Automatic Fault Injection for Driver Robustness Testing

Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie
Department of Computer Science, Portland State University, Portland, OR 97207, USA
{congkai, leil, zhenkun, xie}@cs.pdx.edu

ABSTRACT

Robustness testing is a crucial stage in the device driver development cycle. To accelerate driver robustness testing, effective fault scenarios need to be generated and injected without requiring much time and human effort. In this paper, we present a practical approach to automatic runtime generation and injection of fault scenarios for driver robustness testing. We identify target functions that can fail from runtime execution traces, generate effective fault scenarios on these target functions using a bounded trace-based iterative strategy, and inject the generated fault scenarios at runtime to test driver robustness using a permutation-based injection mechanism. We have evaluated our approach on 12 Linux device drivers and found 28 severe bugs. All these bugs have been further validated via manual fault injection. The results demonstrate that our approach is useful and efficient in generating fault scenarios for driver robustness testing with little manual effort.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance; D.2.5 [**Testing and Debugging**]: Error handling and recovery

General Terms

Experimentation

Keywords

Fault Injection, Fault Scenario Generation, Driver Robustness Testing

1. INTRODUCTION

Robustness testing is a crucial stage in the device driver development cycle. Device drivers may behave correctly in normal system environments, but fail to handle corner cases when experiencing system errors, such as low resource situations, PCI bus errors and DMA failures [32]. Therefore, it is critical to conduct such robustness testing to improve driver reliability. However, such corner cases are usually difficult to trigger when testing drivers. The time-to-market pressure further exacerbates the problem by limiting the time allocated for driver testing [30]. Thus, it is highly desirable to speed-up driver robustness testing and reduce human effort.

Fault injection is a technique for software robustness testing by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be traversed. Recently, fault injection techniques have been widely used for software testing [21, 24]. These techniques have major potential to play a crucial role in driver robustness testing.

Our approach is inspired by Linux Fault Injection Infrastructure (LFII) [19] which has been integrated into the Linux kernel since Version 2.6.19. LFII can cause system faults, such as memory allocation functions returning errors, for system robustness testing. Our concept of faults is consistent with that of LFII. There are also other similar studies focusing on fault injection techniques for driver robustness testing [27, 34]. However, these approaches and tools have obvious limitations. First, they only provide basic frameworks which mainly support low memory situations. Second, they only support random fault injection which is hard to control and inefficient. Third, they require much human effort and time to get good results and are not easy-to-use. This demands an innovative approach to systematic and effective fault generation and injection for driver robustness testing.

We have developed an approach to automatic runtime fault generation and injection for driver robustness testing. Our approach runs a driver test and collects the corresponding runtime trace. Then we identify target functions which can fail from the captured trace, and generate effective fault scenarios on these target functions. Each generated fault scenario includes a fault configuration which is applied to guide further fault injection. Each fault scenario is applied to guide one instance of runtime fault injection and generate further fault scenarios. This process is repeated until all fault scenarios have been tested. To achieve systematic and effective fault injection, we have developed two key strategies. First, a bounded trace-based iterative generation strategy is developed for generating effective fault scenarios. Second, a permutation-based injection strategy is developed to assure the fidelity of runtime fault injection.

We have implemented our approach in a prototype driver robustness testing tool, namely, ADFI (Automatic Driver Fault Injection). ADFI has been applied to 12 widely-used device drivers. ADFI generated thousands of fault scenarios and injected them at runtime automatically. After applying all these generated fault scenarios to driver testing, ADFI detected 28 severe driver bugs. Among these bugs, 8 bugs are caused by low resource situations, 8 bugs are caused by PCI bus errors, 8 bugs are caused by DMA failures and the other 4 bugs are caused by mixed situations.

Our research makes the following three key contributions:

1)Automatic Fault Injection. Our approach to driver robustness testing not only enables runtime fault injection to simulate system errors, but also generates fault scenarios automatically based on the runtime trace to exercise possible error conditions of a driver efficiently. Our approach is easy to use and requires minimum manual efforts, which greatly reduces driver testing costs and accelerates testing process.

2)Bounded Trace-based Iterative Generation Strategy. A bounded trace-based iterative generation strategy is developed to generate unique and effective fault scenarios based on runtime traces. This strategy not only generates effective fault scenarios covering different kinds of error situations in modest time, but also produces efficient fault scenarios with no redundancy.

3)Permutation-based Replay Mechanism. To assure the fidelity of runtime fault injection with generated fault scenarios, a permutation-based replay mechanism is developed to handle software concurrency and runtime uncertainty. The mechanism guarantees that the same driver behaviors can be triggered using the same fault scenario repeatedly at runtime.

The remainder of this paper is structured as follows. Section 2 provides the background. Sections 3 and 4 present the design of our approach. Section 5 discusses its implementation. Section 6 elaborates on the case studies we have conducted and discusses the experimental results. Section 7 reviews related work. Section 8 concludes our work.

2. BACKGROUND

2.1 Driver Robustness Testing

According to the IEEE standard [1], robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions in software testing. The goal of robustness testing is to develop test cases and test environments where the robustness of a system can be assessed.

Kernel modules, especially device drivers, play a critical role in operating systems. It is important to assure that device drivers behave safely and reliably to avoid system crashes. Typically device drivers can work correctly under normal situations. However, it is easy for driver developers to mishandle certain corner cases, such as low resource situations, PCI bus errors and DMA failures.

```
int * p = (int *)kmalloc(size, GFP_ATOMIC);
p[10] = 3;
```

Figure 1: An example with kernel API function call

As shown in Figure 1, the *kmalloc* function is invoked to allocate a block of memory. After the function returns, the returned pointer is directly used without null pointer checking. Under normal system conditions, the *kmalloc* function returns successfully with a correct pointer to the allocated memory. However, when the *kmalloc* function returns a null pointer under a low resource situation, it is possible for the driver to crash the system. To handle such errors, the common approach is to add an error handling mechanism.

```
int * p = (int *)kmalloc(size, GFP_ATOMIC);
if(!p) goto error;
p[10] = 3;
......
error: error_handler();
```

Figure 2: An example with error handler

As shown in Figure 2, after the *kmalloc* function returns, the code checks whether the return value is a null pointer . If the *kmalloc* function returns a null pointer, the corresponding error handler is invoked to handle the error. However, a further concern is whether the error is handled correctly and does not trigger other driver or system errors.

To improve driver robustness, a device driver should be tested to see whether there exist two kinds of bugs: (1) driver error handling code does not exist; (2) driver error handling mechanisms do not handle the error correctly or trigger other driver/system issues. The first kind seems to be easy to avoid as long as driver developers write and check the code carefully. However, it still happens in the real world. The second kind is usually difficult and expensive to test.

