IEICE TRANS. INF. & SYST., VOL.E104–D, NO.11 NOVEMBER 2021

PAPER Special Section on Next-generation Security Applications and Practice

CoLaFUZE: Coverage-Guided and Layout-Aware Fuzzing for Android Drivers

Tianshi MU[†a)], ***Member***, **Huabing ZHANG**[†b)], **Jian WANG**[†c)], ***and*** **Huijuan LI**[†d)], ***Nonmembers***

SUMMARY With the commercialization of 5G mobile phones, Android drivers are increasing rapidly to utilize a large quantity of newly emerging feature-rich hardware. Most of these drivers are developed by third-party vendors and lack proper vulnerabilities review, posing a number of new potential risks to security and privacy. However, the complexity and diversity of Android drivers make the traditional analysis methods inefficient. For example, the driver-specific argument formats make traditional syscall fuzzers difficult to generate valid inputs, the pointer-heavy code makes static analysis results incomplete, and pointer casting hides the actual type. Triggering code deep in Android drivers remains challenging. We present CoLaFUZE, a coverage-guided and layout-aware fuzzing tool for automatically generating valid inputs and exploring the driver code. CoLaFUZE employs a kernel module to capture the data copy operation and redirect it to the fuzzing engine, ensuring that the correct size of the required data is transferred to the driver. CoLaFUZE leverages dynamic analysis and symbolic execution to recover the driver interfaces and generates valid inputs for the interfaces. Furthermore, the seed mutation module of CoLaFUZE leverages coverage information to achieve better seed quality and expose bugs deep in the driver. We evaluate CoLaFUZE on 5 modern Android mobile phones from the top vendors, including Google, Xiaomi, Samsung, Sony, and Huawei. The results show that CoLaFUZE can explore more code coverage compared with the state-of-the-art fuzzer, and CoLaFUZE successfully found 11 vulnerabilities in the testing devices.

key words: *driver fuzzing, structure layout, interface recovery*

1. Introduction

Mobile phones have become our gateways to the connected world. If criminals took control of this device, they could gain enormous power of destruction. Therefore, in order to ensure the safety and reliability of smartphones, researchers have made great efforts. This security is achieved by using a complex application sandbox, which utilizes many attack mitigation techniques for userspace applications, and by regarding security as the most important indicator during the development process. However, the kernel has become one of the weaknesses of the security of mobile phones. Userspace applications have available protection measures, but the modern operating system kernel is relatively vulnerable even if existing protection measures are applied. Therefore, as vulnerabilities in userspace applications become more difficult to discover and exploit, attackers begin to pay attention to kernel vulnerabilities.

The kernel is mainly composed of two types of codes: core kernel code and device driver code. The core kernel code is accessed through the system calls (syscall), allowing users to read and write files (open and write system calls), send and receive messages (send and recv system calls) and manage programs (execve, fork, wait system calls), etc. The driver code on the Android system is usually accessed through the IOCTL interface. This interface is a specific system call that allows the device driver to process the input data.

According to Google's data report, 85% of the Android kernel vulnerabilities are located in driver module codes and developed by third-party device vendors [1]. With the ever-increasing number of mobile phones and the importance of their security, methods for automatically discovering these bugs before attackers exploit them are crucial. Although the interaction with syscalls follows a unified function prototype, the interaction rules with IOCTL depend on the associated device driver. Specifically, the IOCTL interface contains a set of valid commands and corresponding structured parameters. The commands and data structures are determined by the driver developer. This custom, variable, and complex structure brings security risks and makes these codes difficult to analyze. To be effective, any automatic analysis and interaction of these devices must provide the command identifiers and data structures expected by IOCTL.

Thanks to the open source nature of Android [2]–[7], existing approaches have been proposed by analyzing the kernel code. Such as DIFUZE [8] has proposed recovering data structures of IOCTL statically. However, DIFUZE can not handle dynamically generated device names or allocated objects. Furthermore, it lacks coverage information to guide mutations. Charm [9] ports the Android drivers to a x86_64 workstation and accesses the physical device through USB. However, it requires developers to spend days and weeks porting the driver code. EASIER [10] enables off-device analysis by creating an ex-vivo dynamic driver analysis framework for Android devices without porting nor emulation. Nevertheless, only part of the drivers can be initialized in the framework, and false positives cause trouble. Exploring Android driver bugs remains a challenging problem.

To address this problem, we present CoLaFUZE, a technique to enable coverage-guided and structure-layout-aware fuzzing. CoLaFUZE performs an automated dynamic

Manuscript received February 7, 2021.
Manuscript revised May 11, 2021.
Manuscript publicized July 28, 2021.

[†]The authors are with the Digital Grid Research Institute, China Southern Power Grid, China.
a) E-mail: muts@csg.cn
b) E-mail: zhanghuabing@csg.cn
c) E-mail: wangjian3@csg.cn
d) E-mail: lihj1@csg.cn

DOI: 10.1587/transinf.2021NGP0005

analysis of kernel code to recover the device names and symbolic execution to recover the specific IOCTL commands, used to trigger the execution of the driver code. CoLaFUZE recovers structure layout by handling data copy operations on the runtime. The fuzzing engine generates the required size of data and puts them in the desired location, matching the structure layout. Furthermore, the fuzzing engine leverages the coverage information to guide seed mutation.

In summary, this paper has made 3 separate contributions:

Layout-aware driver fuzzing. We provide a novel method to redirect the data copy operations, forward data flow and recover structure layout during fuzzing.
Coverage-guided driver fuzzing. We present a way to feedback the kernel KCOV information to userspace fuzzing engine, enabling the fuzzing engine to mutate seeds effectively.
CoLaFUZE prototype. We introduce a framework to recover driver IOCTL interfaces dynamically and develop a fuzzing tool to explore driver code bugs. We conducted experiments, analyzed 5 mobile phones and found 11 unique vulnerabilities.

2. Technical Background

In this section, we give a brief introduction of device drivers, the IOCTL system call, kernel-userspace data copy operations and code coverage of fuzzing.

2.1 IOCTL System Call

POSIX (Portable Operating System Interface) [11] is a set of standard operating system interfaces based on the Unix operating system. It establishes an interface standard for the interaction between userspace applications and device drivers, including the IOCTL system call. Through the IOCTL system call, the userspace program can issue different operation requests to the driver and transmit and receive formatted information in the form of a structure.

Generally, the prototype IOCTL system call is

int ioctl(int fd, unsigned long request, ...).

