Testing Error Handling Code in Device Drivers Using Characteristic Fault Injection

Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu, *Tsinghua University*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/bai

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Testing Error Handling Code in Device Drivers Using Characteristic Fault Injection

Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu
Department of Computer Science and Technology, Tsinghua University

Abstract

Device drivers may encounter errors when communicating with OS kernel and hardware. However, error handling code often gets insufficient attention in driver development and testing, because these errors rarely occur in real execution. For this reason, many bugs are hidden in error handling code. Previous approaches for testing error handling code often neglect the characteristics of device drivers, so their efficiency and accuracy are limited. In this paper, we first study the source code of Linux drivers to find useful characteristics of error handling code. Then we use these characteristics in fault injection testing, and propose a novel approach named EH-Test, which can efficiently test error handling code in drivers. To improve the representativeness of injected faults, we design a pattern-based extraction strategy to automatically and accurately extract target functions which can actually fail and trigger error handling code. During execution, we use a monitor to record runtime information and pair checkers to check resource usages. We have evaluated EH-Test on 15 real Linux device drivers and found 50 new bugs in Linux 3.17.2. The code coverage is also effectively increased. Comparison experiments to previous related approaches also show the effectiveness of EH-Test.

1. Introduction

As important components of the operating system, device drivers control hardware and provide fundamental supports for high-level programs. During driver execution, different kinds of occasional errors may occur, such as kernel exceptions and hardware malfunctions [31]. Therefore, device drivers need error handling code to assure reliability. But in some drivers, error handling code is incorrect or even missed. In these drivers, serious problems like system crashes and hangs may occur when occasional errors are triggered. According to our study on Linux driver patches, more than 40% of accepted patches add or update corresponding error handling code. It shows that error handling code in device drivers is not reliable enough, so testing error handling code and detecting bugs inside are very necessary.

A challenge of testing error handling code is that occasional errors are infrequent to happen in real execution [34]. For example, "bad address" (EFAULT) is a common error should be handled, but it happens only when the memory or I/O address is invalid. Another example is hardware error, which happens only when the hardware malfunctions. Triggering these errors in real environment is very hard and uncontrollable.

To simulate software and hardware errors at runtime, *software fault injection* (SFI) is often used in driver testing. This technique mutates the code to inject specific errors into the program, and enforces error handling code to be executed at runtime. Linux Fault Injection Capabilities Infrastructure (LFICI) [43] is a well-known project integrated in Linux kernel. It can simulate common errors, such as memory-allocation failures and bad data requests. Inspired by LFICI, other fault injection approaches [7, 13, 23, 32] have been proposed in recent years, and they have shown promising results in driver testing and bug detection. However, these approaches still have some limitations in practical use.

- The representativeness of injected faults is often neglected, and most injected faults are random or manually selected. Random faults can not reflect real errors well. Manually selected faults often omit representative injected faults.
- Numerous redundant test cases are generated. In fact, many generated test cases may cover the same error handling code, but they all need to be actually tested at runtime. For this reason, they often spend much time in runtime testing.
- Only several kinds of faults can be injected, such as memory-allocation failures. But these faults can not cover most error handling code in drivers.
- Much manual effort is needed. The kinds and places of injected faults are often manually decided.

In fact, previous fault injection approaches aim to support general software, but they neglect the characteristics of target programs. To relieve their limitations, we should consider the key driver characteristics in SFI. For example, because drivers are often written in C, so built-in error handling mechanisms (such as "*try-catch*") are not supported. For this reason, the developers often use an *if* check to decide whether the error handling code should be triggered in device drivers. This characteristic can help to decide which functions can actually fail and should be fault-injected.

In this paper, we first study Linux driver code, and find three useful characteristics in error handling code: *function return value trigger*, *few branches* and *check decision*. Then based on these characteristics and SFI, we propose a practical approach named EH-Test[1] to efficiently test error handling code and detect bugs inside. Firstly, EH-Test uses a *pattern-based extraction strategy* to extract target functions which can fail from the captured runtime traces of normal execution. This strategy can automatically and accurately extract real target functions to improve the representativeness of inject faults. Then, we generate test cases by corrupting the return values of target functions. Next, we run each test case on the real hardware, and use a monitor to record runtime information and pair checkers to check resource-usage violations. These pair checkers contain the basic information of resource-acquiring and resource-releasing functions, which can be obtained from specification mining techniques [18, 20, 37, 38] and user configuration. During driver execution, system crashes and hangs can be easily identified through kernel crash logs or user observation. After driver execution, EH-Test can report resource-release omissions. We have implemented EH-Test using LLVM, and evaluated it on 15 Linux drivers of three classes. The results show that EH-Test can accurately find real bugs in error handling code and improve code coverage in runtime testing. Comparison experiments to previous approaches also show its effectiveness.

Compared to previous SFI approaches for testing drivers, our approach have four advantages:

1) Representative injected faults. We design a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults. It uses code patterns to decide whether a function can actually fail in driver execution. This strategy can largely improve the effectiveness of SFI.

2) Efficient test cases. According to our study, many drivers have few branches in error handling code, so injecting a single fault in each test case is enough to cover most error handling code. Moreover, our pattern-based extraction strategy can filter many unrepresentative injected faults. Therefore, the test cases generated by EH-Test are efficient, and the time usage of runtime testing can be largely shortened.

3) Accurate bug detection. By injecting representative faults, EH-Test can realistically simulate different kinds of occasional errors to cover error handling code. Moreover, EH-Test runs on the real hardware and uses exact execution information to perform analysis. These points assure the accuracy of bug detection.

[1] EH-Test program can be downloaded from the link: *http://oslab.cs.tsinghua.edu.cn/EHTest/index.html*

4) High automation and scalability. Most working procedure of EH-Test is automated, including target-function extraction, fault injection and test-case execution. And it can support many kinds of existing drivers.

In this paper, we make the following contributions:

- We study the source code of Linux device drivers, and find three useful characteristics in error handling code.
- Based on the patterns of error handling code in drivers, we design a pattern-based extraction strategy to automatically and accurately real target functions as representative injected faults.
- Based on the characteristics, we design a practical approach named EH-Test, which can efficiently test error handling code in device drivers and detect bugs inside.
- We evaluate EH-Test on 15 device drivers in Linux 3.1.1 and 3.17.2, and respectively find 32 and 50 bugs. All the detected bugs in 3.17.2 have been confirmed by developers. The code coverage is also effectively increased in runtime testing. We also perform comparison experiments to previous approaches, and find that EH-Test can detect the bugs which are missed by them. The experimental results show that EH-Test can efficiently perform driver testing and accurately find real bugs.

