# A Hybrid Interface Recovery Method for Android Kernels Fuzzing

Shuaibing Lu, Xiaohui Kuang, Yuanping Nie, Zhechao Lin
National Key Laboratory of Science and Technology on Information System Security,
Beijing, 100101, China
shuaibinglu@126.com, xiaohui_kuang@163.com, yuanpingnie@nudt.edu.cn, acbuwa@foxmail.com

***Abstract*—Android kernel fuzzing is a research area of interest specifically for detecting kernel vulnerabilities which may allow attackers to obtain the root privilege. The number of Android mobile phones is increasing rapidly with the explosive growth of Android kernel drivers. Interface aware fuzzing is an effective technique to test the security of kernel driver. Existing researches rely on static analysis with kernel source code. However, in fact, there exist millions of Android mobile phones without public accessible source code. In this paper, we propose a hybrid interface recovery method for fuzzing kernels which can recover kernel driver interface no matter the source code is available or not. In white box condition, we employ a dynamic interface recover method that can automatically and completely identify the interface knowledge. In black box condition, we use reverse engineering to extract the key interface information and use similarity computation to infer argument types. We evaluate our hybrid algorithm on on 12 Android smartphones from 9 vendors. Empirical experimental results show that our method can effectively recover interface argument lists and find Android kernel bugs. In total, 31 vulnerabilities are reported in white and black box conditions. The vulnerabilities were responsibly disclosed to affected vendors and 9 of the reported vulnerabilities have been already assigned CVEs.**

***Index Terms*—IoT security, kernel fuzzing, interface recovery, Android security**

## I. Introduction

Recently, “smart” products such as smart phones, watches and TVs, perform a significant role in people’s lives. Android powered products have been in a dominant position among all smart phones. According to an IDC investigation, in 2019, smartphone companies shipped a total of 1.382 billion phones and Android phones capture roughly 86.6% of the worldwide smartphone volume [1]. Android equipment provides rich functionality to satisfy the needs of consumers. Users can communicate with their friends using text, audio and video. They can also query their friends’ locations, navigate to destination, record their individual health data and even pay bills using Android mobile phones. Android mobile phones influence all aspects of people’s lives. Therefore, the security features of Android mobile phones are of great importance. If attackers invade an Android mobile phone, they are not only able to steal user’s private information but they also endanger the user’s identity and property safety. On a social level, Android operating system vulnerabilities can threaten the safety of public facilities and commercial trade.

Android security analysis has become a key area of interest in both industry and academia. Android security can be considered on two levels: userspace application and kernel. Kernel vulnerabilities threaten operation systems and user applications. The kernel of the Android operating system is relatively vulnerable to attack, despite available protections [2].

Fuzzing is a well-known technique that is used for security testing by generating massive, specially designed inputs to the target programs. However, with the Android operating system becoming more and more complex, kernel interface arguments are diverse. Without prior knowledge of argument structures, it is difficult for fuzzing tools to find kernel vulnerabilities. Some researches [3], [4] have been done in order to reduce the random space and allow fuzzing tools to make meaningful choices when mutating the data. Corina et al. [5] proposed Difuze, a kernel interface-aware fuzzing tool used for automatically generating valid inputs and trigger the execution of the kernel drivers. Difuze implements an automatic kernel interface recovery tool and identifies kernel bugs in real phones. However, these approaches rely on the static analysis of the Android kernel source code. Android products must be open-sourced according to the GNU General Public License [6]. However, in fact, researchers cannot always find the custom kernel code for some Android mobile phones. For example, when a) some manufacturers do not provide all the device source codes for some reason, like OPPO, one of the top five Android smartphone manufacturers in sales [1]; b) the Android kernel has many versions, it is not easy to find the corresponding version, especially for small-volume manufacturers; c) there may be a time delay between the release date of the product and the date of the publication of the source codes; and d) the language barrier makes source code retrieval difficult. It is necessary to develop an Android kernel interface reconstruction algorithm in the black box condition. Fuzzing an Android kernel without source code (black box) can be beneficial, as follows:

1. Improvement of fuzzing generalization: Without the kernel source code, the fuzzing relies on fewer input which enables fuzzing to work in very restricted conditions.

2. A broadening of the scope of safety testing: Millions of users are using Android products that are not open source or for which it is difficult to find the source code. Proposing a black box fuzzing tool can help to improve the scope of the fuzzing test.

3. Ease of operation for safety testers: Analyzing the kernel requires testers with professional background knowledge. A

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00052

black box condition fuzzing tool makes it easy for beginners to test the Android mobile phones.

In this paper, we introduce a hybrid kernel interface-aware fuzzing framework for Android drivers that can effectively fuzz Android mobile phones with source codes (white box conditions) and without source code (black box conditions). We open sourced our work on Github[1].

We have designed several experiments to evaluate the proposed fuzzing method. We first analyze the kernel interface recovery results in both white and black box conditions. In the white box conditions, compared with other state-of-the-art algorithms, our method can recover the interface information more automatically and faster.

In summary, our work makes the following contributions to the field:

**Fast and automatic interface recovery in white box conditions**. Interface recovery is the key component of kernel interface-aware fuzzing. We propose a dwarf-based interface recovery method that can quickly and automatically rebuild the interface parameters in white box conditions. The proposed interface recovery technique is more automated and obtain faster results [5].

**The interface structure inference in black box conditions**. Without the Android kernel source code, we propose a binary reverse engineering method to extract the interface structure information and employ a similarity calculation method to obtain interface inference. The proposed method can effectively generate accurate testing samples for a deep fuzzing test.

**Fuzzing Android kernel drivers in both white and black box conditions**. We introduce a hybrid method to facilitate the interface-aware fuzzing test of Android mobile phones in both white and black box conditions. We found 13 vulnerabilities in the mobile phones with source codes and 15 in the mobile phones without source codes in a real Android mobile phone security test. With the manufactures permission, we publish 9 of the bugs on the CVE site [2] [3].

## II. Background

In this section, we will first introduce the common Android fuzzing methods and explain the unique techniques employed by our approach. Then, we will present the existing kernel and driver fuzzing techniques that are the most closely related to our work and compare our method with these previous techniques.

### A. POSIX Driver

The POSIX device standard uses ioctl as an interface for data exchange between the userspace and the kernel driver. Each device is registered by the corresponding driver and provides different functions, such as open(), write(), seek(), ioctl(), and so on. The ioctl interface implements the functions of configuration changes, parameter acquisition, and device state settings. For example, the operation of the microphone modification sampling frequency cannot be done by using regular read and write operations. In the user space program, the ioctl system call is invoked by using the file descriptor, the command identifier, and the required data structure as the parameters. In the kernel, the corresponding driver ioctl handler function is invoked according to the file descriptor. The ioctl handler function executes corresponding operations according to the command identifier and processes data from the incoming user space. In the Linux system, the directory path $'/dev'$ stores the device files, and each device file corresponds to the ioctl processing function. The complexity and diversity make the ioctl system call more vulnerable and challenge valid sample generating as well. In this paper, we attempt to recover the values and types of ioctl arguments in order to guide the deep fuzzing test.