The first argument is a file id of an opened device file, and the second is a request field, a set of command values set by the driver. Each command corresponds to a unique feature of the driver. The third parameter is a pointer, which usually points to a structure defined by the driver developer, and the structure may change when the command value is different. The structure may refer to other structures in a field, causing nested structures. The diversity of IOCTL system call parameters brings developers the possibility of implementing rich functions, and at the same time brings security challenges, attracting many researchers to test and analyze it [12]–[15].

2.2 Kernel-Userspace Data Copy Operation

A parameter pointer must be provided when the IOCTL system call is initiated. By copying data to and from the structure pointed to by the parameter, the driver exchanges data with the userspace program. The kernel provides copy_from_user()/copy_to_user() functions, which accept the source pointer, destination pointer and the number of bytes. The driver must use these functions to ensure the safe copying of data. These functions first perform a security check to ensure that the source address and destination address pointers are in a legal address space.

2.3 Code Coverage

KCOV is a kind of kernel code coverage information used coverage-guided fuzzing [16]. Userspace programs can obtain kernel coverage data through the ”kcov” debugfs file. Coverage collection is based on task granularity, so the userspace program can capture the coverage of a single system call it initiates. KCOV collects more or less stable coverage instead of collecting as much coverage as possible. To achieve this goal, it discards the coverage of soft/hard interrupts and disables the detection of certain uncertain parts of the kernel (such as schedulers, locks). In short, KCOV is stable and suitable for system call fuzzing.

2.4 I2C Driver Example

We take the I2C driver as an example to show how CoLaFUZE automatically recovers the reference relation of these structures and performs layout-aware fuzzing.

I2C is a slow two-wire protocol (variable speed, up to 400 kHz), with a high-speed extension (3.4 MHz). It provides an inexpensive bus for connecting many types of devices with infrequent or low bandwidth communications needs. I2C is widely used with Android systems.

The driver code is presented in Listing 1 (the structure definitions), 2 (the function used to duplicate memory region from userspace), 3 (part of the IOCTL handler function). As shown in Listing 3, function i2c_ioctl_rdwr first

```
1  /* struct i2c_msg - an I2C transaction segment */
2  struct i2c_msg {
3    __u16 addr;     /* slave address            */
4    __u16 flags;
5    __u16 len;      /* msg length               */
6    __u8 *buf;      /* pointer to msg data      */
7  };
8
9  /* This is the structure as used in the I2C_RDWR
10  * ioctl call */
11 struct i2c_rdwr_ioctl_data {
12   struct i2c_msg __user *msgs;
13                   /* pointers to i2c_msgs     */
14   __u32 nmsgs;    /* number of i2c_msgs       */
15 };
```

Listing 1 The structure definitions used by function i2c_ioctl_rdwr

```
1  void *memdup_user(const void __user *src, size_t
2    len)
3  {
4      void *p;
5
6      p = kmalloc_track_caller(len, GFP_KERNEL);
7      if (!p)
8          return ERR_PTR(-ENOMEM);
9
10     if (copy_from_user(p, src, len)) {
11         kfree(p);
12         return ERR_PTR(-EFAULT);
13     }
14
15     return p;
16 }
```

Listing 2 The function to duplicate memory region from userspace

```
1  static noinline int i2cdev_ioctl_rdwr(
2    struct i2c_client *client, unsigned long arg)
3  {
4    struct i2c_rdwr_ioctl_data rdwr_arg;
5    struct i2c_msg *rdwr_pa;
6    u8 __user **data_ptrs;
7    int i, res;
8
9    if (copy_from_user(&rdwr_arg,
10       (struct i2c_rdwr_ioctl_data __user *)arg,
11       sizeof(rdwr_arg)))
12     return -EFAULT;
13
14   /* Put an arbitrary limit on the number of
15    * messages that can be sent at once */
16   if (rdwr_arg.nmsgs > I2C_RDWR_IOCTL_MAX_MSGS)
17     return -EINVAL;
18
19   rdwr_pa = memdup_user(rdwr_arg.msgs,
20     rdwr_arg.nmsgs * sizeof(struct i2c_msg));
21   if (IS_ERR(rdwr_pa))
22     return PTR_ERR(rdwr_pa);
23
24   data_ptrs = kmalloc(rdwr_arg.nmsgs *
25     sizeof(u8 __user *), GFP_KERNEL);
26   if (data_ptrs == NULL) {
27     kfree(rdwr_pa);
28     return -ENOMEM;
29   }
30
31   res = 0;
32   for (i = 0; i < rdwr_arg.nmsgs; i++) {
33     /* Limit the size of the message to a sane
34      * amount */
35     if (rdwr_pa[i].len > 8192) {
36       res = -EINVAL;
37       break;
38     }
39
40     data_ptrs[i] = (u8 __user *)rdwr_pa[i].buf;
41     rdwr_pa[i].buf = memdup_user(data_ptrs[i],
42       rdwr_pa[i].len);
43     if (IS_ERR(rdwr_pa[i].buf)) {
44       res = PTR_ERR(rdwr_pa[i].buf);
45       break;
46     }
47     ...
```

Listing 3 The handler function of the I2C_RDWR IOCTL call

copies the data from userspace to a local variable rdwr_arg, a i2c_rdwr_ioctl_data struct (lines 9-11). Then it invokes function memdup_user to copy data to the buffer rdwr_arg.msgs with the size of a multiplication result (lines 19-20). After it allocates memory space, it enters a loop for fetching data from userspace (lines 41-42). Notice that, the size of the second and third data copy operations are both dynamic, which is beyond the ability of the static method.

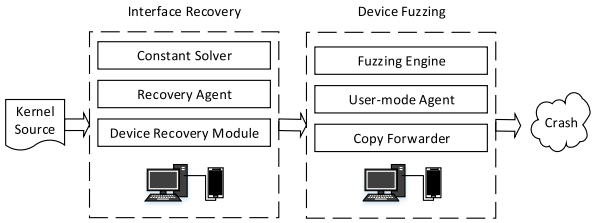


Fig. 1 The CoLaFUZE approach diagram.

3. Overview

We now provide a high-level overview of the design of CoLaFUZE and its practical application to exploring bugs in Android drivers through IOCTL fuzzing. CoLaFUZE requires the kernel code of the mobile phones, released by the vendors according to the GNU General Public License.

