Testing Error Handling Code with Software Fault Injection and Error-Coverage-Guided Fuzzing

Jia-Ju Bai, Zi-Xuan Fu, Kai-Tao Xie, Zu-Ming Jiang

Abstract—Real-world programs require error handling code to handle various kinds of possible errors. However, these errors just infrequently occur due to special conditions, so error handling code is difficult to test. Coverage-guided fuzzing and software fault injection (SFI) are two common techniques that can test error handling code, but they still have major limitations. Specifically, existing fuzzing approaches generate program inputs guided by code coverage, but many occasional errors (such as insufficient memory) are unrelated to inputs, and code coverage cannot effectively reflect the execution contexts of these errors; existing SFI approaches often inject single or random faults, without exploring fault space or using program feedback. In this paper, we propose a new fuzzing framework named EH-Fuzz, to effectively test error handling code. EH-Fuzz uses a context-sensitive SFI-based fuzzing approach to explore fault space and perform fault injection, guided by a new metric named *error coverage*. We evaluate EH-Fuzz on 9 user-level programs and 6 kernel-level modules, and find 45 new real bugs, 31 of which have been confirmed and fixed. We compare EH-Fuzz to existing fuzzing approaches (including AFL, AFL++, Syzkaller, FIZZER and FIFUZZ), and EH-Fuzz finds many real bugs missed by these approaches with higher testing coverage.

Index Terms—Error handling, coverage-guided fuzzing, software fault injection, bug detection, error coverage.

1 Introduction

A program may encounter various kinds of possible errors, and thus it requires error handling code to handle these errors during execution. While error handing code is necessary, it is error-prone in reality. First, error handling code is difficult to correctly implement [1]–[4], because it involves special and complicated semantics. Second, error handling code is difficult to test [5]–[7], because it is infrequently executed and receives insufficient attention. For these reasons, many bugs may exist in error handling code, and they are hard-to-find in runtime testing. Some recent works [8]–[11] have shown that many error handling bugs can cause serious security problems, such as denial of service (DoS) and information disclosure.

Coverage-guided fuzzing is a promising bug-detection technique, which uses program feedback to guide test-case generation in runtime testing. Existing fuzzing approaches [12]–[29] focus on generating program inputs as test cases to cover infrequently-executed code, including error handling code. However, a large part of error handling code is caused by occasional errors (such as insufficient memory and network-connection failures), which are unrelated to program inputs, so existing fuzzing approaches cannot cover such error handling code. Software fault injection (SFI) is a common technique of testing error handling code. It intentionally injects faults or errors into the code of the tested program, and then executes the program to test whether it can correctly handle the injected faults or errors at runtime. However, existing SFI-based approaches [30]–[38] often inject single or random faults, without exploring fault space using program feedback, so they are limited in testing error handling code executed in complicated contexts.

To improve testing of error handling code, several recent approaches [39]–[42] introduce SFI in coverage-guided fuzzing to explore fault space. These approaches perform error mutation to generate injected faults, and use code coverage to guide both SFI-based fuzzing and input-driven fuzzing together. However, these approaches still have two main limitations in practice. On the one hand, an error site can be executed in different execution contexts, and code coverage cannot reflect such context information. For example, if two test cases trigger the same error sites in different execution contexts, these approaches consider the two test cases to be identical for testing error handling code; but many error handling bugs only occur in specific execution contexts, so the two test cases are actually different for error mutation. On the other hand, these approaches fail to identify whether the new code is covered exactly due to error mutation or input mutation, and thus they have to randomly select error mutation or input mutation in test-case generation. For example, some new code branches are covered due to program inputs, but these approaches may mistakenly consider that fault injection increases code coverage and thus identify the current injected faults are interesting for error mutation.

In this paper, to effectively detect bugs in error handling code, we propose a novel *error-coverage-guided SFI-based fuzzing approach*. This approach uses a new metric named *error coverage* to guide fault injection and fault-space exploration, instead of using code coverage. Error coverage is specifically described by *error sequence*, containing the covered error points and their execution states (failure or success), and each error point includes the location and

- *Jia-Ju Bai is with the School of Cyber Science and Technology in Beihang University, Beijing, China. E-mail: baijiaju@buaa.edu.cn.*
- *Zi-Xuan Fu and Kai-Tao Xie are with Tsinghua University, Beijing, China. E-mail: {fuzx20, xkt19}@mails.tsinghua.edu.cn.*
- *Zu-Ming Jiang is with ETH Zurich, Zurich, Switzerland. E-mail: zuming.jiang@inf.ethz.ch.*

calling context of a covered error site. With error coverage, this approach can also smoothly separate SFI-based fuzzing and input-driven fuzzing, to optimize test-case generation of injected faults and program inputs. Specifically, SFI-based fuzzing mutates and generates possible error sequences, each of which contains multiple error points for fault injection, if new error sequences are actually covered; input-driven fuzzing mutates and generates program inputs, if new code branches containing no error site are covered.

Based on our fuzzing approach, we design a new fuzzing framework named EH-Fuzz. At compile time, to reduce manual work of identifying error sites, EH-Fuzz performs a static analysis of the source code of tested programs, to identify possible error sites. The user can select realistic error sites that can actually fail and trigger error handling code. Then, EH-Fuzz uses our error-coverage-guided SFI-based fuzzing approach with input-driven fuzzing, to test both error handling code and normal-execution code.

We have implemented EH-Fuzz based on our previous SFI-based fuzzing tool FIFUZZ [42]. Overall, EH-Fuzz makes two new methodology improvements over FIFUZZ. First, FIFUZZ uses code coverage to guide both SFI-based fuzzing and input-driven fuzzing; while EH-Fuzz uses error coverage and error-unrelated code coverage to guide SFI-based fuzzing and input-driven fuzzing, respectively. Second, FIFUZZ performs SFI-based fuzzing and input-driven fuzzing together, by randomly using error mutation or input mutation; while EH-Fuzz performs SFI-based fuzzing and input-driven fuzzing separately, by smartly scheduling error mutation and input mutation. Due to the two new improvements, compared to FIFUZZ, EH-Fuzz is more effective in both SFI-based fuzzing and input-driven fuzzing to find more deep bugs. Besides, EH-Fuzz has fewer false positives in error-site extraction, and it can also test kernel-level modules that are not supported by FIFUZZ.

Overall, we make the following technical contributions:

- We first reveal the limitations of existing SFI-based fuzzing approaches in testing error handling code. Then, to solve these limitations, we propose a novel error-coverage-guided SFI-based fuzzing approach, with two benefits: (1) it collects the covered error points and their execution states as error coverage, to effectively guide fault injection and fault-space exploration; (2) it smoothly separates SFI-based fuzzing and input-driven fuzzing, to optimize test-case generation of injected faults and program inputs.
- Based on our fuzzing approach, we design a new fuzzing framework named EH-Fuzz, to effectively test error handling code. EH-Fuzz can test both user-level applications and kernel-level modules.
- We evaluate EH-Fuzz on 9 user-level applications of the latest versions and 6 kernel-level modules (including 4 filesystems and 2 device drivers) in Linux 5.16.16. EH-Fuzz finds 45 new real bugs, and 31 of them have been confirmed and fixed by the related developers. Moreover, we compare EH-Fuzz to five existing fuzzing approaches (including AFL, AFL++, Syzkaller, FIZZER and FIFUZZ), and EH-Fuzz finds many bugs missed by these approaches with higher testing coverage.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces our error-coverage-guided SFI-based fuzzing approach. Section 4 introduces EH-Fuzz and its implementation. Section 5 shows our evaluation. Section 6 makes a discussion about EH-Fuzz. Section 7 presents related work, and Section 8 concludes this paper.

2 Background and Motivation

In this section, we first introduce error handling code with a real bug example, and then analyze existing SFI-based fuzzing approaches to reveal their main limitations in testing error handling code.

2.1 Error Handling Code

A program may encounter exceptional situations at runtime, due to special execution conditions like invalid program inputs, insufficient memory and network-connection failures. We refer to such exceptional situations as *errors*, and the code used to handle an error is called *error handling code*.

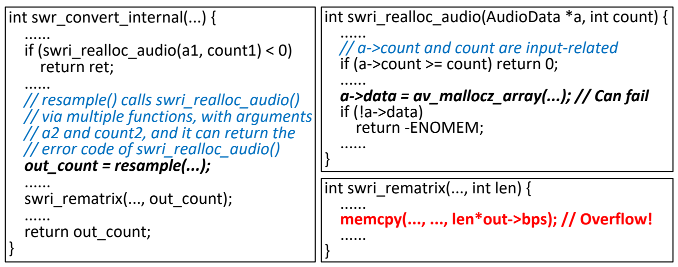
In fact, errors can be classified into two categories: *input-related errors* and *occasional errors*. An input-related error is caused by invalid inputs, such as abnormal commands and bad data. Such an error can be triggered by providing specific program inputs. An occasional error is caused by an exceptional event that occasionally occurs, such as insufficient memory or network-connection failure. Such an error is related to the state of execution environment and system resources (such as memory and network connection), but unrelated to inputs, so it typically cannot be triggered by existing fuzzing that focuses on inputs.

Several recent studies [40], [42] have shown that about 50% of the sites triggering error handling code are related to occasional errors, and existing input-driven fuzzing tools are ineffective in detecting bugs caused by occasional errors.

2.2 Motivating Example

Figure 1 presents a buffer-overflow bug in error handling code of *ffmpeg*. The function `swr_convert_internal` calls `resample`, which returns to the variable *out_count*. The function `resample` calls `swri_realloc_audio`, and it can return the error code of `swri_realloc_audio`, when `av_mallocz_array` in `swri_realloc_audio` fails and returns NULL. In this case, the variable *out_count* is negative and used in the function `swri_rematrix` as the length for memory copying, causing a buffer-overflow bug [43].

Indeed, this bug is hard to find in runtime testing for three main reasons. First, the function `av_mallocz_array` is used to allocate memory, and it can fail only when an occasional error of insufficient memory occurs. Second, even though one can make `av_mallocz_array` fail by fault injection or memory exhaustion, the bug is only triggered in a special execution context, namely this failure happens when `resample` calls `swri_realloc_audio`. We find that the function `swr_convert_internal` also directly calls `swri_realloc_audio` before `resample`, and once `av_mallocz_array` in `swri_realloc_audio` fails before `resample`, the function `swr_convert_internal`



```
int swr_convert_internal(...) {
    ......
    if (swri_realloc_audio(a1, count1) < 0)
        return ret;
    ......
    // resample() calls swri_realloc_audio()
    // via multiple functions, with arguments
    // a2 and count2, and it can return the
    // error code of swri_realloc_audio()
    out_count = resample(...);
    ......
    swri_rematrix(..., out_count);
    ......
    return out_count;
}
```

```
int swri_realloc_audio(AudioData *a, int count) {
    ......
    // a->count and count are input-related
    if (a->count >= count) return 0;
    ......
    a->data = av_mallocz_array(...); // Can fail
    if (!a->data)
        return -ENOMEM;
    ......
}
```

```
int swri_rematrix(..., int len) {
    ......
    memcpy(..., ..., len*out->bps); // Overflow!
    ......
}
```

Fig. 1. Buffer-overflow bug in error handling code of *ffmpeg*.