### B. Kernel Fuzzing

The kernels of modern operating systems are relatively vulnerable to attacks, despite the available protection. The Android kernel fuzzing test in particular has attracted a great deal of research attention due to the urgent need for kernel security. It is practical for fuzzing interface or system calls to test the operating system kernel. Most drivers use ioctl functions, forming a POSIX standard, to communicate with the user space. Ioctl fuzzing requires specific command values and data formats generated by users. It is necessary to identify valid command values and their associated data structures in ioctl fuzzing. Previous researches have been proposed for Windows kernels: Iofuzz [7], ioattack [8], ioctlbf [9] and ioctlfuzzer [10]. Some works [11], [12] introduce fuzzing method for Mac OS kernels. For Linux kernels, the well-known linux syscall fuzzing tools are Trinity [13] and syzkaller [14]. These are reported to perform badly when fuzzing the ioctl handlers of device drivers. There are some previous researches that focus on specific drivers or solely syscall. Some approaches [15], [16] concentrate on wireless drivers. Several researches[17], [18], [19] have been done aiming at USB drivers. Perf-fuzzer [20] introduces a targeted fuzzing of the $perf\_event\_open()$ system call. Stanislas, et al [21] first attempted to extract valid ioctl commands, but the system was unable to scale to real-world kernel modules. Difuze [5] is the most recent state-of-the-art system proposed by Corina et al.. Difuze is the first completely automated system that can be generalized to fuzz all Linux drivers on a device depending on the kernel source code. Furthermore, Difuze can effectively find vulnerabilities in real-world products.

However, Difuze does have the following problems when used to fuzzing Android kernels:

**Frequent Manual interaction:** Many steps are needed to handle manually, reducing the efficiency of the fuzzer. For example, in the step of recovering device names, Difuze cannot restore the dynamically generated device file names, and must be handled manually; Difuze needs to manually specify the driver directory that needs to be analyzed in the kernel code. This is due to the large number of differences in the driver directory of different Android mobile phones. Researchers will

[1]https://github.com/datadancer/HIAFuzz

[2]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11019,11020,...,11025,18318

[3]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8413

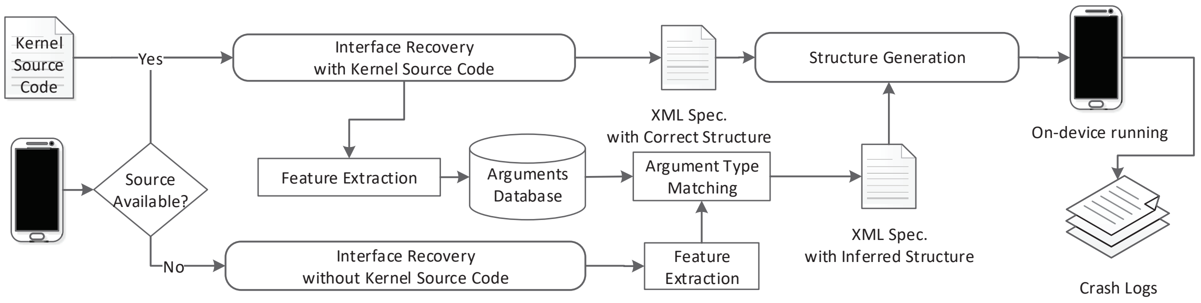

Fig. 1. The Overview of the Hybrid Interface-aware Fuzzing Method

spend much effort to interact with the fuzzer to keep the fuzzer running. **Incomplete support of kernel compilation:** In the step of kernel compilation, Difuze ignores the incomplete supporting for the kernel of LLVM, uses LLVM to recompile the kernel, and causes various compilation errors. To get build instructions for LLVM, system instrument building is needed, and only one thread is permitted to obtain kernel compilation instructions, which is time consuming. The use of LLVM brings much Extra effort to handle the compilation options and errors which are not supported by LLVM. **No support for black box devices:** Difuze rely on the kernel source to obtain the driver interface information, and can not recover the interfaces when the kernel source code is not available.

In this paper, we provide a faster and more automatic interface recovery algorithm which can work both with and without source codes.

## III. Overview

In this section, we will provide an overview of the interface-aware fuzzing approach and its application to kernel ioctl fuzzing. The framework of the interface-aware fuzzing approach is shown in Fig 1. The approach has three main components: Interface recovery with the kernel source code, interface inference without the kernel source code, and interface guided fuzzing. Given a device for fuzzing, we recover the driver interfaces by analyzing its kernel source code (white box conditions) or disassembling its kernel image (black box conditions). In terms of the interface recovery component, to make the recovery complete and more efficient than existing tools, we make use of the GNU binutils tool chain to perform the recovery task. GNU binutils provide original, complete support for kernel compilation, analysis, and disassembly, making the recovery efficient.

In terms of testing a device, we firstly search the original vendor for its kernel source code. If the source code is available, the white box interface recovery is performed, resulting in a list of device file names, commands, and structure definitions. For a device without a kernel source code, we extract the boot image and kernel symbol table, disassembling the relevant ioctl functions. This results in many commands and argument sizes. To achieve the argument type inference, we build an argument type database called Argument Base. We employ a similarity computation algorithm to infer argument types in black box conditions.

After the interface recovery is complete, the testing samples are generated using the recovered structure. In the interface guided fuzzing component, we employ MongoFuzz [5] to find Android kernel vulnerabilities. Because the fuzzing component is not a crucial consideration of this paper, we do not use the latest powerful fuzzing tools such as syzkaller [14] or AFL [22]. Once the target device crashes, we log the input sample and try to use it to reproduce the crash and to locate the bug. In the rest of the paper, we will focus on discussing the interface recovery algorithm in both white and black box conditions.

## IV. Interface Recovery with the Kernel Source Code

Fig 2. demonstrates the process of the interface recovery algorithm in white box conditions. The algorithm has two steps. First, it analyzes the running information of the Android phones, identifying the device files and the corresponding ioctl handler. This step uses the specially designed kernel module to obtain all the dynamically generated device file names and the corresponding addresses and names of the ioctl handlers. A specially designed kernel module is compiled with the kernel source of the target device, and this is loaded during runtime, used to query the ioctl handler of the specified device file (detailed information on this is provided in section 4.1). Second, the command value parameters and the structure parameters of the ioctl handlers are obtained by analyzing the kernel driver code. In this phase, the kernel code is compiled with the -g3 option in order to generate kernel vmlinux with macro dwarf information. Using gdb, objdump, pyelftools [23] and other established tools to analyze the vmlinux, the command values and structure type definitions of the ioctl handlers are obtained. Thus, the exact types of the driver interfaces are restored, and the description files are generated in XML format. Most Android mobile phones comply with the GNU General Public License, and the kernel driver codes have been released. The Android mobile phones of Samsung, SONY, HUAWEI, HTC, among others, can be analyzed using this method. The algorithm obtains the interface of the driver module and can generate accurate testing samples to carry out the deep fuzzing test on the kernel module code.

