and they require specific command values and data formats generated by users. Identifying valid command values and their associated data structures are the two key problems in `ioctl` fuzzing. Some tools have been developed to test `ioctl` interfaces for Windows kernels, such as iofuzz [17], ioattack [44], ioctlbf [67] and ioctlfuzzer [16]. However, these tools depend on the extensive logging and tracing of information provided by the Windows kernel, as well as the format of `ioctl` commands specific to Windows. Moreover, many of these tools are simplistic in nature. They involve simply attaching to processes and hooking the Windows `ioctl` call. Once hooked, the tool mutates the values when a call is made.This is lacking in several aspects e.g. the processes may not exercise the full capability of the drivers, and you cannot know the type information of the incoming data. To solve this problem, DIFUZE analyzes the source code of device drivers to identify valid commands and the corresponding data structure. The analysis techniques that we use require no modification to the actual device.

The extraction of valid `ioctl` commands was previously attempted by Stanislas, et al., but the state-of-the-art system was unable to scale to real-world kernel modules [38]. Conversely, as we show in Section 8, DIFUZE scales to (and finds vulnerabilities in) large kernel modules on real devices.

Trinity [34] and syzkaller [28] are specifically developed for Linux syscall fuzzing. As we show in Section 8, they perform badly when fuzzing `ioctl` handlers of device drivers. Although syzkaller uses additional instrumentation techniques, like Kernel Address Sanitizer [26], to detect more bugs, these techniques cannot be directly used on vendor devices, since they require the analyst to reflash the devices using custom firmware. Several approaches [8, 42, 58, 59, 64] concentrate on fuzzing specifically-chosen syscalls and drivers. However, they only focus on specific functions and cannot be generalized to other syscalls and drivers. DIFUZE is the first completely automated system that can be generalized to fuzz all Linux kernel drivers on a device running an unmodified kernel.

2.4 Other Analyses

Aside from fuzzing, there are two other analysis techniques, symbolic execution, and static analysis, that are related to our work. We will introduce these mechanisms and explain how they affect our design.

Symbolic execution. Symbolic execution is a technique that uses symbolic variables to generate constrained input and satisfy complex checks [10].

DART [24], SAGE [25], Fuzzgrind [11] and Driller [61] combine symbolic execution with random testing to increase the code coverage. BORG [48] uses symbolic execution to generate inputs more likely to trigger buffer overreads. Engineering issues of performing symbolic execution on the raw devices and the fundamental path explosion problem (made all the worse by complex system kernels) render these techniques impractical for kernel drivers.

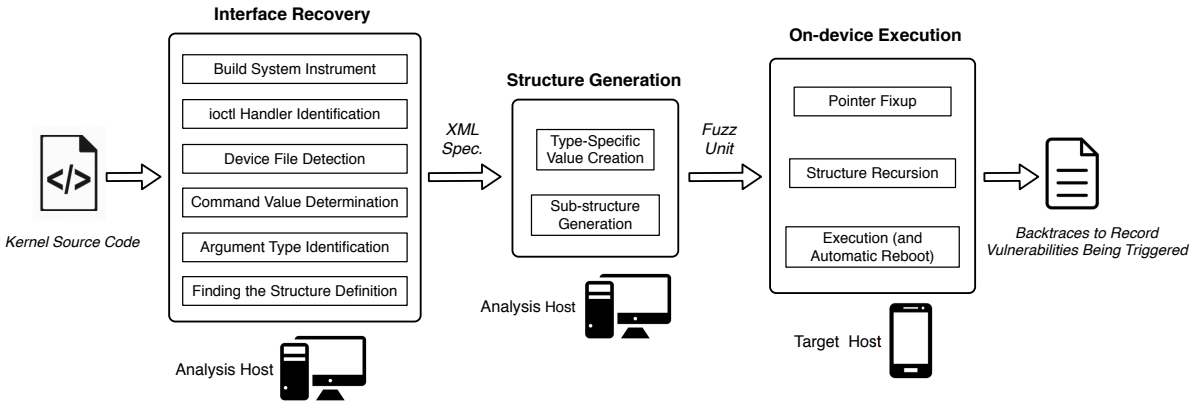

Figure 1: The DIFUZE approach diagram. DIFUZE analyzes the provided kernel sources using a composition of analyses to extract driver interface information, such as valid ioctl commands and argument structure types. It synthesizes instances of these structures and dispatches them to the target device, which triggers ioctl execution with the given inputs and, eventually, finds crashes in the device drivers.

Static analysis. Static analysis is a popular technique to find program vulnerabilities without executing the program in question [2]. To maximize precision, these techniques typically require source code to perform the analysis. Since many system kernels (including the Linux kernel) and device drivers are open-source, kernel security can greatly benefit from static analysis [7]. For example, Ashcraft, et al. developed compiler extensions to catch integers read from untrusted sources in Linux and OpenBSD kernels [5]. Post, et al. used a bounded model checker to find deadlocks and memory leaks in Linux kernels [50]. Ball, et al. built a static analysis tool with a set of rules to prove the correctness of Windows drivers [6].

One limitation of most static analysis tools is the production of many false positives. Since our work leverages fuzzing for the vulnerability detection step, all identified vulnerabilities are actual bugs, and false positives are entirely avoided. Another drawback of static analysis techniques is that the analysis often needs a manual specification of security policies and rules.

3 OVERVIEW

In this section, we will provide an overview of our interface-aware fuzzing approach and its application to vulnerability detection in device drivers through `ioctl` fuzzing. We will also present an example that will be referenced throughout the paper to assist the curious reader in understanding our end-to-end system.

Figure 1 demonstrates the high-level workflow of the system. DIFUZE requires, as input, the source code of the kernel (which will include the source code of the device drivers) of the target host. Since Linux is licensed under the GNU General Public License, any software that is linked against it, such as the kernel-driver interface code, must also be released. Thus, the kernel sources of Android devices are readily available [27, 32, 33, 39, 46, 57, 60, 66] and can be used for our analysis.

Given this input, DIFUZE works through a number of phases to recover the interaction interface for device drivers, generate the correct structures to exercise this interface, and trigger the processing of these structures by the kernel of the target host. Because the triggering of kernel bugs often renders a system unstable (leading to a hang or reboot), only DIFUZE's final stage is done *in vivo* on the target host. The other stages are executed on an external *analysis host*, their results are logged locally (for input replay, in case a bug is triggered), and then transferred over a network connection or debug interface to the target host.

In more detail, these stages are:

Interface recovery. In its first stage, DIFUZE analyzes the provided sources to detect what drivers are enabled on the target host, what device files are used to interact with them, what `ioctl` commands they can receive, and what structures they expect to be passed to these commands. This series of analyses are implemented using LLVM, and are further described in Section 4. The end result of this stage is a set of tuples of the device filename for the target driver, the target `ioctl` command, and structure type definitions.

Structure generation. For each structure, DIFUZE continuously generates *structure instances*: memory contents representing instantiations of the type information recovered from the previous step. These instances are logged and transferred to the target host, along with the associated target device filenames and target `ioctl` command identifiers. This stage is detailed in Section 5.

On-device execution. The actual `ioctl` triggering component resides on the target host itself. Upon receipt of the target device filename, the target `ioctl` command, and the generated structure instances, the executor proceeds to trigger the execution of `ioctls`. We discuss this stage in Section 6.

DIFUZE logs the sequence of inputs that is sent to the target. Thus, when a bug is triggered, and the target device crashes, the inputs can be used for reproducibility and manual triage/analysis.

3.1 Example

To help the reader understand DIFUZE, we provide an example of a simple driver. This example is presented in Listing 1 (the structure definitions), 2 (a wrapper around the `copy_from_user` function, which presents minor complications to the analysis), Listing 4 (the main driver initialization code), and Listing 3 (the `ioctl` handlers themselves).

