Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection

Zu-Ming Jiang, Jia-Ju Bai, Julia L. Lawall, Shi-Min Hu

Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection

Zu-Ming Jiang†, Jia-Ju Bai†*, Julia Lawall‡ and Shi-Min Hu†
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
jjzuming@outlook.com, baijiaju1990@gmail.com, shimin@tsinghua.edu.cn
‡Sorbonne University/Inria/LIP6, Paris, France
Julia.Lawall@lip6.fr

***Abstract*—Device drivers remain a main source of runtime failures in operating systems. To detect bugs in device drivers, fuzzing has been commonly used in practice. However, a main limitation of existing fuzzing approaches is that they cannot effectively test error handling code. Indeed, these fuzzing approaches require effective inputs to cover target code, but much error handling code in drivers is triggered by occasional errors (such as insufficient memory and hardware malfunctions) that are not related to inputs.**

In this paper, based on software fault injection, we propose a new fuzzing approach named FIZZER, to test error handling code in device drivers. At compile time, FIZZER uses static analysis to recommend possible error sites that can trigger error handling code. During driver execution, by analyzing runtime information, it automatically fuzzes error-site sequences for fault injection to improve code coverage. We evaluate FIZZER on 18 device drivers in Linux 4.19, and in total find 22 real bugs. The code coverage is increased by over 15% compared to normal execution without fuzzing.

***Index Terms*—device driver, fuzzing, fault injection, dynamic analysis, bug detection**

I. Introduction

Device drivers are an important part of modern operating systems. They manage hardware devices and provide fundamental support for high-level programs. For this reason, if a device driver has bugs that can occur in real execution, it may cause serious problems, such as information leaks and system crashes. Unfortunately, device drivers remain a main source of bugs found in operating systems. For example, over 50% of Linux kernel bugs are related to device drivers [1].

To detect bugs in drivers, many static and dynamic techniques have been proposed and used. Among these techniques, *fuzzing* [2] is a promising dynamic technique that has been commonly used. Basically, fuzzing generates program inputs to cover infrequently executed code. It starts from some original program inputs, and generates effective inputs that can cover infrequently executed code, by analyzing the feedback (such as code coverage and bug detection results) of program execution. During execution, it uses bug checkers to detect bugs according to the information of program execution. A well-known kernel fuzzing tool is syzkaller [3], which has found hundreds of real bugs in Linux device drivers [4]. Many driver fuzzing approaches [5]–[7] are based on syzkaller.

* Jia-Ju Bai is the corresponding author.

However, existing driver fuzzing approaches still have two main limitations in practical use: *(L1) They cannot directly generate original driver inputs in many cases.* On the one hand, device drivers communicate with hardware devices, and changing hardware outputs is often difficult. Thus, existing fuzzing approaches do not handle the driver inputs from the hardware devices. On the other hand, in many cases, device drivers directly communicate with related kernel modules, instead of user-level applications. But existing fuzzing approaches only generate inputs from system calls at the user level, and thus many generated inputs have been changed by the time they reach device drivers. *(L2) They cannot effectively cover error handling code.* In device drivers, much error handling code is triggered by occasional errors (such as insufficient memory and hardware malfunctions) that are not related to inputs. Due to rare execution, such error handling code is difficult to test in practice [8]–[10]. However, existing fuzzing approaches do not target such error handling code.

Recently, the kernel fuzzing approach PeriScope [11] was proposed to partially address limitation *L1* about generating the driver inputs from the hardware device. PeriScope monitors the communication between the OS kernel and the hardware device, and can simulate and fuzz the driver inputs from the hardware device to perform runtime testing. But PeriScope still relies on the inputs from system calls at the user level. To our knowledge, no driver fuzzing approaches have been proposed to address limitation *L2*.

To address limitation *L2*, a possible way is to use *software fault injection (SFI)* [12] to help driver fuzzing. SFI intentionally injects faults or errors into the tested program, and then runs the program to check whether the program can correctly handle the injected faults or errors at runtime. In this way, SFI can effectively cover error handling code at runtime. Some SFI-based approaches [13]–[16] have shown promising results in testing error handling code and finding bugs in drivers.

To introduce SFI in fuzzing, our basic idea is to fuzz injected faults to test error handling code as much as possible. To achieve this idea, we design a SFI-based fuzzing strategy that has four steps: 1) statically identify the sites that can fail and trigger error handling code in the driver; we call each such site an *error site*; 2) run the driver, and then according to the runtime information of the device driver, use a coverage-based mutation method to generate error-site sequences that

978-1-7281-4982-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSRE.2019.00022

indicate which error sites should fail; 3) inject faults on error sites according to the generated error-site sequences; 4) run the driver, and use the mutation method again, to generate new error-site sequences, making up a fuzzing loop.

Based on our strategy, we propose a new fuzzing approach named FIZZER, to test error handling code in device drivers. At compile time, FIZZER performs static analysis of the driver source code to recommend possible error sites, from which the user should select realistic ones that can actually fail and trigger error handling code. Then, FIZZER uses our SFI-based fuzzing strategy to perform runtime testing. To detect different kinds of bugs, FIZZER uses independent checkers that we have implemented and some third-party checkers. We have implemented FIZZER using LLVM [17] for Linux drivers.

Overall, in this paper, we make four main contributions:

- We perform two studies about error handling code in Linux device drivers, and find that current driver fuzzing approaches may miss many bugs in error handling code, especially those triggered by occasional errors. Thus, it is important to improve fuzzing support for error handling code triggered by different types of errors.
- We propose a SFI-based fuzzing strategy to generate effective injected faults that can cover as much error handling code as possible.
- Based on our strategy, we propose a new fuzzing approach named FIZZER, to effectively test error handling code in device drivers. To our knowledge, FIZZER is the first systematic fuzzing approach targeting error handling code in device drivers.
- We evaluate FIZZER on 18 device drivers in Linux 4.19. It finds 22 new real bugs, and 12 of them have been confirmed by driver developers. The code coverage is increased by over 15% compared to normal execution without fuzzing. We also make a comparison to syzkaller. FIZZER can find many bugs missed by syzkaller and achieve higher code coverage.

The rest of this paper is organized as follows. Section II describes background and our two studies. Section III introduces our basic idea and our SFI-based fuzzing strategy. Section IV introduces FIZZER in detail. Section V shows our evaluation on Linux device drivers. Section VI discusses some possible extensions of FIZZER. Section VII presents related work, and Section VIII concludes this paper.

II. Background

In this section, we first introduce error handling code in device drivers, and then present the results of our studies of driver code and Linux kernel commits.

A. Error Handling and Example Bugs in Device Drivers

Error. A device driver may encounter some exceptional situations due to conditions in its execution environment, such as invalid requests from the user level, insufficient memory or hardware malfunction. We refer to such exceptional situations as *errors*. The code that is used to handle an error is called *error handling code*.