2.2 Runtime Driver Fault Injection

In driver robustness testing, all possible error conditions of a driver ought to be exercised. However, certain error conditions might be difficult and expensive to trigger, but efforts should be made to force or to simulate such errors to test the driver. Fault injection is a technique for software robustness testing by introducing faults to test code paths, in particular error handling code paths that, otherwise, might rarely be followed. Recently, fault injection techniques have been widely explored and studied for software testing and system robustness testing.

```
void * kmalloc(size_t size, int flags) {
    // Memory allocation operations
}

void * kmalloc_fault(size_t size, int flags) {
    return NULL;
}
```

Figure 3: A driver fault injection example

Runtime driver fault injection can be employed to simulate kernel interface failures to trigger and test error handling code. The common approach to driver fault injection is to hijack the kernel function calls, such as *kmalloc* and *vmalloc*. By hijacking these functions, we can call the corresponding fault function to return a false result instead of invoking these functions. As shown in Figure 3, when *kmalloc* is invoked, the corresponding fault function $kmalloc\_fault$ is invoked to return a null pointer instead of a correct pointer to simulate an allocation error. In this way, we can test if device drivers can survive on different error handling code paths to improve driver robustness.

There are two main limitations with current driver fault injection. First, there is no automatic framework to support fault injection for different system function calls. Second, there is no systematic test generation approach to generate effective fault scenarios. Currently most fault injections tools are using random fault injection which is facing major challenges in achieving desired effectiveness and avoiding duplicate fault scenarios.

In our approach, we provide a framework which can automatically generate and inject fault scenarios at runtime. We proposed a trace-based iterative generation strategy to produce unique and effective fault scenarios and developed a permutation-based replay mechanism to inject fault scenarios with high fidelity.

3. BOUNDED TRACE-BASED ITERATIVE FAULT GENERATION

3.1 Preliminary Definitions

To help better understand our approach, we first introduce several definitions and illustrate them with examples.

*Definition 1 (**target function**):* A target function $\tilde{f}$ is a kernel API function which can fail and return an error when $\tilde{f}$ is invoked by a device driver.

As shown in Figure 1, function *kmalloc* is a target function since it can fail and return a null pointer.

A stack trace records a sequence of function call frames at a certain point during the execution of a program which allows tracking the sequence of nested functions called [33].

*Definition 2 (**target stack trace**):* A target stack trace $\tau \triangleq f_1 \rightarrow f_2 \rightarrow ...\rightarrow f_n \rightarrow \tilde{f}$ of a driver consists of a sequence of driver functions and a target function $\tilde{f}$. The sequence of driver functions are called prior to $\tilde{f}$ along a driver path. The first function f_1 is a driver entry function.

```
void Entry_A() { //Driver entry function
    ......
    ret = Target_Function_1();
    if(!ret) goto error;
    Function_X();
    ......
}

void Function_X() {
    ......
    ret = Target_Function_2();
    ......
}

void Entry_B() { //Driver entry function
    ......
    ret = Target_Function_3();
    ......
}
```

Figure 4: A driver function call example

A target stack trace τ records what happened before a target function was invoked. Once a driver/system crash happens, the target stack trace can help the developer better understand the driver behavior. The same target functions can appear in different target stack traces since the same target functions can be invoked along different driver paths.

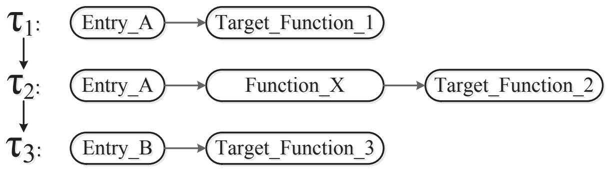

Figure 5: Target stack trace examples

As shown in Figure 4, when driver entry functions *Entry_A* and *Entry_B* are invoked during driver execution, there are three possible target stack traces τ_1, τ_2 and τ_3 shown in Figure 5.

*Definition 3 (**runtime trace**):* A runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow ...\rightarrow \tau_n$ is a sequence of target stack traces. A subsequence ε_k of ε contains the first k target stack traces of ε where $\varepsilon_k \triangleq \tau_1 \rightarrow \tau_2 \rightarrow ... \rightarrow \tau_k$. A runtime trace records all target stack traces during a driver life cycle.

A runtime trace example is shown in Figure 5 which is $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$.

*Definition 4 (**fault configuration**):* A fault configuration $\phi \triangleq \varphi_1, \varphi_2, ..., \varphi_n$ is a sequence of boolean variables. Each boolean variable φ_i (T or F) is used for deciding whether the corresponding target function $\tilde{f}$ of τ_i invokes the kernel API or returns error. A subsequence of ϕ_k of ϕ contains the first k boolean variables of ϕ where $\phi_k \triangleq \varphi_1, \varphi_2, \varphi_3, ..., \varphi_k$.

*Definition 5 (**fault scenario**):* A fault scenario $\sigma \triangleq \langle\varepsilon, \phi\rangle$ is a pair of ε and ϕ. A fault scenario is used to guide an instance of runtime fault injection.

Suppose we capture a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2$ and execution statuses T, T of both target fault functions in τ_1 and τ_2, then one generated fault scenario example is $\sigma \triangleq \langle\varepsilon, \phi\rangle$ where $\varepsilon \triangleq \tau_1 \rightarrow \tau_2$ and $\phi \triangleq T, F$.

*Definition 7 (**fault scenario database**):* A fault scenario database $\Sigma \triangleq \{\langle\sigma, \varsigma\rangle \mid \sigma$ is a fault scenario, ς is the fault simulation result of $\sigma\}$ is a set which saves all unique fault scenarios and their runtime execution results.

We have defined three different kinds of test results: *pass, fail and null.* Before σ is applied, ς is *null.* When the driver handles the fault scenario correctly, ς is *pass.* If the system or the driver crashes during the fault simulation, ς is *fail* and the corresponding crash report is saved for developers to conduct further analysis.

3.2 Challenges

The high-level workflow of our approach is illustrated in Figure 6. ADFI first runs a test suite on a device driver under an empty scenario *Fault0* to capture the runtime trace where *Fault0* includes an empty configuration which does not introduce any runtime fault, and fault scenarios are generated based on the captured trace. Then given one fault scenario *FaultX*, ADFI runs the test to see if *FaultX* triggers a crash. The process of applying one fault scenario is one instance of runtime fault injection. In one instance, ADFI hooks all target function calls. Each time a target function call is captured, ADFI decides to execute the corresponding

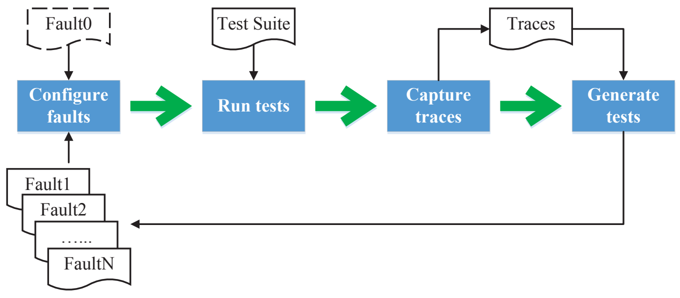

Figure 6: The high-level workflow

target function or inject a fault (return a false result) according to the fault scenario. Simultaneously, ADFI collects the trace executed during this run. Next, ADFI generates more fault scenarios based on the trace. The above process is repeated until all fault scenarios are applied.