As shown in Fig. 1, the workflow of CoLaFUZE composes of two stages: interface recovery and device fuzzing. At the interface recovery stage, CoLaFUZE first uses its device recovery module and recovery agent to obtain the device file names and handler functions to be fuzzed from the kernel space on the fly. Then, CoLaFUZE leverages the constant solver to perform symbolic execution on the obtained handler source code and restores the predefined IOCTL command values.

During the device fuzzing stage, CoLaFUZE uses the copy forwarder to forward the data copy operations from userspace of the driver to the fuzzing engine. Thus the fuzzing data to IOCTL system call is passed through the copy operations, instead of the third argument, enabling CoLaFUZE to feed the driver with the required amount of data, even if the structure definition is unknown. Then, by comparing the source address of data copy operations, the structure layout can be determined. Also, coverage data can be obtained and analyzed for seed selection and mutation with the KCOV module. All the components make CoLaFUZE a coverage-guided and layout-aware fuzzing framework.

4. Interface Recovery

4.1 Recovery Module and Agent

Linux treats everything as files including hardware devices. All hardware files are present in the device folder (/dev) as file-like device nodes which point to different parts of the system (a device driver). Userspace applications can use these device nodes to interface with the system hardware.

The first argument of IOCTL is a file descriptor corresponding to an opened file in the device folder. According to the file descriptor, the IOCTL system call routes the arguments to the drivers.

As shown in Algorithm 1, the recovery module first acquires a kernel struct fd according to the provided file descriptor. The field file of struct fd is another kernel struct file. By accessing the field fop in struct file, the corre-

sponding struct file_operations is obtained, and then the unlocked_ioctl function pointer is obtained.

The recovery module is loaded in kernel space, and the handler address information needs to be transferred to the userspace. A recovery agent is needed to get the address of the IOCTL handler to userspace. As shown in Algorithm 2, firstly, the recovery agent opens the files in the device folder (/dev) and gets the address of the unlocked_ioctl by calling the recovery module. Then it searches the address in the kernel symbol table to get the function name. At last, the function name is added to an ioctl list. The interactions between the recovery module and the recovery agnet is shown in Fig. 2.

With the recovery module and agent, the recovery agent gets a map from a file name in the device folder (/dev) to a handler function name (identified in /proc/kallsyms) on the runtime without analyzing the kernel source code, which improves the degree of automation of the proposed method.

4.2 Command Value Solver

The second argument of IOCTL is a device-dependent request code. Recovering IOCTL commands is necessary since they are usually hard to guess by the fuzzer. Recovering the command values is a typical value-set analysis [17] problem, which can be done in multiple approaches like symbolic execution, range analysis, parsing debug information and pattern matching. We develop a command value solver based on angr [18], a platform-agnostic binary analysis framework for program instrumentation, symbolic execution, control-flow analysis, data-dependency analysis, value-set analysis (VSA), etc. We notice that the command value is only used in the device driver code to determine which operation to take. We extend angr by replacing kernel functions like printk, copy_from_user, copy_to_user, with custom symbolic implementations. Specifically, we replace memory allocation functions with custom symbolic memory allocator. The existing approach to recover IOCTL command values is based on the common convention of having a large switch statement inside IOCTL handlers. However, many drivers use a function table to check the command value like the drm device driver[†]. With our command value solver based on angr, we can do symbolic execution on the kernel driver module and recover the command value set.

We do not recover the third argument structure definition for 2 reasons:

1. Statically recovered definitions are not complete, for pointer casting is frequently used in kernel code, the variable may be cast to another type on the runtime, and dynamically allocated objects and arrays are undetermined in source code;
2. For structure definitions are various, statically recovering structure must handle various code style of driver vendors and must be updated when new code is released, require tools like clang, gcc and a large amount of time.

As a result, we make use of code coverage and structure layout to facilitate seed generation and mutation. Now that the first two arguments are obtained and used to trigger the dynamic execution of the driver IOCTL handler in question.

Algorithm 1 Algorithm of recovery module in kernel

Input: fd: a file descriptor
Output: addr: address of unlocked_ioctl
1: struct fd f = fdget(fd) *//fdget is a kernel function*
2: struct file * filp = f.file
3: **if** filp AND filp->f_op **then**
4: return filp->f_op->unlocked_ioctl
5: **end if**
6: return null

Algorithm 2 Algorithm of recovery agent in userspace

Input: None
Output: ioctl_list: array of IOCTL function names
1: **for all** f in dir(/dev) **do**
2: fd = open(f)
3: addr = recover_module(fd)
4: name = search addr in /proc/kallsyms
5: put name in ioctl_list
6: **end for**

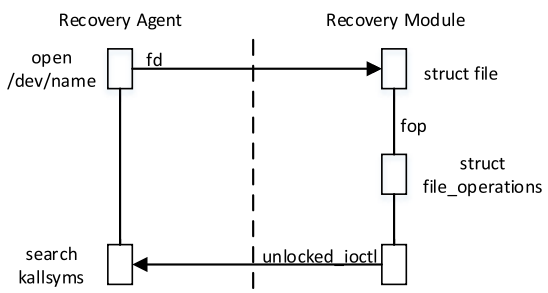


Fig. 2 The interactions between the recovery module and the recovery agent.

5. Device Fuzzing

The device fuzzing consists of three components: the fuzzing engine, the user-mode agent and the data copy forwarder. The fuzzing engine runs in userspace on the analysis host, which generates inputs, processes the coverage data and mutates new inputs. The user-mode agent runs in userspace on the target Android mobile phone, which synchronizes and gathers new inputs from the fuzzing engine and interacts with the target kernel. The data copy forwarder is loaded in the Android kernel in the setup stage and forwards the data copy operation from the driver to the user-mode agent. The user-mode agent and the data copy forwarder establish a data path between the fuzzing engine and the kernel driver. The KCOV module of the kernel is enabled when the Android system boots.