Error type. An error in driver execution can be *input-related* or *occasional*. An input-related error is caused by abnormal inputs, such as unexpected commands and invalid data. Such an error can be conveniently triggered with specific inputs. An occasional error is caused by an exceptional event that occasionally occurs, such as insufficient memory or a hardware malfunction. Such an error is often related to the state of the system and the hardware, instead of inputs, so it is difficult to trigger in real execution.

```
--- a/drivers/net/team/team.c
+++ b/drivers/net/team/team.c
@@ -1167,6 +1167,12 @@
static int team_port_add(struct team *team,
                         struct net_device *port_dev, ...) {
   struct net_device *dev = team->dev;
   ...
   // BUG: Enslaving device itself can cause a system hang
+  if (dev == port_dev) {
+    netdev_err(dev, "Cannot enslave device to itself\n");
+    return -EINVAL;
+  }
   ...
}
```

Fig. 1. Patch A: fixing a hang problem caused by an input-related error.

```
--- a/drivers/net/team/team.c
+++ b/drivers/net/team/team.c
@@ -2395,7 +2395,7 @@
static int team_nl_send_options_get(...) {
   ...
   // COMMENT: This function can free skb when it fails
   err = __send_and_alloc_skb(&skb, ...);
   if (err)
-    goto errout;
+    return err;
   ...
errout:
   // BUG: Free skb again
   nlmsg_free(skb);
   return err;
}
```

Fig. 2. Patch B: fixing a double-free bug caused by an occasional error.

Example bugs in error handling code. Figures 1 and 2 show two patches fixing bugs in error handling code of the Linux *team* driver. The two patches were applied in 2018. Both fixed bugs that had been present since Linux 3.6, released over 5 years earlier. In Figure 1, the original driver code does not handle the error of the device enslaving itself, so a system hang may occur in subsequent execution. To fix this bug, *Patch A* [18] adds an *if* check and corresponding error handling code. This patch mentions that the bug was found by syzkaller. In Figure 2, the function `__send_and_alloc_skb` can fail and free the `skb` data, and then return a non-zero error code. But when handling this error, the function `nlmsg_free` frees the `skb` data again, causing a double-free bug. To fix this bug, *Patch B* [19] deletes the *goto* statement and returns directly. As this patch does not mention any tool, the bug may have been found by manual inspection or real execution.

The bug in Figure 1 involves an input-related error, because the variables `dev` and `port_dev` in the *if* check are related to inputs. The bug in Figure 2 involves an occasional error,

TABLE I
STUDY RESULTS OF ERROR HANDLING CODE IN LINUX DRIVERS.

Driver class	Error site	Input-related error	Occasional error
Ethernet	2893	1268 (44%)	1625 (56%)
Wireless	1264	495 (39%)	769 (61%)
Sound	1751	948 (54%)	803 (46%)
USB	2575	1449 (56%)	1126 (44%)
Character	1588	799 (50%)	789 (50%)
Total	10,071	4959 (49%)	5112 (51%)

TABLE II
STUDY RESULTS OF LINUX KERNEL COMMITS MENTIONING SYZKALLER.

Year	Commit	Driver	Error handling	Occasional error
2016	80	36	6	1
2017	225	62	14	2
2018	599	155	62	13
Total	904	253	82	16

as the function `__send_and_alloc_skb` fails on memory exhaustion, which only occurs occasionally at runtime.

B. Study of Error Handling Code in Device Drivers

To understand the proportion of input-related and occasional errors that can trigger error handling code, we perform a manual study of driver source code in Linux 4.19. Due to the large number of driver source files and due to time constraints, we limit the drivers to five commonly used driver classes (Ethernet controller drivers, wireless controller drivers, sound drivers, USB drivers and character drivers), and randomly select 100 source files in each class to study. We first manually identify the sites that can trigger error handling code by looking for *goto* statements and returned error codes (such as *-EINVAL* in Figure 1), that are often used in error handling code in the Linux kernel [20]. Then, we check whether these sites are related to input-related errors or occasional errors. Table I shows the results.

Table I shows that 51% of the sites that can trigger error handling code are related to occasional errors. Compared to input-related errors, occasional errors are often more difficult to trigger and reproduce, and thus testing error handling code triggered by occasional errors is more challenging.

C. Study of Linux Kernel Commits

To know about the driver bugs found by fuzzing, we perform a manual study of Linux kernel commits. We focus on the commits involving syzkaller [3], as it is a well-known kernel fuzzing tool that has been widely used to test the Linux kernel. Firstly, we select the commits that were submitted in 2016-2018 (3 years) and fixed bugs found by syzkaller, by searching (*git log --grep*) for "syzkaller" in the log message. Then, from the resulting 904 kernel commits, we identify those that affect the *drivers* or *sound* directories, as these directories contain the driver source files. Finally, we manually read the code changes in these driver commits, to check: 1) whether the reported bugs involve error handling code; 2) whether the reported bugs involve occasional errors. Table II shows the results.

Table II shows that 28% of kernel commits fixing bugs reported by syzkaller are for device drivers. Among the driver commits, 32% involve error handling code (such as *Patch A* in Figure 1). Besides, 20% of the driver commits involving error handling code are related to occasional errors. This proportion is much smaller than the proportion (51%) of occasional error sites among all error sites in device drivers. These results suggest that syzkaller may miss many bugs in error handling code, especially those triggered by occasional errors. For this reason, it is important to improve fuzzing support for error handling code triggered by different types of errors.

III. IDEA AND STRATEGY

A. Basic Idea

Software fault injection can help to cover error handling code. An important problem here is how to inject faults to effectively cover as much error handling code as possible. Previous SFI-based approaches for device drivers often inject random faults [21] or perform single fault injection [16], but these strategies still miss much error handling code. To solve this problem, we propose to "fuzz" injected faults according to the runtime information of the driver.

In driver code, there are some sites that can fail at runtime and trigger error handling code, such as the function call to `__send_and_alloc_skb` in Figure 2. We call such sites *error sites* (represented as *Err*). Multiple error sites ordered by their static positions in the driver source code make up a sequence that we call an *error-site sequence*. For fault injection, each error site in an error-site sequence can normally run (indicated as 0) or fail by injecting faults (indicated as 1). As a result, an error-site sequence can be represented as a 0-1 sequence to describe the failure situation of error sites:

$$ErrSeq = [Err_1, Err_2, Err_3, ...Err_x],\ Err_i = \{0, 1\}$$

An error-site sequence is similar to a sequence of program inputs, because they both affect the executed code of the tested program. In fact, error-site sequences can be regarded as the "inputs" of possibly encountered errors. Existing fuzzing approaches fuzz program inputs to cover infrequently-executed code. Thus, accordingly, our basic idea is to *fuzz error-site sequences to cover infrequently-executed error handling code.*

B. SFI-based Fuzzing Strategy

To achieve this basic idea, we propose a SFI-based fuzzing strategy. As shown in Figure 3, this strategy has four basic steps: 1) statically identify error sites in the driver code; 2) run the driver, and then according to the runtime information of the driver, use a coverage-based mutation method to generate error-site sequences to cover error handling code; 3) inject faults on error sites according to the generated error-site sequences; 4) run the driver, and use the mutation method again, to generate new error-site sequences, making up a fuzzing loop. When no new error-site sequences are generated, we terminate the fuzzing process. To detect bugs, we use some independent checkers to analyze runtime information.