The approach described above has two major challenges.

Fault scenario explosion: Generating all feasible fault scenarios does not scale if a large number of target functions exist in a driver. A naïve approach to generating fault scenarios is to explore all target function combinations along a driver runtime trace ε. If there are N target functions along ε, the number of generated fault scenarios can be $2^N - 1$. If we apply all these fault scenarios to driver robustness testing, it can take much time or even forever. Indeed as we tried this approach, it caused a fault scenario explosion after applying a few fault scenarios.

Handling concurrency and runtime uncertainty: ADFI repeatedly runs the same test suite and applies different fault scenarios to guide runtime fault injection. A fault scenario σ is a pair of a reference runtime trace ε and a fault configuration ϕ. To apply σ, ADFI captures a new runtime trace ε_{new} and run each target function $\varepsilon_{new}.\tau_i.\tilde{f}$ according to ϕ. Due to system concurrency and runtime uncertainty, ε and ε_{new} can be different which brings difficulty to find the right $\phi.\varphi_i$ to guide fault injection. This demands a systematic replay mechanism to guarantee that ε_{new} conforms to ε upon a given fault configuration ϕ.

3.3 Trace-based Iterative Strategy

In order to address the fault scenario explosion challenge, we have developed a bounded trace-based iterative generation strategy. For each fault scenario σ, ADFI runs the test suite on the driver and captures the runtime trace $\varepsilon \triangleq \tau_1 \to \tau_2 \to ... \to \tau_n$. In the following, we set n to 3 to illustrate our approach. Although we use a small number as the example, the idea can be applied to any large number. As shown in Figure 7(a), we capture a runtime trace which includes three stack traces and the corresponding execution statuses of target functions in three stack traces: (T, T, T).

By applying the naïve approach, we can generate seven $(2^3 - 1)$ fault scenarios. However, some generated fault scenarios are invalid fault scenarios which are not feasible at runtime. For example, if a generated fault configuration $\phi \triangleq (T, F)$ is applied, the actual trace is $\tau_1 \to \tau_2 \to \tau_4$ shown in Figure 7(c) which is different from $\tau_1 \to \tau_2 \to \tau_3$. In this case, (T, F, F) would be an invalid fault configuration for the trace $\tau_1 \to \tau_2 \to \tau_3$. In order to avoid generating invalid fault scenarios, our trace-based iterative generation strategy only generates one-step fault configurations (F), (T, F) and (T, T, F) in this iteration as shown in Figure 7(b).

Remark: Our approach does not miss any valid fault scenarios. If the driver works as shown in Figure 7(a), our trace-based iterative generation strategy first generates three fault scenarios. Then after the fault scenario including the configuration (F) is applied, the captured fault trace should be (F, T, T) and we can generate new fault configurations (F, F) and (F, T, F). After we apply all fault scenarios, we can cover all eight possibilities eventually.

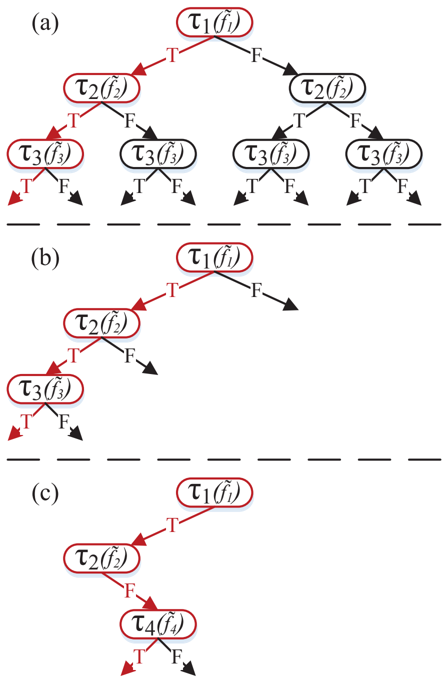

Figure 7: Trace-based iterative generation example

Moreover, our trace-based iterative generation strategy only generates new fault scenarios on a newly captured stack trace. Suppose we apply fault configuration (T, F) generated in Figure 7(b), we can capture the runtime trace $\tau_1 \to \tau_2 \to \tau_4$. As shown in Figure 7(c), we only generate one new fault configuration (T, F, F) from the captured target function execution trace (T, F, T). Here, we do not generate a fault configuration (T, T) because it has been covered. In this way, no duplicate fault scenarios (configurations) are generated.

Algorithm 1 ITERATIVE_GENERATION (ε, σ, Σ)

```
1: $i \leftarrow \varepsilon.size()$; $j \leftarrow \sigma.size()$; $\phi \leftarrow \emptyset$;
2: $\phi \leftarrow buildCompleteConfiguration(\sigma.\phi,\ i,\ j)$;
3: while $i > j$ do
4:     $\phi_{new} \leftarrow \phi_j, 0$; //Build a new configuration
5:     $\Sigma.insert(\langle\varepsilon_{j+1},\ \phi_{new}\rangle)$; //Save the fault scenario
6:     $j \leftarrow j + 1$;
7: end while
```

Algorithm 1 illustrates how to generate new fault scenarios using the trace-based iterative generation strategy. The algorithm takes a runtime trace ε, a reference fault scenario

σ and the fault scenario database Σ as inputs. If the length of the configuration is less than the length of ε, the algorithm first supplements the configuration by adding $(j-i)$ true decisions into ϕ to build a complete configuration (line 2). The algorithm goes through subsequences of the runtime trace ε between ε_j and ε_i. For each subsequence ε_i, the algorithm constructs a new fault decision ϕ_{new} by combining the subsequence ϕ_{i-1} of the previous fault decision ϕ and a false decision. A new fault scenario is created which includes ε_i and ϕ_{new} and saved into the database Σ. Suppose we apply a fault configuration $\phi \triangleq (T, F)$ and capture the corresponding runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$, the corresponding length i is 3 and j is 2. We first supplement the configuration as $\phi \triangleq (T, F, T)$, then we build a new configuration $\phi \triangleq (T, F, F)$.

3.4 Bounded Generation Strategy

We have applied the trace-based iterative generation strategy to device drivers and it can greatly reduce the number of generated tests. However, there are still a large number of fault scenarios generated. After analyzing the captured runtime trace, we found that there are two main reasons.