The function `driver_init` in Listing 4 is the driver initialization function, which will be called as part of kernel initialization. This function registers the device with a name "`example_device`" (line 8) and specifies that the function `ioctl_handler` should be invoked when a userspace application performs the `ioctl` system call (lines 10 and 11) on the device file (in this case, `/dev/example_device`). Although the filename is `example_device`, the absolute path of the file depends on the type of device. The device in the running example is a character device [35] and it will be created under the `/dev` directory, However, there are other types of device files, which will be created in different directories. For instance, proc devices [54] will be created under the `/proc` directory.

We will refer to this example throughout the rest of the paper as the "running example".

```
1  typedef struct {
2    long sub_id;
3    char sub_name[32];
4  } DriverSubstructTwo;
5
6  typedef union {
7    long meta_id;
8    DriverSubstructTwo n_data;
9  } DriverStructTwo;
10
11 typedef struct {
12   int idx;
13   uint8_t subtype;
14   DriverStructTwo *subdata;
15 } DriverStructOne;
```

Listing 1: The structure definitions of our running example. DIFUZE automatically recovers these and performs *structure-aware* fuzzing of the target driver.

```
1  int copy_from_user_wrapper(void *buff, void *userp, size_t size) {
2      // copy size bytes from address provided by the user (userp)
3      return copy_from_user(buff, userp, size);
4  }
```

Listing 2: Like many real-world drivers, our example driver ships with a wrapped copy_from_user function. Because of wrappers like this (and more complex ones), DIFUZE must support the analysis of nested functions.

```
1  DriverStructTwo dev_data1[16];
2  DriverStructTwo dev_data2[16];
3  static bool enable_short; static bool subhandler_enabled;
4
5  long ioctl_handler(struct file *file, int cmd, long arg) {
6    uint32_t curr_idx;
7    uint8_t short_idx; void *argp = (void*) arg;
8    DriverStructTwo *target_dev = NULL;
9    switch (cmd) {
10     case 0x1003:
11       target_dev = dev_data2;
12     case 0x1002:
13       if(!target_dev)
14         target_dev = dev_data1; // program continues to execute
15       if(!enable_short) {
16         if (copy_from_user_wrapper((void*)&curr_idx, argp,
17                  sizeof(curr_idx))) {
18           return -ENOMEM; // failed to copy from user
19         }
20       } else {
21         if (copy_from_user_wrapper((void*)&short_idx, argp,
22                  sizeof(short_idx))) {
23           return -ENOMEM; // failed to copy from user
24         }
25         curr_idx = short_idx;
26       }
27       if(curr_idx < 16)
28         return process_data(&(target_dev[curr_idx]));
29       return -EINVAL;
30     default:
31       if(subhandler_enabled)
32         return ioctl_subhandler(file, cmd, argp);
33   }
34   return -EINVAL;
35 }
36
37 long ioctl_subhandler(struct file *file, int cmd, void *argp) {
38   DriverStructOne drv_data = {0};
39   DriverStructTwo *target_dev;
40   if(cmd == 0x1001) {
41     if(copy_from_user_wrapper((void*)&drv_data, argp,
42              sizeof(drv_data))) {
43       return -ENOMEM; // failed to copy from user
44     }
45     target_dev = dev_data1;
46     if(drv_data.subtype & 1)
47       target_dev = dev_data2;
48     // Arbitrary heap write if drv_data.idx > 16
49     if(copy_from_user_wrapper((void*)&(target_dev[drv_data.idx]),
50                               drv_data.subdata,
51                               sizeof(DriverStructTwo))) {
52       return -ENOMEM; // failed to copy from user
53     }
54     return 0;
55   }
56   return -EINVAL;
57 }
```

Listing 3: The ioctl handlers which expect very specific values for the command identifiers and expect data to be presented in the proper structure for each command. The ioctl processing is split across multiple functions.

```
1  static struct cdev driver_devc;
2  static dev_t client_devt;
3  static struct file_operations driver_ops;
4  __init int driver_init(void)
5  {
6    // request minor number
7    alloc_chrdev_region(&driver_devt, 0, 1, "example_device");
8    // set the ioctl handler for this device
9    driver_ops.unlocked_ioctl = ioctl_handler;
10   cdev_init(&driver_devc, &driver_ops);
11   // register the corresponding device.
12   cdev_add(&driver_devc, MKDEV(MAJOR(driver_devt), 0), 1);
13 }
```

Listing 4: The main driver initialization function of our running example. It dynamically creates the driver file, the name of which must then be recovered by DIFUZE, and registers the top-level ioctl handler, which must also be recovered.

4 INTERFACE RECOVERY

To efficiently fuzz the `ioctl`s of a device driver, DIFUZE needs to recover the *interface* of that driver. The interface of a device driver is comprised of the name/path of the device file used to communicate with the device, the valid values for `ioctl` commands for that device, and the structure definition of the `ioctl` data argument for the different `ioctl` commands.

To recover this data, DIFUZE uses a combination of analyses, implemented in LLVM. As the Linux kernel does not lend itself to analysis (or even compilation) with LLVM, we first developed an alternate build procedure. After this is done, we identify the filename of the device files created by the device driver, find the `ioctl` handler, recover the valid set of `ioctl` command identifiers, and retrieve the structure definitions for the data arguments to those `ioctl` commands.

4.1 Build System Instrumentation

We take several steps to enable DIFUZE to perform LLVM analyses on Linux device drivers.

GCC compilation. First, we perform the manual step of setting up the kernel and driver sources of the target host for compilation, using GCC. While this is generally a well-documented process, the vendors of mobile devices do not go out of their way to make their GPL-mandated source code releases easy to compile, so some manual configuration effort is required. Once the source tree can be compiled with GCC, we run a full compilation and log all executed commands.

GCC-to-LLVM conversion. We process the log of executed commands during the compilation step with a GCC-to-LLVM command conversion utility that we created for DIFUZE. This utility translates command-line flags from the format expected by GCC to the format expected by LLVM utilities and enables the compilation of the kernel source via LLVM. In its compilation, LLVM generates a bitcode file [51] for each source file. We enable debug information to be embedded in the bitcode file, which helps us in extracting the structure definitions as explained in Section 4.6

Bitcode consolidation. The analyses that DIFUZE undertakes operate on each driver separately. As such, we consolidate the various bitcode files to create a single bitcode file per driver. This allows us to carry out interface recovery analyses on a single bitcode file, simplifying the analyses. This consolidated bitcode file is used in the following phases to perform the analyses.

4.2 ioctl Handler Identification

As discussed in Section 2.1, much of the interaction with device drivers happens through the `ioctl` interface. From userspace, the application calls the `ioctl()` system call, passing in a file descriptor to the driver's device file, a command identifier, and the required structured data argument. When this system call is received in kernel space, the corresponding driver's *ioctl handler* is invoked. This handler then dispatches the request to different functionality inside the driver, depending on the command identifier. In the case of our running example, the `ioctl` handler function is `ioctl_handler`.

In order to recover valid command identifiers and the structure definitions of additional `ioctl` arguments, DIFUZE must first identify the top-level `ioctl` handler. Each driver can register a top-level `ioctl` handler for each of its device files, and there are several ways to do this in the Linux kernel. All of these methods, however, involve the creation of one of a set of structures[2] created for this purpose, with one of the fields of these structures being a function pointer to the `ioctl` handler. A full list of these structures, and corresponding field names for one of the kernels are listed in Appendix A.

Our analysis to identify the `ioctl` handler is straightforward: using LLVM's analysis capabilities, we find all uses of any of these structures in the driver and recover the value of the assignment of the `ioctl` handler function pointer. In the case of our running example, we identify the write to the `unlocked_ioctl` field of a `file_operations` structure (Listing 4, line 9). We can then consider the function `ioctl_handler` as an `ioctl` handler.

4.3 Device File Detection

To determine the device file corresponding to an `ioctl` handler, we need to identify the name provided in the registration of the `ioctl` handler (for example, in our running example, the device file would be `/dev/example_device`, from line 7 of Listing 4).