The coverage-based mutation method is the core of our fuzzing strategy. This method takes coverage information into account and drops repeated error-site sequences. But initially

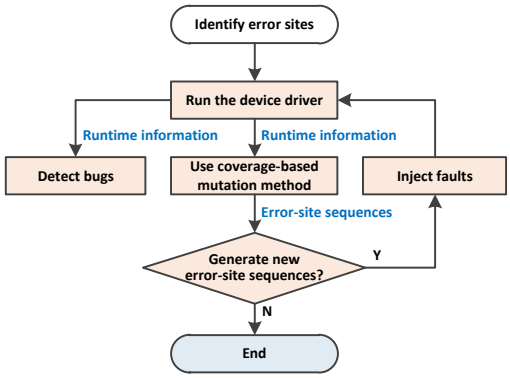

Fig. 3. Basic strategy of fuzzing error handling code.

this information is not available, so this method needs a simplified procedure as the initial mutation, for the first execution of the driver. For all subsequent executions, the method performs the subsequent mutation.

1) Initial mutation: Before the first execution of the driver, the initial error-site sequence is an all-zero sequence, indicating that no faults are injected on error sites. After the execution, by monitoring driver execution, the executed error sites are collected. Then, these executed error sites are used for the initial mutation. The mutation method generates each new error-site sequence by making one executed error site fail (0→1), as each error site may trigger different error handling code. Figure 4 shows an example of the initial mutation for an error-site sequence containing six error sites, of which three error sites are actually executed in the first execution.

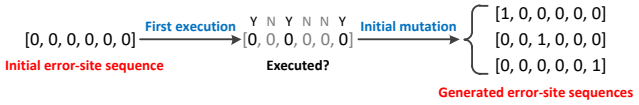

Fig. 4. Example of the initial mutation.

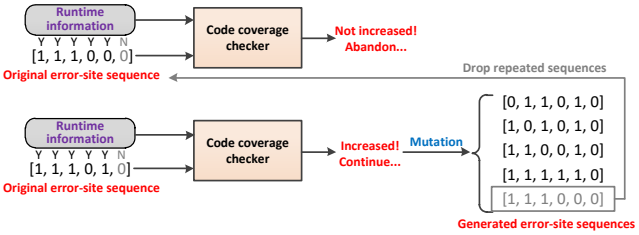

Fig. 5. Example of the subsequent mutation.

2) Subsequent mutation: After a subsequent execution of the driver, if the code coverage is increased (namely new code branches or basic blocks are covered), the mutation method selects the error-site sequence of this execution as a seed for mutation; otherwise, this error-site sequence is abandoned. When this error-site sequence is mutated, only one executed error site is changed (0→1 or 1→0) at a time, because each error site may trigger different error handling code. In this way, the mutation method generates some new error-site sequences. Then, it compares these generated error-site sequences to previously used error-site sequences, and drops repeated ones. Finally, the remaining generated error-site sequences are used for fault injection and subsequent mutations. Figure 5 shows an example of the subsequent mutation for two error-site sequences, of which the last error site is not actually executed.

Note that the mutation method only mutates the executed error sites. Indeed, most segments of error handling code require the failure of only single error site to trigger, and then make the driver abnormally exit without executing other error sites [16]. Thus, only mutating executed error sites can avoid generating many unnecessary error-site sequences.

IV. Approach and Implementation

A. Architecture

Based on our SFI-based fuzzing strategy, we propose a new fuzzing approach named FIZZER, to test error handling code in device drivers. We have implemented FIZZER based on the Clang compiler [22]. To enable fuzzing and perform fault injection, FIZZER performs code analysis and instrumentation on the LLVM bytecode of the driver code. Figure 6 shows the architecture of FIZZER, which consists of four parts:

- ***Error-site analyzer.*** It performs a static analysis of the driver source code to recommend possible error sites, from which the user should select realistic ones that can actually fail and trigger error handling code.
- ***Driver generator.*** It instruments the identified error sites in the driver code and generates a loadable driver.
- ***Runtime fuzzer.*** It uses our SFI-based fuzzing strategy to perform runtime testing. During driver execution, it collects the runtime information about the driver.
- ***Bug checkers.*** They check the information collected by the runtime fuzzer to detect bugs.

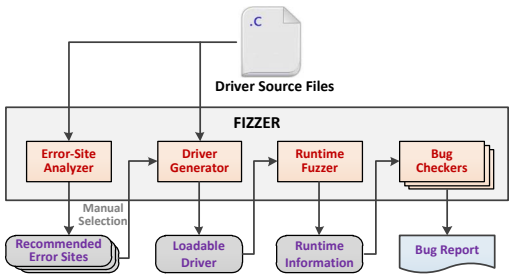

Fig. 6. Overall architecture of FIZZER.

B. Phases

Based on the architecture, FIZZER consists of two phases.

*1) **Compile-Time Analysis**:* This phase performs two tasks:

Identify error sites. In SFI, whether the identified error sites are realistic heavily affects the false positives of bug detection. Thus, many existing SFI approaches [13]–[15] require the user to specify error sites by hand. To reduce manual work, the error-site analyzer performs static analysis to recommend possible error sites, from which the user should select realistic ones. This static analysis focuses on recommending function

calls that can fail as error sites, as a large part of error handling code in drivers is triggered by bad function return values [16].

To make the recommended error sites more realistic, the static analysis is designed according to two kinds of semantic information about the driver source code: 1) according to the Linux kernel documentation [20], a function that can fail often returns a NULL pointer or a non-zero integer to indicate a failure; 2) an *if* statement is often used to check whether an error can occur.[1] The static analysis identifies all function calls that fit both of these forms. Figure 7 shows two examples from the Linux *cmipci* driver.

```
FILE: linux-4.19/sound/pci/cmipci.c
3011. static int snd_cmipci_create(...) {
          .....
          // return a NULL pointer to indicate a failure
3034.     cm = kzalloc(...);
3035.     if (cm == NULL) {
3036.         pci_disable_device(pci);
3037.         return -ENOMEM;
3038.     }
          .....
          // return a non-zero integer to indicate a failure
3202.     err = snd_cmipci_create_fm(...)
3203.     if (err < 0)
3204.         return err;
          .....
3257. }
```

Fig. 7. Examples of function calls that can fail.

Code instrumentation. To perform fault injection, the driver generator instruments an *error probe* on each error site in the driver code, to determine whether this error site should fail in runtime testing. The error probe checks this error site's value in the current error-site sequence. When the error site's value is 1, indicating that this error site should fail, the error probe returns *ERROR* and injects a fault on this error site. In this case, this function call is not executed, and the variable assigned by its return value is assigned by an *error value*. If the called function returns a pointer, this error value is a NULL pointer; if the called function returns an integer, this error value is a random negative integer. When the error site's value is 0, indicating that this error site should succeed, the error probe does not return *ERROR*, and the function call of this site is executed normally. Figure 8 shows the instrumented code in the C language for the examples in Figure 7. Note that the actual instrumentation is performed on the LLVM bytecode.