Figure 3 demonstrates the actions and communica-

[†]https://elixir.bootlin.com/linux/v4.1/source/drivers/gpu/drm/drm_ioctl.c

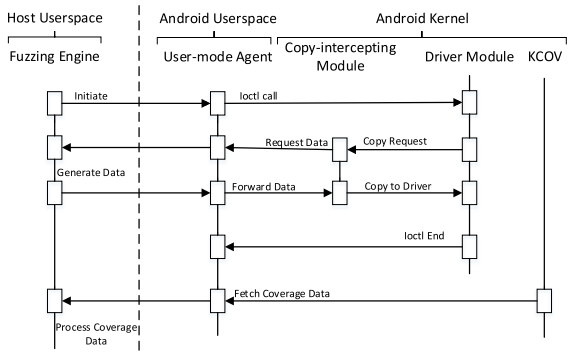


Fig. 3 The interactions during a fuzzing round.

tion during a fuzzing run. The fuzzing engine first initiates a request to the user-mode agent with a recovered file name, a recovered command value and a random value. When receiving parameters, the user-mode agent launches an IOCTL system call and triggers the handler function in the kernel driver code. The driver code checks the command value and tries to fetch the data by calling copy_from_user_forwarder, implemented in the data copy forwarder module. The user-mode agent forwards the request to the fuzzing engine. The fuzzing engine generates the fuzzing data according to the request size and passes it to the user-mode agent. The user-mode agent forwards the data to the data copy forwarder. Then the kernel driver gets the data to be handled and continues to execute. When the IOCTL system call returns, the user-mode agent will get the coverage data from 'kcov' debugfs offered by KCOV and pass it to the fuzzing engine. The fuzzing engine will process the coverage information and determine if the seed is interesting.

5.1 Fuzzing Engine

The fuzzing engine is the core control component, which manages the seed queue, recovers the structure layout, processes coverage information, schedules the tasks, generates and mutates seeds. We implemented the fuzzing engine based on the algorithms proposed by AFL [19]. Unlike the AFL bitmap, we use KCOV coverage data from the kernel KCOV module to decide which inputs triggered interesting behaviors.

Another important feature of the fuzzing engine is that it recovers the layout of structures dynamically and uses it to guide the layout-aware seed generation on request. We notice that the Linux kernel provides a family of functions like copy_from_user to copy data between userspace and kernel space. A kernel module can not access userspace memory directly but copy the required memory through this family of functions. The data copy forwarder forwards this family of functions to the user-mode agent and again to the fuzzing engine. In this way, a data channel between the fuzzing engine and the kernel driver in question is established.

At the beginning of a fuzzing run, the fuzzing engine issues an IOCTL system call with arg set to arbitrary values. When a data copy operation is forwarded to the fuzzing engine, it generates the desired size of memory dynamically. If multiple copy operations are received during one fuzz run, the fuzzing engine is aware that the driver may need a nested structure to handle the current command. The fuzzing engine then searches the source address of the following copy operations in previously generated data to identify the structure pointer offset.

Take the i2c driver as an example. The first field of struct i2c_rdwr_ioctl_data is a pointer to type struct i2c_msg, and the second is a 32-bit unsigned type and the number of i2c_msgs. The field len is a 2-byte unsigned number, indicating the message length. The buf is an unsigned char pointer, which points to the message data. The structure layout is shown in Listing 1. To recover this structure layout, the fuzzing engine first initiates an IOCTL system call request and sets the file name parameter to /dev/i2c-0, the command value to 0x80024001 (I2C_RDWR), and argp to 0x40000000 (a random value). Then the driver enters function i2cdev_ioctl_rdwr. At line 9 of Listing 3, the driver tries to copy data from argp (0x40000000) in userspace. Then the fuzzing engine receives a data request to copy the 16-byte (sizeof rdwr_arg) data at address 0x40000000. The fuzzing engine dynamically generates a 16-byte (size of rdwr_arg) data array and returns it to the user-mode agent. The function i2cdev_ioctl_rdwr continues to line 19 and invokes function memdup_user to duplicate memory from userspace. The fuzzing engine receives data requests to copy x-byte (nmsgs * sizeof i2c_msg) data. The fuzzing engine dynamically generates a x-byte (nmsgs * sizeof i2c_msg) data array and returns it to the user-mode agent. The function i2cdev_ioctl_rdwr continues to line 41 and invokes function memdup_user to duplicate memory from userspace. Also, the fuzzing engine receives data requests one by one, requesting to copy n-byte (rdwr_pa[i].len) data. The fuzzing engine dynamically generates an n-byte (rdwr_pa[i].len) data array and returns it to the user-mode agent. Finally, the function i2cdev_ioctl_rdwr continues to the end and returns. The fuzzing engine records the generated data and source address of each copy operation. By comparing the source in the generated data, the fuzzing engine figures out the nesting relation of the data structure as shown in the Listing 1.

Note that our approach is insensitive to how the length was computed or whether it depends on other user input. Such dependencies can pose a problem to static analysis, but not for our dynamic recovery method. The fuzzing engine continues this algorithm recursively, which allows it to allocate the right amount of memory and deal with nested pointers.

5.2 User-Mode Agent

As the fuzzing engine runs on the host and the driver runs in the Android kernel. This user-mode agent is required to facilitate the fuzzing process. In principle, the user-mode agent must synchronize and forward new inputs from the

Algorithm 3 Algorithm of data copy forwarder in kernel

Input: destination *to*; source *from*; size *n*
Output: size of copied: *n*
1: submit_to_uma(*from*, *n*) // to user-mode agent
2: new_from = wait_uma() // wait user-mod agent for data
3: return copy_from_user(*to*, *new_from*, *n*)

fuzzing engine to the Android kernel and fetch the KCOV information of the target Android kernel to the fuzzing engine. This component connects to the fuzzing engine through a network provided by adb forwarded socket connections. The actions and messages take place in the socket connection during the fuzzing run.

On the other side, it synchronizes with the data copy forwarder through a proc entry in proc file system, for the proc file system acts as an interface to internal data structures in the kernel and can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl).

5.3 Data Copy Forwarder

The traditional driver fuzzing method needs to generate complete input when initiating the IOCTL system call. However, due to the dynamic characteristics of the driver module, the data requirements may be the result of dynamic calculations. The data copy forwarder contains a wrapper of the function copy_from_user, to forward the data copy operations to the user-mode agent. On the other side, the forwarder transmits the data to the driver module, when the required data from the user-mode agent is ready.

As shown in Algorithm 3, the data copy forwarder submits the required source address and size to the user-mode agent and waits for a new source address. The user-mode agent forwards the request to the fuzzing engine, puts the fuzzing data in the target location and then informs the data copy forwarder. When the data is ready, the data copy forwarder calls the build-in copy_from_user to complete the data copy operation.

6. Evaluation

In this section, we answer the following three questions: 1) whether the interface recovery method can effectively recover the driver IOCTL arguments; 2) whether the fuzzing framework can improve the fuzzing performance; 3) whether CoLaFUZE can detect new vulnerabilities in Android device drivers.