TABLE 1
Comparison of state-of-the-art SFI-based fuzzing approaches.

Approach	Tested programs	Contexts of inject faults	Program feedback	Compatible with input mutation
POTUS [39]	Kernel	Neglect	CodeCov	None
FIZZER [40]	Kernel	Neglect	CodeCov	None
iFIZZ [41]	FW App	Consider	CodeCov	None
FIFUZZ [42]	App	Consider	CodeCov	Weak
EH-Fuzz	Kernel+App	Consider	CodeCov+ErrCov	Good

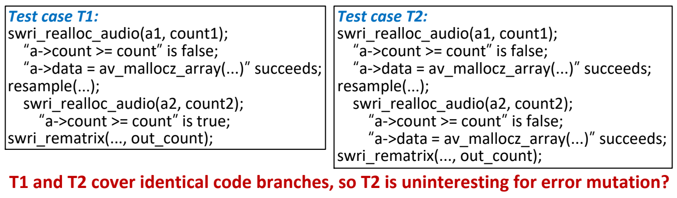
returns directly without calling `swri_rematrix` or triggering the bug. Third, the bug is also related to program inputs, as the variables `a->count` and `count` in `swri_realloc_audio` are input-related. Specifically, if `a->count` is not smaller than `count` due to program inputs, `swri_realloc_audio` directly returns without calling `av_mallocz_array`, and thus the bug cannot occur in this case. Due to the three reasons, the bug had existed for over 7 years before the developer fixed it in December 2021, according to a bug report of our framework EH-Fuzz.

2.3 State of the Art of SFI-Based Fuzzing

Several recent fuzzing approaches [39]–[42] introduce software fault injection (SFI) and "fuzz" injected faults to trigger infrequently-executed error handling code. They first identify error sites with static analysis and manual selection; and then they mutate and generate error-related test cases, which describe the injected faults into the identified error sites, according to code coverage. Table 1 shows the comparison between these approaches. POTUS and FIZZER are designed to test kernel-level drivers, but they neglect the execution contexts of injected faults and lack input mutation. iFIZZ is designed to test IoT firmware applications, and it considers the execution contexts of injected faults, but it has no input mutation. FIFUZZ is the most advanced SFI-based fuzzing approach, by considering the execution contexts of injected faults and supporting input mutation. It is designed to test user-level applications. To be compatible with input-driven fuzzing, FIFUZZ randomly selects error mutation or input mutation, when new code branches are covered.

Although existing SFI-based fuzzing approaches have shown good results in finding error handling bugs, they still have two main limitations in testing error handling code:

L1) Ineffective code-coverage feedback in SFI-based fuzzing. Code coverage is useful to reflecting whether error handling code is covered, but it cannot reflect the runtime contexts of the covered error handling code. Indeed, an error site can be executed in different runtime contexts, and thus bugs hidden in its error handling code can be triggered only in specific runtime contexts. FIFUZZ and iFIZZ consider the calling contexts of error sites in fault injection,



```
Test case T1:
swri_realloc_audio(a1, count1);
    "a->count >= count" is false;
    "a->data = av_mallocz_array(...)" succeeds;
resample(...);
    swri_realloc_audio(a2, count2);
        "a->count >= count" is true;
swri_rematrix(..., out_count);
```

```
Test case T2:
swri_realloc_audio(a1, count1);
    "a->count >= count" is false;
    "a->data = av_mallocz_array(...)" succeeds;
resample(...);
    swri_realloc_audio(a2, count2);
        "a->count >= count" is false;
        "a->data = av_mallocz_array(...)" succeeds;
swri_rematrix(..., out_count);
```

T1 and T2 cover identical code branches, so T2 is uninteresting for error mutation?

Fig. 2. Example test cases for the code in Figure 1.

but fail to consider such context information in program feedback for error mutation. For example in Figure 2, there are two test cases *T1* and *T2* shown in Figure 1, and *T1* covers all the code branches covered by *T2*. Suppose that *T1* is executed before *T2*, so existing SFI-based fuzzing approaches consider that *T2* is uninteresting for error mutation, due to the same code coverage. But in fact, *T2* is actually interesting for error mutation, because the second call to `av_mallocz_array` in *T2* is executed in a different call context (`resample`→`av_mallocz_array`) from *T1*. If error mutation is performed on *T2*, namely making `av_mallocz_array` in `swri_realloc_audio` called by `resample` fail, the buffer-overflow bug in Figure 1 will be found. However, existing SFI-based fuzzing approaches lack this error mutation and thus miss the bug, because they consider *T2* to be uninteresting.

L2) Weak compatibility with input-driven fuzzing. FIFUZZ is the sole existing SFI-based fuzzing approach that is compatible with input-driven fuzzing. It randomly selects error mutation or input mutation in test-case generation, when new code branches are covered. However, this compatibility is weak, because FIFUZZ fails to identify whether these new code branches are covered exactly due to fault injection or program inputs. For example, some new code branches are covered due to program inputs, but FIFUZZ may mistakenly consider that fault injection increases code coverage, and thus identifies the current injected faults to be interesting for error mutation. Even if FIFUZZ can make clear identification for code coverage, it is still limited in SFI-based fuzzing, as code coverage cannot reflect the runtime contexts of the covered error handling code.

Besides the above two methodology limitations, a practical limitation is that existing SFI-based approaches can only test user-level applications or kernel-level modules, without testing them both, which limits the generality.

3 Approach

In this section, we first introduce our basic idea of solving the limitations of existing SFI-based fuzzing, then describe our error model, and finally introduce our SFI-based fuzzing approach and its compatibility with input-driven fuzzing.

3.1 Basic Idea

To improve fuzzing in testing error handling code, we need to address the two main limitations of existing SFI-based fuzzing (described in Section 2.3) from two aspects:

For *L1*, we need to use a new coverage metric named *error coverage*, to replace code coverage in SFI-based fuzzing. This error coverage should reflect the runtime contexts and execution situations of error sites. In our previous approach FIFUZZ, we proposed to use *error points*, each of which includes the location and calling context of a covered error site, for context-sensitive fault injection. Thus, we can combine the covered error points and their execution states (failure or success) as an *error sequence*, to describe error coverage. Specifically, if a new error sequence is actually covered, namely new error points are covered or new execution situations of old error points are covered, we consider that error coverage is increased, and thus the current injected faults are interesting for subsequent error mutation.

For *L2*, we need to separate SFI-based fuzzing from input-driven fuzzing, to optimize test-case generation of injected faults and program inputs. Specifically, we can use error coverage for error mutation to generate new error-related test cases, while use error-unrelated code coverage for input mutation to generate new input-related test cases. This error-unrelated code coverage can be described as the covered code branches containing no error site. In this way, SFI-based fuzzing and input-driven fuzzing can be performed separately, without side effect on each other.

3.2 Error Model

In our basic idea, *error point* is vital for performing context-sensitive fault injection and describing error coverage. Thus, we first introduce error point and the related error model in SFI-based fuzzing before presenting our approach.

An error point represents an execution point where an error can occur and trigger error handling code. During fault injection, each error point can normally run (indicated as 0) or fail by an injected fault (indicated as 1). While the program is tested, multiple error points can be executed, forming an 0-1 *error sequence* that describes the failure situation of error points at runtime:

$$ErrSeq = [ErrPt_1, ErrPt_2, ..., ErrPt_x],\ ErrPt_i = \{0, 1\}$$

In most existing SFI-based approaches, error point is context-insensitive, as they only use the location of each error site in source code to describe an error point, namely $ErrPt =< ErrLoc >$, without considering the execution context of this error site. In this way, if a fault is injected into an error site, this error site will always fail when being executed at runtime. But an error site can be executed in different calling contexts, and some real bugs (such as the buffer-overflow bug shown in Figure 1) can be triggered only when this error site only fails in specific calling context and succeeds in other calling contexts. Thus, existing SFI-based approaches may miss these real bugs. To solve this problem, we make error point context-sensitive, by considering the location and calling context of each error site:

$$ErrPt =< ErrLoc, CallCtx >$$

To describe the calling context of an error site, we consider the runtime call stack when the error site is executed. This runtime call stack includes the information of each function call at the call stack (in order from caller to callee), including the locations of this function call and called function. In this way, a calling context is described as:

$$CallCtx = [CallInfo_1, CallInfo_2, ..., CallInfo_x]$$
$$CallInfo =< CallLoc, FuncLoc >$$

Note that the runtime call stack of an executed error site is related to program execution. Thus, error points cannot be statically determined, and they should be dynamically identified during program execution. Accordingly, when fault injection is performed with a possible error sequence, the faults should be injected into error points during program execution. Moreover, if an error site is executed in N different calling contexts, there will be N different error points, which are used for context-sensitive fault injection and error-coverage collection.

3.3 Error-Coverage-Guided SFI-based Fuzzing

Based on our basic idea and error model, we propose a novel error-coverage-guided SFI-based fuzzing approach, to effectively cover error handling code executed in different runtime contexts and perform fault-space exploration. As shown in Figure 3, our approach has six basic steps:

(S1) At compile time, it identifies possible error sites in the program code, and the user can select realistic ones;

(S2) It runs the tested program and collects runtime information about each executed error site;

(S3) It identifies the covered error points and their execution situations, to collect the covered error sequence as error coverage according to runtime information;

(S4) After program execution, if a new error sequence is actually covered within the time limit, it mutates and generates new possible error sequences containing the covered error points for fault injection.

(S5) It runs the tested program and injects faults into error sites in specific calling contexts, according to the generated possible error sequences.

(S6) It collects runtime information, identifies the covered error sequence and generates new possible error sequences again, which constructs a fuzzing loop.

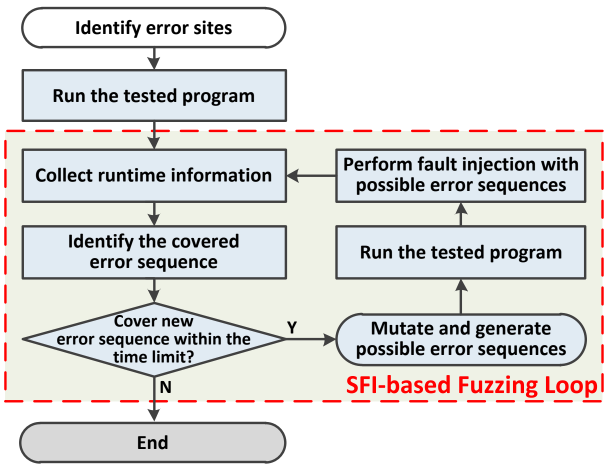

Identify error sites
Run the tested program
Collect runtime information
Perform fault injection with possible error sequences
Identify the covered error sequence
Run the tested program
Cover new error sequence within the time limit?
Y
Mutate and generate possible error sequences
N
SFI-based Fuzzing Loop
End

Fig. 3. Procedure of our SFI-based fuzzing approach.