1)Duplicate stack traces. For some drivers, many duplicate stack traces exist in a runtime trace. There are mainly two reasons for duplicate stack traces. First, the same target function is repeatedly invoked within a loop. For example, a set of ring buffers is usually allocated using a loop when a network driver is initialized. Second, the same target function is invoked along a driver path and the driver path is frequently executed for processing special requests. For example, system resources are allocated and freed in the transmit function for a network driver and the transmit function is called many times during an instance of driver testing.

2)Fault scenario explosion. Although we have applied the trace-based iterative generation strategy to eliminate invalid fault scenarios, fault scenario explosion still exists. As shown in Figure 7(a), eight fault scenarios can be all valid for some drivers. If there are N target functions along a runtime trace, a subset of all N target functions (the number is M, $M < N$) can still bring a large amount of fault scenarios (the number can be $2^M - 1$) in the final result.

To solve these two problems, we have developed a bounded generation strategy to avoid injecting an exponential number of fault scenarios. ADFI supports two kinds of bounds: maximum number of injected faults on the same stack traces in a fault scenario (*MSF*) and maximum number of injected faults in a fault scenario (*MF*).

First we explain how *MSF* works. Suppose *MSF* is 1, we use an example to illustrate the idea. We captured a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ and the corresponding target function execution trace (F, T, T). Within ε, τ_1 and τ_3 are the same stack traces. If we generate fault scenarios following the trace-based iterative strategy, we should generate two fault configurations (F, F) and (F, T, F). The bounded generation strategy does not allow us to inject more than one fault on the same stack trace, which means we only generate one fault configuration (F, F). For another bound *MF*, the idea is straightforward. The number of injected faults in a fault scenario cannot exceed *MF*.

As shown in Algorithm 2, we have extended Algorithm 1 to support bounded generation. There are mainly three

Algorithm 2 BOUNDED_GENERATION (ε, σ, Σ, *bound*)

```
 1: $i \leftarrow \varepsilon.size();\ j \leftarrow \sigma.size();\ \phi \leftarrow \emptyset;\ T \leftarrow \emptyset$
 2: $\phi \leftarrow buildCompleteConfiguration(\sigma.\phi,\ i,\ j);$
 3: $T \leftarrow recordAllFaults(\sigma);$
 4: if $checkMFBound(T,\ MF)$ then
 5:    return
 6: end if
 7: while $i > j$ do
 8:    if $checkMSFBound(T,\ \varepsilon.\tau_{j+1},\ MSF)$ then
 9:       $\phi_{new} \leftarrow \phi_j, 0;$ //Build a new configuration
10:       $\Sigma.insert(\langle \varepsilon_{j+1},\ \phi_{new} \rangle);$ //Save the fault scenario
11:    end if
12:    $j \leftarrow j + 1;$
13: end while
```

differences. First, we go through the reference fault scenario σ to record all fault-related stack traces and the number of faults as a map T before generating tests. Second, before fault scenarios are generated, we check whether the number of faults in the reference fault scenario exceeds *MF*. If yes, we terminate test generation and return directly. Third, during the generation, we check whether the number of faults injected on the same stack traces exceeds *MSF*. If not, we generate the corresponding fault scenario. Otherwise, no fault scenario is generated.

Algorithm 3 RECORDALLFAULTS (σ)

```
 1: $\varepsilon \leftarrow \sigma.\varepsilon;\ \phi \leftarrow \sigma.\phi;\ T \leftarrow \emptyset;\ i \leftarrow \sigma.size();\ j \leftarrow 1;$
 2: while $i \geq j$ do
 3:    if $\phi.\varphi_j == F$ then
 4:       if $T.find(\varepsilon.\tau_j) == T.end()$ then
 5:          $T.insert(\varepsilon.\tau_j, 1);$
 6:       else
 7:          $T.find(\varepsilon.\tau_j) \leftarrow T.find(\varepsilon.\tau_j) + 1;$
 8:       end if
 9:    end if
10:    $j \leftarrow j + 1;$
11: end while
12: return $T;$
```

As shown in Algorithm 3, we process the fault scenario σ to record all fault-related stack traces. $T \triangleq \{\langle \tau, count \rangle \mid \tau$ is a stack trace, *count* is the number of faults injected on $\tau\}$ is a map. We process each boolean variable $\phi.\varphi_j$ in the fault configuration. Once $\phi.\varphi_j$ is false, we insert $\langle \varepsilon.\tau_j, 1 \rangle$ into T or increase the *count* by 1 if $\varepsilon.\tau_j$ exists in T.

4. PERMUTATION-BASED INJECTION STRATEGY

Even if we issue the same test suite to device drivers, two runtime traces ε_1 and ε_2 can be different due to driver concurrency, runtime uncertainty, such as timing issues, memory allocation status and network overload.

There are three kinds of possible differences between ε_1 and ε_2 triggered by the same test suite.

1)Different sequences of stack traces. Device drivers are system software which can handle more than one requests at the same time, which means concurrency widely exists in device drivers. Due to the concurrency, even if two captured runtime traces include the same stack traces, the sequence of stack traces can be different between ε_1 and ε_2.

2)Different length of runtime traces. Due to different system situations or environments, the number of the same stack trace τ can be different between ε_1 and ε_2. For example, if we send the same data over a network driver, there can be different number of calls to the transmit function of the driver. This difference brings different number of the same τ existing in ε_1 and ε_2.

3)Different number of unique stack traces. Due to different faults injected, stack traces captured can be different between ε_1 and ε_2. Since fault scenarios trigger different driver paths, ε_1 and ε_2 along different paths can include different stack traces.

Since a fault scenario σ is generated based on a runtime trace, there are the same differences between $\sigma.\varepsilon$ and the corresponding triggered runtime trace ε_{new}. This makes it difficult to guide runtime fault scenario injection.

We first illustrate how to resolve the first difference. A fault scenario σ includes a runtime trace $\varepsilon \triangleq \tau_1 \to \tau_2 \to ... \to \tau_n$ and a fault configuration $\phi \triangleq \varphi_1, \varphi_2, ..., \varphi_n$. To guide fault injection at runtime, it might trigger a new runtime trace $\varepsilon_{new} \triangleq \tau_{new_1} \to \tau_{new_2} \to ... \to \tau_{new_n}$. Here we assume that ε and ε_{new} have the same stack traces, later we will illustrate how to handle different stack traces. ε should be a permutation of ε_{new}, which means ε_{new} is constructed by all stack traces in ε with a different sequence. As an example, $\tau_1 \to \tau_2 \to \tau_4 \to \tau_3$ is a permutation of $\tau_1 \to \tau_2 \to \tau_3 \to \tau_4$. In the runtime fault injection, we detect such permutations automatically and guide the fault injection.

The second difference is caused by runtime uncertainty. Here we assume that ε and ε_{new} include the same set of unique stack traces and the lengths of ε and ε_{new} can be different, later we will discuss how to handle different set of unique stack traces. Based on the analysis of driver code and our observation, repeatedly injecting faults on the same stack traces caused by runtime uncertainty does not trigger new bugs. Therefore we just ignore such kinds of differences.