Depending on the type of device, there are several ways to register the file name in the Linux kernel [14, 56]. For example, the registration of a character device [35] will use the method `alloc_chrdev_region` to associate a name with the device. For proc devices, the method `proc_create` is used to provide the filename. Furthermore, as mentioned in Section 3.1, depending on the device type, the directory in which the device file is found may vary.

Given an `ioctl` handler, we use the following procedure to identify the corresponding device name.

(1) First, we search for any LLVM `store` instruction that is storing the address into one of the fields of any operations structures listed in Appendix A.
(2) We then check for any reference to the operations structure in any of the registration functions [56].
(3) We analyze the argument value for the device filename and return it if it is a constant.

[2]There are at least 72 variations of these structures.

In case of the running example, Listing 4, we previously determine that the ioctl handler function is ioctl_handler. We identify that ioctl_handler is stored in the file_operations structure (i.e., driver_ops) at line 9 (Step 1), then check for the usage of driver_ops, as parameter for the function cdev_init at line 10 (Step 2). The function cdev_add implies that the device is a character device. We backtrack to the allocation function for the device metadata (alloc_chrdrv_region) at line 7, whose third argument is the device name, detect it as a constant string, and return /dev/example_device as the device name.

```
1 VOS_INT __init RNIC_InitNetCard(VOS_VOID) {
2   ...
3   snprintf(pstDev->name, sizeof(pstDev->name),
4            "%s%s",
5            RNIC_DEV_NAME_PREFIX,
6            g_astRnicManageTbl[ucIndex].pucRnicNetCardName);
7   ...
8 }
```

Listing 5: Dynamically generated device name in RNIC driver on Huawei Honor phone. DIFUZE fails to find the device name for this driver.

A driver could use dynamically created filenames, as shown in Listing 5. Unfortunately, with the limitations inherent to static analysis, we miss such filenames and must fallback to manual analysis (of course, if we wish to remain fully automated we can simply ignore these devices).

Next, we proceed on to identifying valid command identifiers accepted by a given ioctl handler.

4.4 Command Value Determination

Given the ioctl handler, we perform a static inter-procedural, path-sensitive analysis to collect all the equality constraints on the cmd value (i.e., the second argument of the ioctl()). We then use Range Analysis [52] to recover the possible values for the comparison operand. In the case of the ioctl example shown in Listing 3, we collect the following constraints: cmd == 0x1003 (line 10), cmd == 0x1002 (line 12) and cmd == 0x1001 (line 32 → Line 41). As the comparison operands are constants, running Range Analysis on them results in constants: 0x1003, 0x1002 and 0x1001 respectively.

We consider only equality constraints on the cmd value. Based on our observation that almost all the drivers use equality comparison to check for the valid command IDs. There exists special ioctl functions, such as V4L2 drivers, in which the driver specific functions are called in a nested manner by other drivers. We expand our solution for these cases in Appendix B.

4.5 Argument Type Identification

The ioctl command identifiers and the corresponding data structure definitions have a many-to-many relationship: each ioctl command may take several different structures (for example, based on global configuration), and each command structure may be passed to multiple ioctl commands. To find these structures, we first identify all the paths to the copy_from_user function, which the Linux kernel uses to copy data from userspace to kernel space, such as line 16 in Listing 3 → line 3 in Listing 2. We ignore call-sites whose source operand (i.e., the second argument of copy_from_user) is not the passed argument to the ioctl function, since such case cannot help us to determine the ioctl argument type. At each of the remaining call-sites, we find the type of the source operand. This is the type definition to which the user data argument to the ioctl handler must conform.

Note that pointer casting could hide the actual structure type. Consider the running example, where the copy_from_user in line 3 of Listing 2 is reachable from the ioctl handler, ioctl_handler in Listing 3 from multiple paths (like line 16, line 21, and line 32 → line 41). However, the actual type of the source operand at the call-site is void *. In addition, the copy_from_user function might reside in a wrapper function and be called indirectly by the ioctl function (such as line 16 in Listing 3 → line 3 in Listing 2), which is distributed across different functions or files.

To handle this, we perform inter-procedural, path-sensitive type propagation to determine all the possible types that may be assigned to the source operand of a copy_to_user function in each path. This gives us the set of possible types, for each given path, of the user data argument to the ioctl handler.

To associate the command identifier to each of these structure types, we also collect the equality constraints (as explained in Section 4.4) along the path while performing the type propagation. The constraints on the command value on a path reaching a copy_from_user function represent the possible command identifiers associated with the structure type.

For the running example in Listing 3, we first identify all paths reaching a copy_from_user call-site (Note that the actual call happens through the wrapper function copy_from_user_wrapper). Table 1, column 2 shows all the relevant paths. For brevity, we ignored the paths that have the same constraints on cmd and reach the same call-site.

We also ignore Path 6 since the source operand is not the user argument (i.e., at line 49 in Listing 3, the second argument of copy_from_user_wrapper is not argp). Finally, for the remaining paths, we identify the type of the destination operand of the target copy_from_user call-site to determine the command value type. For example, for Path 1 in Table 1, the type of argp is the same as the destination operand curr_idx at line 16 in 3, which is defined as uint32_t at line 6. For each command value, we may get multiple types. For instance, as shown in Table 1, Path 1 and Path 2 have the same cmd constraint values but different argument types. For each command value, we associate all the possible argument types. For example, from Table 1, the command value 0x1003 can be associated with argument types uint32_t and uint8_t. Next, we need to extract the arguments' structure definitions.

4.6 Finding the Structure Definition

Finding the definition of a type requires finding the definition of all the types it is composed of. In the case of our running example, in Listing 1, extracting the definition of type DriverStructOne requires extracting the definitions of both DriverStructTwo and DriverSubstructTwo.

For each of the types identified in Section 4.5, we find the source file name of the corresponding copy_from_user function using the debug information computed in Section 4.1. Knowing the source file,

Table 1: Relevant paths from ioctl handler (of Listing 3) to a `copy_from_user` call-site

Id	Path	cmd constraints	Resolved command id	User argument type
1	Line 10 → Line 11 → Line 16 → Line 3 (of Listing 2)	`cmd == 0x1003`	0x1003	`uint32_t`
2	Line 10 → Line 11 → Line 21 → Line 3 (of Listing 2)	`cmd == 0x1003`	0x1003	`uint8_t`
3	Line 12 → Line 16 → Line 3 (of Listing 2)	`cmd == 0x1002`	0x1002	`uint32_t`
4	Line 12 → Line 21 → Line 3 (of Listing 2)	`cmd == 0x1002`	0x1002	`uint8_t`
5	Line 30 → Line 32 → Line 41 → Line 3 (of Listing 2)	`cmd == 0x1001`	0x1001	`DriverStructOne`
6	Line 30 → Line 32 → Line 49 → Line 3 (of Listing 2)	`cmd == 0x1001`	0x1001	**N/A**

we use our GCC-to-LLVM pipeline to generate the corresponding preprocessed file. As preprocessed files should contain a definition of all the required types, we find the definition of the identified type. Then we run `c2xml` [63] tool to parse the C struct definition into XML format from which the required definition of the types is extracted.

5 STRUCTURE GENERATION

After DIFUZE recovers the `ioctl` interface, it can begin generating instances of structures to pass to the on-device execution engine. The procedure for this is straightforward: DIFUZE instantiates structures, fills their fields with random data, and properly sets pointers to build complex inputs to `ioctls`.

Type-Specific Value Creation: Certain values are more likely to trigger increased code coverage than others. For example, buffer lengths in system code are often aligned to bit boundaries (i.e., buffers of size 128, 256, and so on), so values on or just under a bit boundary are more likely to trigger corner cases (such as single-byte overwrites due to careless string termination). This observation is common wisdom in the fuzzing community, and previous work has widely used it [68]. DIFUZE leverages this concept as well, and favors (but does not confine itself to) integers that are a power of two, one less than a power of two, or one greater than a power of two in its generated integers.

There are some pointers that reference data that is either unstructured (`char *` pointers, for example), or for which the structure definition can't be recovered (`void *` data). For this data, DIFUZE allocates a page of random content.