The rest of this paper is organized as follows. Section 2 introduces the motivation. Section 3 presents the three characteristics of device drivers found by our study on Linux driver code. Section 4 presents our pattern-based extraction strategy. Section 5 introduces EH-Test in detail. Section 6 shows our evaluation on 15 Linux device drivers and comparison experiments to previous approaches. Section 7 introduces the related work and Section 8 concludes this paper.

2. Motivation

To ensure the reliability of device drivers, error handling code should be correctly implemented to handle different kinds of occasional errors. But in fact, error handling code is incorrect or even missed in some drivers, so hard-to-find bugs may occur during execution. In this section, we first reveal this problem using a concrete example and our study on Linux driver patches, and then we sketch the software fault injection technique used in this paper.

2.1 Motivating Example

We first motivate our work using a real Linux driver *bnx2*. This driver manages Broadcom NetXtreme II Ethernet Controller. Figure 1 shows a part of its source code in Linux 3.1.1. The function *bnx2_init_board* calls *pci_request_regions* (line 7906) to request PCI I/O and memory resources when initializing the hardware. If an occasional error occurs when mapping bus memory into CPU, the function *ioremap_nocache* (line 7937) will fail and return NULL. In this situation, the driver calls *pci_release_regions* (line 8248) to release allocated resources in error handling code. Reviewing the code, the function *kzalloc* (line 7885) is called to allocate kernel-space memory. But this memory is not freed in error handling code, so a memory leak occurs. In fact, this bug still remains in Linux 3.17.2.

In this example, we have three findings. Firstly, error handling code in drivers is often used to release allocated resources and undo recent operations [30]. It is because that many drivers are based on the *fail-stop* model [33], namely a simple error can force the driver to exit. Due to this feature, many bugs in error handling are related to resource-usage violations, such as resource leaks and deadlocks. Secondly, error handling code is often written in a separate segment in drivers (line 8274-8253 in Figure 1 is an example), and different "*goto*" target labels handle different errors. This *goto-based* strategy is recommended by the Linux kernel documentation [44], because this strategy can simplify error handling logic and reduce repeated code. Thirdly, bugs in error handling code are hard-to-find. It is because that error handling code is rarely executed, and maintainers pay insufficient attention to it. In the example, from Linux 3.1.1 (released in November 2011) to 3.17.2 (released in October 2014), the memory leak in Figure 1 had not been fixed. Thus, it is very necessary to reveal and detect bugs in error handling code.

Path: linux-3.1.1/drivers/net/bnx2.c

```
7869. static int __devinit bnx2_init_board(....)
7870. {
          ......
7885.     bp->temp_stats_blk = kzalloc(...);
          ......
7906.     rc = pci_request_regions(pdev, ...);
          ......
7937.     bp->regview = ioremap_nocache(...);
7938.     if (!bp->regview) {
7939.         dev_err("Cannot map register space, aborting\n");
7940.         rx = -ENOMEM;
7940.         goto err_out_release;
7941.     }
          ......
8247. err_out_release:
8248.     pci_release_regions(pdev);
8249. err_out_disable:
8250.     pci_disable_device(pdev);
8251.     pci_set_drvdata(pdev, NULL);
8252. err_out:
8253.     return rc;
8254. }
```

Figure 1: Part of the bnx2 driver code in Linux 3.1.1.

Driver Class	Accepted Patches	Error Handling
I2C	29	13(44.83%)
PCI	38	13(34.21%)
PowePC	42	11(26.19%)
RTC	24	8(33.33%)
Network	598	253(42.31%)
Total	731	298(40.77%)

Table 1: Study result of Linux driver patches.

2.2 Study on Linux Patches

To clearly illustrate the reliability of current error handling code in device drivers, we make a study on Linux driver patches. We manually read patches in the Patchwork project[2] and select accepted patches from them in July 2015. These patches are from 5 driver classes, namely I2C bus drivers, PCI bus drivers, PowerPC drivers, real-time clock (RTC) drivers and network drivers. Among them, we identify those which add or update corresponding error handling code. The result is listed in Table 1. The first column presents the driver class name; the second column shows the number of accepted patches; the third column shows the number and percentage of accepted patches add or update corresponding error handling code.

From Table 1, we find that 40% accepted patches add or update corresponding error handling code. In these accepted patches, many are used to fix common bugs, such as memory leaks and null pointer dereferences. One reason for this phenomenon is that complex control flows and different kinds of occasional errors make it difficult to implement correct error handling code. Another reason is that error handling code is often triggered by specific and infrequent conditions (such as insufficient memory and hardware errors), so developers hardly test it well at runtime.

In brief, current error handling code in device drivers is not reliable enough as we expected, and many bugs are hidden in it. Once these bugs are triggered, serious system problems may occur, such as crashes and resource leaks. Therefore, it is important and necessary to test error handling code in device drivers and detect bugs inside.

2.3 Software Fault Injection

Software fault injection (SFI) is a widely used technique of testing error handling code. It intentionally introduces faults or occasional errors into the program, and then tests whether the program can correctly handle the injected faults or errors at runtime. In this paper, we use SFI to test drivers and detect bugs. To help better understand this paper, we explain several terms about SFI.

[2] Patchwork project. *http://patchwork.ozlabs.org/*

Fault Injection. We inject faults or errors to make error handling code executed at runtime. In this paper, fault injection and error injection [17] can be identical, and injected faults can also be called injected errors. As shown in Figure 1, we can inject a fault or error to make the function *ioremap_nocache* (line 7937) fail, and let its error handling code (line 8247-8253) executed.

Fault Representativeness. It reflects whether an injected fault can represent a real fault or error to trigger error handling code. If the injected fault is representative, it means that this fault or error can occur in real execution, so the bugs detected in this situation can be regarded as real bugs. Otherwise, the detected bugs are very probably false. Fault representativeness is a key factor, and it decides the effectiveness of SFI [24].

Target Function. A target function is a called function which can fail and trigger error handling code, so it should be fault-injected in SFI. A target function can be a kernel interface or defined in the driver code. *If a target function is real, its failure can be a representative injected fault, because its failure can cause a real error and actually trigger error handling code*. Namely, the realness of target functions largely decides the fault representativeness of SFI. For example in Figure 1, the function *ioremap_nocache* can actually fail and return a null pointer to trigger error handling code, so it is a real target function.

False Positive. There are two kinds of false positives in this paper. One is the false positive of fault representativeness, which is the injected fault that can not actually trigger error handling code. The other is the false positive of bug detection, which is the false detected bug.

Driver Class	**Number**	**"*Goto*" Statement**	**Return Value**
Wireless	116	5109	3757(73.54%)
Ethernet	219	6749	5192(76.93%)
Block	56	1322	1005(76.02%)
Bluetooth	21	121	89(73.56%)
Clock	117	260	213(81.92%)
PCI	51	467	351(75.16%)
USB	268	4148	2971(71.62%)
Total	848	18176	13578(74.70%)

Table 2: Study result of "*goto*" statements.