Note that each possible error sequence is inferred from the last execution of the tested program to perform fault injection. Thus, after fault injection with this sequence, some error points in it may not be covered in real execution, and even some new error points can be also covered. In this case, the possible error sequence can lead to a different covered error sequence after fault injection. To handle this case, our approach maintains two sets of error sequences,

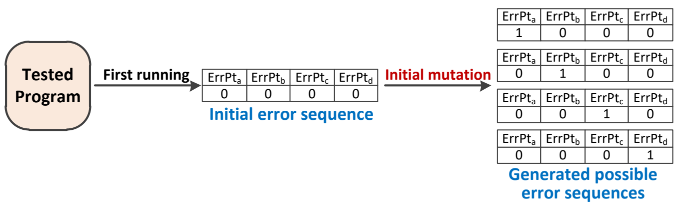


Fig. 4. Example of initial error mutation.

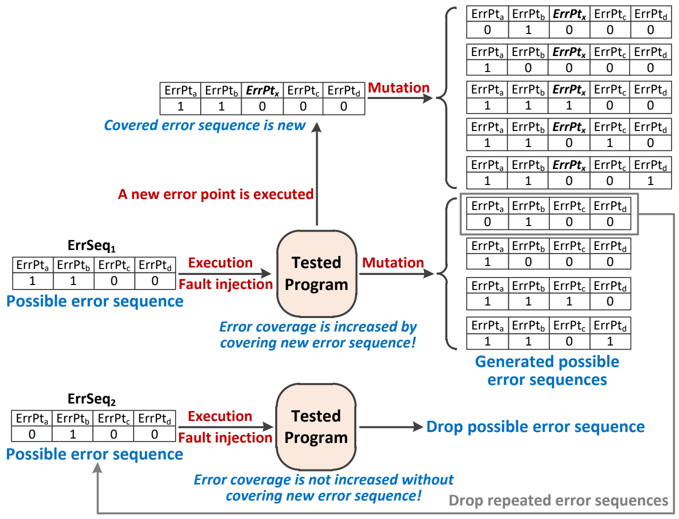

Mutation
Covered error sequence is new
A new error point is executed
ErrSeq1
Execution
Fault injection
Tested Program
Possible error sequence
Error coverage is increased by covering new error sequence!
Generated possible error sequences
ErrSeq2
Drop possible error sequence
Error coverage is not increased without covering new error sequence!
Drop repeated error sequences

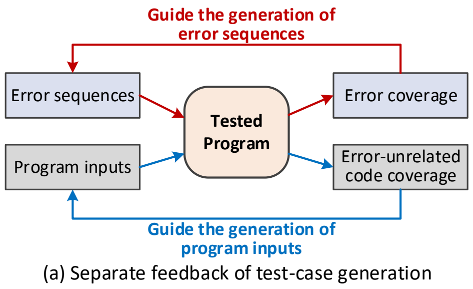
Fig. 5. Example of subsequent error mutation.

namely one set is for the generated possible ones to perform fault injection, and the other set is for the covered ones to describe error coverage.

In our approach, mutating and generating possible error sequences are important operations, which are performed according to the covered error sequences during program execution. Initially such information is unavailable, and thus our approach performs a special initial mutation for the first execution of the tested program. For subsequent executions, our approach performs the subsequent generation and mutation of error sequences. All the generated error sequences that cover the new error points are stored in a pool for error mutation.

Initial error mutation. Our approach first runs the tested program normally, and creates an initial error sequence according to runtime information. This error sequence contains the executed error points, and it is all-zero and used for the initial mutation. The mutation generates each new error sequence by making just one executed error point fail (0→1), as each error point may trigger uncovered error handling code in the related calling context. Figure 4 shows an example of the initial mutation for an error sequence, which generates four new possible error sequences.

Subsequent error mutation. After our approach runs the tested program by injecting faults according to a possible error sequence, it checks whether the covered error sequence of this running is new among the already covered error sequences. If not, our approach drops this possible error sequence for error mutation, as error coverage is not improved; if so, our approach mutates both this possible error sequence and the covered error sequence to generate each new possible error sequence, by changing the value of just one error point (0→1 or 1→0). Then, our approach compares


Guide the generation of error sequences
Error sequences
Tested Program
Error coverage
Program inputs
Error-unrelated code coverage
Guide the generation of program inputs
(a) Separate feedback of test-case generation

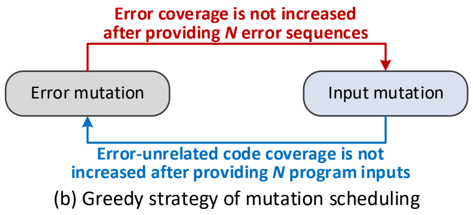

Error coverage is not increased after providing N error sequences
Error mutation
Input mutation
Error-unrelated code coverage is not increased after providing N program inputs
(b) Greedy strategy of mutation scheduling

Fig. 6. Cooperation of SFI-based fuzzing and input-driven fuzzing.

these generated error sequences with all existing possible and already covered error sequences, to drop repeated ones. Figure 5 shows an example of this procedure for two error sequences, For the first possible error sequence $ErrSeq_1$, a new error point $ErrPt_x$ is executed, and a covered error sequence containing this error point is collected. As this covered error sequence is new, which increases error coverage, our approach mutates the possible and covered error sequences to generate nine new possible error sequences. However, one of them is the same with an existing possible error sequence $ErrSeq_2$, so this new error sequence is dropped. For the second possible error sequence $ErrSeq_2$, error coverage is not increased without covering new error sequence, our approach drops it for error mutation.

Note that each error point in a possible error sequence is related to the runtime calling context, and thus when injecting faults into this error point during program execution, our approach needs to dynamically check whether the current runtime calling context and error sites match the target error point. If this error point is not executed during program execution, our approach will ignore this error point in error-coverage collection and error mutation.

3.4 Cooperation with Input-Driven Fuzzing

By using error coverage, our approach can clearly identify how fault injection or program inputs improve the testing coverage, especially for covering error handling code. In this way, our approach can smoothly separate SFI-based fuzzing and input-driven fuzzing, to optimize test-case generation of injected faults and program inputs.

Figure 6(a) shows the separate feedback of test-case generation with different coverage metrics. For a given program input, our approach identifies the covered code branches containing no error site, to describe error-unrelated code coverage. If this coverage is increased, the program input is considered to be interesting, and it is stored in a seed pool for input mutation. For a possible error sequence used in fault injection, our approach identifies the covered error sequence, to describe error coverage. If this coverage is increased, the possible and covered error sequences are both considered to be interesting, and they are stored in a seed pool (different from the input seed pool) for error mutation.

To smartly schedule input mutation and error mutation, our approach uses a greedy strategy shown in Figure 6(b). Our approach first performs error mutation to increase error coverage. If error coverage is not increased after providing N error sequences in fault injection, our approach performs input mutation to increase error-unrelated code coverage. If this code coverage is not increased after providing N program inputs, our approach performs error mutation to increase error coverage again. In this way, our approach can efficiently improve both error coverage and error-unrelated code coverage, to maximize the effectiveness of both SFI-based fuzzing and input-driven fuzzing. Note that we set N as 10% of the executed test cases in the evaluation, according to our experience. We will experimentally study the effect of this value in Section 5.6.

In fact, input mutation and error mutation can also benefit each other. On the one hand, input mutation can help cover more input-related code, which can have new error sites. Thus, these new error sites can be used by error mutation to achieve higher error coverage. On the other hand, error handling code covered by error mutation may have input-related code, which can have code branches affected by program inputs. Thus, these code branches can be covered by input mutation to achieve higher code coverage. However, the latter case is not common in our experience, and thus its effect is not very obvious.

4 EH-Fuzz Framework

Based on our error-coverage-guided SFI-based fuzzing approach, we design and implement a new fuzzing framework named EH-Fuzz, to test error handling code and detect bugs. We have implemented EH-Fuzz based on Clang [44] and our previous approach FIFUZZ [42]. We implement input-driven fuzzing in EH-Fuzz by referring to AFL++ [17] for user-level applications and Syzkaller [45] for kernel-level modules. EH-Fuzz performs code instrumentation and program analysis on the LLVM bytecode of the program. Figure 7 shows its architecture that has six modules:

- ***Error-site extractor.*** It performs an automated static analysis of the source code of the tested program, to identify possible error sites.
- ***Program generator.*** It instruments the program code, including the selected error sites, function calls, function entries and exits, code branches, etc. It generates an executable tested program.
- ***Runtime monitor.*** It runs the tested program with the generated inputs, collects runtime information of the tested program (including error coverage, error-unrelated code coverage, memory access information, etc.), and performs fault injection according to the generated possible error sequences.
- ***Error-sequence generator.*** It performs error mutation to generate new possible error sequences, according to error coverage.
- ***Input generator.*** It performs traditional input mutation to generate new program inputs, according to error-unrelated code coverage.
- ***Bug checkers.*** They check the collected runtime information to detect bugs and generate bug reports.

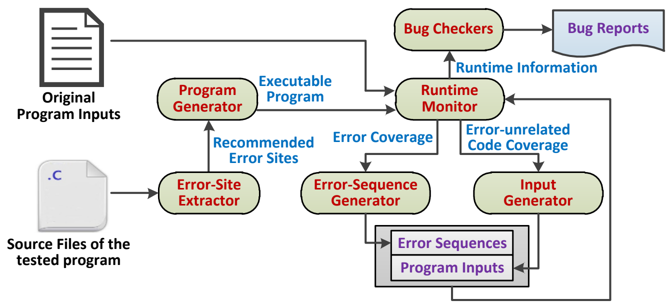


Fig. 7. Overall architecture of EH-Fuzz.

Based on the above architecture, EH-Fuzz consists of two phases, namely compile-time analysis and runtime fuzzing.

4.1 Phase 1: Compile-Time Analysis

In this phase, EH-Fuzz performs two main tasks:

Error-site extraction. For SFI-based approaches, the injected errors should be realistic. Otherwise, the found bugs might be false positives. To ensure that injected errors are realistic, many SFI-based approaches [32], [34], [37] require the user to manually provide error sites, which needs much manual work and cannot scale to large programs. To reduce manual work, the error-site extractor uses a static analysis of the program code to identify possible error sites, from which the user can select realistic ones.

Our analysis extracts specific function calls as error sites, because recent works [30], [40], [42] show that most error sites are code statements checking error-indicating return values of function calls. Our analysis has three steps:

S1: Identifying candidate error sites. In many cases, a function call returns a null pointer or negative integer to indicate a failure. Thus, our analysis identifies a function call as a candidate error site if: 1) it returns a pointer or integer; and 2) the return value is checked by an *if* statement with NULL or zero. The function call to *av_malloc_array* in Figure 1 is an example that satisfies the two requirements. Moreover, to improve accuracy, our analysis also performs an Andersen-style method [46] to identify the variables aliased with each return value. If one of these variables is checked by an *if* statement, the return value is also considered to be checked.