Sub-structure Generation: Inputs to `ioctls` often take the form of nested structures, where a top-level structure contains pointers to other structures. DIFUZE generates these structure instances individually and sends them to the on-device execution component. This component, in the next stage, merges them into a nested structure before passing them to the `ioctl` itself.

6 ON-DEVICE EXECUTION

While prior stages of DIFUZE run on the analysis host, the actual execution of `ioctls` must happen on the target host. As such, the structure generation component sends the generated structures, along with the target device driver filename and `ioctl` command identifier, to the on-device execution component. This component then finalizes these structures and triggers the `ioctl`.

6.1 Pointer Fixup

Some structures comprise multiple memory regions connected by pointers. To save space, the structure generation component transmits the different memory region instances independently, along with metadata about how they can be combined, and the on-device execution component builds the complete structure using this data. This preserves bandwidth between the analysis host and target host, since the same data can be used for differently built structures. For example, since the individual nodes of a tree structure will be sent individually, these nodes can be used to create many different final configurations of the tree structure.

Some structures are recursive. For example, a linked list node may contain a pointer to the next linked list node. To set a bound on the number of combinations of structures that the on-device execution component attempts to create, DIFUZE limits the recursion of such structures to a set threshold.

6.2 Execution

With the structure created, DIFUZE's on-device execution component opens the appropriate device file and triggers the `ioctl` system call with the `ioctl` command identifier and the proper data structure. At this point, DIFUZE watches for any bug in the kernel, which crashes the target device. This is done by maintaining a heartbeat signal between the analysis host and the target host. When DIFUZE finds a bug, it logs the series of inputs that had been sent to the host device for later reproduction and triage.

System restart. When a bug is triggered, the target host will either be in an inconsistent state or will have crashed. In the former case, the on-device execution component triggers a reboot of the device before resuming fuzzing on other `ioctl` commands and other drivers. In the latter case, depending on the way the crash occurred, the device sometimes restarts itself. When that happens, DIFUZE can resume without analyst interaction. Otherwise, an analyst will need to reboot the device before fuzzing can resume.

7 IMPLEMENTATION

As shown in Figure 1, we engineered our system to be completely automated. The user simply provides the compilable kernel source archive, connects the target host (i.e., the mobile phone) to the analysis host, and starts the on-device execution component on the target host. After that, with a single command, our entire pipeline will be run.

Interface Extraction: We used LLVM 3.8 to implement the interface extraction techniques. All components of the interface

extraction are implemented as individual LLVM passes. As mentioned in Section 4.4, We used an existing implementation of Range Analysis [52] to recover valid command identifiers.

7.1 Interface-Aware Fuzzing

Our implementation of Sections 5 and 6 is called MangoFuzz. MangoFuzz is the combination of structure generation on the analysis host and on-device execution of `ioctls`, which together achieve interface-aware fuzzing. It is an intentionally simple prototype designed to test the effectiveness of interface-aware fuzzing, without other optimizations that could influence the results.

MangoFuzz specifically targets `ioctl` system calls on real Android devices. Using the methods described in Section 5, it generates random sequences of `ioctl` calls, along with associated structures, and sends them to the on-device execution component running on the target host.

For a "production-ready" variant of our approach, we also integrated DIFUZE into syzkaller, a state-of-the-art Linux system call fuzzer. This integration has the goal of creating the best possible tool, which we will contribute back to the community as an open-source enhancement of syzkaller.

Syzkaller is a Linux system call fuzzer, which allows analysts to (manually) provide system call descriptions, after which it will fuzz the associated system call. Syzkaller can handle structures as system call arguments if corresponding formats are manually specified. To integrate with DIFUZE, we automatically convert the results of our Interface Recovery step to the format expected by syzkaller, making it interface-aware. Syzkaller is typically used on kernels compiled with coverage information and KASAN (or another memory access sanitizer). However, there is a configuration for running on real, unmodified Android devices, which can be used for our purposes.

Table 2: Android Phones Used in Evaluation (Note that the kernel versions were the latest Android kernels for each phone at the time of our experiment)

Device	Vendor	Chipset Vendor	Android Kernel Version
Pixel	Google	Qualcomm	3.18
E9 Plus	HTC	Mediatek	3.10
M9	HTC	Qualcomm	3.10
P9 Lite	Huawei	Huawei	3.10
Honor 8	Huawei	Huawei	4.1
Galaxy S6	Samsung	Samsung	3.10
Xperia XA	Sony	Mediatek	3.18

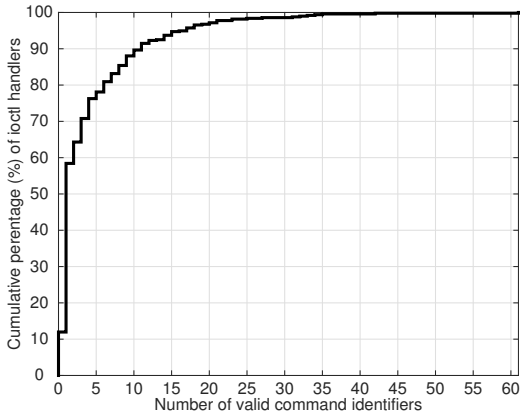

Figure 2: CDF of percentage of ioctl handlers to the number of valid command identifiers

8 EVALUATION

To determine the effectiveness of DIFUZE we evaluate both its interface recovery and bug-finding capabilities. The evaluation is performed on seven different Android phones from five of the most popular vendors, covering a wide range of device drivers. Table 2 shows the specific phones along with the vendor of the chipsets.

First, we evaluate the effectiveness and efficiency of the interface recovery, as it is the core component of the system. To validate the results, we manually analyze a random sampling of `ioctls` and structures and check them against our system's recovered interfaces. We then perform a comparative evaluation of the bug finding capabilities of DIFUZE, using both MangoFuzz and our improvements to syzkaller as the fuzzing component.

8.1 Interface Extraction Evaluation

All the steps of interface extraction are run on the same experiment platform, a machine with an Intel Xeon CPU E5-2690 (3.00 GHz) running Ubuntu 16.04.2 LTS. On average, it took 55.74 minutes to complete the entire interface extraction phase for a kernel.

We evaluate the effectiveness of different steps of our interface extraction on the kernels of the devices listed in Table 2. Table 3 shows the interface extraction results on different kernels. DIFUZE identified a total of 789 `ioctl` handlers in the kernels of seven devices. The number of handlers also closely correspond to the number of drivers on the corresponding phone.

Device Name Identification: Our approach for device name identification (Section 4.3) is able to work on different vendor-specific devices. DIFUZE can automatically identify 469 device names, accounting for 59.44% of the `ioctl` handlers. Most of the identification failures come from kernel mainline drivers. For example, our name recovery on only vendor drivers of the Xperia XA was able to recover more than 90% of the names. The reason for this discrepancy is that mainline drivers tend to use dynamically generated names (Listing 5 and Section 4.3) whereas vendor drivers tend to use static names. We manually extracted those dynamically created device names.

Valid Command Identifiers: The fourth column of Table 3 shows the number of valid command identifiers extracted across all the entry points of the corresponding kernels. In total, DIFUZE found 3,565 valid command identifiers across all the drivers of all kernels. The numbers of valid command identifiers vary considerably across different kernels. As we will show in Tables 3 and 5, the number of crashes the fuzzer found is positively correlated with the number of valid command identifiers.