Driver Class	**Number**	**Error handling**	**Without Branch**
Wireless	116	3903	3111(79.71%)
Ethernet	219	2587	1941(75.03%)
Block	56	149	127(85.23%)
Bluetooth	21	330	239(72.42%)
Clock	117	467	422(90.36%)
PCI	51	470	371(78.94%)
USB	268	701	493(70.32%)
Total	848	8607	6704(77.89%)

Table 3: Study result of branches in error handling code.

3. Characteristics

Previous SFI approaches often have limitations in practical use, such as reporting many false bugs and needing much manual effort. One reason is that they are often used for general software, but neglect key characteristics of device drivers. To improve SFI in testing drivers, we first study the source code of Linux device drivers to find key characteristics of error handling code.

3.1 Function Return Value Trigger

Occasional errors in drivers are often triggered with the function failures, which are reflected as bad return values (null pointers or negative integers of error codes). As shown in Figure 1, when an error occurs in memory mapping, *ioremap_nocache* returns a null pointer. In the example, we find that *error handling code is triggered by a bad function return value*. To know about the proportion of this specific form, we write a program to automatically analyze the source code of 848 Linux (version 3.17.2) device drivers from 7 driver classes. These driver classes are all commonly used, so the study result on them can be applicative to most drivers. In the study, we search for "*goto*" statements in the code, because they are often the entries of error handling code according to the *goto-based* strategy [30]. The result is shown in Table 2. The first column shows the driver class name; the second column shows the number of drivers in each class; the third column shows the number of "*goto*" statements; the fourth column shows the number and proportion of "*goto*" statements in the "*if*" branches of bad function return values.

From Table 2, we find that about 75% of "*goto*" statements are in the "*if*" branches of bad function return values. It indicates that most error handling code in device drivers is triggered by bad function return values. There are two common data types of function return values in device drivers, namely pointer and integer. According to the Linux kernel documentation [44], a null pointer or non-zero integer indicates the operation failure. Moreover, different non-zero integers represent different failure types. For example, -EIO indicates an input/output error and -ENODEV indicates no such device. As for the remaining 25% "*goto*" statements, they are triggered by data failures in the code, such as erroneous data read from registers and bad device states.

3.2 Few Branches

In user-mode applications, error handling code often contains many *if* branches [39]. The main reason is that most user-mode applications are based on *fail-recovery* model. During recovery, error handling code should handle other errors. Therefore, multiple faults need to be injected in user-mode applications to cover most error handling code in runtime testing.

Different from user-mode applications, many device drivers are based on the *fail-stop* model [33]. Namely, when an error occurs, the driver only handles it and prepares to exit, but other errors are never handled at that time. Thus, *there are few if branches in error handling code of device drivers*. To validate this characteristic, we also write a program to automatically analyze the source code of these 848 Linux drivers. In the study, we first filter out all annotations and blank lines, and then count source code lines with and without *if* branches in error handling code. The result is shown in Table 3. The first column shows the driver class name; the second column shows the number of drivers in each class; the third column presents the number of source code lines in error handling code; the fourth column presents the number and proportion of source code lines without *if* branches in error handling code.

From Table 3, we can see that nearly 78% of error handling code is not in *if* branches in these drivers. It indicates that injecting a single fault in each test case is enough to cover most error handling code. This characteristic can help to simplify the complexity of injected faults and improve the efficiency of SFI. The remaining 22% error handling code is in *if* branches because different resource-usage states or device states need to be separately handled in the same error handling code.

This characteristic commonly exists in *fail-stop* drivers. However, some drivers like SATA are based on the *fail-recovery* model, namely they will restart when an error occurs. Thus, many branches are needed to handle the recovery procedure. For these drivers, injecting a single fault is not enough to cover most error handling code. In this paper, we mainly focus on *fail-stop* drivers, because they occupy a large part of existing drivers [34].

3.3 Check Decision

Linux drivers are often implemented in C, so built-in error handling mechanisms (such as "*try-catch*") are not supported. *To check whether an occasional error occur, an if check is often used in the source code*. The *if* statement checks whether the key data is erroneous and decides whether error handling code should be executed. This key data can be a common variable or a function return value. Thus, the characteristic in Section 3.1 can be regarded as an aspect of it. For example in Table 2, all "*goto*" statements triggered by bad function return values are in *if* checks. Particularly, most *if* checks for function return values only check whether the value is a null pointer or non-zero integer (line 7938 in Figure 1 is an example). Namely, different bad function return values are often handled by the same error handling code.

This *if* check decision characteristic is also recommended by the Linux kernel documentation [44]. It can help us inject more representative and efficient faults for SFI. Specifically, we can inject faults in the data checked by these *if* checks, to simulate more realistic errors in device drivers.

```
Procedure: Pattern-based extraction strategy
1:  func_set := ø; cand_set := ø; fault_set := ø;
2:  func_set := called functions in normal execution traces;
3:  foreach func in func_set do
4:      if GetRetType(func) == integer or pointer then
5:          AddSet(cand_set, func);
6:      end if
7   end foreach
8:  foreach func in cand_set do
9:      if func's RetVal is checked by "if" in the driver then
10:         AddSet(fault_set, func);
11:     else if func's RetVal is checked in other drivers then
12:         AddSet(fault_set, func);
13:     else if func's RetVal is specified to be checked then
14:         AddSet(fault_set, func);
15:     end if
16: end foreach
```

Figure 2: Procedure of extracting target functions.

4. Pattern-based Extraction

The representativeness of injected faults is a key factor of SFI [24]. This property largely determines the accuracy of bug detection and the efficiency of runtime testing. Injecting representative faults can simulate realistic errors to trigger real error handling, so detected bugs are very probably real. Meanwhile, useless test cases are less generated when the injected faults are representative, so the time usage can be largely reduced.

A common strategy is to inject random faults, which has been used in many previous SFI approaches [11, 14, 21, 22]. But some studies [15, 17, 24] have proved this strategy can not well represent real errors, and they also introduces many false positives in bug detection. Because most error handling code in drivers is triggered by bad function return values (in Section 3.1), it is feasible to inject faults in some manually selected target functions which can fail at runtime. This strategy has been used in some previous approaches [7, 32, 43] to test drivers, but it has three problems. Firstly, new target functions should be manually selected when testing a new driver. Secondly, it is hard to assure the selected target functions can actually trigger realistic errors at runtime. Thirdly, many real target functions may be omitted in manual selection.

Based on the characteristics mentioned in Section 3.1 and 3.3, we propose a *pattern-based extraction strategy* to automatically and accurately extract real target functions from the source code. Figure 2 shows the main procedure of this strategy, which consists of two phases.