Algorithm 4 GET_FAULT_CONFIGURATION (τ, σ, *Flags*)

```
1: $i \leftarrow 0$; $n \leftarrow Flags.size()$;
2: $i \leftarrow findNextStackTrace(\tau, \sigma, i)$;
3: while $i \neq n$ do
4:     if $Flags[i] \neq true$ then
5:         $Flags[i] \leftarrow true$;
6:         return $\sigma.\phi.\varphi_i$;
7:     end if
8:     $i \leftarrow findNextStackTrace(\tau, \sigma, i)$;
9: end while
10: return $true$;
```

As shown in Algorithm 4, a permutation-based injection mechanism is developed to guide the fault configuration. The algorithm takes a stack trace τ, the fault scenario σ and a flag array *Flags* as inputs. The array *Flags* has the length of $\sigma.\varepsilon$ and each element is initialized as false at the beginning of an instance of fault injection. Each time a target function is invoked, we determine whether the function should be executed normally or return an error with the corresponding stack trace τ. We first find τ from the beginning of σ and return the index i. Then we check $Flags[i]$ to see whether the fault decision $\sigma.\phi.\varphi_i$ has been conducted or not. If it is not conducted, we return $\sigma.\phi.\varphi_i$. Otherwise, we continue to get the index of the next stack trace from the position i. If we can not get the index from a position, $findNextStackTrace$ function returns n which means all fault decisions for τ have been covered. Therefore we return true to let the target function execute normally.

The third difference is caused by different faults injected. A set of unique stack traces in ε and ε_{new} is represented as S_ε and $S_{\varepsilon_{new}}$. There can be three kinds of cases: $S_\varepsilon \subsetneq S_{\varepsilon_{new}}$, $S_{\varepsilon_{new}} \subsetneq S_\varepsilon$ and ($S_\varepsilon \nsubseteq S_{\varepsilon_{new}}$ *and* $S_{\varepsilon_{new}} \nsubseteq S_\varepsilon$). According to our experiments, only the first case $S_\varepsilon \subsetneq S_{\varepsilon_{new}}$ occurs. There are two reasons. First, the same test suite is used for different rounds of fault injections. Second, a fault injected can trigger some new stack traces. Currently we also detect two other kinds of cases in our tool. Once any case is found, a warning is given.

5. IMPLEMENTATION

5.1 Overview

As illustrated in Figure 8, our automatic fault injection framework includes three key components:

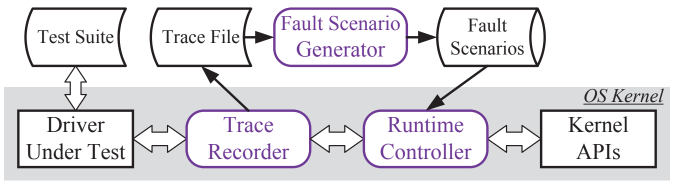

Figure 8: Runtime fault injection framework

1)Trace Recorder. The trace recorder captures runtime traces and kernel function return values while the driver is tested under a test suite. The trace recorder fully hooks the kernel API function calls so that all function calls and return values are intercepted and recorded in the trace files.

2)Fault Scenario Generator. The fault scenario generator takes a trace file as the input to generate fault scenarios. A trace-based iterative generation algorithm is implemented and employed by the generator to deliver high-quality fault scenarios. Generated fault scenarios are saved in the fault scenario database for guiding further fault injection.

3)Runtime Controller. The runtime controller applies a fault scenario in the driver testing process by emulating a fault return according to the fault configuration. The runtime controller is a kernel-level module working with the trace recorder together. It intercepts all target function calls invoked by device drivers. Once a kernel API function call is captured, it determines if a fault should be injected. If it is, the runtime controller returns a false result instead of invoking the real kernel API function.

5.2 Fault Injection on Kernel API Interface

In this paper, we mainly focus on the kernel API functions provided by the kernel since we want to test whether device drivers can survive under different system situations. Since operating systems provide lots of kernel API functions to support drivers, so far we have conducted our research on three main categories of kernel API functions:

1)Memory Allocation Functions. The Linux kernel offers a rich set of memory allocation primitives which can be used by drivers to allocate and optimize system memory resources. Different kinds of memory allocation functions can be used for allocating different kinds of memory. For example, the "*kmalloc*" function is used to grab small pieces of memory in kernel space and the "*get_free_page(s)*" function is used to allocate larger contiguous blocks of memory.

2)Memory Map and DMA Functions. A modern operating system is usually a virtual memory system, which means that the addresses seen by user programs do not directly correspond to the physical address used by the hardware devices. Memory map functions are needed for the conversion between virtual address and physical address. For example, the "*mmap*" function establishes a mapping between a process address space and a device. DMA is the hardware mechanism used for data transfer between device drivers and hardware devices without the need of involving the system processor. For example, the "*dma_set_mask*" function is used for checking if the mask is possible and updates the device parameters if it is.

3)PCI Interface Functions. The PCI bus is a widely-used system bus for attaching hardware devices. To support PCI device control and management, a set of functions are provided by the kernel and used by device drivers. For example, the "*pci_enable_device*" function is used for initializing device before it is used by a driver.

5.3 Filter Mechanism

When we first applied ADFI, we observed that the same crashes happened repeatedly. After analyzing these crashes, we found two key reasons.

1)Caused by a target function. If a fault is injected into a target function $\tilde{f}$, the corresponding error handling code for $\tilde{f}$ is tested. If the error handling mechanism is not correct, there is always a crash if a fault is injected on $\tilde{f}$ in a fault scenario.

2)Caused by a sequence of stack traces. Suppose a fault scenario σ_1 includes a runtime trace $\varepsilon \triangleq \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$ and a fault configuration $\phi \triangleq T, F, T, F$, it triggers a crash. If another fault scenario σ_2 includes the same runtime trace and a different fault configuration $\phi \triangleq F, F, T, F$, σ_2 possibly causes the same crash. In σ_1, two faults are injected in τ_2 and τ_4 which cause a crash. Since the same two faults are injected in τ_2 and τ_4 within σ_2, the same crash usually happens according to our experiments.

The target function $\tilde{f}$ is included in different stack traces. The stack trace τ is included in different fault scenarios. If we detect a bug triggered by a specific target function or a stack trace or a sequence of stack traces, we do not want to trigger the same crash repeatedly by other fault scenarios. Currently we provide two kinds of filter mechanisms to avoid such kinds of repeated crashes.

1)Function-Call-based Filter. A function call can be labeled as a filter pattern. As long as a fault needs to be injected into this function call according to the fault configuration, the fault scenario is ignored and not applied.