Figure 2 shows the distribution of the number of valid command identifiers per `ioctl` handler. 11% of the `ioctl` handlers do not expect any command. The code of these `ioctls` is conditionally

Table 3: Interfaces recovered by DIFUZE on different kernels of the Phones.

	ioctl handlers	Device Names Automatically Identified	Valid Command Identifiers	User Argument Types			
				no copy_from_user	Scalar	Struct	Struct with pointers
Pixel	193	136	611	270	87	151	103
E9 Plus	77	36	610	272	101	195	42
M9	171	122	563	216	83	149	115
P9 Lite	71	30	384	187	56	118	23
Honor 8	86	33	376	208	70	87	11
Galaxy S6	106	70	364	243	23	67	31
Xperia XA	85	42	657	292	106	194	65
Total	789	469	**3,565**	1,688	526	961	390

compiled and guarded by kernel configurations. During our compilation, the `ioctl` handler code is disabled, so the corresponding `ioctl` handler appears empty in the generated bitcode file, which leads to zero command value in our command identification process. There are 50% of the `ioctl` handlers that expect a single command identifier. Most of them are attributed to the `v4l2_ioctl_ops`. As explained in Appendix B, these are nested handlers that manage a (single) specific command. The majority (98.3%) of the `ioctl` handlers have less than 20 valid command identifiers. We manually investigate the rest (1.7%) of the `ioctls` with more than 20 command identifiers, and find that our approach over approximates the function pointers for some of the `ioctl` functions. Although such over estimation causes extra invalid fuzz units in our subsequent fuzzing steps, it has marginal impact on the overall performance (especially given that we have a small percentage of such cases).

User argument types: The last four columns in Table 3 show how an argument passed by the user (third argument to the `ioctl` handler) is treated.

For 1,688 (47%) of command identifiers, we do not find any `copy_from_user`. This places us in one of two categories. (1) the user argument is treated as C type long, and thus argument type identification is not needed since the user argument is treated as a raw value (and hence no `copy_from_user` is present). (2) Or, there is instead a `copy_to_user`, where the user is meant to supply a pointer to some type for which the kernel will copy information to the user. We do not care about type identification here either, as the kernel will not be processing the user data.

For the rest 1,877 (53%) of the command identifiers, the user argument is expected to be a pointer to a specific data type. i.e., a `copy_from_user` call should be used to copy the data. Such pointer arguments can be further categorized as the following three cases.

(i) 526 (15%) of the command identifiers expect a scalar pointer argument, For example, in our running case, as shown in Table 1, command IDs 0x1003 and 0x1002 belongs to this category since they expect the user argument pointing to scalar types `uint32_t` or `uint8_t`. (ii) 961 (30%) of the command identifiers expect the user argument to point to a C structure with no embedded pointers. e.g., `DriverStructTwo` in Listing 1. (iii) For 390 (11%) of the command identifiers, the date type is a C structure which contains embedded pointers. In the case of our running example, as shown in Table 1, command ID 0x1001 belong to this category and expects the user argument to point to `DriverStructOne`, which contains embedded pointers (Listing 1). These commands are extremely hard to effectively fuzz without the argument type information, because the user argument is expected to point to a structure, which itself contains pointers (which should also be valid pointers).

Random Sampling Verification: We picked a random sample of five `ioctls` for each of the seven Android phones in our test set and manually verified that the extracted types were correct. These 35 `ioctls` had a total of 327 commands, of which we correctly identified the argument and commands for 294 of them, yielding a 90% accuracy.

8.2 Evaluation Setup

To determine how well DIFUZE can find actual bugs in device drivers, and the effects of using the extracted interface information on this ability, we test it both using our prototype fuzzer, MangoFuzz, and using syzkaller. We will use the identifiers DIFUZE^{m} and DIFUZE^{s} to represent DIFUZE when it is using MangoFuzz and syzkaller, respectively, as the fuzzing and on-device execution component. Additionally, we evaluate the system by varying the amount of the interface that we provide to syzkaller; this way, we can examine how different levels of interface extraction influence the results. Specifically, we run the following configurations of DIFUZE:

Syzkaller base. We specify that syzkaller should only fuzz using the system calls `open()` (to open the device files) and `ioctl` (to trigger `ioctl` handlers). Its default configuration contains several standard device filenames and the structures of some standard types along with `ioctls` for common Linux devices.

Syzkaller+path. In this configuration, we add the specifications of extracted driver paths which syzkaller should try to fuzz. However, the rest of the interface information is not provided.

DIFUZE^{i}. Here, the extracted interface information of the device paths and `ioctls` is used with syzkaller as the fuzzer. We expect that this configuration would be able to trigger `ioctl` command handling, but will be unable to explore code that handles complex structures.

DIFUZE^{s}. This configuration integrates all of the interface recovery, including automatic identification of `ioctl` argument structure formats, with syzkaller. We expect this to be the best-performing configuration, as it is able to leverage many of the optimizations found in syzkaller.

DIFUZE^{m}. The final configuration integrates our interface recovery with our simple fuzzer prototype, MangoFuzz. This configuration is meant to explore the effect that interface-aware fuzzing has on the number of discovered bugs, even when other state-of-the-art optimizations are absent.

We evaluated the system on seven modern Android devices, including the current "flagship" model of Google, and other popular phones from Samsung Sony, and HTC. For each device, we first updated it to the latest available version and then rooted the device. The on-device execution component is run as root to ensure that we can fuzz all drivers, and not just those accessible from app-level permissions. However, as discussed in Section 9, this component could also take the form of a standard application, though this would come at the cost of lower accessibility to device files (and their `ioctl` handlers). With this setup, we do not have code coverage feedback or KASAN enabled, as this would require re-compiling the kernels and flashing a non-stock kernel. More discussion on these compile-time instrumentations can be found in Section 9. Every one of the aforementioned DIFUZE configurations is run on each Android device for five hours. If a crash occurs frequently in a single driver, we blacklist the buggy `ioctl` handler to prevent the phone from repeatedly crashing and the resulting reboots interfering with the experiment.

8.3 Results

We collected all crash logs and crashing sequences of system calls, manually triaged them, and filtered out the small number of duplicates. In total, DIFUZE was able to find 36 unique bugs in the seven Android devices that were used for testing. An overview of the found bugs is shown in Table 5.

We were unable to get syzkaller to work on the Galaxy S6 and DIFUZE^{m} was unable to trigger any bugs on it, making it the only Android device for which we found zero bugs. On all the other devices, we found anywhere from two vulnerabilities (in the Honor 8) to fourteen vulnerabilities in the Xperia XA.

The base configuration of syzkaller (without interface information) was unable to find any bugs in our tests. Giving it the correct paths of drivers (syzkaller+path) only yielded three crashes across all devices. This suggests that blindly fuzzing kernel drivers is not very effective, which is likely because such testing is undertaken by the vendor before these devices are shipped.

When we add partial interface information in the form of the extracted `ioctl` numbers, DIFUZE^{i} is able to find 22 bugs. Although this is impressive on its own, adding the remaining interface information (the `ioctl` argument structure definitions) to the interface substantially increased the number of bugs found by 54.5%, to a total of 34 bugs. This result shows the effectiveness of interface-aware fuzzing and, moreover, shows the importance of both the recovered `ioctl` command identifiers and the structure information to the analysis of `ioctl` handlers.

A particularly interesting result from our experiments is that DIFUZE^{m} only found four fewer bugs than DIFUZE^{s}. Syzkaller is a state-of-the-art tool with a large number of fuzzing strategies and optimizations built in, while MangoFuzz is a simple fuzzing prototype with no optimizations except those described in Section 6. We believe this shows that fuzzing with accurate interface information is quite powerful.

Table 4: Types of Crashes Found by DIFUZE

Crash Type	Count
Arbitrary read	4
Arbitrary write	4
Assert Failure	6
Buffer Overflow	2
Null dereference	9
Out of bounds index	5
Uncategorized	5
Writing to non-volatile memory	1

We briefly triaged each of the crashes and quickly classified the reason that the device crashes. These results are shown in Table 4. These are often serious bugs even when the crash itself might seem benign. For example, an assertion error could be triggered by a more serious underlying bug that a malicious user could carefully craft to gain a more powerful primitive. Adding to this, one of the more interesting bugs discovered was that we could bypass most of the asserts encountered. The ability to bypass these checks allowed for many would-be thwarted scenarios to become a reality. To demonstrate the severity of our results, we exploited one of the arbitrary write vulnerabilities to gain code execution in the kernel and escalate from app-level privileges to root.

We are currently working on responsibly disclosing the vulnerabilities to the vendors. While doing so, we found that four of the bugs were patched during the course of the experiments. To the best of our knowledge the remaining 32 of the 36 bugs are 0-days.