S2: Selecting interface functions. In most cases, a function defined in the tested program can fail, as it calls specific interface functions that can fail. For user-level applications, these interface functions are defined in external libraries; for kernel-level modules, these interface functions are defined in the kernel core. If the calls to these interface functions and their callers are both considered as error sites for fault injection, repeated faults may be injected. To avoid such repetition, from all the identified function calls, our analysis only selects those whose called functions are interface function externally defined outside the tested program.

S3: Performing statistical analysis. In some cases, a function can actually fail and trigger error handling, but the return values of several calls to this function are not checked by *if* statements. To handle such cases, our analysis uses a statistical method to extract functions that can fail from the identified function calls, and we refer to such a function as an *error function*. At first, this method classifies the selected function calls by called function, and collects all function calls to each called function in the tested program code. Then, for the function calls to a given function, this method calculates the percent of them whose return values are checked by *if* statements. If this percent is larger than a threshold *R* (set as 0.6 in our evaluation), this method identifies this function as an error function. Finally, this method extracts all function calls to this function are possible error sites. For accuracy and generality, when testing multiple user-level applications or kernel modules, this method analyzes these applications' code together or the whole kernel code.

In *S3*, the value of the threshold *R* heavily affects the identified error functions and identified error sites (function calls). If *R* is too small, many unrealistic error functions will be identified, which may introduce many false positives in bug detection; if *R* is too large, many realistic error functions will be missed, which may introduce many false negatives in bug detection. Thus, we select *R* as 0.6 in the evaluation, as it is not too small or large. Besides, we will experimentally study the effect of this value in Section 5.2.

Code instrumentation. This task is performed for two purposes: intercepting error points and collecting error-unrelated code coverage.

Error points are intercepted to collect error coverage and perform fault injection. To record the runtime calling context of each error site, the program generator instruments code before and after each function call to each function defined in the tested program code, and at the entry and exit of each function definition. During program execution, the runtime calling context of each error site and its location are collected to create an error point. To monitor the execution states of error points and perform fault injection at runtime, the program generator instruments code before each identified error site. For execution monitoring, if an error point fails at runtime, its value in the covered error sequence is 1; otherwise, its value is 0. For fault injection, if an error point's value in the possible error sequence is 1, a fault is injected into this error point. To inject a fault, the function call of the error site is not executed, and its return value is assigned to a null pointer or a random negative integer. If the error point's value in the possible error sequence is 0, the function call of the error site is normally executed.

To collect error-unrelated code coverage, the program generator needs to identify error-related code branches. For this purpose, the program generator first extracts the basic blocks containing the identified error sites, and then identifies the code branches passing any of these basic blocks to be error-related. After that, the program generator instruments all the code branches in the tested program, except for the identified error-related ones.

4.2 Phase 2: Runtime Fuzzing

In this phase, with the identified error sites and instrumented program, EH-Fuzz performs our error-coverage-guided SFI-based fuzzing approach with traditional input-driven fuzzing, to test the program and detect bugs.

The runtime fuzzer executes the tested program using the program inputs generated by input-driven fuzzing process, and injects faults into the program using the possible error sequences generated by our SFI-based fuzzing approach. It also collects runtime information about executed error points, error-unrelated code branches, etc. The error-sequence generator mutates and generates new possible error sequences, according to error coverage; the input generator mutates and generates new program inputs, according to error-unrelated code coverage. Then, EH-Fuzz combines the generated possible error sequences and program inputs together, and uses them to execute the tested program again. To detect bugs, the bug checkers (such as ASan [47] and MSan [48]) analyze the collected runtime information about memory accesses during fuzzing.

To improve testing performance and correctness, we use two practical techniques in the runtime fuzzer:

Performance enhancement of information collection. The runtime fuzzer can affect the performance of the tested program, because the tested program's threads need to execute much extra code and perform extra synchronization for information collection. To enhance the performance, the runtime fuzzer uses a separate thread and message passing to perform information collection, which can minimize the extra code executed by the tested program's threads and avoid extra synchronization. Specifically, each thread of the tested program executes a little extra code to wrap useful code information in a message that is put in a message queue, and the separate thread fetches all the messages from the message queue to perform information ordering and collection with synchronization.

Interrupt handling in kernel-level modules. A kernel module can raise interrupts during execution, causing the interrupt-handler code to be executed in the current thread's context. If the current thread belongs to the tested kernel module, the runtime fuzzer may record an incorrect calling context that mixes interrupt handler context with the kernel module context. For example, when an interrupt occurs while a kernel module function *ModFunc* is executed, the kernel module can suspend executing *ModFunc* to execute the interrupt handler function *IntrFunc*. In this case, the runtime fuzzer may mistakenly consider *IntrFunc* is called by *ModFunc*, and thus may identify incorrect calling contexts of the executed error sites. To solve this problem, the runtime fuzzer calls the Linux kernel interface `in_interrupt` to check whether the currently executed function is in interrupt handler. If so, the runtime fuzzer regards the interrupt handler as being executed in a separate thread, to maintain a special calling context of the interrupt handler.

4.3 Improvements over FIFUZZ

Compared to our previous fuzzing approach FIFUZZ [42], EH-Fuzz has two new methodology improvements. First, FIFUZZ uses code coverage to guide both SFI-based fuzzing and input-driven fuzzing; while EH-Fuzz uses error coverage and error-unrelated code coverage to guide SFI-based fuzzing and input-driven fuzzing, respectively. Second, FIFUZZ performs SFI-based fuzzing and input-driven fuzzing together, by randomly using error mutation or input mutation; while EH-Fuzz performs SFI-based fuzzing and input-driven fuzzing separately, by smartly scheduling error mutation and input mutation. Due to the two new improvements, compared to FIFUZZ, EH-Fuzz is more effective in both SFI-based fuzzing and input-driven fuzzing to find more deep and complex bugs.

TABLE 2
Basic information of the tested programs.

Program	Description	Version	LOC
vim	Text editor	v8.2.3595	540K
bison	Parser generator	v3.8.1	92K
ffmpeg	Solution for media processing	v4.4.1	1.1M
nasm	80x86 and x86-64 assembler	v2.15.05	105K
catdoc	MS-Word-file viewer	v0.95	4K
clamav	Antivirus engine	v0.104.1	219K
cflow	Code analyzer of C source files	v1.7	34K
gif2png+libpng	File converter for pictures	v2.5.14+v1.6.3	59K
openssl	Cryptography library	v3.0.0	516K
btrfs	Linux BTRFS filesystem	Linux 5.16.16	105K
xfs	Linux XFS filesystem	Linux 5.16.16	97K
jfs	Linux JFS filesystem	Linux 5.16.16	17K
cephfs	Linux Ceph filesystem	Linux 5.16.16	22K
xhci	Intel USB 3.0 host driver	Linux 5.16.16	65K
vmxnet3	Vmware virtual network driver	Linux 5.16.16	5K

TABLE 3
Results of error-site extraction.

Program	Function call	Identified	Realistic
vim	83,129	334 (0.40%)	265 (79.34%)
bison	16,464	187 (1.14%)	129 (68.98%)
ffmpeg	473,450	198 (0.04%)	103 (52.02%)
nasm	11,162	62 (0.56%)	28 (45.16%)
catdoc	1,322	101 (7.64%)	69 (68.32%)
clamav	124,350	2,125 (1.71%)	1,247 (58.68%)
cflow	3,738	117 (3.13%)	84 (71.79%)
gif2png+libpng	16,317	129 (0.79%)	65 (50.39%)
openssl	372,715	135 (0.04%)	102 (75.56%)
btrfs	158,098	929 (0.59%)	351 (37.78%)
xfs	165,786	201 (0.12%)	171 (85.07%)
jfs	24,003	114 (0.47%)	100 (87.72%)
cephfs	34,470	460 (1.33%)	140 (30.43%)
xhci	14,419	180 (1.25%)	104 (57.78%)
vmxnet3	5,896	98 (1.66%)	43 (43.88%)
Total	1,505,319	5,370 (0.36%)	3,001 (55.88%)

In addition, EH-Fuzz has two implementation improvements over FIFUZZ. First, EH-Fuzz handles the alias relationship of each function-call return value while FIFUZZ neglects, and thus EH-Fuzz produces fewer false positives in error-site extraction, which can reduce manual work of selecting realistic error sites. Second, FIFUZZ can only test user-level applications; while EH-Fuzz is applicable to both user-level applications and kernel-level modules, by enhancing information-collection performance and considering interrupt handling.

5 Evaluation

5.1 Experimental Setup

To validate the effectiveness of EH-Fuzz, we evaluate it on 9 popular user-level C applications of the latest versions as of our evaluation, as well as 6 common kernel-level modules (4 filesystems and 2 device drivers) in Linux 5.16.16. The basic information of these 15 tested programs are listed in Table 2 (the lines of source code are counted by CLOC [49]). The experiment runs on a regular desktop with eight Intel i7-3770@3.40G processors and 16GB physical memory. The code compiler is Clang 12.0 [44], and the OS is Ubuntu 20.04.

5.2 Error-Site Extraction

Before testing the 15 programs, EH-Fuzz first performs a static analysis of their source code to identify possible error sites, and then we manually select realistic ones that can actually fail and trigger error handling code, according to the related source code. Table 3 shows the results, when we set R as 0.6. The first column presents the program name; the second column presents the number of all function calls in the program code; the third column presents the number of error sites identified by EH-Fuzz; the last column presents the number of realistic error sites that we manually select.

In total, EH-Fuzz identifies 157 error functions, and extract 5,370 function calls to these functions as possible error sites. Among them, we manually identify 122 realistic error functions and 3,001 function calls to these functions as realistic error sites, achieving a false positive rate of 44%. This rate is lower than that of our previous approach FIFUZZ (81%), as EH-Fuzz handles the alias relationship of each function-call return value while FIFUZZ neglects. In fact, the manual selection of realistic error sites is easily manageable and not hard, as many error sites call the same functions and checking each error site does not require much program-specific knowledge. One master student spent 4 hours on the manual selection of these error sites in the 15 tested programs, and this manual work is similar to that of FIFUZZ. Considering that there are over 1.5M function calls in the tested programs, EH-Fuzz drops over 99% of them, because they are considered not to fail or trigger error handling code, according to their contexts in source code. The results indicate that EH-Fuzz can dramatically help reduce the manual work of identifying realistic error sites.