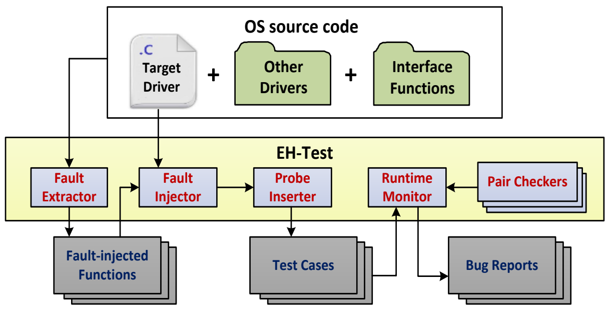

Figure 3: Overall architecture of EH-Test.

Firstly, we run the driver normally on the hardware, and record the runtime traces during normal execution. All functions whose return values are pointers and integers are selected from the recorded runtime traces. These functions are regarded as *candidate functions* for fault injection. Secondly, for each candidate function, we judge whether it can trigger a realistic error. According to the code patterns of Linux device drivers, a candidate function can be regarded as a real target function under three patterns.

Pattern 1. The return value of the candidate function is checked by an *if* statement in the driver. Because an *if* check is often used to decide whether error handling code should be triggered (in Section 3.3). In most cases, this *if* check is often closely behind the function call.

Pattern 2. The return value of the candidate function is checked in other drivers. In some cases, the developer may forget to check the return value of a certain function in the driver. But this function's bad return value can be deemed to trigger a realistic error, when it is checked by an *if* statement in other drivers.

Pattern 3. The return values of some kernel interface functions are clearly specified to be checked in their declarations or annotations, because they can trigger errors. For example in the Linux kernel code, a specific macro "*__must_check*" is defined. If this macro is noted in the declaration of a function, its return value must be checked. The function *pci_request_regions* in Figure 1 uses this macro. Besides, some key phrases in the function annotation also indicate the function return value should be checked. Therefore, the declaration and annotation of candidate functions should be checked as well.

This strategy has three advantages. Firstly, when the driver source code and hardware are available, this strategy can automatically extract target functions without manual effort. Secondly, by using exact runtime information and common code patterns, many unreal target functions are filtered out. Thirdly, no real target functions in the captured runtime traces are omitted. By using this strategy, we can automatically and accurately extract real target functions as representative injected faults to improve the effectiveness of SFI.

5. Approach

To efficiently test error handling code in device drivers, we propose EH-Test based on driver characteristics, code instrumentation and dynamic analysis. Figure 3 shows the overall architecture of EH-Test, which consists of five modules:

- ***Fault extractor.*** This module uses the pattern-based extraction strategy to automatically extract target functions. It needs the source code of the target driver, other drivers and kernel interface functions as input, which can be obtained from the OS source code.
- ***Fault injector.*** This module uses code instrumentation to inject faults by corrupting the return values of target functions. A single fault is injected in each test case.
- ***Probe inserter.*** This module instruments probes in the driver code to collect runtime information and count code coverage during execution. It outputs test cases of the tested driver. Each test case is a loadable driver, which can be directly installed in the operating system.
- ***Runtime monitor.*** This module runs test cases and records the runtime information of the tested driver during execution. It also detects bugs at runtime.
- ***Pair Checkers.*** They are used to check resource usages in drivers. Each pair checker contains the basic information of a pair of resource-acquiring and resource-releasing functions. We have written some pair checkers in EH-Test based on the result of specification mining techniques.

Based on the architecture, two phases are performed when EH-Test works, namely test case generation and runtime testing. The manual work only includes writing pair checkers, checking extracted target functions and rebooting the system when crash bugs are detected.

5.1 Test Case Generation

In this phase, we have two tasks, namely extracting target functions from the code and generating test cases of the driver by injecting faults on target functions. The detailed steps are as follows.

Firstly, we input the driver code and OS source code to the fault extractor. It uses the pattern-based extraction strategy to extract target functions. After extraction, the user also is allowed to check and modify target functions as needed.

Secondly, we inject faults into target functions. A key question is that how many faults should be injected in each test case. Many previous approaches [7, 22, 35, 39] inject multiple faults in each test case, because they aim to cover as much error handling code as possible. But fault scenario explosion may occur in this situation,

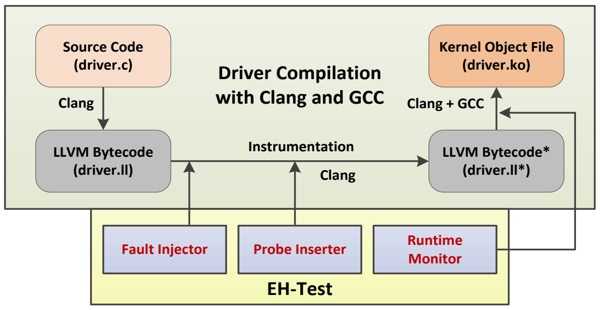
Source Code (driver.c)
Driver Compilation with Clang and GCC
Kernel Object File (driver.ko)
Clang
Clang + GCC
LLVM Bytecode (driver.ll)
Instrumentation
Clang
LLVM Bytecode* (driver.ll*)
Fault Injector
Probe Inserter
Runtime Monitor
EH-Test

Figure 4: Compilation procedure of the tested driver.

which can largely reduce testing efficiency. To relieve this problem and speed up testing, these approaches have to use some expedients, such as limiting the number of injected faults (or searching paths) [7, 39] and resorting to user guidance [22]. For many Linux drivers, a key characteristic is that there are few *if* branches in error handling code (in Section 3.2). Namely, the error handling code in many device drivers only handles a single error at a time. Thus, to cover most error handling code with less testing time, we only corrupt the return value of one target function in each test case. The target function call is replaced by an *error function* in the code. What this error function does is only returning a bad value. If the return value of the target function is a pointer, the error function will return a null pointer; if the return value of the target function is an integer, the error function will return a random negative number.

Thirdly, we instrument probes to collect runtime information and count code coverage during execution. The runtime information is used to detect bugs in the next phase. The code coverage is used to quantify the effectiveness of runtime testing. Finally, driver test cases are generated. Each test case is a kernel object file, namely a loadable driver.

In the second and third steps, code instrumentation is used. We implement it at compile time using the Clang [40] compiler. Figure 4 shows the compilation procedure of the tested driver. Firstly, we use the Clang compiler to compile the C source code of the driver into the LLVM bytecode. Secondly, we utilize the fault injector and probe inserter to instrument our handled code in the bytecode. Thirdly, we use the Clang compiler to compile the bytecode into the assembly code, and then build the object file using GCC. Finally, we link the object file and the runtime monitor's program together, and generate a kernel object file as a test case.

5.2 Runtime Testing

In this phase, we run each test case on the real hardware and detect bugs during execution. Three kinds of bugs are detected in current implementation, namely crashes, hangs and resource-release omissions.