In the next few subsections, we will present case studies of two bugs found during our experiments, demonstrating their impact and the necessity of interface information in their detection.

8.4 Case Study I: Design issue in Honor 8

One of the most interesting bugs in our collection was found not through an OS crash (as is typical for kernel bugs), but by noticing very strange behavior from the target host. After several fuzzing rounds on the Huawei Honor 8, we noticed that the serial number of the device had changed, as shown in Listing 6. The serial number of the device should be a read-only property which only the boot loader (which runs at the high EL3 privilege level [4]) should be able to change. However, this occurrence shows that the serial number can actually be changed from a userspace application (running at the least privilege level EL0) on Android by exploiting this kernel driver. Thus, this represents a design-level vulnerability.

This bug was found while fuzzing the driver `nve`. The Honor 8 has a partition on flash, `nvme`, which stores the device configuration information. Some of these configuration options are unprivileged and can be modifiable by Android. This includes whether the device unlock is enabled and whether ramdump is allowed, but notably excludes properties such as the board identifier and serial number, which should only be modifiable by the boot loader. However, the `ioctl` handler for the device `/dev/nve` provides a way to read and write these options. Additionally, it does not check the type of

Table 5: Bugs found by each fuzzing configuration per device

	syzkaller base	syzkaller+path	DIFUZE^{i}	DIFUZE^{s}	DIFUZE^{m}	total unique
Pixel	0	1	2	5	3	5
E9 Plus	0	0	4	6	6	6
M9	0	0	3	3	2	3
P9 Lite	0	0	2	5	5	6
Honor 8	0	0	1	2	2	2
Galaxy S6	-	-	-	-	0	0
Xperia XA	0	2	10	13	12	14
Total	0	3	22	34	30	36

configuration option, and a malicious userspace application can read or write the privileged configuration options.

Adding a check to disallow modifications to privilege configuration options could fix this issue. It should not be possible for Android kernel running at privilege level EL1 to read or write options that belong to the boot loader running at higher privilege. Of course, the truly correct fix for this is to separate privileged and unprivileged options, and store them on different partitions accessible by differently-privileged code.

```
# before fuzzing
HWFRD:/ $ getprop ro.serialno
RNV0216811001641
# after fuzzing
HWFRD:/ $ getprop ro.serialno
^Rï£¡DO>l
```

Listing 6: A design issue found by DIFUZE while fuzzing nve driver.

8.5 Case Study II: qseecom bug

In this section, we walk through an example of a bug that was found only with the highest level of interface extraction (that is, type recovery/complex structure instantiation). The relevant source is shown below, which we will reference.

```
1  static int qseecom_mdtp_cipher_dip(void __user *argp)
2  {
3          struct qseecom_mdtp_cipher_dip_req req;
4          u32 tzbuflenin, tzbuflenout;
5          char *tzbufin = NULL, *tzbufout = NULL;
6          int ret;
7
8          do {
9                  ret = copy_from_user(&req, argp, sizeof(req));
10                 if (ret) {
11                         pr_err("copy_from_user failed, ret= %d\n",
12                                ret);
13                         break;
14                 }
15                 ...
16                 /* Copy the input buffer from
17                    userspace to kernel space */
18                 tzbuflenin = PAGE_ALIGN(req.in_buf_size);
19                 tzbufin = kzalloc(tzbuflenin, GFP_KERNEL);
20                 if (!tzbufin) {
21                         pr_err("error allocating in buffer\n");
22                         ret = -ENOMEM;
23                         break;
24                 }
25
26                 ret = copy_from_user(tzbufin, req.in_buf,
27                                      req.in_buf_size);
28                 ...
29         } while (0);
30         ...
31         return ret;
32 }
33
34 long qseecom_ioctl(struct file *file, unsigned cmd,
35                    unsigned long arg)
36 {
37         int ret = 0;
38         void __user *argp = (void __user *) arg;
39         switch (cmd) {
40         ...
41         case QSEECOM_IOCTL_MDTP_CIPHER_DIP_REQ: {
42                 ...
43                 ret = qseecom_mdtp_cipher_dip(argp);
44                 break;
45         }
46         ...
47         }
48         return ret;
49 }
```

Our example is that of CVE-2017-0612 (this is one of the four bugs which was patched during the course of the experiments) [1]. This bug was found by our system on Google's flagship Android phone, the Pixel. The `ioctl` function for the driver starts at line 31 and follows the common design of `ioctls`. The userspace application specifies cmd and arg.

Given the cmd `QSEECOM_IOCTL_MDTP_CIPHER_DIP_REQ`, we enter `qseecom_mdtp_cipher_dip` on line 39. Inside this function, on line 9, we see our user data copied into a `struct qseecom_mdtp_cipher_dip_req req`. In line 16, we see the bug. `tzbuflenin` is calculated by calling `PAGE_ALIGN` on our user controlled value of `req.in_buf_size`. If a userspace application provides a large value here, `PAGE_ALIGN` will overflow, resulting in a value smaller than `req.in_buf_size`, specifically zero. Next, on line 17, we see an attempt to `kalloc` this calculated size. Finally, on line 24, the driver attempts to copy_from_user an embedded pointer in our struct to the allocated buffer. This copy_from_user will result in a crash, as the size of the buffer was improperly calculated. Note, however, for this crash to be observed, the user supplied `req.in_buffer` must be a *valid* pointer (else copy_from_user will fail gracefully, and return an error). Thus, without a properly instantiated argument to the `ioctl`, this crash will never be triggered.

Table 6: Performance of coverage-guided fuzzing with and without interface information.

ioctl cmd ID	Interface type	Basic Blocks covered		Percentage increase
		No interface	Interface Aware	
SCSI_IOCTL_SEND_COMMAND	Simple Structure	3811	4629	21.46%
CDROM_SEND_PACKET	Complex Structure	3956	5582	41.10%

8.6 Augmenting with Coverage-guided Fuzzing

Coverage-guided fuzzing is a well-studied technique and was shown to be an effective method to achieve good coverage [9]. Thus, a natural question arises: is interface awareness still needed if coverage-guidance can be used? The answer is yes: providing interface information for coverage-guided fuzzing will significantly improve its performance on drivers.

To demonstrate, we ran syzkaller in the coverage-guided mode on an x86-64 kernel, fuzzing `ioctl`s `SCSI_IOCTL_SEND_COMMAND` (which has a simple interface) and `CDROM_SEND_PACKET` (which has a complex interface) with and without structure interface information for four hours per combination. Table 6 shows the results of these combinations, where the last column shows the percentage increase in basic blocks reached when the interface information was provided. This shows that interface information would still significantly improve coverage-guided fuzzing performance.

Scaling this evaluation to our commercial devices is difficult due to the necessity to recompile, often backporting kcov [36], and re-flashing the kernel. This requires significant engineering effort, outside the scope of this project.

9 DISCUSSION

We have shown that by using interface-aware fuzzing DIFUZE can improve kernel security by uncovering potentially harmful bugs. However, there are still some weaknesses of this approach and directions for improvement, which we will review in this section.

9.1 Weaknesses

One problem, which we discovered while fuzzing, was that buggy drivers could crash early on, preventing the fuzzer from exploring deeper functionality in the driver. There are likely bugs that we never hit, simply because an earlier bug is triggered frequently, and each time we hit that bug the phone rebooted. With current techniques, our only recourse was to stop fuzzing that particular command identifier, or at times, even the whole `ioctl` handler and move on to others.

Another weakness of DIFUZE is the inability to extract complex relationships between fields of structures in the interface. It is not uncommon that one field of a structure relates to another: for example, a length field could specify the size of a buffer. However, our system does not recognize these relationships, which could potentially provide valuable information to the fuzzer.