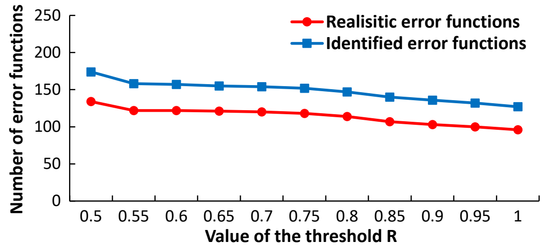


Fig. 8. Variation of error functions affected by the value of R.

As described in Section 4.1, the value of the threshold R in our static analysis heavily affects the identified error functions. The above results are obtained with R = 0.6. to understand the effect of this value, we test it from 0.5 to 1 with 0.05 step, and show the results in Figure 8. We find that the number of identified error functions and realistic error functions are both decreased when R becomes larger. In this case, more unrealistic error functions are dropped, but more realistic ones are also missed. Thus, if R is too small, many unrealistic error functions will be identified, which may introduce many false positives in bug detection; if R is too large, many realistic error functions will be missed, which may introduce many false negatives in bug detection.

5.3 Runtime Testing

With the 3,001 realistic error sites shown in Table 3, we use EH-Fuzz to test the 15 programs. Following the recommendations of [50], we fuzz each program with ASan [47] (for applications) or KASan [51] (for kernel modules) for three times, and the fuzzing time limit is 24 hours. To count unique bugs, we manually check their root causes with the bug reports and source code. Table 4 shows the results. The columns *"Error sequences"* and *"Program inputs"* show

TABLE 4
Fuzzing results.

Program	Error sequence		Program input		Testing coverage		Found bug							
	Generate	*Interest*	*Generate*	*Interest*	*ErrSeq*	*Branch*	*NullPtr*	*UAF*	*DoubleFree*	*Overflow*	*MemLeak*	*InvalidPtr*	*Assert*	*All*
vim	650,594	2,638	108,735	3,626	379,636	60,318	1	1	0	2	0	1	0	5
bison	931,504	13,552	1,379,860	1,329	210,157	20,465	0	0	0	0	0	0	0	0
ffmpeg	171,121	1,048	54,650	2,429	109,413	82,250	7	1	2	3	0	0	0	13
nasm	153,205	1,447	2,444,814	1,766	1,451	11,454	0	0	0	0	0	0	0	0
catdoc	3,027	233	130,025	148	866	1,283	4	0	0	1	0	0	0	5
clamav	150,231	753	216,588	954	41,894	24,153	0	0	1	1	0	1	0	3
cflow	2,460,190	35,771	4,189,353	596	87,824	4,581	0	0	0	1	0	0	0	1
gif2png+libpng	1,276	89	2,241,618	298	91	3,268	0	0	0	0	0	0	0	0
openssl	237,717	129,366	6,297,639	97	204,785	18,196	2	1	1	0	0	0	0	4
btrfs	1,041	881	106,546	1,228	1,041	42,427	0	1	1	0	1	0	0	3
xfs	1,448	677	157,089	1,262	990	26,413	1	0	0	0	0	0	0	1
jfs	1,725	1,019	137,838	365	1,396	9,835	1	0	0	0	0	0	1	2
cephfs	949	720	232,802	2,187	871	11,660	3	0	0	0	1	0	0	4
xhci	1,330	1,052	143,340	2,299	1,211	4,342	1	0	0	0	0	0	0	1
vmxnet3	1,694	1,088	145,498	2,440	1,474	2,553	1	1	0	0	0	0	1	3
Total	4,767,052	190,334	17,986,395	21,024	1,043,100	323,198	21	5	5	8	2	2	2	45

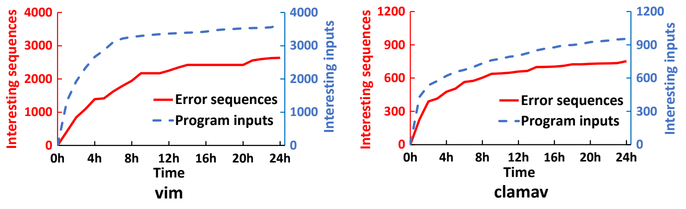


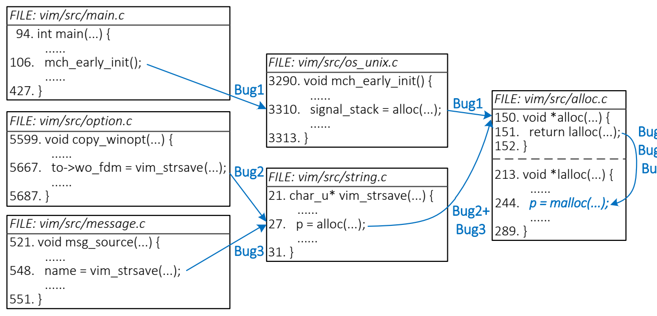
Fig. 9. Interesting error sequences and inputs for *vim* and *clamav*.

the results about generated error sequences and inputs; the column *"Generate"* shows the number of generated ones, and the column *"Interest"* shows the number of interesting ones that increase error coverage or error-unrelated code coverage. The columns *"ErrSeq"* and *"Branch"* show the number of the covered error sequences and code branches, respectively. From Table 4, we make some observations.

Error coverage. EH-Fuzz covers many error sequences in different contexts, and they are generated by the error mutation according to our error coverage. In total, 4% of the generated error sequences increase error coverage, so they are considered to be interesting. This proportion is larger than that (0.01%) of the generated program inputs, so error mutation outperforms input mutation. Particularly, the proportion of interesting error sequences in kernel-level modules is much larger than user-level applications. Indeed, an application test (0.16s on average) is much shorter than a kernel test (63s on average), causing many more application tests to be executed; because uninteresting error sequences are often generated in the later tests, running more tests can cause a larger proportion of uninteresting error sequences.

To better understand the variation of interesting error sequences and program inputs, we select *vim* and *clamav* as examples to study their results in Figure 9. We find that the number of interesting error sequences increases quickly during earlier tests, and then tends to be stable in the later tests. This trend is quite similar to program inputs.

Found bugs. EH-Fuzz in total finds 45 new and unique bugs in terms of their root causes, including 31 bugs in user-level applications and 14 bugs in kernel-level modules. Specifically, 39 of these bugs are caused by incorrect error handling, and they are found by SFI-based fuzzing; the remaining 6 bugs are caused by incorrect handling of program inputs, and they are found by input-driven fuzzing. We have reported all these bugs to the related developers.



```
FILE: vim/src/main.c
 94. int main(...) {
       ......
106.   mch_early_init();
       ......
427. }

FILE: vim/src/option.c
5599. void copy_winopt(...) {
        ......
5667.   to->wo_fdm = vim_strsave(...);
        ......
5687. }

FILE: vim/src/message.c
521. void msg_source(...) {
       ......
548.   name = vim_strsave(...);
       ......
551. }

FILE: vim/src/os_unix.c
3290. void mch_early_init() {
        ......
3310.   signal_stack = alloc(...);
        ......
3313. }

FILE: vim/src/string.c
21. char_u* vim_strsave(...) {
      ......
27.   p = alloc(...);
      ......
31. }

FILE: vim/src/alloc.c
150. void *alloc(...) {
151.   return lalloc(...);
152. }
- - - - - - - - - - - -
213. void *lalloc(...) {
       ......
244.   p = malloc(...);
       ......
289. }
```

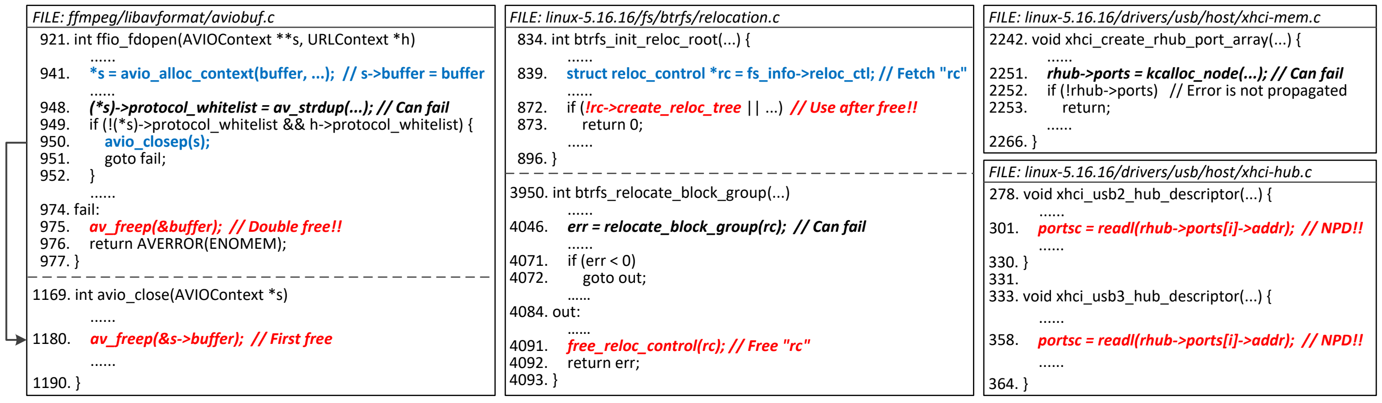
Bug1 (NullPtr): main -> mch_early_init(106) -> alloc(3310) -> lalloc(151) -> *malloc(244)*
Bug2 (Overflow): copy_winopt -> vim_strsave(5667) -> alloc(27) -> lalloc(151) -> *malloc(244)*
Bug3 (UAF): msg_source -> vim_strsave(548) -> alloc(27) -> lalloc(151) -> *malloc(244)*

Fig. 10. Example bugs caused by the same error site.

31 of them have been confirmed and fixed (including 19 in applications and 12 in kernel modules), and we are still waiting for the response of the remaining ones.

Error handling bugs. The 39 found bugs related to incorrect error handling are caused by only 28 error sites but in different calling contexts. 26 of the error sites are related to memory-allocation errors. Figure 10 shows such examples of three bugs found in *vim*. The three bugs are a null-pointer dereference (*Bug1*), a buffer-overflow bug (*Bug2*) and a use-after-free bug (*Bug3*), and they have different root causes according to our manual checking. Additionally, the developer fixes each of these bugs by modifying separate error handling code. The text in each line presents the call stack of error site, including the function name and code line number of function call. The three bugs are all caused by the failures of the function call to `alloc`, but the failures occur in different calling contexts. The results confirm the importance of considering runtime contexts of error sites in SFI-based fuzzing.