To perform code instrumentation, the driver generator modifies the normal compilation process of the tested driver. It first uses the Clang compiler to compile the C source code into the LLVM bytecode. Then, it instruments the LLVM bytecode. Finally, it uses the Clang compiler to generate a loadable driver from the instrumented LLVM bytecode.

*2) **Runtime Testing**:* In this phase, with the identified error sites, the runtime fuzzer uses our SFI-based fuzzing strategy in runtime testing. We explain some main technical details used in this phase:

Runtime monitoring. As our mutation method is coverage-based, the runtime fuzzer collects the information about the basic blocks and code branches executed by the driver. Because device drivers can run concurrently, the runtime fuzzer

[1]We do not consider *ERR_PTR()*, *PTR_ERR()* and *IS_ERR()* at present.

```
FILE: linux-4.19/sound/pci/cmipci.c
3011. static int snd_cmipci_create(...) {
          .....
++++.     if (error_probe() == ERROR)
++++.         cm = NULL;
++++.     else
3034.         cm = kzalloc(...);  // error site
3035.     if (cm == NULL) {
3036.         pci_disable_device(pci);
3037.         return -ENOMEM;
3038.     }
          .....
++++.     if (error_probe() == ERROR)
++++.         err = random_negative_integer;
++++.     else
3202.         err = snd_cmipci_create_fm(...)  // error_site
3203.     if (err < 0)
3204.         return err;
          .....
3257. }
```

Fig. 8. Examples of code instrumentation.

uses locks to synchronize the information collected in different running threads.

Bug detection. The runtime fuzzer uses independent bug checkers to detect bugs according to the collected runtime information. We have implemented two checkers to detect resource leaks and double-lock bugs, and use two third-party checkers, namely KASAN [23] to detect memory-corruption bugs and Kmemleak [24] to detect memory leaks.

Distributed deployment. During runtime testing, the tested driver may hang or crash. In this case, if error-site-sequence generation and runtime testing are both performed on the same machine, the intermediate results (including the generated error-site sequences and code coverage) can be lost, making it necessary to restart the fuzzing process from the beginning. To solve this problem, we deploy FIZZER on multiple machines, including one server and multiple clients, as shown in Figure 9. The server receives and records the runtime information collected by the clients, generates error-site sequences, and then sends them to the clients. Each client receives the error-site sequences, performs fault-injection testing of the driver accordingly, and sends the collected runtime information and bug reports to the server. This deployment has two advantages: 1) when the driver crashes or hangs, the fuzzing process can be continued according to last intermediate results, after rebooting the corresponding test clients; 2) the elapsed time can be largely reduced by using multiple clients.

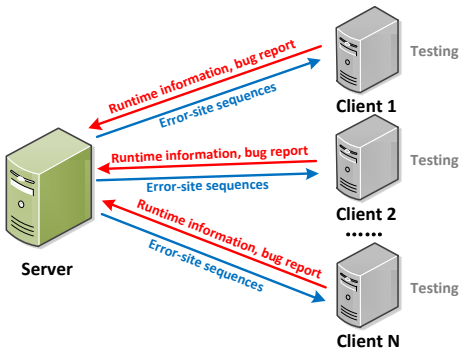

Fig. 9. Deployment of FIZZER.

TABLE III
TESTED DEVICE DRIVERS

Class	Driver	Hardware device	LOC
Ethernet	e1000	Intel 82540EM Ethernet Controller	17.1K
	e1000e	Intel 82572EI Ethernet Controller	29.2K
	3c59x	3Com 3c905B Ethernet Controller	3.4K
	via-rhine	VIA VT6106S Ethernet Controller	2.7K
	r8169	Realtek RTL8169 Ethernet Controller	7.5K
	tg3	Broadcom BCM5721 Ethernet Controller	21.8K
	dl2k	ICPlus IP1000 Ethernet Controller	2.3K
Wireless	b43	Broadcom BCM4322 Wireless Controller	57.1K
	iwlwifi	Intel 7260 Wireless Controller	13.2K
	ath9k	Atheros AR5418 Wireless Controller	87.8K
	iwl4965	Intel 4965AGN Wireless Controller	28.6K
	rtl8723be	Realtek RTL8723BE Wireless Controller	55.8K
USB	ehci_hcd	Intel USB 2.0 EHCI Controller	16.0K
	xhci_hcd	Intel USB 3.0 xHCI Controller	24.0K
	usb_storage	Kingston 8GB USB disk	23.3K
Sound	cmipci	C-Media CM8738 Sound Card	3.4K
	maestro3	ESS ES1988 Allegro-1 Sound Card	2.8K
	ymfpci	Yamaha YMF754 Sound Card	3.2K

TABLE IV
WORKLOADS OF TESTING DRIVERS

Class	Workload Description	Command
Ethernet	Network configuration	*ifconfig, dhcp, nmcli, route*
	Data transmission	*ping, ssh, scp, ftp, wget*
Wireless	Network configuration	*iwconfig, dhcp, nmcli, route*
	Data transmission	*ping, ssh, scp, ftp, wget*
USB	Device control	*mount, umount, lsusb*
	Data transmission	*cp, dd*
Sound	Sound playback	*aplay, mplayer*
	Sound recording	*arecord*

V. EVALUATION

A. Experimental Setup

To validate the effectiveness of FIZZER, we evaluate it on 18 commonly used device drivers in Linux 4.19, including 7 Ethernet controller drivers, 5 wireless controller drivers, 3 USB drivers and 3 sound card drivers. Table III lists the drivers.

The experiment runs on three common Lenovo PCs with eight Intel i7-3770@3.40G processors and 8GB physical memory. One PC is used as the server, and the other two PCs are used as the test clients. Note that when testing an Ethernet controller driver, we use another Ethernet controller different from that of the tested driver to perform the communication between the server and clients. The driver source code is compiled using Clang 6.0. For each tested driver, we install it and run it with some workloads, and then uninstall it. The workloads are shown in Table IV.

B. Error Site Identification

FIZZER first performs static analysis of the driver source code to recommend possible error sites. Then, according to our driver-specific knowledge and related driver documentations, we manually select the realistic error sites that can actually fail and trigger error handling code. Table V shows the results. The first column shows the driver name; the second column shows the number of all function calls in the drivers; the third column shows the number of error sites (function calls) recommended by FIZZER; the last column shows the number of realistic error sites (function calls) that we manually select.

TABLE V
RESULTS OF SELECTING ERROR SITES.