When driver crashes occur, the OS outputs the dump information into the kernel crash log. Therefore, we can check the kernel crash log to detect and locate crash bugs like null pointer dereferences. For driver hangs, we can detect them by observing whether the system freezes. These two kinds of bugs are easy to observe in real execution.

Function Names	Description	Data
request_irq	Enable / disable the interrupt line and IRQ handling	Para1
free_irq		Para1
pci_enable_device	Initialize / disable the device on the PCI bus	Para1
pci_disable_device		Para1
dma_pool_alloc	Allocate / free a block of consistent memory for DMA	RetVal
dma_pool_free		Para1

Table 4: Selected paired functions in device drivers.

As for resource-release omissions, they are hard-to-find in real execution, because they rarely lead to obvious exceptions. However, they often cause resource-usage problems, such as resource leaks and memory leaks. Moreover, resource-release omissions often occur in device drivers, especially in error handling code [31]. For these reasons, EH-Test should detect resource-release omissions in device drivers. A resource-release omission occurs when a resource-acquiring function is successfully called but its resource-releasing function is not called. For example in Figure 1, *kzalloc* is a resource-acquiring function and it is used to allocate kernel memory, but the resource-releasing function *kfree* is not called, which leads to a resource-release omission. A resource-acquiring function and its resource-releasing function should be called in pairs, so they can be called *paired functions* [20]. Besides, they should operate the same mapped data (parameter or return value) as the handled resource. In EH-Test, we implement some pair checkers to detect resource-release omissions. Each pair checker contains the basic information of a pair of paired functions, including function names and mapped data. Some previous approaches for specification mining [18, 20, 37, 38] can be used to extract paired functions from the code. In this paper, we use the mining result of *PF-Miner* [20], which is a static approach for mining paired functions in Linux drivers, to build the pair checkers. Table 4 shows some selected paired functions in the checkers. The first column shows function names; the second column shows the description; the third column shows the mapped data.

During driver execution, the runtime monitor uses the inserted probes to record the runtime information of function calls and maintains a resource-usage list. For each function call, the monitor checks whether it is in the pair checkers. When a resource-acquiring function is called, the monitor checks its return value to judge whether the resource is successfully allocated. If it is true, the monitor will create a node containing the function name and mapped data, and add it into the resource-usage list. When a resource-releasing function is called, the monitor scans the list to match the node with the function information. If it is matched, the node will be deleted to indicate the resource is released. When the driver is removed, the monitor checks the nodes in the list. If the list is not empty, it indicates resource-release omissions occur, so the monitor will report them.

6. Evaluation

6.1 Experimental Setup

To validate the effectiveness of EH-Test, we evaluate it on real device drivers. The tested drivers should satisfy three criteria. Firstly, they should be commonly used in practice. Secondly, they should be within the driver classes in Section 3, because they can satisfy the characteristics found by the study. Thirdly, they should run as kernel modules, because the test cases of them can be directly installed and removed without rebooting the operating system. According to these criteria, 15 Linux device drivers are selected, including wireless, USB and Ethernet drivers. Table 5 shows the basic information of tested drivers in Linux 3.17.2.

Class	Driver	Hardware	Lines
Wireless	rtl8180	Realtek RTL8180L Wireless Controller	4.6K
	b43	Broadcom BCM4322 Wireless Controller	57.5K
	iwl4965	Intel 4965AGN Wireless Controller	29.1K
	rt2800	Ralink RT3060 Wireless Controller	22.5K
USB	usb_storage	Kingston 4GB USB disk	7.6K
	uhci_hcd	Intel USB UHCI Controller	7.2K
	ehci_hcd	Intel USB2 EHCI Controller	11.2K
Ethernet	e100	Intel 82559 Ethernet Controller	3.2K
	e1000e	Intel 82572EI Ethernet Controller	28.3K
	igb	Intel 82575EB Ethernet Controller	24.9K
	r8169	Realtek RTL8169 Ethernet Controller	7.4K
	8139too	Realtek RTL8139D Ethernet Controller	2.7K
	3c59x	3Com 3c905B Ethernet Controller	3.4K
	sky2	Marvell 88E8056 Ethernet Controller	7.7K
	ipg	ICPlus IP1000 Ethernet Controller	3.0K

Table 5: Tested drivers in Linux 3.17.2.

The experiment runs on a Lenovo PC with two Intel i5-3470@3.20G processors and 2GB physical memory. GCC 4.8 and Clang 3.2 are used for compilation. We write 75 pair checkers based on the result of *PF-Miner*. For each test case of the drivers, we install it in the system, run it on the workload, and finally remove it. The workload consists of three kinds. For wireless drivers, we turn on WiFi, ping another computer and turn off WiFi; for Ethernet drivers, we ping another computer; For USB drivers, we copy a 4MB file to the USB disk.

Driver	Candidate	Target	Real
rtl8180	39	18	17 (14)
b43	260	55	55 (46)
iwl4965	497	79	74 (64)
rt2800	185	65	57 (48)
usb_storage	60	20	15 (15)
uhci_hcd	120	24	19 (10)
ehci_hcd	160	23	21 (14)
e100	80	33	27 (26)
e1000e	175	62	56 (41)
igb	247	59	51 (51)
r8169	77	15	15 (14)
8139too	64	9	8 (7)
3c59x	59	15	14 (14)
sky2	86	30	25 (25)
ipg	74	16	16 (15)
Total	2183	523	470 (404)

Table 6: Result of the pattern-based extraction.

6.2 Target Function Extraction

The representativeness of injected faults is a key factor of SFI. In this paper, injected faults are bad return values of target functions, and all target functions are automatically extracted by our pattern-based extraction strategy. Thus, the fault representativeness of SFI largely depends on the effectiveness of our pattern-based extraction strategy. There are three important research questions about its effectiveness:

RQ1: *How many unrepresentative candidate functions are automatically filtered out?*
RQ2: *How much is the false positive rate of the strategy?*
RQ3: *How many real target functions are omitted?*

To answer these questions, we first evaluate EH-Test on the 15 drivers to extract candidate functions and target functions. Then we manually check the extracted target functions to judge their realness. Table 6 shows the result in Linux 3.17.2. The first column presents the driver name; the second column shows the number of candidate functions; the third column shows the number of extracted target functions; the fourth column shows the number of real target functions.