### A. Device Files and Ioctl Handler Module

In order to fuzz the driver ioctl handler, the associated device files need to be identified. Existing methods statically

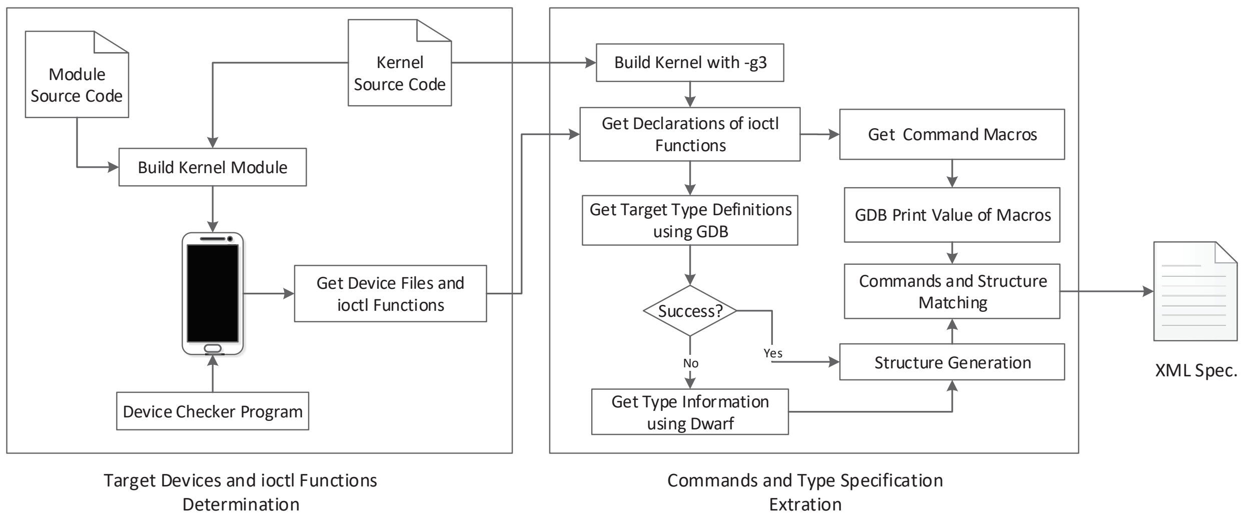
Module Source Code
Kernel Source Code
Build Kernel Module
Get Device Files and ioctl Functions
Device Checker Program
Build Kernel with -g3
Get Declarations of ioctl Functions
Get Command Macros
Get Target Type Definitions using GDB
GDB Print Value of Macros
Success?
Yes
No
Commands and Structure Matching
Structure Generation
Get Type Information using Dwarf
XML Spec.
Target Devices and ioctl Functions Determination
Commands and Type Specification Extration

Fig. 2. Interface Recovery with the Kernel Source Code

analyze the kernel source code to restore the device name strings, while the dynamically generated device file names cannot be determined. Even worse, the static analysis used in such approaches relies on a large number of manual analyses, resulting in low efficiency. In order to handle this, a dynamic method is proposed in order to obtain the device names and the ioctl handler names of the corresponding drivers.

The ioctl system call is implemented in the $fs/ioctl.c$ in the Linux kernel. The ioctl system call first acquires a kernel structure struct file according to the file descriptor argument. By accessing the fop field in struct file, the corresponding struct $file\_operations$ is obtained, and then the $unlocked\_ioctl$ function pointer is obtained, the location of the memory address. Mutating the processing of ioctl, a kernel module is developed. As shown in Fig 2, using the file descriptor argument, the corresponding $unlocked\_ioctl$ is obtained in the kernel space and returned to the user space. In the userspace, by reading and comparing the addresses in the kernel symbol table ($'/proc/kallsyms'$), the function names are obtained. All device file names and corresponding driver ioctl handler names are obtained without analyzing the kernel source code, which improves the degree of automation of the proposed method.

## B. Command Value Determination

After having obtained the driver ioctl handler function, it is necessary to obtain the command values and structure parameters. Given that there can be various macro definitions used in an ioctl handler, we selected macros corresonding to a command value by parsing the switch case statement. The implementation framwork of the ioctl handler functions follows the switch statement programming pattern. The values of command cases are defined by macros. To obtain the values of the command macros, it is necessary to compile the kernel by using -g3 and by replacing the corresponding position -g2 in the Makefile with -g3. Thus, a vmlinux image containing the macro definition debug information is generated. By parsing the $.debug\_macro$ section of the DWARF debugging information, the values of the macro definitions can be obtained. Gdb is used to load the vmlinux image, list the ioctl handler function code context, and print the macro command values. Occasionally, for ioctl handler functions without a switch case statement, we just select all the macros used for commands to prevent missing valid values.

## C. Argument Type Determination

Depending on the command value, arguments may have different types. The relation between commands and argument types is determine by the control flow. When dealing with commands, the driver uses $copy\_from\_user$ to pass the data from the user space and $copy\_to\_user$ to transmit data to the user space. The switch case statement in the source maps the argument type to corresponding command values. The argument types corresponding to the commands will be determined by parsing the source code and the image debug information. By analyzing the location and parameters of the $copy\_from\_user$ function in the driver code, the variable name of the incoming data is obtained. The types of the variables are obtained from the DWARF debug information of vmlinux according to the variable names. Dependent types are retrieved as well. While anonymous structures and unions cannot be parsed by the $ptype$ command of gdb, it is necessary to parse the $.debug\_info$ section of the vmlinux. By determining the compilation unit corresponding to the ioctl handler function, we can locate the data information entry(DIE) and find the ioctl handler function information. Fig.3 shows a driver program with a simplified graphical representation of its DWARF description. The top most DIE represents the compilation unit. It has some “children”, including the DIE describing $ashmem\_ioctl$ and the DIE describing the structure type $ashmem\_area$ which is the value needed by $ashmem\_ioctl$. The subprogram DIE is a child of the compilation unit DIE, while the structure type DIE is referenced by the type attribute in the subprogram DIE.

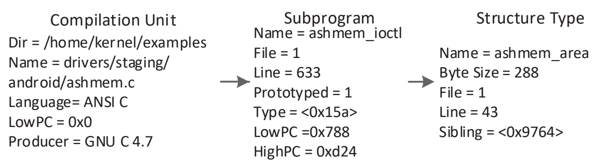

Fig. 3. Graphical Representation of DWARF data

Given an ioctl handler function, variable names, and a vmlinux image, the following steps must be taken in order to identify the DIE of the specified variable type:

1. Use pyelftools to load vmlinux in order to access the DWARF debug information and find the compilation unit (CU) and DIE.

2. Find the DIE that belongs to type $DW\_AT\_subprogram$ and the attribute $DW\_AT\_name$ value that is equal to the ioctl handler function name. Let the current CU be $device\_CU$ and the current DIE be $ioctl\_DIE$, that is, the $device\_CU$ corresponds to the source code file of the driver and $ioctl\_DIE$ corresponds to the code area of the ioctl handler function.

3. Parse the $ioctl\_DIE$ and search the child DIE that belongs to type $DW\_AT\_variable$ and contains a $DW\_AT\_name$ value that is equal to the given variable name. Name this DIE as $variable\_DIE$.

4. Read the $DW\_AT\_type$ attribute of $variable\_DIE$, to get the corresponding type which may be identified as $DW\_TAG\_structure\_type$, $DW\_TAG\_base\_type$ and $DW\_TAG\_enumeration\_type$, etc.. Finally, the type of the specified variable DIE is the type that we have tried to find.

### D. Structure Definition Recovery

The structure DIE that needs to be recovered is determined along with the ioctl handler function names and the variable names. When the structure is a base type, like long, char, short, and float, the structure recovery phrase is not necessary. When the structure type is a custom structure of the program, it is usually possible to use the ptype command of gdb to establish a definition of the structure. However, ptype can only print the type of the global structure definition, and the anonymous union or struct defined in the function body cannot be obtained using ptype. The DIE of the anonymous union or structure in DWARF is needed to recover the structure or union. The DIE of the structure has various attributes, such as name, definition file, line number, and child member information DIEs.

Given the DIE of a structure, the method of recovery of the DIE structure is as follows:

1. Identify the structure name according to the $DW\_AT\_name$ attribute and ascertain the size of the structure according to the $DW\_AT\_byte\_size$ attribute.