Driver	Function call	Recommended	Selected
e1000	9104	285 (3%)	156 (55%)
e1000e	14,133	532 (4%)	323 (61%)
3c59x	2069	41 (2%)	41 (100%)
via-rhine	1543	24 (2%)	24 (100%)
r8169	4269	51 (1%)	51 (100%)
tg3	12,726	825 (6%)	308 (12%)
dl2k	1358	34 (3%)	15 (44%)
b43	20,120	239 (1%)	115 (48%)
iwlwifi	4313	227 (5%)	79 (35%)
ath9k	12,272	181 (1%)	93 (51%)
iwl4965	11,921	262 (2%)	99 (38%)
rtl8723be	27,627	111 (0.4%)	44 (40%)
ehci_hcd	5368	128 (2%)	83 (65%)
xhci_hcd	12,314	419 (3%)	278 (66%)
usb_storage	2914	82 (3%)	57 (70%)
cmipci	1884	52 (3%)	52 (100%)
maestro3	1106	36 (3%)	36 (100%)
ymfpci	2137	68 (3%)	63 (93%)
Total	147,178	3597 (2%)	1917 (53%)

From Table V, 98% of function calls in the tested drivers are automatically dropped by FIZZER, as they are identified not to trigger errors according to their calling contexts in driver code. The remaining 2% of function calls are recommended by FIZZER as possible error sites, and we manually select 53% of them as realistic error sites for fault injection. The selected function calls are all related to resource allocation and hardware control, because these operations can indeed fail at runtime. One masters student spent 2 hours on the selection for the 18 tested drivers. Reviewing the selected realistic error sites, we find that over half of them are related to occasional errors. The results indicate that FIZZER can reduce the manual work of dropping unrealistic error sites.

C. Runtime Testing

With the identified realistic error sites, we test the 18 device drivers. We try to fuzz each driver using all generated error-site sequences, and abort the fuzzing with a 5-hour timeout. To measure the increased code coverage provided by FIZZER, we collect the covered basic blocks and code branches in both normal execution without fuzzing and runtime testing with FIZZER. Table VI shows the testing results. The *"All"* column in the *"Error-site sequence"* columns shows the number of all generated error-site sequences used for fault injection. Until reaching the limit of the timeout, each generated error-site sequence is used in exactly one test; after the timeout, all remaining sequences are discarded. The *"Useful"* column shows the number of error-site sequences that increase code coverage. Table VI shows:

Code coverage. FIZZER covers 15% more basic blocks and 21% more code branches in the tested drivers, compared to normal execution without fuzzing. However, it still generates many error-site sequences that do not increase code coverage. Indeed, our mutation does not consider that two different error-site sequences may cover the same error handling code. Even

TABLE VI
RESULTS OF RUNTIME TESTING.

Driver	Error-site sequence		Basic block			Code branch			Detected bug			
	All	*Useful*	*Normal*	*Test*	*Increase*	*Normal*	*Test*	*Increase*	*Crash*	*Corrupt*	*Leak*	*All*
e1000	2192	102	1527	1813	19%	1705	2122	24%	0	0	1	1
e1000e	4059	158	2047	2709	32%	2504	3540	41%	0	0	0	0
3c59x	314	27	697	770	10%	801	902	13%	0	0	0	0
via-rhine	207	23	366	406	11%	417	484	16%	0	0	0	0
r8169	534	36	497	549	10%	582	670	15%	0	0	0	0
tg3	1516	115	2429	2657	9%	3184	3609	13%	0	0	0	0
dl2k	43	15	343	363	6%	401	437	9%	0	0	0	0
b43	1832	125	3090	3294	7%	4091	4510	10%	1	0	0	1
iwlwifi	250	45	744	834	12%	968	1155	19%	0	2	0	2
ath9k	514	110	2378	2657	12%	2848	3275	15%	3	0	0	3
iwl4965	3087	235	3959	4436	12%	4756	5605	18%	0	0	0	0
rtl8723be	468	62	4212	4650	10%	5266	5918	12%	8	0	4	12
ehci_hcd	1064	53	1162	1341	15%	1547	2052	33%	0	0	0	0
xhci_hcd	4142	190	2316	3111	34%	2799	4097	46%	1	0	2	3
usb_storage	73	21	469	639	36%	523	750	43%	0	0	0	0
cmipci	136	35	355	395	11%	450	534	19%	0	0	0	0
maestro3	300	34	311	354	14%	408	500	23%	0	0	0	0
ymfpci	257	54	577	643	11%	736	906	23%	0	0	0	0
Total	20,988	1440	27,479	31,621	15%	33,986	41,066	21%	13	2	7	22

though there is no additional code coverage, the two error site sequence may cause the error handling code to be executed in different contexts, e.g. with respect to acquired locks and allocated memory regions, and thus using these error-site sequences can still reveal new bugs. It may be possible to use static analysis to detect contexts that have no impact on the execution of error handling code, and thus reduce the set of error-site sequences considered.

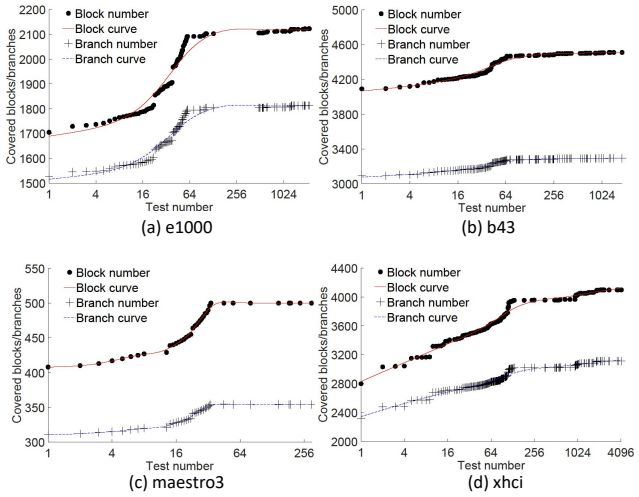

Fig. 10. Detailed results of code coverage for four drivers.

To know about the relationship between the code coverage and the number of tests, we collect the numbers of covered basic blocks and code branches in each test. Figure 10 shows the results for four randomly selected drivers (*e1000*, *b43*, *xhci* and *ymfpci*), each from a different driver class. The X axis shows the number of tests on a log scale; the Y axis shows the number of covered basic blocks and code branches. Note that in each graph, the Y axis does not start at 0. We only show the points that increase code coverage. The results show that code coverage increases quickly in the earlier tests, and then tends to be stable in the later tests.

Bug detection. FIZZER finds 22 new real bugs, including 13 crashes caused by NULL pointer dereferences, 2 use-after-free bugs and 7 resource leaks (including memory leaks). Crashes are found by observing driver execution and kernel logs; use-after-free bugs are found by KASAN; and resource leaks are found by Kmemleak and our resource-leak checkers. We find that all these found bugs are related to occasional errors, thus they are hard to detect and reproduce in real execution. We have reported these bugs to driver developers. 12 of them have been confirmed, and our 4 patches fixing 7 bugs have been applied by the kernel maintainers. We have not yet received a reply for the other bugs.