6.1 Interface Recovery Evaluation

The experiments were taken on a machine running Ubuntu 18.04 with an Intel Xeon Gold 6128 processor and 64 GB DDR4 RAM. The devices we used to evaluate our interface recovery method is listed in Table 1, including products of Google, Samsung, Huawei, Sony and Xiaomi.

Table 1 Android devices used for evaluation. We choose products of the top mobile phone vendors and chipset vendors across the world.

	Android	Kernel	Chipset
Pixel 4	9	4.14.114	Snapdragon 670
Samsung A51	10	4.14.113	Exynos 980
Honor 20	10	4.14.116	Kirin 710F
Xperia 10	9	4.14.16	Snapdragon 630
Mi Play	9	4.9.77	MediaTek P35

The IOCTL handlers are recovered by the kernel recovery module of CoLaFUZE, which runs in kernel space and extracts the kernel address of handlers. The valid results recovered is shown in Table 2. In total, 504 IOCTL handlers are identified by CoLaFUZE in the testing mobile phone, corresponding to the number of running drivers. The command value solver of CoLaFUZE extracts the commands by angr, and the number of recovered command values is positively correlated with the number of IOCTL handlers. For the device names are dynamically extracted by traversing the device folder (/dev), hundreds of block and character devices are gathered. We then remove the devices without IOCTL handlers, which can only be operated by reading or writing. In total, 512 valid device names are selected out.

To show the effectiveness of the proposed interface recovery method, we compare the recovery result of CoLaFUZE against the result of DIFUZE, an interface-aware kernel fuzzing tool. We run the open source version of DIFUZE[†] from github on the kernel code of the mobile phones. As DIFUZE is not actively maintained, it can not fully support these new devices. We developed patches and extended them by adjusting compiler flags, adding new driver file patches, new structures used to register an IOCTL handler and so on. After the porting work, we use DIFUZE to recover the interfaces from the code of testing devices.

We found that, even if DIFUZE successfully recovered the correct interface information, the recovered interface may be useless. The dynamically running kernel on the mobile phone does not actually activate the driver module in the source code. In other words, the driver module is redundant. It is mainly because some vendors of smartphones released the code that works, but not the tidiest one. This causes problems to the static method DIFUZE, but not to the dynamic method CoLaFUZE.

The recovered interface is valid only if it is recovered correctly and does be activated on the testing mobile phones. Compared with DIFUZE, CoLaFUZE recovers more valid interfaces, especially on the aspect of device name recovery. As the device names could be dynamically generated instead of hard-coded in the source. DIFUZE recovers less than 65.8% of the total device names recovered by CoLaFUZE.

We evaluate the interface recovery results by manually extracting a random sampling of IOCTL interfaces and compare them with the recovered results. The picked drivers contain 64 IOCTL handlers and 623 commands, of which CoLaFUZE recovers all 64 handlers and 94.06% (586) correct commands.

[†]https://github.com/ucsb-seclab/difuze

Table 2 Valid interfaces recovered on the testing mobile phones by CoLaFUZE and DIFUZE

	Valid results of CoLaFUZE			Valid results of DIFUZE		
	IOCTL handlers	Commands	Device Names	IOCTL handlers	Commands	Device Names
Pixel 4	125	589	161	105	424	147
Samsung A51	102	276	97	73	372	69
Honor 20	91	281	78	69	216	31
Xperia 10	91	533	81	76	311	44
Mi Play	95	475	95	77	334	46
Total	504	2154	512	400	1657	337

6.2 Fuzzing Performance

To determine how well CoLaFUZE can improve fuzzing performance like execution speed, code coverage and the ability to trigger bugs, we carry on a comparative evaluation of CoLaFUZE, the latest kernel fuzzers including syzkaller [20] and DIFUZE. To find out the effects of using coverage information on the ability to explore new paths, we test CoLaFUZE with and without the KCOV module. We use LaFUZE as the layout-aware fuzzing (CoLaFUZE without coverage module). Also, we test CoLaFUZE by disabling the data copy forwarder module, identified by CoFUZE, to show the effects of structure layout information. Specifically, we run the following 5 types of fuzzers:

- **Syzkaller**. Syzkaller is an unsupervised coverage-guided kernel fuzzer. We only enable syzkaller to use IOCTL and system calls it depends on. The interfaces of common Linux devices are already listed in the description files of syzkaller but no third-party devices.
- **DIFUZE**. DIFUZE is an interface aware fuzzing tool. It recovers the interface information, including data structure definitions by data-flow analysis, type propagation, constraints checking and definition extracting methods. On the fuzzing stage, it employs a fuzzer named MangoFuzz based on Peach.
- **LaFUZE**. In this configuration, we enable the data copy forwarder and the user-mode agent to perform layout-aware fuzzing. In this situation, when a data copy request is received, the correct size of data is generated and transferred to the driver.
- **CoFUZE**. This configuration enables the KCOV module of the target kernel and disables the data copy forwarder module. The fuzzing engine can obtain the coverage information, generate and mutate seeds accordingly.
- **CoLaFUZE**. Here, CoLaFUZE integrates all the modules in this paper. It performs coverage-guided and layout-aware fuzzing to validate the improvement of code coverage and the ability of bug finding.

We evaluated these fuzzers on real Android products listed in Table 1. We search and download the kernel source code from the official site of each vendor. Then we port the recovery and the data copy forwarder module, enable the KCOV and KASAN kernel module, and build the kernel image using AOSP tools [21]. To flash the generated boot image onto the physical device, the bootloader should

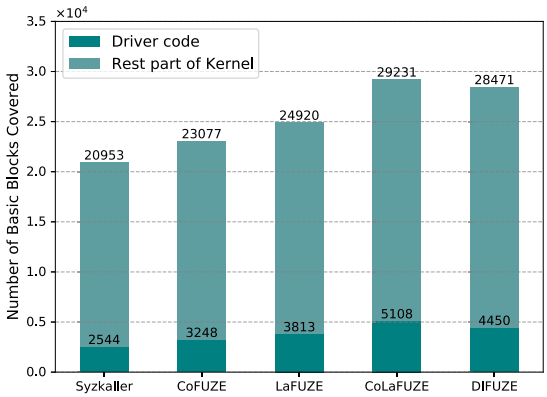


Fig. 4 The coverage results of fuzzers on Samsung S10 for 24 hours.

be unlocked [22]. The user-mode agent is running as root for some driver files can only be opened with root permissions. To compare the overall performance of these fuzzers and ensure comparable results, we run each fuzzer on Samsung S10 for 24 hours with a single process suggested in [23]. The coverage of the device drivers and the rest of the kernel of each fuzzer is shown in Fig. 4. We can see that, for lacking interface information, syzkaller can not perform effective fuzzing on drivers of mobile phones, and it covers the least basic blocks. For no third-party devices are included in the description files of syzkaller, it fails to trigger the hidden handlers. We believe that if the recovered interfaces are added to the specifications of syzkaller, the fuzzer could cover much more basic blocks. Also, the results show that CoLaFUZE achieves the most code coverage in the driver. CoLaFUZE outperforms CoFUZE due to the availability of structure layout needed for seed generation and outperforms LaFUZE due to the benefit from coverage information for seed mutation. For example, a valid input for i2cdev_ioctl_rdwr in Listing 3 requires the fileds to be valid pointers. For no layout informatioin available, CoFUZE my generate invalid address values in field data_ptrs and trigger error at line 8 in Listing 2 and line 43 in Listing 3.