From Table 6, we can find that 523 target functions are extracted from 2183 candidate functions. It indicates that 76% candidate functions are automatically filtered out because they are unrepresentative, which can answer RQ1. By manually checking the documents and implementations of the extracted target functions, we find that 470 target functions are real, which means they can actually fail and trigger error handling code. It indicates that the false positive rate of our pattern-based extraction strategy is only 10%, which can answer RQ2. Many false target functions return integers which are also checked by *if* statements, but they reflect different driver configurations or states but never trigger occasional errors. Answering RQ3 is difficult, because target

Driver	Linux 3.1.1					Linux 3.17.2				
	Test case	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*	*Test case*	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*
rtl8180	16	03:14	0 / 0	1 (0)	1	18	04:21	0 / 0	3 (2)	3
b43	62	23:57	0 / 0	1 (1)	1	55	26:34	0 / 0	1 (1)	1
iwl4965	100	36:42	5 / 0	7 (7)	12	79	25:18	5 / 0	8 (8)	13
rt2800	62	19:21	1 / 0	1 (0)	2	65	21:37	0 / 0	1 (0)	1
usb_storage	25	03:35	0 / 0	0 (0)	0	20	03:07	0 / 0	0 (0)	0
uhci_hcd	22	03:47	0 / 0	0 (0)	0	24	03:20	0 / 0	0 (0)	0
ehci_hcd	24	03:50	0 / 0	1 (0)	1	23	03:58	0 / 0	10 (9)	10
e100	33	03:02	1 / 0	0 (0)	1	33	02:28	1 / 0	1 (1)	2
e1000e	66	11:01	0 / 0	0 (0)	0	62	10:30	3 / 0	3 (0)	6
igb	62	10:09	0 / 0	0 (0)	0	59	12:56	0 / 1	6 (6)	7
r8169	15	01:24	0 / 0	0 (0)	0	15	01:43	0 / 0	0 (0)	0
8139too	9	00:45	0 / 0	1 (0)	1	9	00:46	0 / 0	1 (0)	1
3c59x	18	01:26	0 / 0	2 (2)	2	15	01:24	0 / 0	2 (2)	2
sky2	26	01:43	3 / 0	8 (0)	11	30	02:14	4 / 0	0 (0)	4
ipg	17	01:16	0 / 0	0 (0)	0	16	01:28	0 / 0	0 (0)	0
Total	557	125:12	10 / 0	22 (10)	32	523	121:42	13 / 1	36 (29)	50

Table 7: Bug-detection result of EH-Test.

functions are extracted from normal execution traces, but different execution paths may have different runtime traces. Thus, the real target functions within the unexecuted paths will be omitted. However, we find that all target functions in the captured runtime traces are extracted by our strategy.

Reviewing the result, we also find an interesting phenomenon. Most target functions are in the initialization procedure. The data in the parenthesis of the fourth column show the numbers of these functions. They occupy 86% of all target functions. Namely, most kinds of occasional errors in drivers occur in the initialization procedure. In fact, it has been noted in the Linux driver manual [9], and our results can successfully verify it. The explanation for this phenomenon is that different kinds of configurations need to be made in the initialization, and each configuration can cause a kind of occasional error. After the driver is initialized, only several kinds of errors can occur in the running procedure.

6.3 Bug Detection

With the extracted target functions, we perform runtime testing to detect bugs in error handling code. Each test case is generated by making one target function fail. To validate whether EH-Test can find the known bugs having been fixed, we first use EH-Test to test the 15 drivers in an older Linux version 3.1.1 (released in November 2011). Then we test these drivers in a newer Linux version 3.17.2 (released in October 2014) to validate whether EH-Test can find new bugs. Table 7 shows the result. The first column shows the driver name; the second and seventh columns ("*Test case*") show the number of generated test cases; the third and eighth columns ("*Time usage*") present the time usage of the runtime testing; the fourth and ninth columns ("*Crash / Hang*") show the number of detected crashes and hangs; the fifth and tenth columns ("*Resource*") present the number of detected resource-release omissions; the sixth and eleventh columns ("*Bugs*") show the number of detected bugs. Specifically, the number of memory leaks is shown in the parenthesis of the fifth and tenth columns ("*Resource*"), because the memory leak is an important kind of resource-release omission.

From Table 7, we make the following observations:

Firstly, EH-Test finds 32 bugs in the 15 drivers in Linux 3.1.1, including 10 crashes and 22 resource-release omissions. Among these resource-release omissions, 10 are memory leaks. Reviewing the driver code, 8 resource-release omissions (*sky2* driver) and 1 crash (*rt2800* driver) have been fixed in Linux 3.17.2. It indicates that EH-Test can find the known bugs.

Secondly, EH-Test finds 50 bugs in the 15 drivers in Linux 3.17.2, including 13 crashes, 1 hang and 36 resource-release omissions. Among the resource-release omissions, 29 are memory leaks. Moreover, 23 bugs are reserved from the legacy code in 3.1.1, and 27 bugs are introduced due to new implementations. We send all the bugs to the driver developers, and all of them have been confirmed. We also send 17 patches[3] to fix them, and 15 have been applied by the maintainers. It indicates that EH-Test can accurately find new bugs in drivers.

Actually, a threat to validity is that false extracted target functions may introduce false bugs. In our evaluation, no false bugs are detected for this reason.

Thirdly, the time usage of EH-Test is short. About 2 hours are spent in totally testing the 15 drivers, and only several minutes are spent for most drivers. This time usage is shorter than many previous SFI approaches [7, 13, 23] for testing drivers. One reason is that EH-Test uses the pattern-based extraction strategy to filter out many unrepresentative candidate functions, so no redundant test cases are generated. Thus, the test cases are efficient, and the testing time is largely shortened.

[3] The patches can be found in the link: *http://oslab.cs.tsinghua.edu.cn/EHTest/patch.html*

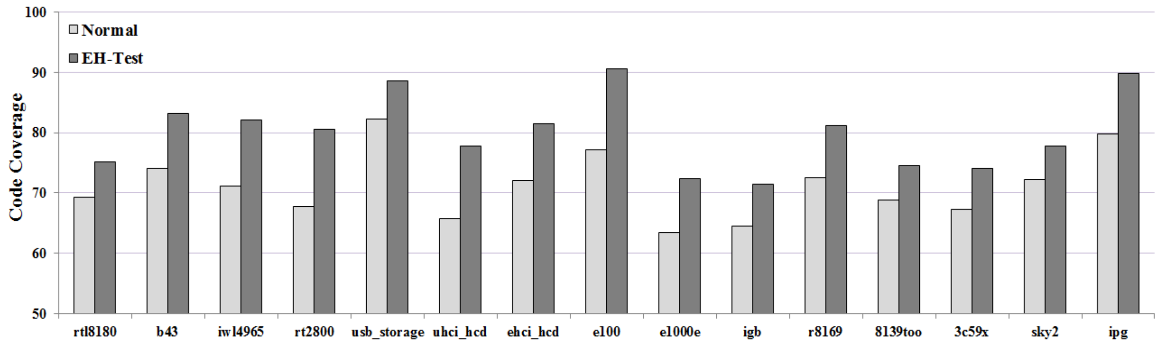

Figure 6: Code coverage of the tested drivers.

Fourthly, resource-release omissions occupy a large part of detected bugs. The main reason is that resource-release omissions often get little attention by developers. The complex execution paths make it difficult to correctly manage resources in error handling code. Meanwhile, resource-release omissions rarely lead to obvious exceptions, so they are hard-to-find in runtime testing.