2. Traverse the children nodes to obtain the member variable information and get the name, size and type of child DIEs.

3. If the child node is not a base type, analyze the type of child node further.

4. Repeat the above steps until all the child nodes are recovered.

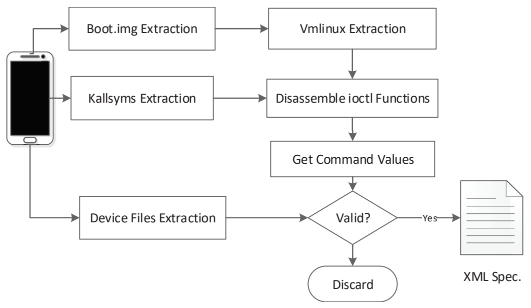

Fig. 4. Interface Reconstruction without the Kernel Source Code

## V. Interface Reconstruction Without the Kernel Source Code

Without the kernel source code, the interface cannot be detected by analyzing the kernel driver source code, and the debugging information cannot be obtained. Therefore, the kernel module cannot be loaded to the kernel, and the structure cannot be recovered. A black-box interface recovery algorithm is proposed. As shown in Fig 4, without the source code, only the kernel binary image and kernel symbol table are available. We use a reverse engineering technique to reconstruct the interface. Then, we can obtain the command values and part of the structure size information, as discussed in section 5.2. The command values are candidates of input samples. Furthermore, all the candidate commands are attempted with each device file to determine if the command is valid for the device file.

### A. Kernel Image and kallsyms Extraction

When the kernel source is compiled, the ELF format vmlinux executable file is generated. The vmlinux is handled by objcopy to generate the binary object file, and then gzip or other compression algorithms are applied to compress the binary image and generate a $piggy.gzip$ file. To make self-decompression occur at booting time, $piggy.gzip$ is converted into an object file linked with an initializer head.o and a self-decompressing program $misc.o$, and this process generates a compressed vmlinux. The final binarization is undertaken in order to obtain a bootloader boot image named zImage.

By analyzing the generation process of zImage, we find that the kernel executable code can be obtained by extracting and decompressing the kernel image, located in the device. Generally, the boot.img is extracted from the corresponding directory of the device directory $'/dev/block/'$ and the boot.img is decompressed with the binary analysis and kernel building tools such as binwalk or unpackbooting. The bzImage is obtained, and the unzip tool is further used to obtain the vmlinux containing the code.

The symbol information of the vmlinux is in the $'/proc/kallsyms'$ kernel symbol table. If the addresses in $'/proc/kallsyms'$ are all zeros, use this command $'echo \quad 0 > /proc/sys/kernel/kptr\_restrict'$ to unlock the restriction.

## B. Commands Recovery

According to the address and function name of the kernel symbol table, the extracted code segment is disassembled, and the control flow diagram is analyzed using IDA. The ioctl handler function uses switch case statement to perform different operations on different commands, and the gcc compiles the switch statement to a series of compare and jump assembly instructions. The constant values in the comparison instruction of the disassembly code are the handler commands.

As the command values are macros in the kernel source, they are loaded into the register as constants in the binary code of the compiled kernel image. In the disassembly code, the loaded and compared immediate digits are recorded as a candidate set of command values for the ioctl handler.

For example, the disassembled *qseecom_ioctl* codes contains codes in this pattern:

```
mov    w0, #0x9711
movk   w0, #0xc024, lsl #16
cmp    w20, w0
b.eq   0x5815f8
```

The operand of *mov* and *movk* are constants that are loaded in register $w0$, and $w0$ is compared with an argument $w20$. The value of $w0$ is a command which is $0xc0249711$. All the commands for the relevant ioctl handlers can be found in this pattern.

## C. Argument Size Recovery

If the ioctl handler function calls *copy_from_user*, the parameter analysis is performed to obtain the size of third parameter. According to the control flow graph of the program, the command value corresponding to the data structure is obtained. Finally, we get a list of triplets in form (ioctl command structure size).

For example, in *qseecom_ioctl*, a *copy_from_user* is used,

```
add    x0,x29, #0x70
mov    x1, x21
mov    x2,#0x20
bl     _arch_copy_from_user
```

The third argument of *copy_from_user* is the size to be copied and passed in register $x2$. The value of $x2$ is the size of the target structure which is $0x20$ in this case.

## D. Device Files and Commands Matching

The obtained triplets list is filtered to match the specified device file. To determine whether each command is valid for the device, perform the following steps:

1. Open the device file, use command and struct size in the triplets list and generate test cases as the arguments of ioctl system call.

2. If an error message such as Invalid command is returned, the currently invalid command value is discarded. Otherwise, the command value will be added to the valid command set of the current devices.

3. The above steps are repeated to finally obtain all the command values that are valid for the device file.

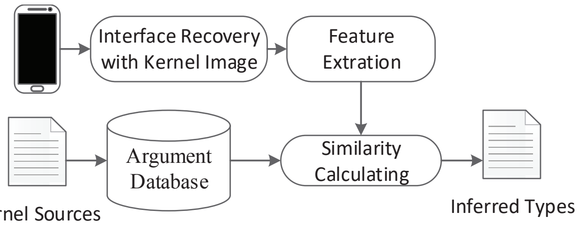

Fig. 5. The process of type inference

After these steps, the corresponding relationship between (device file, ioctl handler, command, struct size) is obtained and the redundant command values are discarded.

## E. Type Inference

It is challenging to obtain the argument types of the ioctl functions from binary image by reverse engineering. However, the argument type is key information for generating valid samples. It indicates the base type of struct and the reference relation. In fact, the drivers with similar functions tend to use similar argument types. For example, the *uart_ioctl* of Huawei P10 and Samsung Galaxy S both use *serial_struct* as the argument. Note that, the drivers used by the close-sourced Android phones are related to the open-sourced drivers. To infer argument types of ioctl handlers in black-box conditions, we build a database of argument types from open-sourced Android kernels, called Argument Base. The process of type inference is shown at Fig.5.

The Argument Base is built by extracting argument information of ioctl handlers from open-sourced Android kernels. We extract 310 Android phone models from most widely used vendors, such as Samsung, Huawei, Sony and HTC. 8730 ioctl handlers and their argument types are extracted and stored in the Argument Base. The argument information is defined by features listed in Table 1. In black-box conditions, argument types can be inferred by retrieval the Argument Base by calculating the feature similarities. We define $f$ as a similarity function. Given a black-box ioctl function feature vector $x = x_1, x_2, ..., x_n$ and a target function feature vector $y = y_1, y_2, ..., y_n$ which is in the Argument Base, we can calculate the similarity between $x$ and $y$:

$$Similarity = f(x, y) \tag{1}$$

,similarity function $f$ is defined in Table 1.

## VI. Interface Guided Fuzzing

In this section, we introduce the interface structure generation and the Fuzz tool employed in our method. The interface recovery component outputs a XML file with commands and correct argument structures.

In white-box condition, we generate test samples according to the type of each structure element. Type-specific values are assigned to each part of a structure according to some specific values like all zeros, all ones, some powers of two and

TABLE I
FEATURES OF IOCTL HANDLERS AND CORRESPONDING SIMILARITY FUNCTIONS

| Feature | Notation | Feature Type | Similarity |
|---|---|---|---|
| Function Name | $S_n$ | String | Cosine |
| Commands | $S_c$ | Int Set | Jaccard |
| Arg Type Name | $S_t$ | String | Cosine |
| Arg Size | $N_a$ | Int | Ratio |
| Function Size | $N_f$ | Int | Ratio |
| Strings | $S_s$ | String | Cosine |
| Opcode Graph | G | Graph | Opcode Sim |
| Device Name | $S_d$ | String | Cosine |
| Sub-function | $S_s$ | String | Cosine |

corner values. Some pointers like void $*$, char $*$ are generated randomly.