9.2 Future Work

A valuable technique for fuzzing, found in many of the best fuzzers, is using run-time coverage to guide the fuzzer. Currently, we do not use this technique (though as we show in Section 8.6, we can vastly improve coverage guided techniques with interface awareness). To use run-time coverage information, we would need to re-compile and flash the kernel to the device, which presents several challenges. First, to get fine-grained coverage information, a development board is needed. This can be expensive or, in many cases, simply unavailable for real-world devices. Second, it is not always possible to find the latest kernel sources to recompile. This is acceptable for DIFUZE, as it is unlikely that `ioctl` interfaces change radically between minor kernel updates, and the actual execution will still be performed on the latest version of the software on the target host. However, if an older (instrumented) kernel is flashed onto the target host, the bugs discovered as a result might already be obsolete. Finally, some vendors do not make it easy to flash a new kernel to the device by locking the bootloader and performing other security checks.

For these reasons, we did not instrument the kernel to insert code coverage measurements or the Kernel Address Sanitizer (KASAN). Both KASAN and coverage information could further improve the results of DIFUZE. KASAN helps to find bugs by detecting memory corruption and triggering an assertion failure. Without it, exploitable bugs may be triggered without causing the device to crash, simply because the corrupted memory is not used by other functionality, or because no important data was corrupted. Coverage information could improve the system by enabling deeper exploration of drivers as it will try to mutate inputs that trigger previously-neglected driver functionality.

10 CONCLUSION

In this paper, we proposed *interface aware fuzzing* to increase the effectiveness of automated analysis on interface-sensitive code such as Linux kernel drivers. We provided a set of techniques to recover the `ioctl` interface specifications for fuzzing such code. We implemented all our techniques in an automated pipeline that works directly on the kernel source archive with a single command. We show that our technique is efficient and effective in recovering components, device file names, valid command identifiers and corresponding argument types of the interface for most drivers. We carry out a thorough evaluation, using several different configurations of DIFUZE on seven models of Android phones, to demonstrate that our implementation of interface aware fuzzer is effective, finding 36 bugs, of which 32 are previously unknown vulnerabilities.

We are open sourcing our DIFUZE to provide the community with a tool to help ensure the safety of modern mobile devices.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This material is based on research sponsored by the Office of Naval Research under grant

numbers N00014-15-1-2948, N00014-17-1-2011 and by DARPA under agreement number FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

This work is also sponsored by a gift from Google's Anti-Abuse group.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] 2016. Android Security BulletinâĂŤMay 2017. (2016). https://source.android.com/security/bulletin/2017-05-01.

[2] Alfred Aho, Jeffrey Ullman, Monica S. Lam, and Ravi Sethi. 1986. *Compilers: Principles, Techniques, and Tools.* "Addison-Wesley".

[3] Dave Aitel. 2002. The Advantages of Block-Based Protocol Analysis for Security Testing. (2002). https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html.

[4] ARM. 2013. ARM Exception levels. (2013). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/CHDHJIJG.html.

[5] K. Ashcraft and D. Engler. 2002. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02)*. 143–159. https://doi.org/10.1109/SECPRI.2002.1004368

[6] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *Proceedings of the 2006 ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*. ACM, New York, NY, USA, 73–85. https://doi.org/10.1145/1217935.1217943

[7] Peter T. Breuer and Simon Pickin. 2006. *One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code.* Springer Berlin Heidelberg, Berlin, Heidelberg, 56–70. https://doi.org/10.1007/11767077_5

[8] Laurent Butti and Julien Tinnes. 2008. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology* 4, 1 (2008), 25–37.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 2008 USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[11] Gabriel Campana. 2009. Fuzzgrind: un outil de fuzzing automatique. *Actes du* (2009), 213–229.

[12] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 725–741. https://doi.org/10.1109/SP.2015.50

[13] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 110–125. https://doi.org/10.1109/SP.2009.14

[14] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers: Where the Kernel Meets the Hardware.* " O'Reilly Media, Inc.".

[15] International Data Corporation. 2016. Smartphone OS Market Share. (2016). http://www.idc.com/promo/smartphone-market-share/os.

[16] Cr4sh. 2011. IOCTL Fuzzer - Windows kernel drivers fuzzer. (2011). https://github.com/Cr4sh/ioctlfuzzer.

[17] debasishm89. 2014. A mutation based user mode (ring3) dumb in-memory Windows Kernel (IOCTL) Fuzzer. (2014). https://github.com/debasishm89/iofuzz.

[18] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 482–493. https://doi.org/10.1109/ASE.2015.65

[19] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary Fuzzing for Black-box XSS Detection. In *Proceedings of the 2014 ACM Conference on Data and Application Security and Privacy (CODASPY '14)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2557547.2557550

[20] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A Survey of Mobile Malware in the Wild. In *Proceedings of the 2011 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2046614.2046618

[21] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 2009 International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[22] GNU. 2007. GNU General Public License. (2007). https://www.gnu.org/licenses/gpl-3.0.en.html.

[23] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[25] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *Proceedings of the 2008 Symposium on Network and Distributed System Security (NDSS '08)*. San Diego, CA, USA.

[26] Google. 2016. The Kernel Address Sanitizer. (2016). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[27] Google. 2017. Goole Android Kernel Sources. (2017). https://android.googlesource.com/kernel.

[28] Google. 2017. syzkaller - linux syscall fuzzer. (2017). https://github.com/google/syzkaller.

[29] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 2016 International Symposium on Haskell (Haskell '16)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2976002.2976017

[30] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 2013 USENIX Security Symposium (SEC '13)*. Washington, DC, USA, 49–64.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments.. In *Proceedings of the 2012 USENIX Security Symposium (SEC '12)*. Bellevue, WA, USA, 445–458.

[32] HTC. 2017. HTC Android Kernel Sources. (2017). https://www.htcdev.com/devcenter/downloads.

[33] Huawei. 2017. Huawei Android Kernel Sources. (2017). http://consumer.huawei.com/ng/support/downloads/index.htm.

[34] Dave Jones. 2011. Trinity: A system call fuzzer. In *Proceedings of the 2011 Ottawa Linux Symposium (OLS '11)*.

[35] kernel. 2001. Character device registration. (2001). http://www.makelinux.net/ldd3/chp-3-sect-4.

[36] Paul Larson, Nigel Hinds, Rajan Ravindran, and Hubertus Franke. 2003. Improving the Linux Test Project with kernel code coverage analysis. In *Proceedings of the 2003 Ottawa Linux Symposium (OLS '03)*.

[37] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*. ACM, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[38] Stanislas Lejay. 2016. Fuzzing IOCTLs with angr. (2016). https://thunderco.re/project/security/2016/07/18/fuzzing-ioctls/.

[39] LG. 2017. LG Android Kernel Sources. (2017). http://opensource.lge.com/osList/list?m=Mc001&s=Sc002.

[40] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 2010 Annual Information Security Symposium (CERIAS '10)*. CERIAS - Purdue University, West Lafayette, IN, Article 5, 1 pages. http://dl.acm.org/citation.cfm?id=2788959.2788964

[41] Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and Zhuo-Chun Tang. 2008. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *Proceedings of the 2008 Convergence and Hybrid Information Technology (ICCIT '08)*, Vol. 2. IEEE, 491–497.

[42] Manuel Mendonça and Nuno Neves. 2008. Fuzzing wi-fi drivers to locate security vulnerabilities. In *Proceedings of the 2008 Dependable Computing Conference (EDCC '08)*. IEEE, 110–119.

[43] Alessio Merlo, Gabriele Costa, Luca Verderame, and Alessandro Armando. 2016. Android vs. SEAndroid. *Pervasive Mob. Comput.* 30, C (Aug. 2016), 113–131. https://doi.org/10.1016/j.pmcj.2016.01.006

[44] Microsoft. 2017. How to Perform Fuzz Tests with IoSpy and IoAttack. (2017). https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack.

[45] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[46] Motorola. 2017. Motorola Android Kernel Sources. (2017). https://github.com/MotorolaMobilityLLC.