Reviewing the bugs found by FIZZER, we find that 4 bugs (2 crashes in the *rtl8723be* driver and 2 resource leaks in the *xhci* driver) involve function pointer calls in the code paths triggering these bugs. Detecting these 4 bugs is quite challenging for static analysis, as without runtime information, statically identifying the correct function(s) called by a function pointer call is often difficult. Besides, we observe that 14 bugs (7 resource leaks, 5 crashes and 2 use-after-free bugs) are caused by improper resource release. Specifically, the 7 resource leaks are caused by lacking necessary resource-release operations; the 5 crashes are caused by operating on unsuccessfully allocated resources (the related variables contain NULL pointers); and the 2 use-after-free bugs are caused by doubly releasing the same memory.

Example found bug. Figure 11 illustrates a NULL pointer dereference found in the *xhci* driver. The function `xhci_debugfs_create_slot` calls `kzalloc` on line 479 to allocate and zero memory. When `kzalloc` fails, `xhci_debugfs_create_slot` abnormally returns on line 481, without assigning the pointer `dev->debugfs_private` on line 486, so this pointer remains NULL at that time. After that,

the function `xhci_debugfs_create_endpoint` assigns `dev->debugfs_private` to `spriv` on line 441, and then dereferences `spriv` on line 443. But `spriv` is NULL, thus a NULL pointer dereference occurs and causes a system crash. This bug has been confirmed by the USB driver developers. To fix this bug, whether `spriv` is NULL should be checked before dereferencing it on line 443. Our patch making this change has been applied by the kernel maintainers [25].

```
FILE: linux-4.19/drivers/usb/host/xhci-debugfs.c
     /*** Executed later ***/
436. void xhci_debugfs_create_endpoint(...) {
       ......
       // dev->debugfs_private is NULL
441.   struct xhci_slot_priv *spriv = dev->debugfs_private;
442.
443.   if (spriv->eps[ep_index])  // Use a NULL pointer "spriv"
444.     return;
       ......
456. }
-----------------------------------------------------------
     /*** Executed first ***/
474. void xhci_debugfs_create_slot(...) {
       ......
       // Function call fails
479.   priv = kzalloc(sizeof(*priv), GFP_KERNEL);
480.   if (!priv)
481.     return;
       ......
       // dev->debugfs_private has not been assigned
486.   dev->debugfs_private = priv;
       ......
492. }
```

Fig. 11. A found NULL pointer dereference in the *xhci* driver.

D. Comparison to Existing Approaches

We select syzkaller [3] to make a detailed comparison with FIZZER. We select it for four reasons: 1) it is a start-of-the-art kernel fuzzing tool, and has been widely used to test the Linux kernel; 2) it has found hundreds of real bugs in Linux device drivers; 3) many driver fuzzing approaches [5]–[7] are based on syzkaller; 4) it is open-source and can be conveniently deployed on virtual machines or physical machines.

In design, syzkaller fuzzes system calls to test the Linux kernel, while FIZZER fuzzes error-sites sequences for fault injection to test device drivers. Thus, syzkaller test more kinds of infrequently executed code, but it cannot effectively cover error handling code, especially the code triggered by occasional errors. FIZZER is more specific and effective in testing error handling code, but is less effective in testing other kinds of infrequently executed code.

We run syzkaller on physical machines to test six of the network device drivers that were tested by FIZZER. We choose to run syzkaller on physical machines to use the same execution environment as FIZZER. The tested device drivers are randomly selected, including 3 Ethernet controller drivers and 3 wireless controller drivers. We configure syzkaller by enabling network-related syscalls, including `sys_accept`, `sys_socket`, `sys_connect` and so on. We run syzkaller on three PCs, namely one that is used as the server, and the other two that are used as the test clients. We run each device driver for 5 hours. The experimental results are shown in Table VII. The *"Block"* columns show the number of the covered basic blocks and the percentage increase in covered basic blocks as compared to normal execution; the *"Branch"* columns show the number of covered code branches and the percentage increase in covered code branches as compared to normal execution.

TABLE VII
COMPARISON OF SYZKALLER AND FIZZER.

Driver	syzkaller			FIZZER		
	Block	***Branch***	***Bug***	***Block***	***Branch***	***Bug***
e1000	1571 (+3%)	1742 (+2%)	0	1813 (+19%)	2122 (+24%)	1
e1000e	2074 (+1%)	2549 (+2%)	0	2709 (+32%)	3540 (+41%)	0
tg3	2474 (+2%)	3298 (+4%)	0	2657 (+9%)	3609 (+13%)	0
b43	3153 (+2%)	4168 (+2%)	0	3294 (+7%)	4510 (+10%)	1
ath9k	2526 (+6%)	3048 (+7%)	0	2657 (+12%)	3275 (+15%)	3
rtl8723be	4485 (+6%)	5603 (+6%)	0	4650 (+10%)	5918 (+12%)	12
Total	16,283 (+4%)	20,408 (+4%)	0	17,780 (+13%)	22,974 (+17%)	17

We find that FIZZER achieves higher code coverage than syzkaller, and finds many real bugs missed by syzkaller. Indeed, the tested drivers have much error handling code, especially triggered by occasional errors. But syzkaller cannot effectively cover error handling code by generating system calls, and thus misses related bugs. FIZZER performs SFI-based fuzzing that can effectively cover error handling code and detect related bugs. However, by checking the covered code, we also find that syzkaller covers some driver code that is uncovered by FIZZER. This code is related to driver configuration and control commands. syzkaller can mutate the arguments of system calls to cover this code, but FIZZER cannot do it.

VI. DISCUSSION

In this section, we discuss four questions about FIZZER:

Q1: *Can FIZZER automatically identify realistic error sites from driver code?*

Whether the identified error sites are realistic heavily affects the bug-detection results of FIZZER. If the error sites are unrealistic, the found bugs may be false. To reduce the manual work of identifying error sites, FIZZER performs an automated static analysis of driver code to recommend possible error sites. This static analysis drops many error sites considered to be unrealistic, but cannot ensure that all the recommended ones are realistic. Thus, the user is still required to manually select realistic ones from them. For example, in our evaluation, we manually select 53% of the recommended error sites for fuzzing, because we find that the remaining 47% may be unrealistic. In fact, automatically identifying realistic injected faults (namely error sites for FIZZER) is important for all SFI-based approaches, but no systematic and practical solutions are available. We believe that automatically dropping unrealistic error sites is useful to explore practical solutions.

Q2: *Can FIZZER find yet more bugs in the tested drivers?*

On the one hand, implementing more runtime checkers or using more third-party runtime checkers is helpful to detecting other kinds of bugs. For example, we plan to investigate the use of DataCollider [26] to detect data races. On the other hand, running more complex workloads can increase code coverage, which can increase the possibility of finding more bugs.

Q3: *Can FIZZER find false bugs?*

Though in our evaluation, all the bugs found by FIZZER are identified as real bugs, FIZZER may still find false bugs in some cases. For example, the user may select unrealistic error sites, which may cause FIZZER to find bugs that cannot possibly occur in real execution. Besides, the bug checkers used by FIZZER may also introduce false positives.