Figure 5 shows a crash detected by EH-Test in the *e100* driver. The function *pci_pool_create* (line 2967) is called to create a pool of consistent memory blocks for the PCI device, and this function returns a pointer (*nic->cbs_pool*) to this memory area. But the function *pci_pool_create* may fail when the memory is insufficient, and it will return a null pointer in this situation. Thus, a null pointer dereference will occur, when the function *pci_pool_alloc* (line 1910) uses this pointer to allocate a memory block. This crash is detected when we inject a fault in the function *pci_pool_create*. This function is extracted as a target function in our pattern-based extraction strategy, because many other drivers check its return value in the code (pattern 2 in Section 4). To fix this bug, we add an *if* check after the function *pci_pool_create* (line 2967) to check its return value and implement the corresponding error handling code.

Path: linux-3.17.2/drivers/net/ethernet/intel/e100.c

```
1901. static int e100_alloc_cbs(...)
1902. {
            ......
1910.       nic->cbs = pci_pool_alloc(nic->cbs_pool, ...);
1911.       if (!nic->cbs)
1912.           return -ENOMEM;
            ......
1929. }
-------------------------------------------------
2843. static int e100_probe(....)
2844. {
            ......
2967.       nic->cbs_pool = pci_pool_create(...);
2968.       netif_info(...);
            ......
2990. }
```

Figure 5: A detected bug in the *e100* driver[4].

[4] The applied patch for this bug is in the link: *http://marc.info/?l=linux-netdev&m=143993218231729&w=2*

6.4 Code Coverage

Code coverage is a key criterion in runtime testing. To calculate code coverage, we use the inserted probes to count executed instructions at runtime. Because most target functions are in the initialization procedure, we focus on measuring the code coverage in this procedure. Figure 6 shows the results in Linux 3.17.2. The average code coverage of EH-Test is increased by 8.82% compared to the normal execution. It indicates that hundreds of more instructions are executed in runtime testing.

In fact, not all error handling code can be covered by EH-Test. Firstly, EH-Test only injects faults in target functions, but some error handling code is triggered by erroneous data read from hardware registers. Secondly, our approach injects a single fault in each test case, but some error handling code is triggered by multiple errors. Thirdly, target functions in unexecuted paths are not extracted in our pattern-based strategy, so their error handling code can not be covered. These points cause that the bugs in the uncovered code will be missed.

6.5 Comparison Experiments

Software fault injection and symbolic execution are two runtime techniques which are often used to test drivers.

Software Fault Injection. We compare EH-Test to ADFI [7], a state-of-the-art SFI approach for testing drivers. It uses a bounded trace-based iterative strategy to relieve fault scenario explosion and a permutation-based replay mechanism to assure the fidelity of fault injection. Similar to EH-Test, it injects faults in some target functions and generates test cases to detect bugs. But there are two main differences between ADFI and EH-Test. Firstly, the target functions in ADFI are all manually selected. Only memory, DMA and PCI related interfaces are considered. Thus, much manual work is needed, and many real target functions may be omitted. EH-Test can automatically and accurately extract all target functions in the captured runtime traces without omissions, and the only optional manual work is checking the extracted target functions. Secondly, ADFI injects multiple faults in each test case. The advantage is that much more configuration and error handling code can be covered to detect more bugs. But numerous test cases are generated, so it spends much more time (often several hours) than EH-Test when testing a driver.

ADFI program and its detailed bug reports are not available, thus we compare the number of its detected bugs from the paper. ADFI and EH-Test both test three drivers with the same workload. For the *e100* and *r8169* drivers, they both find the same number of bugs. For the *ehci_hcd* driver, EH-Test finds 10 bugs, but ADFI does not find any bug in this driver. Reviewing the code, we find that these bugs are triggered by the target functions which are not memory, DMA or PCI related functions, so ADFI omits them.

Symbolic Execution. We select SymDrive [28], a famous symbolic execution approach to make the comparison. This approach uses a symbolic device and some checkers to detect bugs, including memory leaks and null pointer dereferences.

SymDrive program is open-source, and we successfully run it to test the *e100* driver. It runs for nearly 80 minutes and searches 4838 paths, finally exits due to insufficient disk space. In the experiment, SymDrive does not find the bug shown in Figure 5. The reason is that the return value of the function *pci_pool_create* is not marked as a symbolic value in SymDrive, so the corresponding error handling path is not searched. Besides, SymDrive can only test the drivers whose devices are supported by QEMU [4]. But many devices are not supported by the QEMU used in SymDrive, so the drivers for these devices can not be directly tested. The *sky2* and *iwl4965* drivers are the typical examples.

7. Related Work

7.1 Software Fault Injection

In software testing, software fault injection is a technique for testing rarely executed code by deliberately injecting faults. In particular, it is often used to test error handling code in software systems.

Many approaches [11, 14, 21, 22] use random fault injection in software testing. They replace the program data with random faulty data or inject faults into random places, and then run test cases to validate whether the software can properly handle the faults. But random fault injection often leads to poor code coverage and low bug-detection accuracy. To relieve this problem, some approaches [3, 12, 39] use program information to guide fault injection and generate efficient test cases. Moreover, to improve the representativeness of injected faults, much research [10, 17, 24, 25] gives useful solutions through empirical studies. In fact, these approaches are often used for general software, especially user-mode applications. They often neglect the characteristics of device drivers, so their effectiveness may be largely limited when directly testing device drivers.

Besides user-mode applications, SFI is also carefully designed to test drivers [7, 13, 23, 29, 32, 35]. Mendonca et al. [23] perform robustness testing for Windows drivers based on random fault injection. Frequently used kernel interfaces in drivers are called with random parameters. ADFI [7] uses a bounded trace-based iterative strategy to relieve fault scenario explosion and a permutation-based replay mechanism to assure the fidelity of fault injection. But these approaches still have limitations when testing device drivers. A typical limitation is that they often neglect the representativeness of injected faults, and their injected faults are often random or manually selected. To relieve this limitation, EH-Test uses a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults.

7.2 Symbolic Execution

Some approaches [5, 8, 16, 28] introduce symbolic execution in driver testing without the real hardware. DDT [16] is a tool for testing binary drivers against undesired behaviors, such as resource leaks and race conditions. It combines virtualization with selective symbolic execution to test drivers using some modular dynamic checkers. SymDrive [28] provides a symbolic device based on QEMU [4] to simulate real hardware behaviors. It utilizes a favor-success path-selection algorithm in execution, which can increase the exploration priority of executing path at every successful function return within drivers and kernel interfaces. Some checkers are provided to detect common bugs like memory leaks.