2)Stack-Trace-based Filter. A stack trace (or a sequence of stack traces) can be defined as a filter pattern. As long as a fault (or a sequence of faults) needs to be injected into a stack trace (or a sequence of stack traces, respectively) according to the fault configuration, the fault scenario is ignored and not applied.

The filter mechanism provides flexibility for driver developers to define filters to avoid repeated crashes. It has been applied to both fault scenario generation and injection.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

As shown in Table 1, we applied ADFI to 12 drivers in 3 categories: Wireless, USB controller and Ethernet. These three categories represent the most important three types of PCI devices.

Table 1: Summary of target drivers

Category	*Driver*	*Size*	*Description*
Wireless	ath9k	4.3M	Qualcomm AR9485 Wireless Driver
	iwlwifi	12M	Intel Wireless AGN Driver
USB	ehci_hcd	10M	USB 2.0 Host Controller Driver
	xhci_hcd	13M	USB 3.0 Host Controller Driver
Ethernet	e100	655K	Intel(R) PRO/100 Network Driver
	e1000	2.3M	Intel(R) PRO/1000 Network Driver
	ixgbe	5.9M	Intel(R) 10 Gigabit Network Driver
	i40e	8M	Intel(R) 40 Gigabit Network Driver
	tg3	2.1M	Broadcom Tigon3 Ethernet Driver
	bnx2	1.3M	Broadcom NetXtreme II Driver
	8139cp	537K	RealTek Fast Ethernet driver
	r8169	1.1M	RealTek Gigabit Ethernet Driver

As the workloads of the experiments, we created different test suites for different categories. There is one requirement that each test suite must start with a "load driver" command and end with a "remove driver" command. Between them, any test cases are allowed. A partial list of test cases for each category is shown in Table 2. Of these drivers, Intel ethernet network drivers are downloaded[1]. The other drivers are from Linux kernel source code.

Table 2: Summary of workload

Category	*Test Applications*
Wireless	Basic network commands (e.g. *ifup*, *ifconfig*, *ifdown*)
	Data transfer commands (e.g. *scp*, *ping*)
	Wireless config tools (e.g. *iw*, *iwconfig*)
USB	Basic USB control commands (e.g. *lsusb*)
	Enable/disable a USB device on the USB hub
	Transfer data to a USB disk
Ethernet	Basic network commands (e.g. *ifup*, *ifconfig*, *ifdown*)
	Data transfer commands (e.g. *scp*, *ping*)
	Ethernet config tools (e.g. *ethtool*, *scapy*)

6.2 Bug Findings

After testing all 12 drivers, we found the 28 distinct bugs described in Table 3. Of these bugs, 8 bugs are triggered by

[1]The latest version of Intel ethernet network drivers can be download in the following link: http://sourceforge.net/projects/e1000/files/

Table 3: Bug results

Category	Wireless Driver		USB Driver		Ethernet Driver								Total
	ath9k	iwlwifi	ehci_hcd	xhci_hcd	e100	e1000	ixgbe	i40e	tg3	bnx2	8139cp	r8169	
PCI	0	0	0	0	0	0	2	2	0	2	2	0	8
Memory	0	1	0	0	1	4	0	2	0	0	0	0	8
DMA	1	0	0	4	1	0	0	0	0	1	1	0	8
Mixed	0	0	0	0	0	4	0	0	0	0	0	0	4
Total	1	1	0	4	2	8	2	4	0	3	3	0	*28*

PCI interface faults, 8 bugs are triggered by memory allocation faults, 8 bugs are triggered by DMA function faults, and the other 4 bugs are triggered by mixed PCI/Memory/DMA faults. All these bugs can result in serious driver/system issues which include driver freeze, driver crash, system freeze and system crash. Moreover, all these bugs are difficult to find under normal situations.

These results show the effectiveness of our fault injection approach. We summarize the failure outcomes as follows:

*1)**System crash.*** The fault results in a kernel panic or fatal system error which crashes the entire system.

*2)**System hang.*** The fault results in a kernel freeze where the whole system ceases to respond to inputs.

*3)**Driver crash.*** The fault only results in a driver crash while the system can still work correctly.

*4)**Driver freeze.*** The fault only results in a driver freeze where the driver can not be loaded/removed.

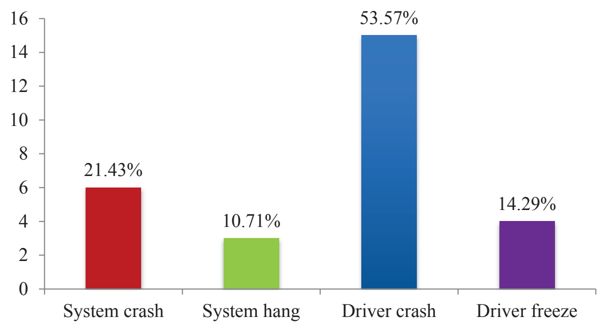

Figure 9: Outcomes of experiments

Figure 9 provides the distributions of failure types. Of the 28 bugs, 9 bugs result in system failures including 6 crashes and 3 hangs. The other 19 bugs result in driver failures including 15 crashes and 4 freezes.

Bug Validation. To verify if all these bugs are valid, we manually injected bug-correlated faults into device drivers. For example, if there is a "*kmalloc*" fault, we manually injected the fault. We modified the original statement

$$\text{“}void\ *\ p = kmalloc(size,\ GFP\_KERNEL);\text{”}$$

to

$$\text{“}void\ *\ p = NULL;\text{”}$$

Then we recompiled the driver and ran the driver under the test suite. The above example is just a simple fault scenario. Some fault scenarios are quite involved and require more modifications to the driver code to reproduce. All 28 bugs can be triggered the same way as they are triggered by ADFI. By this manual validation, we are better assured that all 28 bugs are valid and they can happen in a real system environment.

6.3 Human Efforts

One goal of ADFI is to minimize the human effort in testing the robustness of a driver. The necessary effort of our approach comes from three sources: (1) a configuration file to prepare ADFI for testing a driver; (2) crash analysis; (3) compilation flag modification to support coverage. The first two efforts are required while the third one is optional.

Configuration file. Only a few parameters need to be defined in a configuration file. They include driver name, runtime data folder path, test suite path and several runtime parameters. One example is shown in Figure 10. Such configuration is easy to create. In our experiments, only a few minutes are needed to set up one configuration file.

```
[target]                ; Target Module Info
name = ath9k

[ADFI]
runtime_folder = /home/kai/ADFI/ath9k/data/
test_suite_cmd = /home/kai/ADFI/ath9k/testsuite.sh
max_same_faults = 1
max_faults = 3
max_retry = 3
```

Figure 10: A sample configuration

Crash analysis. Once a crash happens, the developer needs to figure out the cause of the crash. Our approach can inject the same fault and trigger the same behavior repeatedly. When there is a crash, our approach can tell what faults have been injected into the driver. Furthermore, the whole driver stack is provided by ADFI to support crash analysis. This information can help driver developers understand and figure out the root cause of the crash. In our experiments, the average time for understanding each of the 28 bugs is less than 10 minutes using the ADFI debug facilities.