In black-box condition, the difference is that we can't obtain the argument type immediately. The black-box argument types can be inferred by matching the most similar argument types in Argument Base. The rest is same as the white condition fuzzing method.

The test samples are input to the fuzz tool. One part of fuzz tool is an executor on target device, receiving commands, device filenames and instantiated structures from host part. The executor open device files and trigger the ioctl with the arguments. Some pointers may exist in structure, the pointer is assigned to an address that contains meaning values to trigger deep path in driver. Once the kernel crashes, the host part will stop fuzzing and record the sample for reproducing the bug.

In this paper, we select MongoFuzz as our fuzz tool. Since the fuzzing component is not the key of our contribution in this paper, we do not use latest powerful fuzz tools.

## VII. EVALUATION

In this section, we evaluate several experiments that we undertook to determine the effectiveness of the proposed method. The evaluation aims to answer the following three questions:

Q1. Compared to other state-of-the-art algorithms, will our method perform better in interface recovery?

Q2. Can the proposed method find real vulnerabilities in the Android mobile phones with kernel source codes and those without?

Q3. What are the advantages and disadvantages of the black box and white box methods?

To answer these questions, we undertook three experiments. All the components of our method run on the same platform, a desktop machine: Ubuntu 16.04.2 LTS OS. The hardware configuration is: Intel core i7 4Ghz CPU with 32G memory. The target Android mobile phones include six open-sourced Android products [24], [25], [26], [27], [28] from five different vendors and five Android products from four different vendors without source code. Note that, Xiaomi MIX 2 is an open source device[29], however, we use black-box method to fuzz MIX 2. The device information is shown at Table 2.

[4]Xiaomi MIX 2 is an open source deivce, however, in this experiment we use black-box method to evaluate the Xiaomi MIX 2.

TABLE II
THE ANDROID DEVICES USED IN OUR EXPERIMENTS

| Device | Android Version | White&Black-box |
|---|---|---|
| Huawei Mate 9 | 7.0 | White |
| Samsung Note 3 | 5.0 | White |
| Huawei Honor 8 | 7.0 | White |
| Sony Xperia | 7.0 | White |
| Amazon Fire HD 3 | Fire OS 4.5.5.3 | White |
| Yotaphone 2 | 5.0 | White |
| Xiaomi MIX 2[4] | 8.0 | Black |
| Xiaomi 5c | 7.1.2 | Black |
| Oppo R11 | 7.1.1 | Black |
| Oppo A37m | 5.1 | Black |
| 360 N6 Pro | 7.1.1 | Black |
| Jianguo Pro2 | 7.1.1 | Black |

TABLE III
IOCTL RECOVERY RESULTS FOR DEVICES WITH KERNEL SOURCE CODES

| | Ioctl Handler | Device Names | Valid Commands |
|---|---|---|---|
| Huawei Mate 9 | 56 | 962 | 308 |
| Samsung Note 3 | 63 | 1165 | 241 |
| Huawei Honor 8 | 58 | 977 | 249 |
| Sony Xperia | 49 | 804 | 512 |
| Amazon Fire HD 3 | 24 | 211 | 277 |
| Yota Yotaphone 2 | 51 | 1020 | 305 |

### A. Interface Recovery Experiment

In this experiment, we evaluate the performance of our interface approaches known as white box interface recovery and black box interface reconstruction.

**In white box conditions**, our approach uses a dynamic device name identification method. The device name identification results are shown in Table 3. Every device corresponds to a sole ioctl handler and every ioctl handler corresponds to one or more device names. Different kernels from different vendors have a large gap between them in terms of the number of valid commands. It is important to note that our interface recovery, in white box conditions, can significantly decrease this cost. Our approach can automatically identify 100% of the device names and 98% of the valid commands. Theoretically, the proposed algorithm can recover 100% of the valid commands; however, in practice, a small proportion of the valid commands cannot be printed by gdb. It could be that some gcc compilation options affected the results in this regard.

Compared with the latest state-of-the-art method, Difuze, our algorithm achieves better performance in device name identification at 100% (Difuze: 59.44%). The reason for this improvement is that our method can capture the dynamically generated device names, while Difuze relies on manual analysis of kernel driver codes.

We note that the kernel driver codes provided by the vendors contain some invalid driver modules that do not exist in the target device. Thus, static analysis is used to identify the invalid ioctl handler and device names, which increases the time involved. Our method can recover the interface faster. The time cost comparison is shown at Table 4. Our running time for interface recovery is 3 minutes and 51 seconds on average, and the Difuze running time is 28 minutes and 18 seconds for the same process. Our method can obviously decrease the time

TABLE IV
TIME COST COMPARISON OF INTERFACE RECOVERY

| | Our method time cost | Difuze time cost |
|---|---|---|
| Huawei Mate 9 | 3min 5s | 42min 19s |
| Samsung Note 3 | 4min 36s | 32min 3s |
| Huawei Honor 8 | 4min 29s | 28min 59s |
| Sony Xperia | 5min 16s | 22min38s |
| Amazon Fire HD 3 | 1min 51s | 15min 34s |
| Average time cost | 3min 51s | 28min 18s |

TABLE V
INTERFACE RECONSTRUCTION RESULTS FOR DEVICES WITHOUT SOURCE CODES

| | Device Names | Candidate Commands |
|---|---|---|
| Xiaomi MIX 2 | 1052 | 1326 |
| Xiaomi 5c | 658 | 875 |
| OPPO R11 | 953 | 1728 |
| OPPO A37m | 867 | 1212 |
| 360 N6 Pro | 954 | 1350 |
| Jianguo Pro2 | 973 | 1176 |

cost for interface recovery.

In practice, we use some additional steps for the Samsung Note 3 and the Sony Xperia. This is because both of these devices do not support kernel module loading in run time. We compiled their kernel source codes to generate a kernel image, which enabled us to load the kernel module in run time. We repacked the boot image with the compiled kernel image and flashed the boot image to the Android device.

**In black box conditions**, for there is no kernel source code, we cannot recover all the interface knowledge. Our black box reconstruction approach can totally recover all the device names and the candidate commands, containing the real, valid commands. The black box interface reconstruction results are shown in Table 5. Similar to white box conditions, different Android phones have different numbers of device names. There are $2^{64}$ possible command values, in theory. Our method can reduce the $2^{64}$ possible command values to around 2,000 candidate values. However, this is still too many to fuzz. We further match the candidate command values with the device files by checking the return value of the ioctl. If it is not a valid command, the error message "Invalid Command" will be returned. Using this method, we can narrow the range of candidate commands for each device file. The minimum number of candidate commands is 10 and the maximum candidate commands around 300.

## B. White Box Interface Recovery Fuzzing

In this experiment, we test our interface recovery fuzzing in white box. We apply the recovered white-box interface structure (in sec 6.1) as the test sample prototype to the fuzzing tool. We employ the MongoFuzz as fuzzing engine. Fuzzing time is set to 12 hours. We test 6 Android mobile phones with source code and recheck all the crashes and filtered out duplicates for each device. The fuzzing engine of Difuze is also configured as Mango, ie, the opensourced version of Difuze. This configuration utilizes the power of interface-aware fuzzing, without advanced optimizations techniques. We root the device to make the executor on device able to fuzz all drivers. Results are shown at Table 6. For HUAWEI honor 8 is already tested in Difuze and repaired all the bugs, our method and Difuze can not find bugs on it. 7 bugs are found on kindle fire HD 3rd tablet, making the most one.