Bug features. Reviewing the bugs found by EH-Fuzz, we find three interesting features. First, among the 39 error handling bugs, only 2 are triggered by two error points' failures, and the remaining 37 bugs are triggered by only one error point's failure. The results indicate that error handling bugs are often triggered by just one error in specific context. Second, among the 10 use-after-free and double-free bugs, 8 of them are caused by missing null-pointer assignments after freeing the pointers. Indeed, there are NULL checks be-



```
FILE: ffmpeg/libavformat/aviobuf.c
921. int ffio_fdopen(AVIOContext **s, URLContext *h)
         ......
941.     *s = avio_alloc_context(buffer, ...);  // s->buffer = buffer
         ......
948.     (*s)->protocol_whitelist = av_strdup(...); // Can fail
949.     if (!(*s)->protocol_whitelist && h->protocol_whitelist) {
950.        avio_closep(s);
951.        goto fail;
952.     }
         ......
974. fail:
975.     av_freep(&buffer);  // Double free!!
976.     return AVERROR(ENOMEM);
977. }
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
1169. int avio_close(AVIOContext *s)
         ......
1180.    av_freep(&s->buffer);  // First free
         ......
1190. }
```

(a) Double-free bug in *ffmpeg* application

```
FILE: linux-5.16.16/fs/btrfs/relocation.c
834. int btrfs_init_reloc_root(...) {
         ......
839.     struct reloc_control *rc = fs_info->reloc_ctl; // Fetch "rc"
         ......
872.     if (!rc->create_reloc_tree || ...)  // Use after free!!
873.        return 0;
         ......
896. }
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
3950. int btrfs_relocate_block_group(...)
         ......
4046.    err = relocate_block_group(rc);  // Can fail
         ......
4071.    if (err < 0)
4072.       goto out;
         ......
4084. out:
         ......
4091.    free_reloc_control(rc); // Free "rc"
4092.    return err;
4093. }
```

(b) Use-after-free bug in *btrfs* filesystem

```
FILE: linux-5.16.16/drivers/usb/host/xhci-mem.c
2242. void xhci_create_rhub_port_array(...) {
          ......
2251.     rhub->ports = kcalloc_node(...); // Can fail
2252.     if (!rhub->ports)   // Error is not propagated
2253.        return;
          ......
2266. }
```

```
FILE: linux-5.16.16/drivers/usb/host/xhci-hub.c
278. void xhci_usb2_hub_descriptor(...) {
         ......
301.     portsc = readl(rhub->ports[i]->addr);  // NPD!!
         ......
330. }
331.
333. void xhci_usb3_hub_descriptor(...) {
         ......
358.     portsc = readl(rhub->ports[i]->addr);  // NPD!!
         ......
364. }
```

(c) Null-pointer dereference in *xhci* driver

Fig. 11. Example bugs found by EH-Fuzz.

TABLE 5
Security impact classified by bug type.

Bug type	*Crash/DoS*	*Data corruption*	*Memory overread*
Null-pointer dereference	21	0	0
Use after free	0	5	0
Double free	0	5	0
Buffer overflow	0	3	5
Memory leak	2	0	0
Invalid-pointer access	2	0	0
Assertion failure	2	0	0
Total	27	13	5

fore the pointers are used or freed, namely these bugs can be avoided if these pointers are assigned to NULL. Finally, 28 of the found bugs are caused by incorrect error propagation across functions, such as neglecting or repeatedly handling the errors propagated from callee functions. For example in Figure 1, the failure of the call to `av_mallocz_array` is correctly handled in `swri_realloc_audio`, which returns an error code *-ENOMEM*; but when this error is propagated to `swr_convert_internal` through the function call to `resample`, the error is neglected and passed to `swri_rematrix` without error handling, causing a buffer-overflow bug. Indeed, error propagation across functions is error-prone, due to complex calling contexts of error sites.

5.4 Security Impact of the Found Bugs

We manually review the 45 found bugs to estimate their security impact. The results are shown in Table 5, classified by bug type, including null-pointer dereference, use after free, double free, buffer overflow, memory leak, invalid-pointer access and assertion failure. These bugs can cause serious security problems, including denial of service (DoS), data corruption and memory overread. Figure 11 shows three example bugs, which have been confirmed and fixed.

Double-free bug in *ffmpeg* application. In Figure 11(a), the function `av_strdup` on line 948 can fail due to insufficient memory. In this case, `avio_closep` is called to free the pointer `s->buffer` via `av_freep` on line 1180. Then, `av_freep` is called again to free the pointer `buffer` on line 975. Because `s->buffer` has been assigned with `buffer` in `avio_alloc_context` on line 941, `av_freep` is actually called twice to free the same pointer `buffer`, causing a double-free bug. Once this bug is triggered, it can be exploited to corrupt memory area pointed by `buffer`, and thus to inject malicious data into the media file.

Use-after-free bug in *btrfs* filesystem. In Figure 11(b), the function `relocate_block_group` on line 4046 can fail due to insufficient memory. In this case, the pointer `rc` is freed by `free_reloc_control` on line 4091, and then an error code is returned on line 4092. However, this error code is not properly handled by the filesystem, causing that when the function `btrfs_init_reloc_root` is called in another thread, the pointer `rc` can be still fetched via `fs_info->reloc_ctl` on line 839. Then, `rc` is used to access `rc->create_reloc_tree` on line 872, causing a use-after-free bug. Once this bug is triggered, it can be exploited to corrupt the memory area pointed by `rc`, and thus to inject malicious data into the filesystem.

Null-pointer dereference in *xhci* driver. In Figure 11(c), the function `kcalloc_node` on line 2251 can fail due to insufficient memory, causing the pointer `rhub->ports` to be NULL, but this error is not propagated to other functions. Thus, when the functions `xhci_usb2_hub_descriptor` and `xhci_usb3_hub_descriptor` are called to access `rhub->ports[i]->addr`, a null-pointer dereference occurs on lines 301 and 358, respectively. Once this bug is triggered, it can be exploited to perform DoS attack.

5.5 Comparison to Existing Fuzzing Approaches

We select five state-of-the-art fuzzing approaches for comparison, including three popular input-driven fuzzing approaches (AFL [16], AFL++ [17] and Syzkaller [45]) and two recent SFI-based fuzzing approaches (FIZZER [40] and FIFUZZ [42]). As AFL, AFL++ and FIFUZZ can only test user-level applications, we use them for the 9 applications in Table 2; as Syzkaller and FIZZER can only test kernel-level programs, we use them for the 6 kernel modules in Table 2. Note that FIZZER lacks input-driven fuzzing, so we run common benchmarks *fstest* [52], *usb-tester* [53] and *netperf* [54], for the four filesystems (*btrfs*, *xfs*, *jfs* and *cephfs*), *xhci* USB host driver and *vmxnet3* network driver, respectively. Following the recommendations of [50], we use EH-Fuzz and these approaches to test each program with ASan [47] (for applications) or KASan [51] (for kernel modules) for three times, and the fuzzing time limit is 24 hours. Table 6 shows the results, and we make several observations.

Code coverage. Compared to the five existing fuzzing approaches, EH-Fuzz has higher code coverage in most of the 15 tested programs, as it covers more error handling code.

TABLE 6
Comparison results of EH-Fuzz and five existing fuzzing approaches.

Program	AFL		AFL++		Syzkaller		FIZZER			FIFUZZ			EH-Fuzz		
	Branch	*Bug*	*Branch*	*Bug*	*Branch*	*Bug*	*Branch*	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*
vim	49,010	0	64,717	0	-	-	-	-	-	34,621	1,855	1	60,318	379,636	5
bison	18,240	0	20,583	0	-	-	-	-	-	18,107	498	0	20,465	210,157	0
ffmpeg	42,201	0	63,988	2	-	-	-	-	-	54,009	789	5	82,250	109,413	13
nasm	10,772	0	11,949	0	-	-	-	-	-	10,563	148	0	11,454	1,451	0
catdoc	1,167	2	1,210	2	-	-	-	-	-	1,257	168	3	1,283	866	5
clamav	23,115	0	24,105	0	-	-	-	-	-	18,023	137	2	24,153	41,894	3
cflow	4,332	0	4,577	1	-	-	-	-	-	4,350	98	0	4,581	87,824	1
gif2png+libpng	2,534	0	3,044	0	-	-	-	-	-	2,200	66	0	3,268	91	0
openssl	17,167	0	17,479	0	-	-	-	-	-	17,068	806	4	18,196	204,785	4
btrfs	-	-	-	-	38,468	1	11,540	791	1	-	-	-	42,427	1,041	3
xfs	-	-	-	-	21,234	0	8,322	525	1	-	-	-	26,413	990	1
jfs	-	-	-	-	8,207	0	4,006	814	0	-	-	-	9,835	1,396	2
cephfs	-	-	-	-	6,621	0	5,493	233	1	-	-	-	11,660	871	4
xhci	-	-	-	-	2,941	0	1,980	357	1	-	-	-	4,342	1,211	1
vmxnet3	-	-	-	-	1,672	0	1,012	294	2	-	-	-	2,553	1,474	3

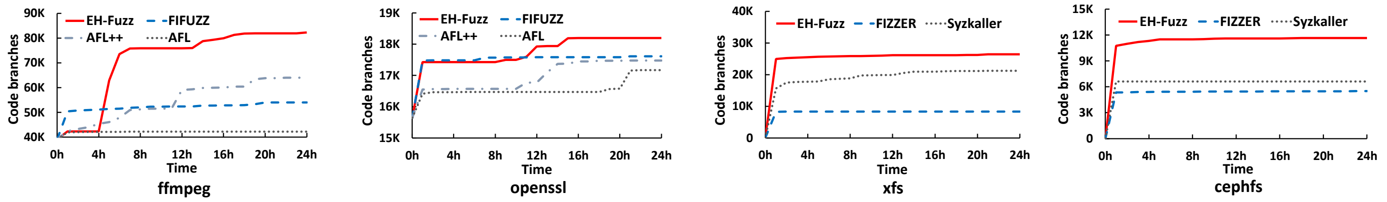


Fig. 12. Covered code branches in the comparison.

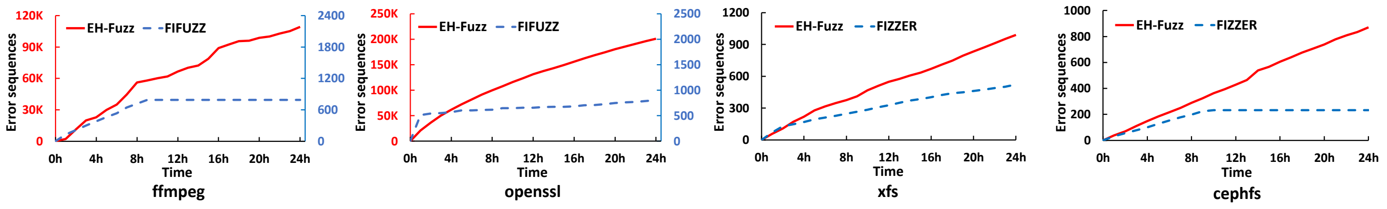


Fig. 13. Covered error sequences in the comparison.

As for *vim*, *bison* and *nasm*, EH-Fuzz covers slightly fewer code branches than AFL++. Indeed, EH-Fuzz performs both input mutation and error mutation, while AFL++ always performs input mutation. For the three programs, within the same testing time, input mutation contributes more in code coverage than error mutation, though they cover many different code branches. We select four programs (*ffmpeg*, *openssl*, *xfs* and *cephfs*) as examples to study code coverage in Figure 12. The number of covered code branches increases quickly during earlier tests, and then tends to be stable in the later tests; and EH-Fuzz achieves higher code coverage than the other approaches in the four programs.