Symbolic execution is very time consuming, and it needs much programmer guidance to avoid path explosion. Moreover, many devices can not be simulated well in virtual machines, so their drivers can not be directly tested using symbolic execution.

7.3 Static Analysis

Static analysis is often used to detect bugs in device drivers. It only analyzes the driver source code or binary code without actually running target drivers. Some approaches [1, 2, 6, 26, 27, 36, 41, 42] are based on program verification. For example, SDV [2] is a famous static tool to verify Windows drivers. It abstracts the C source code to a simpler form encoded as a state machine, and checks violations of kernel API usage rules. Some approaches [18, 19, 20, 31] mine implicit specifications from the driver code and detect related bugs. For example, PR-Miner [19] exploits data mining techniques to automatically extract implicit programming rules from software code and detect violations against these extracted rules.

Compared to runtime testing, static analysis lacks precise context information of real execution, so some false positives may be introduced in bug detection.

8. Conclusion

In this paper, we first study the source code of Linux drivers, and find three useful characteristics of error handling code. Then based on these characteristics, we propose a practical approach named EH-Test to efficiently test error handling code and detect bugs inside. It uses a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults. It has been evaluated on 15 Linux drivers and found 50 new real bugs. Our work shows that by introducing the characteristics of target programs, software testing can be more effective.

9. Acknowledgments

We would like to express our gratitude to our shepherd Jon Howell and other reviewers for their helpful comments and suggestions. We also thank the Linux driver developers for their useful feedbacks. This work is supported by Research Grant of Beijing Higher Institution Engineering Research Center and Tsinghua University Initiative Scientific Research Program (2014z09102). Shi-Min Hu is the corresponding author of this paper.

References

[1] S. Amani, L. Ryzhyk, A. F. Donaldson, G. Heiser, A. Legg and Y. Zhu. Static analysis of device drivers: we can do better. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, 2011.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems*, pages 75-88, 2006.

[3] R. Banabic and G. Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th European conference on Computer Systems*, pages 281-294, 2012.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41-46, 2005.

[5] V. Chipounov, V. Kuznetsov and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265-278, 2011.

[6] E. Clarke, D. Kroening and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168-176, 2004.

[7] 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*, pages 361-372, 2015.

[8] K. Cong, F. Xie and L. Lei. Symbolic execution of virtual devices. In *Proceedings of the 13th International Conference on Quality Software*, pages 1-10, 2013.

[9] J. Corbet, A.Rubini and G. K. Hartman, In *Linux Device Drivers*, 3rd ed. O'Reilly Media, pp. 32-35, 2005.

[10] J. A. Duraes and H. S. Madeira. Emulation of software faults: a field data study and a practical approach. In *IEEE Transactions on Software Engineering*, volume 32, issue 11, pages 849-867, 2006.

[11] 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*, pages 23-34, 2004.

[12] 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*, pages 31-40, 2013.

[13] A. Johansson, N. Suri and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 502-511, 2007.

[14] P. Joshi, H. S. Gunawi and K. Sen. PREFAIL: a programmable tool for multiple-failure injection. In *Proceedings of the 2011 International Conference on Object Oriented Programming Systems Languages and Applications*, pages 171-188, 2011.

[15] N. Kikuchi, T. Yoshimura, R. Sakuma and K. Kono. Do injected faults cause real failures? A case study of Linux. In *Proceedings of the 2014 International Symposium on Software Reliability Engineering Workshops*, 2014.

[16] V. Kuznetsov, V. Chipounov and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.

[17] 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*, pages 397-408, 2014.

[18] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart and G. Muller. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks*, pages 43-52, 2009.

[19] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering*, pp. 306-315, 2005.

[20] H. Liu, Y. Wang, L. Jiang and S. Hu. PF-Miner: a new paired functions mining method for Android kernel in error paths. In *Proceedings of the 38th International Computer Software and Applications Conference*, pages 33-42, 2014.

[21] 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*, pages 379-388, 2009.

[22] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. In *ACM Transactions on Computer Systems*, volume 29, issue 4, 2011.

[23] M. Mendonca and N. Neves. Robustness testing of the Windows DDK. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 554-564, 2007.

[24] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Maderia. On fault representativeness of software fault injection. In *IEEE Transactions on Software Engineering*, volume 39, issue 1, pages 80-96, 2013.

[25] R. Natella, D. Cotroneo, J. Duraes and H. Maderia. Representativeness analysis of injected software faults in complex software. In *Proceedings of the 40th International Conference on Dependable Systems and Networks*, pages 437-446, 2010.

[26] J. Obdrzalek, J. Slaby and M. Trtik. STANSE: bug-finding framework for C programs. In *Mathematical and Engineering Methods in Computer Science*, volume 7119, pages 167-178, 2012.

[27] H. Post and W. Kuchlin. Integrated static analysis for Linux device driver verification. In *Proceedings of the 6th International Conference on Integrated Formal Methods*, pages 518-537, 2007.

[28] M. J. Renzelmann, A. Kadav and M. M. Swift. SymDrive: testing drivers without devices. In *Proceedings of the 10th International Conference on Operating Systems Design and Implementation*, pages 279-292, 2012.

[29] V. V. Rubanov and E. A. Shatokhin. Runtime verification of Linux kernel modules based on call interception. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 180-189, 2011.

[30] S. Saha, J. 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*, pages 41-50, 2011.

[31] S. Saha, J. P. 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*, pages 1-12, 2013.

[32] S. D. Shekar, B. B. Meshram and M. P. Varshapriya. Device driver fault simulation using KEDR. In *International Journal of Advanced Research in Computer Engineering and Technology*, pages 580-584, 2012.

[33] M. M. Swift, M. Annamalai, B. N. Bershad and H. M. Leny. Recovering device drivers. In *ACM Transactions on Computer Systems*, volume 24, pages 333-360, 2006.

[34] M. M. Swift, B. N. Bershad and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207-222, 2003.

[35] S. Winter, M. Tretter, B. Sattler and N. Suri. simFI: From single to simultaneous software fault injections. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks*, pages 1-12, 2013.

[36] T. Witkowski, N. Blanc, D. Kroening and G. Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 501-504, 2007.

[37] Q. Wu, G. Liang, Q. Wang, T. Xie and H. Mei. Iterative mining of resource-releasing specifications. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 233-242, 2011.

[38] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, pages 282-291, 2006.

[39] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, pages 595-605, 2012.

[40] Clang compiler.
http://clang.llvm.org/

[41] Clang Static Analyzer.
http://clang-analyzer.llvm.org/

[42] Cppcheck: A tool for static C/C++ code analysis.
http://cppcheck.sourceforge.net/

[43] Linux Fault Injection Capabilities Infrastructure.
http://www.kernel.org/doc/Documentation/fault-injection/ fault-injection.txt

[44] Linux Kernel Coding Style.
http://www.kernel.org/doc/Documentation/CodingStyle