Compared with CoLaFUZE, DIFUZE covers less basic blocks. We analyzed the blocks covered by CoLaFUZE but not DIFUZE and found cases that DIFUZE fails to handle. For example, a length field specifies a dynamic size array for data input, and a statement casts the integer to a pointer. However, DIFUZE does not recognize the relationships between the fields of structures, can not figure out the dynamically changing request and fails to trigger code behind. On

Table 3 The bugs found by different fuzzers per mobile phone. For each mobile phone, every fuzzer was running in 24 hours. To avoid frequently crashing and rebooting, we would blacklist the interface if it caused too many crashes.

	Syzkaller	DIFUZE	LaFUZE	CoFUZE	CoLaFUZE	Total Unique
Google Pixel 4	0	0	0	0	1	1
Galaxy S10	0	0	0	0	0	0
Honor 20	0	2	2	1	2	2
Xperia XZ3	1	3	2	1	4	4
Mi Play	1	2	2	3	4	4
Total	2	7	6	5	11	11

the other hand, DIFUZE is a generation-based fuzzer, and it generates input seeds according to the statically recovered templates, without a coverage-guided mutation stage. For example, in Listing A· 1 the type cmdqU32Ptr_t is defined as unsigned long long, DIFUZE recovers the struct cmdqCommandStruct and treats the field prop_addr as a number. DIFUZE generates random numbers instead of valid pointers at this field, causing the function to return at line 20.

6.3 Detected Vulnerabilities

Hundreds of crashes were triaged during fuzzing. The crash logs, input seeds and system calls are analyzed, manually triggered and filtered. The overall result of each fuzzer is shown in Table 3.

The default configuration of syzkaller contains only the common part of valid interfaces, which has been fully analyzed and verified. Without vendor-customized interface information, the syzkaller fuzzer finally triggered 2 bugs. This shows that blindly fuzzing the driver interface is not effective because the vendor testers may have already performed similar testing work.

CoFUZE uses the recovered interfaces by our recovery module to trigger target IOCTL handler execution and mutate inputs of the third argument. CoFUZE successfully found 5 bugs in the testing mobile phones, showing that with valid commands, the fuzzer can trigger more code execution. As no structure information and reference relations are provided, CoFUZE can only generate and mutate the bytes array of input at the beginning of each fuzzing run.

LaFUZE uses the recovered interfaces by our recovery module to trigger target IOCTL handler execution and generate inputs to copy_from_user requests. LaFUZE successfully identified 6 bugs in the mobile phones. The result suggests that layout information can effectively help to explore the driver code.

When we combine CoFUZE and LaFUZE, CoLaFUZE is able to find 11 bugs, making it the most one. Compared with DIFUZE, CoLaFUZE detects 4 more bugs. We manually analyzed these bugs and found that DIFUZE fails to generate valid pointers or data buffers. In one bug driver, an integer field is cast to userspace pointer, but DIFUZE regards this field as an integer. The generated pointer data is not valid, making the driver not able to obtain the data to process. In another bug driver, the array size is dynamically dependent on the value previously provided as in Listing 1. In each run, when DIFUZE generates a new input, the size of the second data array is specified in one field of the first struct, and DIFUZE should generate the correct size of the data array corresponding to the field. However, DIFUZE always failed to satisfy the size constraint, for the size is different in each fuzzing round. In Appendix, We present a case study of bugs triggered in Mi Play, which has been reported and confirmed, to show the necessity of the proposed methods and the reason why DIFUZE failed to find.

For the kernel source code may not be the latest, we are in the process of responsibly disclosing these vulnerabilities to corresponding vendors and checking whether the bugs are previously known, silently fixed or still present in the most recent kernel version.

6.4 Limitations

The method proposed in this paper has three main limitations. First, because of the necessity to recompile and reflash the kernel, the engineer needs to make efforts to prepare the devices ready for fuzzing. This work can not be eliminated for the production version of Android does not contain debugging features or vulnerability sanitizers. Second, if a buggy driver is causing the kernel to crash frequently, the fuzzer must wait until the kernel reboots, decreasing the fuzzing efficiency. Finally, an early bug could also prevent the fuzzer from reaching deeper bugs in the same interface. Each time the fuzzer select the bug command, the early bug will be hit, and the phone will reboot. Even if there is a deeper bug after the early bug, the deeper one will never be reached.

7. Related Work

As an effective dynamic analysis technique, fuzzing has been widely applied to kernel and device drivers.

DIFUZE [8] deals with the problem of interface recovery by leveraging static analysis to compose correctly-structured input. Such a static approach can only deal with the hard-coded names and commands but failed to handle the dynamically generated arguments. Also, it is not able to extract complex relationships between fields of structures in the interface. Often in complex driver code, one field of a structure relates to another: for example, a length field could specify a buffer size. Our approach successfully recognizes the relationships, which provide valuable information during the fuzzing stage.

EASIER [10] creates an ex-vivo dynamic driver analysis framework for Android devices by developing an evasion kernel that enables driver initialization. It evades hardware and kernel dependencies instead of precisely emulating the features. The upside of evasion is that it enables dynamic analysis without the actual devices. The downside is that it is imprecise and is blamed for producing a significant amount of false positives that troubles the researchers. Only part of driver modules can be successfully initialized in the evasion kernel, limiting the scope of appliance.

Charm [9] proposes a remote device driver execution method that enables the device driver to execute in a virtual machine on a workstation to facilitating dynamic analysis of device drivers of mobile phones. It uses the actual mobile phone device only for servicing the low-level and infrequent I/O operations through a low-latency and customized USB channel. A drawback of Charm is that it requires porting every driver in question to a specific version of the kernel. The time needed to port the driver for an expert is from days to weeks.