Error coverage. Compared to the two SFI-based fuzzing approaches FIZZER and FIFUZZ, EH-Fuzz has higher error coverage by covering more error sequences. Indeed, FIZZER lacks input mutation and performs context-insensitive fault injection, so it fails to cover error handling code triggered by specific inputs or executed in different contexts. FIFUZZ covers more error sequences than FIZZER, by performing input mutation and context-sensitive fault injection. However, it guides SFI-based fuzzing according to code coverage, which cannot reflect the context information of error handling code; and it randomly selects error mutation or input mutation, without knowing whether the coverage is increased by error mutation or input mutation. Thus, it still misses much error handling code in many contexts. To solve the two problems, EH-Fuzz uses error coverage and error-unrelated code coverage to guide SFI-based fuzzing and input-driven fuzzing, respectively; and it smartly schedules error mutation or input mutation, to maximize error coverage and code coverage. Thus, EH-Fuzz achieves higher error coverage and code coverage than FIZZER and FIFUZZ. We select four programs (*ffmpeg*, *openssl*, *xfs* and *cephfs*) as examples to study error coverage in Figure 13. In this figure, the number of covered error sequences increases along with testing time, EH-Fuzz achieves higher error coverage than the other approaches in the four programs.

Found bugs in applications. In the 9 user-level applications, AFL, AFL++ and FIFUZZ finds 18 unique bugs. Specifically, AFL++ find 3 input-related bugs missed by AFL and FIFUZZ, because AFL++ integrates better strategies of input mutation and seed selection; FIFUZZ finds 13 error handling bugs missed by AFL and AFL++, because its fault injection coverages more error handling code; the 2 input-related bugs found by AFL are found by both AFL++ and FIFUZZ. EH-Fuzz finds 31 bugs in the 9 applications, including the 18 bugs found by these fuzzing approaches, and 13 bugs missed by them, due to higher testing coverage.

Found bugs in kernel modules. In the 6 kernel-level modules, Syzkaller and FIZZER find 7 unique bugs. Specifically, Syzkaller finds one input-related bug missed by FIZZER, due to input mutation; and FIZZER finds 6 error handling bugs missed by Syzkaller, due to error mutation. EH-Fuzz finds 14 bugs in the 6 kernel modules, including the 7 bugs found by these fuzzing approaches, and 7 bugs missed by them, due to higher testing coverage.

TABLE 7
Evaluation results of our ablation study.

Program	FIFUZZ			FIFUZZ$_{alias}$			FIFUZZ$^{err\_cov}_{alias}$			FIFUZZ$^{mut\_sch}_{alias}$			EH-Fuzz		
	Branch	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*	*Branch*	*ErrSeq*	*Bug*
vim	34,621	1,855	1	66,931	16,710	2	66,862	316,482	4	68,587	15,574	1	60,318	379,636	5
bison	18,107	498	0	20,219	2,501	0	20,086	85,501	0	20,311	2,292	0	20,465	210,157	0
ffmpeg	54,009	789	5	104,621	15,272	9	98,069	68,600	8	105,117	19,624	7	82,250	109,413	13
nasm	10,563	148	0	10,061	812	0	10,379	1,539	0	10,047	805	0	11,454	1,451	0
catdoc	1,257	168	3	1,270	37	4	1,278	507	4	1,273	49	4	1,283	866	5
clamav	18,023	137	2	28,116	10,998	2	28,907	34,802	3	28,127	5,403	2	24,153	41,894	3
cflow	4,350	98	0	4,533	34,481	1	4,532	69,812	1	4,533	34,446	1	4,581	87,824	1
gif2png+libpng	2,200	66	0	3,283	79	0	3,283	90	0	3,283	79	0	3,268	91	0
openssl	17,068	806	4	18,027	98,422	4	18,029	155,538	4	18,027	99,228	4	18,196	204,785	4
Total	160,198	4,565	15	257,061	179,312	22	251,425	732,871	24	259,305	177,500	19	225,968	1,036,117	31

5.6 Sensitivity Analysis

Ablation study on FIFUZZ. EH-Fuzz has two new methodology improvements over our previous approach FIFUZZ: (1) using a new program feedback error coverage to guide SFI-based fuzzing, and (2) smartly scheduling error mutation and input mutation according to error coverage and error-unrelated code coverage. Besides, EH-Fuzz has an implementation improvement of considering the alias relationship of each function-call return value.

To clearly understand the value of each improvement made by EH-Fuzz, we perform an ablation study in the evaluation. In detail, we implement three tools by adding each improvement in FIFUZZ: (1) FIFUZZ$_{alias}$ uses alias analysis of function-call return values (namely FIFUZZ + alias analysis); (2) FIFUZZ$^{err\_cov}_{alias}$ uses alias analysis of function-call return values, and randomly schedules error mutation and input mutation using error coverage and code coverage, respectively (namely FIFUZZ + alias analysis + error coverage), and (3) FIFUZZ$^{mut\_sch}_{alias}$ uses alias analysis of function-call return values, and adaptively schedules error mutation and input mutation using just code coverage (namely FIFUZZ + alias analysis + mutation scheduling). In the ablation study, we evaluate these tools on the 9 tested user-level applications, and compare their evaluation results with FIFUZZ and EH-Fuzz in Table 7.

Compared to FIFUZZ, the three implemented tools find more bugs by covering more code branches and error sequences, reflecting the effectiveness of each improvement made by EH-Fuzz. Specifically, with alias analysis, FIFUZZ$_{alias}$ identifies many realistic error sites missed by FIFUZZ, and they are used to cover more error handling code; FIFUZZ$^{err\_cov}_{alias}$ uses error coverage as program feedback, which can cover complex error handling situations; FIFUZZ$^{mut\_sch}_{alias}$ identifies the contribution of error mutation and input mutation to code coverage, for adaptive mutation scheduling, which is smarter than random mutation scheduling. Moreover, EH-Fuzz produces the best results in error coverage and bug detection, indicating that the combination of these improvements is indeed useful to test error handling code and find deep bugs.

Effect analysis of mutation scheduling. As described in Section 3.4, EH-Fuzz switches to error mutation or input mutation after running N error sequences or program inputs. In the above evaluation, we set N as 10% of the executed test cases, according to our experience. To understand the effect of this value, we test it from 10% to 50% with 10% step, and show the results of bug detection for the 9 user-level applications in Figure 14. We find that this value has effect on bug detection result, as it decides the number of error mutation and input mutation; and EH-Fuzz produces the best result in bug detection, when N is set as 10% of the executed test cases. In fact, each tested program should have a unique optimal value of N, because different programs have different error sites and possible program inputs. In our evaluation, for convenience, we just use the same criterion of N for all the tested programs.

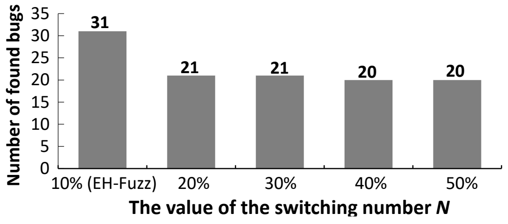


Fig. 14. Variation of found bugs affected by the value of *N*.

6 Discussion

6.1 False Positive and Negative Analysis

False positives. EH-Fuzz could report false bugs, if the selected error sites for fault injection are unrealistic. In the evaluation, we manually select realistic error sites with the aid of our static analysis in Section 4.1, so there is no false positive of bug detection. This static analysis can automatically identify possible error sites from the program code, but it still has some false positives as shown in Table 3. Indeed, some functions that return pointers or integers never cause errors, though their return values are often checked in the code. The functions `strcmp` and `strstr` are examples. To reduce such cases, a blacklist or a precise dataflow analysis can be used to drop these functions. In our opinion, to avoid false positives of bug detection, like our previous FIFUZZ, the user still needs to manually check the identified possible error sites and select realistic ones.

Although with realistic error sites, our error mutation can still produce wrong error sequences, which are actually impossible in real execution. However, they never cause false positives of bug detection, as error handling code of these error sequences are never triggered in real execution.

False negatives. EH-Fuzz can still miss real bugs in error handling code due to three possible reasons. First, some error handling code is triggered by erroneous data (such as bad data read from hardware registers or DMA buffers), but EH-Fuzz just injects faults into the return values of specific function calls, and thus such error handling code is not covered at present. Second, some error sites are executed only when specific program inputs and configuration are provided. In the evaluation, EH-Fuzz cannot provide all possible program inputs and configuration, and thus some error sites may not be executed to cover the related error handling code. Finally, we use ASan and KASan to report just memory bugs in the evaluation, Other checkers can be used to detect other kinds of bugs, such as MSan [48] to detect uninitialized uses, UBSan [55] to detect undefined behaviors, and TSan [56] to detect data races.

6.2 Exploitability of Error Handling Bugs

To detect bugs in error handling code, EH-Fuzz injects errors in specific orders according to calling contexts. Thus, to actually reproduce and exploit a bug found by EH-Fuzz, two requirements should be satisfied: (1) being able to actually produce related errors: (2) controlling the occurrence order and time of related errors.

For the first requirement, different kinds of errors can be produced in different ways. We have to manually look into the error-site function to understand its semantics. However, most of the bugs found in our evaluations are related to memory-allocation errors. Thus, an intuitive exploitation way is to exhaustively consume the memory, which has been used in some approaches [57], [58] to perform attacks.

For the second requirement, as we have the error sequence that triggers the bug, we can know when to and when not to produce the errors. A key challenge here is, when errors are dependent to each other, we must timely produce an error in a specific time window, similar to exploiting use-after-free bugs [59], [60].

6.3 Advice on Avoiding Error Handling Bugs

After studying the root causes of error handling bugs found by EH-Fuzz, we have some suggestions on how to avoid such bugs in common programs:

First, the developers should clearly understand which code operations (especially function calls) can cause errors, to add necessary security checks and related error handling of these operations, to avoid missing-check bugs.

Second, the developers should correctly propagate errors across functions through function return values or reference parameters to caller functions, to avoid the bugs caused by missing error handling of ancestor functions, such as the null-pointer dereference in Figure 11(c).

Finally, the developers should obey several implicit rules of error handling. For example, our previous study [61] provides two useful rules: (R1) resource-release order should be reversed to resource-allocation order; and (R2) when a function encounters an error, all successfully acquired resources in this function should be released in the error handling code of this function, not its caller or callee functions. For examle in Figure 11(a), if R2 is obeyed in the code, we believe that the double-free bug could be avoided.

7 Related Work

7.1 Fuzzing

Fuzzing [62] is a useful technique of runtime testing to detect bugs and discover vulnerabilities. It generates lots of program inputs in a specific way to cover infrequently executed code. A typical fuzzing approach can be generation-based, mutation-based, or a hybrid of them.