TABLE VI
WHITE BOX FUZZING BUGS

| | Our method | Difuze |
|---|---|---|
| Huawei Mate 9 | 1 | 1 |
| Samsung Note 3 | 0 | 0 |
| Huawei Honor 8 | 0 | 0 |
| Sony Xperia | 3 | 2 |
| Amazon Fire HD 3 | 7 | 6 |
| Yotaphone 2 | 1 | 1 |
| Total | 12 | 10 |

Both Difuze and our method found vulnerabilities in these Android mobile phones. Our method found 12 bugs in total. It is important to note that our method found two more vulnerabilities than Difuze for the Sony Xperia and Kindle Fire HD 3rd. We analyzed the vulnerabilities and found that they exist in the driver modules, whose device names are generated dynamically at run time. Difuze uses a static interface recovery method that cannot capture these dynamically generated device names. Therefore, Difuze cannot identify these vulnerabilities. We present a case study of white-box fuzzing at Appendix A.

## C. Black Box Interface Reconstruction Fuzzing

In this experiment, we analyzed the performance of our black box interface reconstruction fuzzing method. We applied the recovered black box interface knowledge (in section 6.1) as the test sample prototype for the fuzzing tool. The engine used was also MongoFuzz. MongoFuzz needs structure as an input to generate test samples which cannot be obtained in black box conditions. We employed a random generation strategy to construct the test samples.

We chose 11 different Android mobile phones from nine vendors. The devices came from the US (Amazon), Russia (Yota Devices), China (Huawei, Xiaomi, Oppo, 360, and Smartisan), Japan (SONY), and Korea (Samsung). Most of them are the "flagship" products of these vendors. We rooted these 11 Android mobile phones, allowing us to fuzz the kernel drivers. Each Android device was run on MongoFuzz for 12 hours. If one driver crashed multiple times, we commented on the driver and went on to fuzz other drivers. If a device crashed, we recorded the input sample and verified the bug.

We tested the Android mobile phones without source codes. To compare the effect of the black box method and the white box method, we applied the black box fuzzing method to Android mobile phones with source codes. The results are shown in Table 7.

We found 12 vulnerabilities among 6 Android mobile phones using the white box method and 27 bugs among 11 Android mobile phones using the black box method. Android mobile phones without source codes, in total, had 15 vulnerabilities. The total unique number of vulnerabilities is 31. We did not find any vulnerabilities in the Samsung Note 3 and the Huawei Honor 8. The Kindle Fire HD 3rd had 8 vulnerabilities in total. It is important to note that the black box method can

TABLE VII
BLACK BOX FUZZING BUGS

| Device | White Box Fuzzing | Black Box Fuzzing | Total Unique |
|---|---|---|---|
| Huawei Mate 9 | 1 | 1 | 1 |
| Samsung Note 3 | 0 | 0 | 0 |
| Huawei Honor 8 | 0 | 0 | 0 |
| Sony Xperia | 3 | 2 | 3 |
| Amazon Fire HD 3 | 7 | 5 | 8 |
| Yotaphone2 | 1 | 1 | 1 |
| Xiaomi MIX 2 | - | 3 | 3 |
| Xiaomi 5c | - | 3 | 3 |
| Oppo R11 | - | 2 | 2 |
| Oppo A37m | - | 3 | 3 |
| 360 N6 Pro | - | 4 | 4 |
| Jianguo Pro2 | - | 3 | 3 |
| Total | 12 | 27 | 31 |

identify 75% of the vulnerabilities that the white box method found. Furthermore, one vulnerability found by the black box method was ignored by the white box method. This could be because the white box interface recovery method and the Difuze interface recovery algorithm both tried to recover the structure needed by the ioctl interface. However, the structure contains several pointers to other structures which are very complex and contain yet more pointers to more structures. Since the number of dependent structures is too large and the relationship between structures is too complex, our method and Difuze can only recover incomplete structures. This leads to runtime errors using MongoFuzz when generating the testing samples. However, the black box method simply generates a random input and ignores the complex structures and relationship. Thus, MongoFuzz can fuzz the corresponding device and find the vulnerability.

We have disclosed the vulnerabilities to the vendors. While doing so, we found 10 of them were patched in latest updates. 21 of them are 0-days. In the next subsections, we will present the case studies of 3 bugs found, showing the effectiveness of our black box method. A case study of black-box fuzzing is shown at Appendix B.

### D. Results of Code-Coverage

Coverage-guided fuzzing is an effective way to improve code coverage. Can coverage-guidance be benefit from interface information? To answer this question, we provide interface information to syzkaller for fuzzing ioctls of an x86-64 based Android kernel. The target command is $FBIOPUTCMAP$ and $FBIOPAN\_DISPLAY$ with and without interface information for 3 hours (see Table 8). The results are shown in table 9, the basic blocks covered were increased by 14.45% and 35.08% when interface information available.

For most Android mobile phones do not support $CONFIG\_KCOV$ option for their kernels and re-flashing the kernel requires much engineering effort, we leave this component to the future work.

### E. Evaluation Discussion

In summary, we answer the questions enumerated in section 7 affirmatively as follows:

Q1: Compared to other state-of-the-art algorithms, will our method perform better in interface recovery?

A1: Our method can recover the ioctl interface in phones with kernel source codes and those without, while the latest state-of-the-art algorithm Difuze can only reconstruct the interface in white box conditions. Furthermore, in white box conditions, our algorithm performs much better than Difuze in device name identification. Our interface recovery algorithm can dynamically capture the dynamically generated device names well, while Difuze relies on manual kernel driver code analysis. The difference between the two methods is also reflected in the time involved. Our method, in white box conditions, can reduce the running time from 28 minutes 18 seconds to 3 minutes 51 seconds on average. This is because the static analysis used by Difuze will identify the invalid ioctl handler and device names, which is time consuming.

Q2: Can the proposed method find real vulnerabilities in mobile phones with kernel source codes and those without?

A2: The experimental results show that the proposed method can find real vulnerabilities in both white and black box conditions. In the white box fuzzing experiment, we tested six Android mobile phones, "flagship" products from five well-known vendors. We found 12 vulnerabilities in total in this experiment. In the black box fuzzing experiment, we analyzed 11 Android products, six Android mobile phones with kernel source codes and five Android mobile phones without kernel source codes. We found 24 bugs that made the mobile phone crash in the black box fuzzing experiment. The total number of vulnerabilities found is 28. We have communicated with the security departments of the above vendors. Most vulnerabilities were confirmed by the vendors.

Q3: What are the advantages and disadvantages of the black box and white box methods?

A3: In the black box fuzzing experiment, we applied our black box algorithm to Android mobile phones with kernel source codes. Black box fuzzing can find 75% of the vulnerabilities that white box fuzzing can find. This is because black box interface recovery cannot exactly recover the interface knowledge exactly. The black box interface recovery method provides candidate commands which contain the valid command. This affects the precision of the fuzzing test samples. However, we were surprised to find that black box fuzzing can find one bug that the white box fuzzing method could not find. This illustrates that the black box method can find some vulnerabilities that the white box fuzzing method or Difuze will ignore.