Compilation flag. To evaluate the driver code coverage, we need to compile the driver with additional compilation flags. We can achieve this in two ways. First, we can add the flags into the Linux kernel compilation process. Second, we can add the flags into the driver compilation Makefile. Both ways are easy to implement. In our experiments, we manually added the flags into each driver Makefile.

6.4 Evaluation of Fault Generation and Injection Strategy

ADFI allows two kinds of bounds, the maximum faults (MF) and the maximum same faults (MSF) in a test case.

Table 4: Results under different MF (MSF = 1)

Category	MF	Wireless Driver		USB Driver		Ethernet Driver							
		ath9k	iwlwifi	ehci_hcd	xhci_hcd	e100	e1000	ixgbe	i40e	tg3	bnx2	8139cp	r8169
PCI	1	1	3	0	0	2	5	5	5	7	3	2	2
	2	1	3	0	0	2	9	8	8	10	3	2	3
	3	1	3	0	0	2	9	9	8	10	3	2	3
Memory	1	5	24	4	1	3	13	11	32	11	9	1	3
	2	5	164	10	1	3	49	53	136	25	34	1	3
	3	5	840	12	1	3	117	156	414	29	51	1	3
DMA	1	3	4	1	6	5	11	9	17	4	13	3	8
	2	3	9	1	6	6	40	51	69	6	77	7	24
	3	3	10	1	6	6	95	171	177	6	221	8	37
ALL	1	9	31	5	7	10	28	25	54	22	25	6	13
	2	9	235	15	7	12	180	209	268	84	234	10	56
	3	9	1375	18	7	12	858	924	1365	175	980	11	130

We first set MSF as 1 and then generated faults under different MFs. Table 4 shows the number of generated fault scenarios where MF is 1, 2 and 3.

We have generated fault scenarios on all functions in the three categories (c.f. Section 5.2). As shown in Table 4, different number of fault scenarios were generated for different device drivers. For drivers such as *ath9k* and *8139cp*, only about 10 fault scenarios were generated. For drivers such as *iwlwifi* and *i40e*, more than 1000 fault scenarios were generated. The number of generated faults depends on how many target functions are used in a device driver.

Another observation from the results is that there are no generated fault scenarios for *ehci_hcd* and *xhci_hcd* under PCI category. After analyzing the source code of *ehci_hcd* and *xhci_hcd* code, we did not find PCI-related functions invoked by these drivers directly. The fact is that both these drivers only invoke some PCI wrapper functions directly and these PCI wrapper functions are defined in the kernel.

We further tried to generate fault scenarios while setting MSF as 2 on *e1000* and *iwlwifi* drivers. We generated more test cases on both drivers, however no new bugs were detected and almost no coverage improvement was achieved.

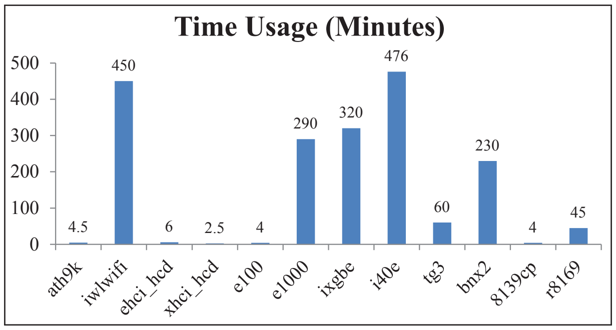

Figure 11: Time usage

In order to evaluate the efficiency of ADFI, we summarized total time usage for fault generation and injection in Figure 11. All these time usages were summarized while generating fault scenarios on all functions in three categories. ADFI can deliver high quality fault scenarios and find bugs effectively with a modest amount of time.

6.5 Coverage Improvement

As shown in Table 5, the generated fault scenarios led to decent test coverage improvement. Our approach focuses on the error handling mechanism and capability of device drivers. The error handling code only takes up a small portion of driver code. Even if we can trigger all error handling mechanisms in a driver, it does not mean that the improved coverage is very high.

As shown in Table 5, the improved coverage is from 0.1% to 6.5%. However, our approach can cover many error handling branches. Particularly, for *iwlwifi* and *i40e*, the statement coverage can be improved by more than 200 new statements and the branch coverage can be improved by more than 150 new branches. After going through all 150 new branches, we found that most of them are error handling branches.

6.6 Further Potentials

Although our approach is only evaluated on Linux drivers in our experiments, the idea can be applied in other domains. We list three potential applications in the following:

1)Linux kernel module testing. While ADFI mainly focuses on device drivers, the principles can easily apply to other kernel modules. The only effort is to identify necessary categories of target functions for different kernel modules.

2)Windows driver testing. The Windows drivers have similar structures to Linux drivers. Once we can figure out how to migrate ADFI into the Windows environment, it can be used for Windows driver robustness testing.

3)User-level program/library testing. The user-level program/library needs to invoke certain functions which can fail at runtime, for example "*malloc*" function. Our idea can be further applied to test the robustness of user-level program/library to improve reliability.

7. RELATED WORK

There has been much research on device driver testing since drivers account for a major portion of operating systems and are a major cause of operating system crashes [12]. Our work is related to past work in several areas, including static analysis, reliability testing and fault injection.

7.1 Static Analysis

Model checking, theorem proving, and program analysis have been used to analyze device drivers to find thousands of bugs [4, 11, 17, 25]. Nevertheless, these tools take time

Table 5: **Summary of coverage improvement**

Driver	Statement					Branch				
	#	Test Suite		Generated Tests		#	Test Suite		Generated Tests	
		#	%	#	%		#	%	#	%
ath9k	6146	3147	51.20%	3208	52.20%	3171	1059	33.40%	1268	39.99%
iwlwifi	11966	6761	56.50%	7000	58.50%	6458	2454	38.00%	2648	41.00%
ehci_hcd	2763	1307	47.30%	1323	47.88%	1586	568	35.81%	588	37.07%
xhci_hcd	4772	2114	44.30%	2119	44.40%	2485	721	29.01%	723	29.09%
e100	1258	721	57.31%	743	59.06%	617	206	33.39%	231	37.44%
e1000	5496	2215	40.30%	2259	41.10%	3530	787	22.29%	833	23.60%
ixgbe	13234	4222	31.90%	4301	32.50%	7288	1414	19.40%	1479	20.29%
i40e	9666	3557	36.80%	3886	40.20%	4882	1089	22.31%	1255	25.71%
tg3	7865	2580	32.80%	2658	33.80%	4990	983	19.70%	1043	20.90%
bnx2	3856	1828	47.41%	1859	48.21%	2217	643	29.00%	687	30.99%
8139cp	856	498	58.18%	506	59.11%	314	117	37.26%	126	40.13%
r8169	2596	1241	47.80%	1264	48.69%	848	294	34.67%	319	37.62%

to run and the results require time and expertise to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development. Numerous approaches have proposed to statically infer so-called protocols, describing expected sequences of function calls [11, 17, 26]. These approaches have focused on sequences of function calls that are expected to appear within a single function, rather than the specific interaction between a driver and the rest of the kernel.