Q4: *Can FIZZER test other kinds of programs?*

The basic idea of our SFI-based fuzzing strategy is to fuzz injected faults according to the program's runtime information, to effectively cover error handling code. We only implement our strategy for device drivers at present. We believe that our strategy can be used to test other kinds of programs, including other kernel-level modules (such as file systems and network modules) and user-level applications, because they also have much error handling code (especially the code triggered by occasional errors, such as insufficient memory and file-operation failures). Thus, we plan to apply our strategy to testing these programs and detecting their bugs.

VII. Related Work

A. Fuzzing

Fuzzing [2] is a popular and promising technique of runtime testing. It generates a large number of inputs as test cases in a specific way to test programs. Fuzzing can be dumb or smart [27]. Dumb fuzzing generates inputs without knowledge of program behavior, and randomly changes the inputs. Smart fuzzing generates inputs according to program behavior, and thus is more effective in improving code coverage and detecting bugs.

Almost all the modern fuzzing approaches are smart, and many of them [28]–[35] are used to test user-level applications. AFLFast [29] models the procedure of coverage-based grey-box fuzzing as a systematic exploration of a Markov chain's state space, to select more powerful seeds that can generate effective test cases. CollAFL [30] solves the problem of hash collision that largely harms the accuracy of fuzzing large-scale applications, and thus enables more accurate edge coverage information. Based on the accurate coverage information, CollAFL uses three new seed selection policies to fuzz directly towards non-explored paths, which can achieve good efficiency and find deep bugs.

To find bugs and vulnerabilities in operating systems, some fuzzing approaches [3], [5]–[7], [11], [36]–[38] target kernel-level programs. For example, syzkaller [3] is a well-known kernel fuzzing tool developed and maintained by Google. It is a coverage-based fuzzer, and fuzzes system calls to test the OS kernel. It can perform fuzzing on slave virtual machines or physical machines. syzkaller has found over one thousand real bugs in the Linux kernel, including hundreds of bugs in device drivers [4]. Many approaches (such as [5]–[7], [36]) for fuzzing kernel-level programs are based on syzkaller. MoonShine [36] is a kernel fuzzing approach, that distills system call traces while still maintaining the dependencies across the system calls to maximize code coverage. It first executes the tested kernel-level programs and records their system call traces along with the code coverage achieved by each system call. Then, it greedily selects the system calls that contribute the most new code coverage, identifies their dependencies using lightweight static analysis and groups them into seed programs. MoonShine is implemented as an extension of syzkaller, and it can achieve higher code coverage and detect more deep bugs compared to syzkaller. DIFUZE [5] is an interface-aware driver fuzzing approach, which uses static analysis to generate correctly-structured inputs from userspace to explore device drivers. It first performs static analysis of the driver source code, and identifies possible effective inputs for driver interfaces. Then, it generates these inputs to fuzz device drivers through `IOCTL` related system calls. In this way, it can reduce many unnecessary inputs, and thus can achieve higher code coverage within less testing time. DIFUZE is implemented based on syzkaller and finds dozens of new previously unknown vulnerabilities on seven modern Android smartphones.

Previous driver fuzzing approaches generate inputs from system calls to cover infrequently executed code in device drivers, but this strategy cannot effectively cover error handling code, especially the code triggered by occasional errors. To solve this problem, our approach introduces software fault injection, and fuzzes injected faults according to runtime information. It can effectively cover error handling code.

B. Software Fault Injection

Software fault injection (SFI) [12] is a classical technique of runtime testing. It deliberately injects faults into the tested program, to cover infrequently executed code at runtime. It is often used to test error handling code in software systems.

Many SFI-based approaches [39]–[45] target user-level applications. Some of them [39]–[42] perform random fault injection, namely inject faults on random sites or replace program data with random faulty data, to validate whether the tested program can properly handle these faults. However, some studies [46]–[49] have shown that random fault injection often injects unrealistic faults, and thus it covers many infeasible code paths and reports many false bugs. To solve this problem, some approaches [43]–[45] use program information to inject more realistic faults and achieve higher code coverage within less testing time. For example, AFEX [43] uses a fitness-guided and feedback-based algorithm to search for high-impact faults in a fault space. It monitors program execution and uses the effect of previously injected faults to dynamically learn the structure of the fault space, and adaptively chooses new injected faults for subsequent tests.

Some SFI-based approaches [14]–[16], [21], [50], [51] can test kernel-level device drivers. Mendona et al. [21] propose a robustness-testing approach for Windows device drivers. This approach injects random faults on the arguments of frequently called kernel interfaces in device drivers. ADFI [14] performs a bounded trace-based iterative generation strategy to relieve the problem of fault scenario explosion, and exploits a permutation-based replay mechanism to guarantee the fidelity of runtime fault injection. EH-Test [16] uses a pattern-based

strategy to analyze driver code and extract possibly realistic injected faults, and uses single fault injection to cover error handling code in drivers.

Similar to ADFI, FIZZER uses the runtime information of the driver to guide fault injection, but they have different goals. ADFI aims to reduce the exploration of fault scenarios, while FIZZER aims to generate error-site sequences that can improve code coverage. To achieve this goal, FIZZER uses a new SFI-based fuzzing strategy that is different from the bounded trace-based iterative generation strategy used by ADFI. Similar to EH-Test, FIZZER analyzes driver source code to recommend possible error sites. But EH-Test only performs single fault injection and does not use runtime information to guide fault injection, and thus error handling code only triggered by multiple errors cannot be covered. Different from EH-Test, FIZZER can inject multiple faults according to runtime information, thus it can cover more error handling code and has better ability to find deep bugs. For example, in our evaluation, the two resource leaks found by FIZZER in the *xhci* driver are triggered by injecting two faults, and thus EH-Test cannot find these resource leaks.

C. Static Analysis of Error Handling Code

Some approaches [8], [52]–[55] use static analysis to detect problems in error handling code. PF-Miner [52] identifies error handling paths in Android kernel source code, and then respectively collects function call sequences in normal execution paths and error handling paths. By using statistical analysis to compare the collected function call sequences in these two kinds of code paths, PF-Miner mines frequently used resource-acquiring and resource-release function pairs as API rules. According to the mined rules, it again analyzes error handling code to detect related violations. EDP [53] performs a dataflow analysis to check how errors are propagated through file systems and storage device drivers. It constructs a function-call graph that covers all possible cases in which error codes propagate via return values or function arguments. By checking this call graph, EDP detects violations related to the handling of error codes.

Static analysis can conveniently analyze more error handling code without executing the tested program. However, because it lacks exact runtime information, static analysis often reports many false positives (for example, PF-Miner has a false positive rate of 27%). For FIZZER, it could be interesting to introduce static analysis of driver source code to drop useless generated error-site sequences, which can help to improve fuzzing efficiency and reduce testing time.