Generation-based fuzzing approaches [12]–[15] generate inputs according to the specific input formats. Csmith [12] is a randomized test-case generator to fuzz C-language compilers. According to C99 standard, Csmith randomly generates a large number of C programs as inputs for the tested compiler, and these programs contain complex code using different kinds of C-language features.

Mutation-based fuzzing approaches [16]–[25] start from some original seeds, and perform mutation of the selected seeds, to generate new inputs, without the requirement of specific formats. To improve code coverage, these approaches often mutate existing inputs according to the feedback of program execution, such as code coverage and bug-detection results. AFL [16] and AFL++ [17] are two well-known coverage-guided fuzzing frameworks, which have been widely-used in industry and research. They use many effective fuzzing strategies and technical tricks to reduce runtime overhead and improve fuzzing efficiency.

Some approaches [26]–[29] combine generation-based and mutation-based fuzzing for efficient bug detection. AFLSmart [26] uses a high-level structural representation of the seed file to generate new files. It mutates on the file-structure level not on the bit level, which can completely explore new input domains without breaking file validity.

Existing fuzzing approaches focus on generating inputs to cover infrequently executed code, but they are limited in covering error handling code triggered by non-input errors. To solve this problem, EH-Fuzz introduces software fault injection in fuzzing, and generates injected faults according to a new feedback *error coverage*. In this way, EH-Fuzz can effectively test error handling code and find deep bugs.

7.2 Software Fault Injection

Software fault injection (SFI) is a classical technique of testing error handling code. SFI intentionally injects faults or errors into the tested program, and then executes the program to test whether it can correctly handle the injected faults or errors during execution.

Existing SFI-based approaches inject single fault [30]–[32] or multiple faults [33]–[38] in each test case to cover error handling code. Some of these approaches [33]–[35] inject faults on random error sites or randomly change program data, but some studies [63]–[65] have shown that random fault injection introduces much uncertainty, causing that the code coverage is low and many detected bugs are false. To solve this problem, some approaches [36]–[38] analyze program information to guide fault injection, which can achieve higher code coverage and detect more bugs.

Existing SFI-based approaches perform only context-insensitive fault injection, namely they inject faults based on the locations of error sites in source code, without considering the execution contexts of these error sites. Thus, these approaches may miss many error handling bugs that only occur in specific calling contexts of error sites. Moreover, these approaches require proper workloads, because they do not generate the test cases of program inputs.

7.3 SFI-Based Fuzzing

Several recent fuzzing approaches [39]–[42] introduce software fault injection (SFI) and "fuzz" injected faults to trigger infrequently-executed error handling code. However, these approaches still have two limitations in testing error handling code. First, these approaches all use code coverage to guide SFI-based fuzzing, but code coverage cannot reflect the runtime contexts of the covered error handling code. Thus, these approaches may miss many interesting test cases of error mutation, which can trigger real error handling bugs. Second, these approaches have weak compatibility with input-driven fuzzing. FIFUZZ [42] is the sole one that is compatible with input-driven fuzzing. It randomly selects error mutation or input mutation when new code branches are covered, without knowing whether these new code branches are covered exactly due to fault injection or program inputs. As a result, FIFUZZ may make mistakes in selecting between error mutation and input mutation for test case generation, and thus miss many real bugs.

To solve these limitations, EH-Fuzz uses two new techniques. First, EH-Fuzz uses the covered error sequences as error coverage, to reflect the runtime contexts of the covered error handling code. Second, EH-Fuzz smartly schedules SFI-based fuzzing and input-driven fuzzing, according to error coverage and error-unrelated code coverage. Thanks to these techniques, EH-Fuzz can achieve higher testing coverage and find more bugs than existing SFI-based fuzzing approaches, as shown in Section 5.5.

7.4 Static Analysis of Error Handling Code

Static analysis can conveniently analyze the source code of the target program without actually executing the program. Thus, some existing approaches [5], [7], [8], [11], [66] use static analysis to detect bugs in error handling code. EDP [7] statically validates the error propagation through file systems and storage device drivers. It builds a function-call graph that shows how error codes propagate through return values and function parameters. By analyzing this call graph, EDP detects bugs about incorrect operations on error codes. APEx [66] infers API error specifications from their usage patterns, based on a key insight that error paths tend to have fewer code branches and program statements than regular code.

Due to lacking exact runtime information, static analysis often reports many false positives. For example, the false positive rate of EPEx [11] is 22%. However, we believe static analysis is still useful to SFI-based fuzzing, such as error-site extraction in EH-Fuzz.

8 Conclusion

In this paper, we propose a new fuzzing framework named EH-Fuzz to effectively test error handling code. EH-Fuzz introduces software fault injection (SFI) in coverage-guided fuzzing, with two new techniques. First, EH-Fuzz guides SFI-based fuzzing using a new feedback named error coverage, which reflects the runtime contexts of the covered error handling code. Second, EH-Fuzz smartly schedules SFI-based fuzzing and input-driven fuzzing, according to error coverage and error-unrelated code coverage. We have evaluated EH-Fuzz on 9 user-level programs and 6 kernel-level modules, and in total finds 45 new real bugs. 31 of these bugs have been confirmed and fixed by the related developers. We also compare EH-Fuzz to existing fuzzing approaches, and EH-Fuzz finds many real bugs missed by these approaches with higher testing coverage. EH-Fuzz is available on https://sites.google.com/view/eh-fuzz/.

EH-Fuzz can be still improved in some aspects. First, the static analysis of identifying possible error sites still has some false positives. We plan to reduce these false positives, by using a precise dataflow analysis. Second, we plan to improve the fuzzing efficiency using some techniques, such as snapshot-based [67]–[69] or parallel ways [70]–[72]. Finally, we implement EH-Fuzz for only C programs at present, and we plan to apply it to testing the programs of other programming languages (such as Java and Python).

Acknowledgment

We thank the reviewers who provided very useful suggestions, and we also thank the open-source-software developers who gave useful feedback to our bug reports. This work was supported by National Natural Science Foundation of China under Project 62002195.

References

[1] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th International Workshop on Exception Handling (WEH)*, 2008, pp. 62–68.

[2] B. Cabral and P. Marques, "Exception handling: A field study in java and. net," in *Proceedings of the 2007 European Conference on Object-Oriented Programming (ECOOP)*, 2007, pp. 151–175.

[3] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *Proceedings of the 13th International Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 484–487.

[4] F. Ebert and F. Castor, "A study on developers' perceptions about exception handling bugs," in *Proceedings of the 2013 International Conference on Software Maintenance (ICSM)*, 2013, pp. 448–451.

[5] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: detecting resource-release omission faults in error-handling code for systems software," in *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[6] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *Proceedings of the 19th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2004, pp. 419–431.

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

[8] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller, "Finding error handling bugs in openssl using Coccinelle," in *Proceedings of the 2010 European Dependable Computing Conference (EDCC)*, 2010, pp. 191–196.

[9] A. Askarov and A. Sabelfeld, "Catch me if you can: permissive yet secure error handling," in *Proceedings of the 4th International Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 45–57.

[10] C. Zuo, J. Wu, and S. Guo, "Automatically detecting SSL error-handling vulnerabilities in hybrid mobile web apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 591–596.

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

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd International Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 283–294.

[13] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th International Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 197–208.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 206–215.

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

[16] "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[18] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

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

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

[21] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: a practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 745–761.

[22] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[23] "Honggfuzz: security oriented fuzzer with powerful analysis options," https://github.com/google/honggfuzz.

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 445–458.

[25] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, 2018, pp. 475–485.

[26] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering (TSE)*, 2019.

[27] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[28] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 2019 International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 329–340.

[29] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.

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

[31] J.-J. Bai, Y.-P. Wang, H.-Q. Liu, and S.-M. Hu, "Mining and checking paired functions in device drivers using characteristic fault injection," *Information and Software Technology (IST)*, vol. 73, pp. 122–133, 2016.

[32] M. Susskraut and C. Fetzer, "Automatically finding and patching bad error handling," in *Proceedings of the 2006 European Dependable Computing Conference (EDCC)*, 2006, pp. 13–22.

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

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

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

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

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

[38] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 595–605.

[39] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "POTUS: probing off-the-shelf USB drivers with symbolic fault injection," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[40] Z.-M. Jiang, J.-J. Bai, J. Lawall, and S.-M. Hu, "Fuzzing error handling code in device drivers based on software fault injection," in *Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 128–138.

[41] P. Liu, S. Ji, X. Zhang, Q. Dai, K. Lu, L. Fu, W. Chen, P. Cheng, W. Wang, and R. Beyah, "iFIZZ: deep-state and efficient fault-scenario generation to test IoT firmware," in *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, 2021, pp. 805–816.

[42] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2595–2612.

[43] "Buffer overflow bug in FFmpeg," https://www.mail-archive.com/ffmpeg-devel@ffmpeg.org/msg128726.html.

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

[45] "Syzkaller: a kernel fuzzer," https://github.com/google/syzkaller.

[46] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[47] "ASan: address sanitizer," http://github.com/google/sanitizers/wiki/AddressSanitizer.

[48] "MSan: memory sanitizer," http://github.com/google/sanitizers/wiki/MemorySanitizer.

[49] "CLOC: count lines of code," https://cloc.sourceforge.net.

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th International Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.

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

[52] "Posix filesystem test suite," https://github.com/zfsonlinux/fstest.

[53] "Black box usb testing," https://github.com/h0rac/usb-tester.

[54] "Netperf: networking benchmark," https://github.com/Hewlett-Packard/netperf.

[55] "UBSan: undefined behavior sanitizer," http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[56] "TSan: thread sanitizer," http://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[57] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, 2016, pp. 1675–1689.

[58] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, 2016, pp. 1663–1674.

[59] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 781–797.

[60] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: unleashing use-after-free vulnerabilities in Linux kernel," in *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)*, 2015, pp. 414–425.

[61] J.-J. Bai, Y.-P. Wang, and S.-M. Hu, "Automated and reliable resource release in device drivers based on dynamic analysis," *Journal of Systems and Software (JSS)*, vol. 137, pp. 463–479, 2018.

[62] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: a

survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 11, pp. 2312–2331, 2019.

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

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

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

[66] Y. Kang, B. Ray, and S. Jana, "APEx: automated inference of error specifications for C APIs," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, 2016, pp. 472–482.

[67] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: greybox hypervisor fuzzing using fast snapshots and affine types," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 2597–2614.

[68] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2541–2557.

[69] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, 2022, pp. 166–180.

[70] Y. Wang, Y. Zhang, C. Pang, P. Li, N. Triandopoulos, and J. Xu, "Facilitating parallel fuzzing with mutually-exclusive task distribution," in *Proceedings of the 17th International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2021, pp. 185–206.

[71] V.-T. Pham, M.-D. Nguyen, Q.-T. Ta, T. Murray, and B. I. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1337–1341.

[72] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos, "Collabfuzz: a framework for collaborative fuzzing," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 1–7.