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a hybrid Android kernel driver interface-aware fuzzing method. The method has two components. One is the white box interface recovery model which can automatically and completely recover interface knowledge. The other is the black box interface reconstruction model which uses reverse engineering to reconstruct the key information of the kernel driver interfaces without kernel source codes. After having obtained the structure of the kernel driver interface, we used MongoFuzz to test the Android mobile

TABLE VIII
PERFORMANCE OF CODE COVERAGE WITH AND WITHOUT INTERFACE INFORMATION

| Ioctl cmd | Interface Type | Basic Blocks Covered Without Interface | Basic Blocks Covered With Interface | Performance Increase |
|---|---|---|---|---|
| FBIOPUTCMAP | Simple | 2318 | 2653 | 14.45% |
| FBIOPAN_DISPLAY | Complex | 2491 | 3365 | 35.08% |

phones. Compared with state-of-the-art methods, our algorithm can recover the interface better in white box conditions. Furthermore, in the same conditions, our algorithm can find more vulnerabilities. Our hybrid interface-aware fuzzing found 28 vulnerabilities in total from 11 Android mobile phones. The experimental results illustrate that our method works effectively in Android kernel fuzzing.

In the future, we will improve our black box interface reconstruction ability that can narrow the candidate commands and propose a new method to recover the structure of kernel interface without kernel source codes. Furthermore, we will try to extend our method to other operating systems.

## REFERENCES

[1] *Smartphone Market Share*, IDC, Jan 2020, https://www.idc.com/promo/smartphone-market-share/os.
[2] A. Merlo, G. Costa, L. Verderame, and A. Armando, "Android vs. seandroid: An empirical assessment," *Pervasive & Mobile Computing*, vol. 30, pp. 113–131, 2016.
[3] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations," in *Usenix Conference on Security*, 2013, pp. 49–64.
[4] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *IEEE International Conference on Software Engineering*, 2009, pp. 474–484.
[5] J. Corina, A. Machiry, C. Salls, S. Yan, H. Shuang, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *ACM Sigsac Conference*, 2017, pp. 2123–2138.
[6] *GNU General Public License*, GNU, 2007, https://www.gnu.org/licenses/gpl-3.0.en.html.
[7] debasishm89, *A mutation based user mode (ring3) dumb in-memory Win-dows Kernel (IOCTL) Fuzzer*, 2014, http://developer.android.com/tools/help/monkey.html.
[8] *How to Perform Fuzz Tests with IoSpy and IoAttack*, Microsoft, 2017, https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack.
[9] Xst3nZ, *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/.
[10] Cr4sh, *IOCTL Fuzzer - Windows kernel drivers fuzzer.*, 2011, https://github com/Cr4sh/ioctlfuzzer.
[11] H. S. Han and K. C. Sang, "Imf: Inferred model-based fuzzer," in *ACM Sigsac Conference*, 2017, pp. 2345–2358.
[12] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," 2017.
[13] D. Jones, "Trinity: A system call fuzzer," in *Proceedings of the 2011 Ottawa Linux Symposium*, 2011.
[14] *Syzkaller - linux syscall fuzzer*, Google, 2017, https://github.com/google/syzkaller.
[15] L. Butti and J. Tinns, "Discovering and exploiting 802.11 wireless driver vulnerabilities," *Journal in Computer Virology*, vol. 4, no. 1, pp. 25–37, 2008.
[16] M. Mendonca and N. Neves, "Fuzzing wi-fi drivers to locate security vulnerabilities," in *IEEE High Assurance Systems Engineering Symposium*, 2007, pp. 379–380.
[17] R. S. ergej Schumilo and H. Schwartke, "Don't trust your usb! how to find bugs in usb device drivers," in *Blackhat Europe (2014)*, 2014.
[18] K. Y. Sim, F. C. Kuo, and R. Merkel, "Fuzzing the out-of-memory killer on embedded linux:an adaptive random approach," in *ACM Symposium on Applied Computing*, 2011, pp. 387–392.
[19] R. Van Tonder and H. Engelbrecht, "Lowering the usb fuzzing barrier by transparent two-way emulation," in *Proceedings of the 8th USENIX conference on Offensive Technologies*, 2014, pp. 13–13.
[20] V. M. Weaver and D. Jones, "perffuzzer: Targeted fuzzing of the perf vent open() system call," in *Technical Report UMAINEVMW-TR-PERF-FUZZER,University of Maine*, 2015.
[21] S. Lejay, *Fuzzing IOCTLs with angr*, 2016, https://thunderco.re/project/security/2016/07/18/fuzzing-ioctls/.
[22] M. Zalewski, *American Fuzzy Lop.*, 2014, http://lcamtuf.coredump.cx/afl.
[23] E. Bendersky, *pyelftools*, 2019, https://github.com/eliben/pyelftools.git.
[24] Samsung, *Source code*, 2019, http://opensource.samsung.com/reception.do.
[25] Amazon, "Source code," 2019, https://www.amazon.com/gp/help/customer/display.html.
[26] Huawei, *Source code*, 2019, https://consumer.huawei.com/en/opensource/.
[27] Sony, *Source code*, 2019, https://developer.sony.com/develop/open-devices/downloads/open-source-archives/.
[28] Yota, *Source code*, 2018, https://github.com/SteadyQuad/android_kernel_yotaphone2.
[29] Xiaomi, *Source code*, 2019, https://github.com/MiCode/Xiaomi_Kernel_OpenSource.

## APPENDIX A: CASE STUDY OF WHITE-BOX FUZZING

We select 7 bugs found by our white-box fuzzing method and publish them in CVE site. The bugs are listed in table 9.

A detailed analysis of the bug CVE-2018-11020 found in OMX offloading remote processor driver in Kindle Fire HD 3rd tablet was provided in this section. Difuze failed to recover the driver name for the name is assigned in runtime and also failed to recover the complete struct definitions of related sub structs for the sub structs each has lots of sub sub structs. Difuze tried to recover arguments for the driver relating to hundreds of structs, most of which are not necessary. Our method dynamically recovered the driver name and used the main structs to generate fuzzing arguments and the bug was triggered.

```
static
long rpmsg_omx_ioctl(struct file *filp,
unsigned int cmd, unsigned long arg)
{
...
  case OMX_IOCIONUNREGISTER:
  {
    struct ion_fd_data data;
    struct rpmsg_buffer *buffer;

    if (copy_from_user(&data,
    (char __user *) arg, sizeof(data))) {
      dev_err(omxserv->dev,
        "\%s: \%d: copy_from_user fail:
        \%d\n", __func__,
        _IOC_NR(cmd), ret);
      return -EFAULT;
    }
    buffer=(struct rpmsg_buffer *)data.handle;
    if (_rpmsg_buffer_validate(omx, buffer))
      _rpmsg_buffer_free(omx, buffer);
    else
      ion_free(omx->ion_client,data.handle);
    if (copy_to_user((char __user *) arg,
     &data, sizeof(data)))
     {
      dev_err(omxserv->dev,
        "\%s: \%d: copy_to_user fail:
```

TABLE IX
DETAILS OF BUGS FOUND BY WHITE-BOX FUZZING METHOD

| CVE ID | Device Name | Command | Driver Module | Bug Type |
|---|---|---|---|---|
| CVE-2018-11019 | /dev/gcioctl | 3221773726 | misc/gcx/gcioctl/gcif.c | NULL pointer dereference |
| CVE-2018-11020 | /dev/rpmsg-omx1 | 3221772291 | rpmsg/rpmsg_omx.c | Bad paging request |
| CVE-2018-11021 | /dev/dsscomp | 1118064517 | video/omap2/dsscomp/device.c | NULL pointer dereference |
| CVE-2018-11022 | /dev/gcioctl | 3224132973 | misc/gcx/gcioctl/gcif.c | Bad paging request |
| CVE-2018-11023 | /dev/gcioctl | 3222560159 | misc/gcx/gcioctl/gcif.c | Deadlock |
| CVE-2018-11024 | /dev/gcioctl | 1077435789 | misc/gcx/gcioctl/gcif.c | Bad paging request |
| CVE-2018-11025 | /dev/twl6030-gpadc | 24832 | mfd/twl6030-gpadc.c | Bad paging request |
| CVE-2019-8413 | /dev/elliptic0 | 1074316661 | elliptic/elliptic.c | NULL pointer dereference |

```
        \%d\n", __func__,
       _IOC_NR(cmd), ret);
      return -EFAULT;
    }
    break;
  }
...
void ion_free(struct ion_client *client,
struct ion_handle *handle)
{
 bool valid_handle;
 BUG_ON(client != handle->client);
 ...
```

This bug was found by our system on Amazons kindle fire pad. The ioctl function for the driver is rpmsg_omx_ioctl, following the common design of ioctl. The application in user space specifies command value and argument content.