[47] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746
[48] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The BORG: Nanoprobing Binaries for Buffer Overreads. In *Proceedings of the 2015 ACM Conference on Data and Application Security and Privacy (CODASPY '15)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/2699026.2699098
[49] Peach. 2017. The Peach Fuzzer. (2017). http://www.peachfuzzer.com/.
[50] Hendrik Post and Wolfgang Küchlin. 2007. *Integrated Static Analysis for Linux Device Driver Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 518–537. https://doi.org/10.1007/978-3-540-73210-5_27
[51] LLVM Project. 2003. LLVM Bitcode File Format. (2003). http://llvm.org/docs/BitCodeFormat.html.
[52] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*. IEEE Computer Society, 1–11.
[53] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*. San Diego, CA, USA.
[54] redhat. 2017. Proc device registration. (2017). https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Reference_Guide/s2-proc-devices.html.
[55] Juha Röning, Marko Laakso, and Ari Takanen. 2002. PROTOS – Systematic Approach to Eliminate Software Vulnerabilities. *Invited presentation at Microsoft Research* (May 2002).
[56] Alessandro Rubini and Jonathan Corbet. 2001. *Linux device drivers.* " O'Reilly Media, Inc.".
[57] Samsung. 2017. Samsung Android Kernel Sources. (2017). http://opensource.samsung.com/reception/receptionSub.do?method=sub&sub=T&menu_item=mobile&classification1=mobile_phone.
[58] Sergej Schumilo, Ralf Spenneberg, and H Schwartke. 2014. DonâĂŹt trust your USB! How to find bugs in USB device drivers. *Blackhat Europe* (2014).
[59] Kwan Yong Sim, F-C Kuo, and R Merkel. 2011. Fuzzing the out-of-memory killer on embedded Linux: an adaptive random approach. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*. ACM, 387–392.
[60] Sony. 2017. Sony Android Kernel Sources. (2017). https://github.com/sonyxperiadev/kernel.
[61] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. San Diego, CA, USA.
[62] Jeffrey Vander Stoep. 2016. Android: protecting the kernel. In *Linux Security Summit*. Linux Foundation.
[63] Linus Torvalds. 2011. C2XML - Converting source code to XML. (2011). http://c2xml.sourceforge.net/.
[64] Vincent M Weaver and Dave Jones. 2015. *perf fuzzer: Targeted fuzzing of the perf event open () system call.* Technical Report. Technical Report UMAINEVMW-TR-PERF-FUZZER, University of Maine.
[65] Wiki. 2017. TanenbaumâĂŞTorvalds debate. (2017). https://en.wikipedia.org/wiki/Tanenbaum%E2%80%93Torvalds_debate.
[66] Xiaomi. 2017. Xiaomi Android Kernel Sources. (2017). https://github.com/MiCode/Xiaomi_Kernel_OpenSource.
[67] Xst3nZ. 2012. IOCTLbf is just a small tool (Proof of Concept) that can be used to search vulnerabilities in Windows kernel drivers. (2012). https://code.google.com/archive/p/ioctlbf/.
[68] Michal Zalewski. 2014. Binary fuzzing strategies: what works, what doesn't. (2014). https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.
[69] M. Zalewski. 2017. American Fuzzy Lop. (2017). http://lcamtuf.coredump.cx/afl/technical_details.txt.

A IOCTL REGISTRATION STRUCTURES

There are several structures that could be used by the Linux kernel drivers to register an ioctl handler. Listing 7 shows the list of structures in the Kernel running on the Huawei P9.

```
struct.media_file_operations
struct.video_device
struct.v4l2_file_operations
struct.block_device_operations
struct.tty_operations
struct.posix_clock_operations
struct.security_operations
struct.file_operations
struct.v4l2_subdev_core_ops
struct.snd_pcm_ops
struct.snd_hwdep_ops
struct.snd_info_entry_ops
struct.adf_obj_ops
struct.net_device_ops
struct.kvm_device_ops
struct.ide_disk_ops
struct.ide_ioctl_devset
struct.hdlcdrv_ops
struct.uart_ops
struct.fb_ops
struct.proto_ops
struct.tty_ldisc_ops
struct.watchdog_ops
struct.atmdev_ops
struct.atmphy_ops
struct.atm_ioctl
struct.vfio_device_ops
struct.vfio_iommu_driver_ops
struct.rtc_class_ops
struct.usb_gadget_ops
struct.ppp_channel_ops
struct.cdrom_device_info
struct.cdrom_device_ops
```

Listing 7: List of structures that can be used to register an ioctl handler.

B HANDLING V4L2 DRIVERS

There are certain ioctl functions whose commands and arguments are first verified by the Linux kernel before the driver specific functions are invoked. This includes Video for Linux (`v4l2`) ioctls as shown in Listing 8, where the driver provides a standardized, overrideable ioctl handler (set by drivers using the `v4l2_ioctl_ops` structure, line 2 in Listing 8) to ease the creation of video devices (such as cameras). The Linux kernel implements the ioctl handler function `video_ioctl2` (line 10), which checks the provided ioctl identifier and calls specific v4l2 handler functions provided by the driver itself. Similar to other ioctl handlers, `video_ioctl2` also expects specific structures from the user, depending on the command identifier. Furthermore, the dispatched v4l2 handler functions themselves also expect properly formatted input with proper command codes passed in.

This poses two analysis challenges. First, as mentioned in Section 4.1, we consider only the functions defined by the driver. As such, we would miss the ioctl handler `video_ioctl2`, which is defined by the kernel. To handle this, we identify the v4l2 registration function `video_register_device` (line 32) and traverse the structures of its arguments to identify the `v4l2_ioctl_ops` data structure (line 32 → 29 → 13 → 17 → 2), treating each function pointer in the structure as analogous to a top-level ioctl handler. However, we need to tackle a second problem. In order to trigger the execution of any of the functions registered via `v4l2_ioctl_ops`, the proper standardized v4l2 ioctl command identifier must be provided. Furthermore, the sub-handlers provided by the driver introduce their own command identifiers as well. Thus, DIFUZE keeps track of a *nested interface* for such devices..

```
1  // v4l2_ioctl_ops initialized with required functions.
2  static const struct v4l2_ioctl_ops iris_ioctl_ops = {
3    .vidioc_querycap = iris_vidioc_querycap,
4    .vidioc_s_tuner = iris_vidioc_s_tuner
5  }
6
7  static const struct v4l2_file_operations iris_fops = {
8    // here video_ioctl2, implemented by kernel
9    // is the main ioctl handler.
10   .unlocked_ioctl = video_ioctl2
11 };
12
13 static struct video_device iris_viddev_template = {
14   //initialize file operations.
15   .fops  = &iris_fops,
16   // initialize ioctl operations.
17   .ioctl_ops = &iris_ioctl_ops
18 };
19
20 static int __init driver_init() {
21   struct iris_device *radio;
22   int radio_nr = -1;
23   radio = kzalloc(sizeof(struct iris_device), GFP_KERNEL);
24   if (!radio) {
25     FMDERR(": Could not allocate radio device\n");
26     return -ENOMEM;
27   }
28   // copy the video_device structure.
29   memcpy(radio->videodev, &iris_viddev_template,
30           sizeof(iris_viddev_template));
31   // register the v4l2 device
32   video_register_device(radio->videodev, VFL_TYPE_RADIO, radio_nr);
33 }
```

Listing 8: Example of a v4l2_ioctl_ops initialization and registering of a v4l2 device.

```
1  vidioc_querycap:2154321408
2  vidioc_s_tuner:1079268894
```

Listing 9: An example *v4l2-function-mapping*, which contains entries in <function name>:<command id> format.

To handle this, we first create a mapping between the command ID and the function pointer, to identify which function in the set will be called for a given command value. DIFUZE automatically extracts such information with LLVM. For the example v4l2 driver in Listing 8, we generate a mapping called *v4l2-function-mapping*, as shown in Listing 9. DIFUZE associates the sub-handler functions `iris_vidioc_querycap` and `iris_vidioc_s_tuner` (line 3 and line 4 in Listing 8), with v4l2-standard ioctl command identifiers of 2154321408 and 1079268894 (line 1 and line 4 in Listing 9). These functions would then be further analyzed to recover nested interface information.