Some safety holes in drivers can be eliminated by the use of advanced type systems. For example, Bugrara and Aiken propose an analysis to differentiate between safe and unsafe userspace pointers in kernel code [6]. They focus, however, on the entire kernel, and thus may report to the driver developer about faults in code other than his own.

7.2 Reliability Testing

There has been much research for operating systems reliability testing [2, 5, 8, 10, 16, 28, 29, 31]. Reliability testing of operating systems has been focused on device drivers since drivers are usually developed by a third party. Previous research on device driver reliability has mainly targeted detecting, isolating, and avoiding generic programming errors and errors in the interface between the driver and the OS.

7.3 Fault Injection Techniques

In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed. Fault injection techniques are widely used for software and system testing [13, 20, 21, 22, 24], ranging from testing the reliability of device drivers to testing operating systems, embedded systems and real-time systems [3, 7, 14, 15, 18, 23, 27].

There are several fault injection frameworks provided on both Windows and Linux platforms.

Windows Driver Verifier: Driver Verifier provides options to fail instances of the driver's memory allocations, as might occur if the driver was running on a computer with insufficient memory. This tests the driver's ability to respond properly to low memory and other low-resource conditions.

Linux Fault Injection Framework: This framework [9] can cause memory allocation failures at two levels: in the slab allocator (where it affects *kmalloc* and most other small-object allocations) and at the page allocator level (where it affects everything, eventually). There are also hooks to cause occasional disk I/O operations to fail, which should be useful for filesystem developers. In both cases, there is a flexible runtime configuration infrastructure, based on debugfs, which will let developers focus fault injections into a specific part of the kernel.

KEDR Framework: KEDR [27] is a framework for dynamic (runtime and post mortem) analysis of Linux kernel modules, including device drivers, file system modules, etc. The components of KEDR operate on a kernel module chosen by the user. They can intercept the function calls made by the module and, based on that, detect memory leaks, simulate resource shortage in the system as well as other uncommon situations, save the information about the function calls to a kind of "trace" for future analysis by the user-space tools.

There are three major limitations in the frameworks above. First, these frameworks mainly support memory-related fault injection to simulate low resource situations. Second, these frameworks mainly provide random fault simulation. Third, these frameworks require high manual efforts. Our approach extends the above framework to support more fault situations, such as DMA-related operations and PCI-related operations. Our approach provides an easy-to-use approach with little human effort which can systematically enumerate different kinds of fault scenarios to guide fault simulation.

8. CONCLUSIONS

We have presented an approach to runtime fault injection for driver robustness testing. We have evaluated our approach on 12 widely-used device drivers. Our approach was able to generate and inject effective fault scenarios in a modest amount of time using the trace-based iterative fault generation strategy. We have detected 28 bugs which have been further validated by manually injecting these bugs into device drivers. We have also measured test coverage and found that ADFI led to decent improvement in statement and branch coverage in drivers.

9. ACKNOWLEDGMENT

This research received financial support from National Science Foundation (Grant #: 1422067). The authors thank the anonymous reviewers for their helpful comments.

References

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[2] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *International Conference on Dependable Systems and Networks*, 2004.

[3] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *International Conference on Dependable Systems and Networks*, 2004.

[4] 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 *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[5] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Conference on USENIX Annual Technical Conference*, 2010.

[6] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, 2008.

[7] G. Cabodi, M. Murciano, and M. Violante. Boosting software fault injection for dependability analysis of real-time embedded applications. *ACM Trans. Embed. Comput. Syst.*, 2011.

[8] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella. Sabrine: State-based robustness testing of operating systems. In *International Conference on Automated Software Engineering*, 2013.

[9] R. J. Drebes and T. Nanya. Limitations of the Linux fault injection framework to test direct memory access address errors. In *Pacific Rim International Symposium on Dependable Computing*, 2008.

[10] J. Duraes and H. Madeira. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *Transactions of IEICE*, 2003.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles*, 2001.

[12] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Conference on Large Installation System Administration*, 2006.

[13] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Pacific Rim Int'l Symposium on Dependable Computing*, 2013.

[14] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into Linux by three fault injection techniques. In *International Conference on Dependable Systems and Networks*, 2002.

[15] A. Johansson and N. Suri. On the impact of injection triggers for os robustness evaluation. In *International Symposium on Software Reliability Engineering*, 2007.

[16] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[17] J. Lawall, J. Brunel, N. Palix, R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in Linux code. In *International Conference on Dependable Systems Networks*, 2009.

[18] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Conference on Design, Automation and Test in Europe*, 2009.

[19] Linux. Fault Injection Capabilities Infrastructure. http://lxr.linux.no/linux+v3.14/Documentation/fault-injection/.

[20] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Conference on USENIX Annual Technical Conference*, 2010.

[21] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *International Conference on Dependable Systems and Networks*, 2009.

[22] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 2011.

[23] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi. Scfit: A FPGA-based fault injection technique for SEU fault model. In *Conference on Design, Automation and Test in Europe*, 2012.

[24] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott. Fault injection framework for system resilience evaluation: Fake faults for finding future failures. In *Workshop on Resiliency in High Performance*, 2009.

[25] H. Post and W. Küchlin. Integrated static analysis for Linux device driver verification. In *International Conference on Integrated Formal Methods*, 2007.

[26] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *International Conference on Software Engineering*, 2007.

[27] V. V. Rubanov and E. A. Shatokhin. Runtime verification of Linux kernel modules based on call interception. In *International Conference on Software Testing, Verification, and Validation*, 2011.

[28] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *ACM European Conference on Computer Systems*, 2009.

[29] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser. Improved device driver reliability through hardware verification reuse. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[30] C. Sârbu, A. Johansson, F. Fraikin, and N. Suri. Improving robustness testing of cots os extensions. In *International Conference on Service Availability*, 2006.

[31] C. Sarbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of OS device Drivers. In *International Symposium on Software Reliability Engineering*, 2008.

[32] V. Shakti D Shekar, B B Meshram. Device driver fault simulation using kedr. *International Journal of Advanced Research in Computer Engineering and Technology*, 2012.

[33] Wikipedia. Stack trace. http://en.wikipedia.org/wiki/Stack_trace.

[34] Windows. Low Resources Simulation. http://msdn.microsoft.com/en-us/library/windows/hardware/ff548288(v=vs.85).aspx.