Given the command OMX_IOCIONUNREGISTER, the code enters the corresponding case statement of rpmsg_omx_ioctl. Following the control flow of this ioctl function, the ion_free statement is reached, where the data.handle variable is used as the second argument of function ion_free. The data is used as a pointer type without validating the value in function ion_free. If a userspace application provides an invalid value here, the kernel will be failed to handle kernel paging request at the invalid virtual address.

## APPENDIX B: CASE STUDY OF BLACK-BOX FUZZING

With the manufacturers' permission, we publish 1 of the bugs on the CVE site[5]. The driver of device file /dev/block-/mmcblk0rpmb in the kernel of Qiku 360 Phone N6 Pro 1801-A01 device allows attackers to cause a denial of service attack (NULL pointer dereference and mobile phone crash) via a crafted 0xc0d8b300 ioctl call. However, the detailed reason of this bug is hard to figure out for the lack of kernel source.

To show the ability of triggering bugs powered by our black box method, we demonstrate an example of bugs that were found by our black box method. There 3 bugs exist in OPPO A37m with kernel version 3.10.72, one of them was reported earlier (CVE-2016-6492) and another two remain secret. Both of them are repaired in the latest updates. The procedure of finding the bugs of our black box method is as follows: We first extract the boot image and kallsyms from OPPO A37m. Then we disassemble interesting ioctl related functions including $FDVT\_ioctl$ as shown in Fig 6. The assembly instructions of $FDVT\_ioctl$ contain commands values as their constants.

[5]http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2018-18318

```
stp    x29, x30, [sp,#-144]!
lsr    w0, w1, #30
mov    x29, sp
stp    x19, x20, [sp,#16]
stp    x21, x22, [sp,#32]
stp    x23, x24, [sp,#48]
stp    x25, x26, [sp,#64]
str    x27, [sp,#80]
mov    w19, w1
mov    x22, x2
cbz    w0, 0x3732f4
tbnz   w19, #30, 0x3733d8
mov    w0, #0x4e03
movk   w0, #0x4018, lsl #16
cmp    w19, w0
b.eq   0x37373c
mov    w0, #0x4e03
movk   w0, #0x4018, lsl #16
cmp    w19, w0
b.ls   0x37337c
mov    w0, #0x4e02
movk   w0, #0x8004, lsl #16
cmp    w19, w0
b.eq   0x373618
mov    w0, #0x4e04
movk   w0, #0xc018, lsl #16
cmp    w19, w0
b.eq   0x373454
mov    w0, #0x4e05
movk   w0, #0x4018, lsl #16
cmp    w19, w0
b.eq   0x373754
```

Fig. 6. A Part of Disassembly Code of Function $FDVT\_ioctl$

```
static int MT6573FDVT_SetRegHW
(MT6573FDVTRegIO*a_pstCfg)
{
    MT6573FDVTRegIO *pREGIO = NULL;
    u32 i=0;
    static UINT8 illegalWRLogTimes = 0;

    if (NULL == a_pstCfg) {
        LOG_DBG("Null input argrment \n");
        return -EINVAL;
    }

    pREGIO = (MT6573FDVTRegIO*)a_pstCfg;

    if(copy_from_user(
    (void*)pMT6573FDVTWRBuff.u4Addr,
    (void *) pREGIO->pAddr,
    pREGIO->u4Count * sizeof(u32))) {
      LOG_DBG("ioctl copy from user failed\n");
    return -EFAULT;
    }

    if(copy_from_user(
    (void*)pMT6573FDVTWRBuff.u4Data,
    (void *) pREGIO->pData,
    pREGIO->u4Count * sizeof(u32))) {
      LOG_DBG("ioctl copy from user failed\n");
       return -EFAULT;
   }
   return 0;
}
```

For example, instructions at position 13 and 14 give low 16 bits and high 16 bits of command 0x40184e03. Seven commands (19968, 19969, 2147765762, 1075334659, 3222818308, 1075334661, 20096) were recovered at this step and added to candidate command set. After the matching step, we find the 7 commands are valid after applying ioctl on device /dev/camera-fdvt. Then we use Mango to fuzz the ioctl interface on /dev/camera-fdvt. Finally, we get 3 crashes when using command 3222818308, 1075334661 and 1075334659.

We then analyze the crash log and found the bug caused by command 1075334659 is reported in CVE-2016-6492. The command 1075334659 is the value of $MT6573FDVTIOC\_T\_SET\_FDCONF\_CMD$ which is used to set registers of MT6573FDVT device. The source code related to CVE-2016-6492 is in a camera driver of Mediatek named $camera\_fdvt$.c.

```
static int MT6573FDVT_ReadRegHW
(MT6573FDVTRegIO*a_pstCfg)
{
    int ret = 0;
    int size = a_pstCfg->u4Count * 4;
    int i;

    if (size > buf_size)
        LOG_DBG("size too big\n");

    if (copy_from_user(
    pMT6573FDVTRDBuff.u4Addr,
    a_pstCfg->pAddr, size) != 0) {
        LOG_DBG("copy_from_user failed\n");
        ret = -EFAULT;
        goto mt_FDVT_read_reg_exit;
    }
...
  if (copy_to_user(a_pstCfg->pData,
  pMT6573FDVTRDBuff.u4Data, size) != 0)
  {
        LOG_DBG("copy_to_user failed\n");
        ret = -EFAULT;
        goto mt_FDVT_read_reg_exit;
    }
  mt_FDVT_read_reg_exit:
    return ret;
}
```

Function $MT6573FDVT\_SetRegHW$ uses $copy\_from\_user$ without checking the length of copied data. The length is determined by pREGIO $->$ u4Count and can be used by user to gain privileges via a crafted payload.

The command 1075334661 is corresponding to command $MT6573FDVTIOC\_T\_SET\_SDCONF\_CMD$ which is used to set another register of MT6573FDVT device and function $MT6573FDVT\_SetRegHW$ is called as the same as command 1075334659 does.

While there is no information on why command 3222818308 causes kernel crash in CVE List. We then analyze the other functions in source code file $camera\_fdvt.c$ and found that command 3222818308 is the value of command $MT6573FDVTIOC\_G\_READ\_FDREG\_CMD$ and calls another function $MT6573FDVT\_ReadRegHW$. What interesting is that, the length of copied data is checked in line 7, but just a warning message is given. The $copy\_from\_user$ function will try to copy data no matter whether the size is bigger than upper limit $buf\_size$. In case a userspace application provides a large size value, $copy\_from\_user$ will try to fetch unexpected data which may cause kernel crash.