Many existing works make various contributions to different stages of kernel fuzzing. Agamotto [24] improved the kernel driver fuzzing performance by dynamically creating multiple checkpoints and skipping parts of test cases using the checkpoints. USBFuzz [25] leverages an emulation USB device to fuzz the USB drivers of modern operating systems. POTUS [26] also aims to find vulnerabilities in USB device drivers using fault injection, concurrency fuzzing, and symbolic execution. FIZZER [27] tries to explore the error handling code in device drivers using software fault injection. Unicorefuzz [28] is an emulation-based fuzzing approach and uses CPU emulation to fuzz device drivers and kernel components with coverage-based feedback. IMF [29] inferred a model for the system in question to facilitate input generation. It extracts the model by inspecting API sequences by actual applications running on the system. Krace [30] aims to explore data race bugs in the kernel file system by capturing the exploration progress in the concurrency dimension and precisely data race detecting. Muzz [31] hunts for bugs in multithreaded programs using thread-aware instrumentations to obtain runtime feedback to accentuate execution states caused by thread interleavings and preserves more valuable seeds that expose bugs under a multithreading context. Razzer [32] employs a static analysis and a deterministic thread interleaving technique to find race bugs in kernel efficiently. JANUS [33] employs two techniques to find bugs efficiently: mutating metadata on a large image and emitting image-directed file operations. PeriScope [34] focuses on the hardware-OS boundary and leverages fine-grained analysis of device-driver interactions.

Static analysis is also used by many existing methods to identify bugs in source code or binaries of kernel or device drivers. IDEA [35] is a static analysis tool for finding bugs in Apple driver binaries that is able to effectively recover C++ classes, resolve indirect calls on the Apple platform and find the unique paradigms through which Apple drivers interact with userspace programs. DR. CHECKER [36] performed flow-sensitive, context-sensitive and field-sensitive taint analyses on the Linux driver source code and identified driver bugs. Moonshine [37] leverages lightweight static analysis for efficiently detecting dependencies across different system calls and generates good seeds for OS fuzzing according to the OS kernel state created by the previously executed system calls.

8. Conclusion

This Android driver interface exposes the kernels and device drivers to attacks by malicious programs. The main challenge to kernel driver fuzzing is how to generate valid inputs, how to handle the dependency in data fields and how to mutate the inputs. This paper proposes a coverage-guided and layout-aware kernel driver fuzzing framework CoLaFUZE to facilitate the generation of the seed, kernel-userspace data copy, and seeds mutation. We show that the proposed methods can recover the device file names, solve the command values, forward the data request and detect vulnerabilities effectively and efficiently. We carry out a comparative evaluation on different configurations of CoLaFUZE with state-of-the-art fuzzers on 5 modern Android mobile phones and show that CoLaFUZE makes a significant improvement in code coverage and successfully detects 11 vulnerabilities.

References

[1] J.V. Stoep, “Android: protecting the kernel,” Linux Security Summit, 2016.
[2] Copyright Google Inc., Android Sources. http://android.googlesource.com/kernel, accessed Jan. 21. 2021.
[3] Copyright Huawei Inc., Open Source Release Center. https://consumer.huawei.com/en/opensource/, accessed Jan. 21. 2021.
[4] Copyright Samsung Inc., Samsung Open Source. http://opensource.samsung.com, accessed Jan. 21. 2021.
[5] Copyright Sony Inc., Open Source Archives - Open Devices - Sony Developer World. https://developer.sony.com/develop/open-devices/downloads/open-source-archives/, accessed Jan. 21. 2021.
[6] Copyright Xiaomi Inc., Xiaomi Phone Kerenl OpenSource. https://github.com/MiCode/Xiaomi_Kernel_OpenSource, accessed Jan. 21. 2021.
[7] Copyright Amazon Inc., Source Code Notice. https://www.amazon.com/gp/help/customer/display.html, accessed Jan. 21. 2021.
[8] J. Corina, A. Machiry, C. Salls, S. Yan, H. Shuang, C. Kruegel, and G. Vigna, “Difuze: Interface aware fuzzing for kernel drivers,” ACM Sigsac Conference, pp.2123–2138, Oct. 2017.
[9] S.M.S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A.A. Sani, and Z. Qian, “Charm: Facilitating dynamic analysis of device drivers of mobile systems,” 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, pp.291–307, USENIX Association, Aug. 2018.
[10] I. Pustogarov, Q. Wu, and D. Lie, “Ex-vivo dynamic analysis framework for android device drivers,” 2020 IEEE Symposium on Security and Privacy (SP), pp.1088–1105, 2020.
[11] “Information technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7,” standard, ISO Org., Sept. 2009.
[12] Cr4sh, IOCTL Fuzzer - Windows kernel drivers fuzzer., 2011. https://github.com/Cr4sh/ioctlfuzzer, accessed Jan. 21. 2021.