VIII. Conclusion

Fuzzing has been commonly used for runtime testing, and it has found many real bugs in device drivers. However, existing driver fuzzing approaches cannot effectively test error handling code in device drivers, which limits the ability to find bugs. To solve this problem, based on software fault injection (SFI), we propose a new fuzzing approach named FIZZER, to effectively test error handling code in device drivers. At compile time, FIZZER uses static analysis to recommend possible error sites, from which the user should select realistic ones that can actually trigger error handling code. During driver execution, by analyzing runtime information, FIZZER automatically fuzzes error-site sequences for fault injection to improve code coverage in runtime testing. We have evaluated FIZZER on 18 device drivers in Linux 4.19. It in total finds 22 real bugs, and 12 of them have been confirmed by driver developers. Besides, the code coverage is increased by over 15% compared to normal execution without fuzzing. Comparison to syzkaller shows that FIZZER can find many bugs missed by syzkaller and achieve higher code coverage.

FIZZER can be improved in some aspects. Firstly, FIZZER cannot ensure that all of the recommended error sites are realistic, and thus the user is still required to manually select realistic ones from them. To further reduce manual work, we will explore automated solutions for identifying realistic error sites. Secondly, FIZZER still generates many error-site sequences that are useless in improving code coverage. To solve this problem, we will introduce static analysis to drop these error-site sequences, which can help to improve fuzzing efficiency and reduce testing time. Finally, we only implement FIZZER for Linux device drivers at present. We plan to apply it to testing other programs, such as kernel-level file systems and user-level applications, because they also have much error handling code.

Acknowledgment

We would like to thank the Linux driver developers who gave helpful feedback on our bug reports. This work was supported by the China Postdoctoral Science Foundation under Project 2019T120093.

References

[1] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.
[2] "Fuzzing," https://en.wikipedia.org/wiki/Fuzzing.
[3] "Syzkaller: a kernel fuzzer," https://github.com/google/syzkaller.
[4] "Bugs found by syzkaller," https://syzkaller.appspot.com/upstream/fixed.
[5] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: interface aware fuzzing for kernel drivers," in *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)*, 2017, pp. 2123–2138.
[6] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: facilitating dynamic analysis of device drivers of mobile systems," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 291–307.
[7] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "RAZZER: finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019, pp. 279–293.
[8] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: detecting resource-release omission faults in error-handling code for systems software," in *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.
[9] S. Saha, J. L. Lawall, and G. Muller, "An approach to improving the structure of error-handling code in the Linux kernel," in *Proceedings of the 2011 International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2011, pp. 41–50.
[10] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 87–98.

[11] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: an effective probing and fuzzing framework for the hardware-OS boundary," in *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)*, 2019, pp. 1–15.

[12] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS)*, 1993, pp. 208–217.

[13] "Linux fault injection capabilities infrastructure," http://www.kernel.org/doc/Documentation/faultinjection/fault-injection.txt.

[14] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 361–372.

[15] V. S. D. Shekar, B. Meshram, and M. Varshapriya, "Device driver fault simulation using KEDR," *International Journal of Advanced Research in Computer Engineering and Technology*, pp. 580–584, 2012.

[16] J.-J. Bai, Y.-P. Wang, J. Yin, and S.-M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *Proceedings of 2016 USENIX Annual Technical Conference*, 2016, pp. 635–647.

[17] "LLVM compiler infrastructure," https://llvm.org/.

[18] "Commit 471b83bd8bbe: forbid enslaving team device to itself in the team driver," https://github.com/torvalds/linux/commit/471b83bd8bbe.

[19] "Commit cbcc607e1842: fix double free in error path in the team driver," https://github.com/torvalds/linux/commit/cbcc607e1842.

[20] "Linux kernel coding style," http://www.kernel.org/doc/Documentation/process/coding-style.rst.

[21] M. Mendonca and N. Neves, "Robustness testing of the Windows DDK," in *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 554–564.

[22] "Clang: a LLVM-based compiler for C/C++," https://clang.llvm.org/.

[23] "The kernel address sanitizer," https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[24] "The kernel memory leak detector," https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html.

[25] "Commit 5bce256f0b52: usb: xhci: Fix a potential null pointer dereference," https://github.com/torvalds/linux/commit/5bce256f0b52.

[26] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proceedings of the 9th International Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 151–162.

[27] C. Miller, "How smart is intelligent fuzzing or how stupid is dumb fuzzing?" *Independent Security Evaluators*, 2007.

[28] "American Fuzzy Lop (AFL)," http://lcamtuf.coredump.cx/afl/.

[29] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 5, pp. 489–506, 2017.

[30] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018, pp. 679–696.

[31] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: automatically generating pathological inputs," in *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 254–265.

[32] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, 2017, pp. 50–59.

[33] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: data-driven seed generation for fuzzing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: application-aware evolutionary fuzzing," in *Proceedings of the 24th Network and Distributed Systems Security Symposium (NDSS)*, 2017, pp. 1–14.

[35] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.

[36] S. Pailoor, A. Aday, and S. Jana, "MoonShine: optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 729–743.

[37] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 167–182.

[38] M. Mendonça and N. Neves, "Fuzzing Wi-Fi drivers to locate security vulnerabilities," in *Proceedings of the 7th European Dependable Computing Conference (EDCC)*, 2008, pp. 110–119.

[39] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott, "Testing of Java web services for robustness," in *Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 23–34.

[40] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: a programmable tool for multiple-failure injection," in *Proceedings of the 26th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 171–188.

[41] P. D. Marinescu and G. Candea, "LFI: a practical and general library-level fault injector," in *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 379–388.

[42] S. Ghosh and J. L. Kelly, "Bytecode fault injection for Java software," *Journal of Systems and Software (JSS)*, vol. 81, no. 11, pp. 2034–2043, 2008.

[43] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, 2012, pp. 281–294.

[44] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *Proceedings of the 19th Pacific Rim Symposium on Dependable Computing*, 2013, pp. 31–40.

[45] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code: an extended study in the mobile application domain," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 32, 2014.

[46] N. Kikuchi, T. Yoshimura, R. Sakuma, and K. Kono, "Do injected faults cause real failures? a case study of Linux," in *Proceedings of the 25th International Symposium on Software Reliability Engineering Workshops (ISSRE-W)*, 2014, pp. 174–179.

[47] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 397–408.

[48] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 1, pp. 80–96, 2013.

[49] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "Representativeness analysis of injected software faults in complex software," in *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 437–446.

[50] A. Johansson, N. Suri, and B. Murphy, "On the impact of injection triggers for OS robustness evaluation," in *Proceedings of the 18th International Symposium on Software Reliability Engineering (ISSRE)*, 2007, pp. 127–126.

[51] ——, "On the selection of error model(s) for OS robustness evaluation," in *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 502–511.

[52] H.-Q. Liu, Y.-P. Wang, J.-J. Bai, and S.-M. Hu, "PF-Miner: a practical paired functions mining method for Android kernel in error paths," *Journal of Systems and Software (JSS)*, vol. 121, pp. 234–246, 2016.

[53] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: error handling is occasionally correct." in *Proceedings of the 6th International Conference on File and Storage Technologies (FAST)*, 2008, pp. 207–222.

[54] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 496–506.

[55] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 345–362.