[13] 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/, accessed Jan. 21. 2021.
[14] C.S. Lejay, Fuzzing IOCTLs with angr. https://thunderco.re/project/security/2016/07/18/fuzzing-ioctls/, accessed Jan. 21. 2021.
[15] debasishm89, A mutation based user mode (ring3) dumb in-memory Win-dows Kernel (IOCTL) Fuzzer, 2014. http://developer.android.com/tools/help/monkey.html, accessed Jan. 21. 2021.
[16] The kernel development community, kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html, accessed Jan. 21. 2021.
[17] W.H. Harrison, "Compiler analysis of the value ranges for variables," IEEE Trans. Softw. Eng., vol.SE-3, no.3, pp.243–250, May 1977.
[18] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," 2017 IEEE Cybersecurity Development (SecDev), pp.8–9, 2017.
[19] M. Zalewski, American Fuzzy Lop. http://lcamtuf.coredump.cx/afl, Accessed: Jan. 21, 2021.
[20] Copyright Google Inc., Syzkaller - linux syscall fuzzer, 2017. https://github.com/google/syzkaller, accessed Jan. 21. 2021.
[21] Copyright Google Inc., Android Flash Tool, 2021. https://flash.android.com, accessed Jan. 21. 2021.
[22] Copyright Google Inc., AOSP - Locking/Unlocking the Bootloader, 2021. https://source.android.com/devices/bootloader/locking_unlocking, accessed Jan. 21. 2021.
[23] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, pp.2123–2138, Association for Computing Machinery, New York, NY, USA,Oct. 2018.
[24] D. Song, F. Hetzelt, J. Kim, B.B. Kang, J.P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," 29th USENIX Security Symposium (USENIX Security 20), pp.2541–2557, USENIX Association, Aug. 2020.
[25] H. Peng and M. Payer, "USBfuzz: A framework for fuzzing USB drivers by device emulation," 29th USENIX Security Symposium (USENIX Security 20), pp.2559–2575, USENIX Association, Aug. 2020.
[26] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "POTUS: Probing off-the-shelf USB drivers with symbolic fault injection," 11th USENIX Workshop on Offensive Technologies (WOOT 17), Vancouver, BC, USENIX Association, Aug. 2017.
[27] Z. Jiang, J. Bai, J. Lawall, and S. Hu, "Fuzzing error handling code in device drivers based on software fault injection," 2019 IEEE 30th Int. Symp. Software Reliability Engineering (ISSRE), pp.128–138, 2019.
[28] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," 13th USENIX Workshop on Offensive Technologies (WOOT 19), Santa Clara, CA, USENIX Association, Aug. 2019.
[29] H.S. Han and K.C. Sang, "IMF: Inferred model-based fuzzer," ACM Sigsac Conference, pp.2345–2358, Oct. 2017.
[30] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," 2020 IEEE Symposium on Security and Privacy (SP), pp.1643–1660, 2020.
[31] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," 29th USENIX Security Symposium (USENIX Security 20), pp.2325–2342, USENIX Association, Aug. 2020.
[32] D.R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," 2019 IEEE Symposium on Security and Privacy (SP), pp.754–768, 2019.
[33] W. Xu, H. Moon, S. Kashyap, P.N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," 2019 IEEE Symposium on Security and Privacy (SP), pp.818–834, 2019.
[34] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, Feb. 24-27, 2019, The Internet Society, 2019.
[35] X. Bai, L. Xing, M. Zheng, and F. Qu, "iDEA: Static analysis on the security of apple kernel drivers," Proc. 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, pp.1185–1202, Association for Computing Machinery, New York, NY, USA, Oct. 2020.
[36] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, pp.1007–1024, USENIX Association, Aug. 2017.
[37] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," Proc. 27th USENIX Conference on Security Symposium, SEC'18, USA, p.729–743, USENIX Association, Aug. 2018.

Appendix: Case Study

We walk through an example of a bug that was triggered only with the proposed framework. The relevant source is shown in Listings A· 1 and A· 2.

The example is CVE-2020-0069, a bug of MediaTek command queue driver triggered in Mi Play during the experiments. The handler cmdq_ioctl of the driver invokes function cmdq_driver_ioctl_exec_command when cmd is specified as CMDQ_IOCTL_EXEC_COMMAND. At lines 8-9 in Listing A· 2, the user data is copied into a struct cmdqCommandStruct command. At line 14, it invokes cmdq_driver_copy_handle_prop_from_user to copy user data from command.prop_addr with pointer casting. Notice that the field prop_addr of struct cmdqCommandStruct is cmdqU32Ptr_t, an unsigned long long type defined at line 2 of Listing A· 1. CoLFUZE forwards this copy operation to the fuzzing engine and fetches new inputs on the fly. The data is transferred into cmdq_driver_process_command_request and be parsed as a sequence of CMDQ instructions. The bug is triggered when parsing a CMDQ_CODE_WRITE (0x04) instruction, for the target address is not sanitized properly, causing an out of bounds write bug.

However, DIFUZE fails to trigger this bug. DIFUZE recovers struct cmdqCommandStruct and regards the field prop_addr as a number instead of a userspace pointer, for

```
1  /* defined in cmdq_def.h */
2  #define cmdqU32Ptr_t unsigned long long
3
4  struct cmdqCommandStruct {
5    /* Previous fields. */
6    /* task property */
7    uint32_t prop_size;
8    cmdqU32Ptr_t prop_addr;
9    /* Other fields. */
10 };
```

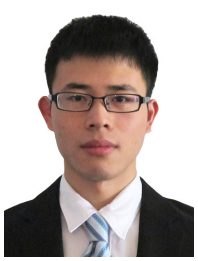
Listing A· 1 The structure definition used by driver CMDQ

```
1  static s32 cmdq_driver_ioctl_exec_command(struct file
2      *pf, unsigned long param)
3  {
4    struct cmdqCommandStruct command;
5    struct task_private desc_private = {0};
6    s32 status;
7
8    if (copy_from_user(&command, (void *)param,
9      sizeof(struct cmdqCommandStruct)))
10       return -EFAULT;
11
12   /* Some command checking code here*/
13
14   status = cmdq_driver_copy_handle_prop_from_user(
15     (void *)CMDQ_U32_PTR(command.prop_addr),
16     command.prop_size,
17     (void *)CMDQ_U32_PTR(&command.prop_addr));
18   if (status < 0) {
19     CMDQ_ERR("copy prop_addr failed, err=%d\n",
           status);
20     return status;
21   }
22
23   /* insert private_data for resource reclaim */
24   desc_private.node_private_data = pf->private_data;
25   command.privateData =
26     (cmdqU32Ptr_t)(unsigned long)&desc_private;
27
28   status =
29     cmdq_driver_process_command_request(&command);
30 ...
31 };
```

Listing A· 2 The function to execute user-provided program instructions of driver CMDQ

the type cmdqU32Ptr_t is defined as unsigned long long. DIFUZE fails to generate a valid pointer at field prop_addr, and the function always returns at line 20. Thus, thanks to the dynamic forwarding module, this bug is finally reached.

Tianshi Mu was born in 1980. He received the M.S. degree in communication and information system in 2005. He is now the director of IEEE PES power system communication and network security technical committee. Currently, He is engaged in application security, data security and other directions of forward-looking security technology research, as well as the construction of enterprise-level security platform in Digital Grid Research Institute, CSG.



Huabing Zhang was born in 1988. He is currently a senior engineer and director of the Cyber Security Company of CSG Digital Grid Research Institute. His main research interests include network security situation awareness and security automation.

Jian Wang was born in 1978. He is currently a deputy director with Digital Grid Security R&D Center of Digital Grid Research Institute Cyber Security Branch, Guangzhou, China. His main research interests include wireless communication, network security and computer security.

Huijuan Li was born in 1983. She received the M.S degree in pattern recognition and intelligent systems from the South China University of Technology in 2008. Her main research interests include computer and information technology and network security.